UNIT-2

Functions: Built-In Functions, Commonly Used Modules, Function Definition and Calling the Function, return Statement and void Function, Scope and Lifetime of Variables, Default Parameters, Keyword Arguments, Command Line Arguments.
Strings: Creating and Storing Strings, Basic String Operations, Accessing Characters in String by Index Number, String Slicing and Joining, String Methods, Formatting Strings.
Lists: Creating Lists, Basic List operations, Indexing and slicing in Lists, Built in Function used in List, List Methods, Advanced List processing - list comprehension. Illustrative programs: selection sort, insertion sort, merge sort.

Introduction

Modular programming refers to the process of breaking a large programming tasks into separate, smaller, more manageable subtasks or modules.

Advantages of modular programming are:

- Simplicity
- Maintainability
- Reusability
- Scoping
- Avoids naming conflicts

In Python Functions, Modules and Packages are the language constructs in Python that promote code modularization.

Functions

Functions are one of the fundamental building blocks in Python programming language which used to perform a specific task and grouped under a name. When a block of statements that needs to be executed multiple times within the program, functions are used.

- This block of code can be reused anywhere in the program any number of times.

Functions can be either Built-in Functions or User-defined functions.

Built-In Functions

Python language has several predefined functions that are always available.

- **abs(number):**
  - Return the absolute value of a number. The argument may be an integer, a floating-point number or an object that implements __abs__().
- **all(iterable):**
  - Return True, if all elements of the iterable are true (or) if the iterable is empty).
- **any(iterable):**

    - Return True, if any element of the iterable is true. If the iterable is empty, return False.
- **bin(integer):**
    - Convert an integer number to a binary string prefixed with “0b”.
- **callable(object):**
    - Return True if the object argument appears callable, False if not.
- **chr(integer):**
    - Return the string representing a character of the specified integer.
- **compile(source, filename, mode):**
    - Compiles source into a code object.
    - code = compile('print("Hello World")', 'test.py', 'exec')
    - exec(code) # Output: Hello World
- **divmod(a, b):**
    - Take two (non-complex) numbers as arguments and return a pair of numbers consisting of their quotient and remainder.
- **enumerate(iterable, start=0):**
    - Return an enumerate object. iterable must be a sequence.
- **eval():**
    - Evaluates a string as a Python expression
    - exec('print("Hello World")') # Output: Hello World
- **dir():**
    - Return the list of names in the current local scope
- **dir(object):**
    - If the object is a module object, the list contains the names of the module’s attributes.
    - If the object is a type or class object, the list contains the names of its attributes, and recursively of the attributes of its bases.
- **int()**
    - Convert string or float values into integer
- **float()**
    - Converts string or integer value into float value
- **str()**
    - Converts any value into string
- **complex(real, img)**
    - Used to create complex number, real can be a string or integer, img must be an integer
- **input()**
    - Used to read data from the user
- **print()**
- **print(f".....{ }.....")**
- **print(string.format(positional|key=value))**
    - Used to display string content on the console
- **ord()**

    - Returns an integer representing Unicode code
- **oct()**
    - Returns the octal value of the decimal
- **hex()**
    - Returns the hexa decimal value of the decimal
- **map(function, iterable)**
    - Applies the function for every element of the iterable
    - l = list(map(lambda x:x**2, [2,3,4,5])
    - print(l) # output: [4,9,16,25]
- **filter(function, iterable)**
    - Filters out the iterable elements using the function
    - l = list(filter(lambda x:x>0, [-5, -2, 0, 1, 2, 3, -1])
    - print(l) #Output: [1,2,3]

Commonly Used Modules

Modules are files containing predefined Python code that contains functions, classes, and variables.

There are three types of modules in Python:

- Built-in modules: Modules that comes with python installation
- User-defined Modules: Modules defined by the user or programmer.
- External/Third party Modules: Provided by external sources, we have to download and install them using pip.

There are actually three different ways to define a module in Python:

- A built-in module is contained in the interpreter.
- A module can be written in C and loaded dynamically at run-time.
- A module can be written in Python itself.

1. Built-in Modules:

Built-in modules are pre-defined and come with the Python Standard Library. These modules are written in C language and provide various functionalities that can be used directly by importing them in the program.

To use a module, it is to be imported into the program using the ***import*** keyword.

Syntaxes for importing modules:

import module_name

import module1, module2, ...

used to import all the contents of the module, multiple modules can also be imported at a time.

import module as alias_name

used to import a module with a short name.

Note: Module name must ne prefixed with content in the above two ways

from module import specific_method

from module import method1, method2,.....

from module import *

used to import a specific methods from the module

Some common examples of built-in modules are:

- math: Provides mathematical functions like sqrt, log, etc.
- os: Provides functions to interact with the operating system.
- random: Provides functions to generate random numbers.
- datetime: Provides classes for manipulating dates and times.

Examples:

```
import math
print(math.sqrt(16)) # Output: 4.0
-----------------------------------------------------------------
import math as m
print(m.sqrt(16)) # Output: 4.0
-----------------------------------------------------------------
from math import sqrt
print(sqrt(16)) # Output: 4.0
```

The dir() function

The built-in function dir() returns a list of defined names in a module. Without arguments, it produces an alphabetically sorted list of names.

```
import math
dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'cbrt', 'ceil', 'comb', 'copysign', 'cos', 'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc', 'exp', 'exp2', 'expm1', 'fabs', 'factorial', 'floor', 'fma', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'isqrt', 'lcm', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'nextafter', 'perm', 'pi', 'pow', 'prod', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'sumprod', 'tan', 'tanh', 'tau', 'trunc', 'ulp']
```

2. User defined modules:

User-defined modules are custom-made modules created by the user. Creating a user-defined module involves writing Python code in a file and saving it with a **.py** extension.

For example, creating a module named sample.py:

```
# sample.py
def greet(name=""):
print(f"Hello! Nice to meet you {name}")
```

To use this module in another file, import it using the import keyword:

```
# main.py
import sample
sample.greet("John") # Output: Hello! Nice to meet you John
```

3. Third party modules:

Third-party modules in Python are libraries that are not included in the standard Python distribution. These modules are created by developers other than the core Python developers and can be used to extend the functionality of Python without having to write the code from scratch**.**

Few third-party modules are:

- **NumPy**: This module is used for numerical computing in Python. It provides support for arrays, matrices, and many mathematical functions to operate on these data structures.
- **Pandas**: This module is used for data manipulation and analysis. It provides data structures like DataFrame and Series, which are essential for handling and analyzing structured data.
- **Matplotlib**: Matplotlib is a plotting library that is used to visualize large datasets.

To download and install third party modules we have to use package manager ***pip*** in the command prompt.

```
c:\> pip install numpy
```

Third-party modules are an essential part of the Python ecosystem. They provide additional functionalities that are not available in the standard library and help in building robust and efficient applications.

User Defined Function

User can create his own function and use them as and where it is needed. User-defined functions are reusable code blocks created by users to perform some specific task in the program.

Function Definition

A function in python is a block of statements which are used to perform a specific task.

The syntax for function definition:

```
def function_name(parameter_1, parameter_2, ..., parameter_n):
    statement(s)
```

In Python, a function definition consists of the

1. ***def*** keyword is used to inform the interpreter that it is a user defined function.

2. The name of the function:

- Unique name for each user defined function
- Single word, can contains a combination of letters, digits and underscore. Can have multiple words combined with '_' symbol.
- Cannot start with digit.
- keywords cannot be used as function names.

3. List of parameters enclosed in parentheses and separated by commas.

- These parameters are called as formal parameters
- A function may or may not contain parameters.

4. A colon is required at the end of function header.

5. Block of statements that defines the body of the function.

Calling the Function

Defining a function does not execute it. Defining simply names the function and specifies what to do when the function is called. Calling the function actually performs the specified actions with the indicated parameters.

The syntax for function call or calling function is,

function_name(argument_1, argument_2, ..., argument_n)

Arguments are the actual value that is passed into the calling function. These arguments are called as actual arguments. There must be a one-to-one correspondence between the formal parameters in the function definition and the actual arguments of the calling function.

When a function is called, the formal parameters are temporarily “bound” to the actual arguments.

Example:

```
#program to find the factorial of a given number.
def fact(n): #function definition
        f = 1
        for i in range(1, n+1):
                f = f * i
        print(f"factorial of {n} is {f}")

n = int(input("Enter an integer value"))
fact(n) #function call
```

Note: A function should be defined before it is called and the block of statements in the function definition are executed only after calling the function.

Return Statement

Functions in python may return a value(s) to the calling statement. So that it can be stored in a variable and used later. This can be achieved using the optional return statement in the function definition.

The syntax for return statement is:

return variable[s]

or

return [expression_list]

The return statement terminates the execution of the function definition in which it appears and returns control to the function calling statement.

Example:

```
def func1(a, b):
  c = a + b
  d = a - b
  return c, d

add, sub= func1(15, 5)
print("addition=", add, "Subtraction=", sub)
```

Void Function

In Python, it is possible to define functions without a return statement. Functions like this are called void functions, and they return None.

```
#Program to Check If a Number Is Armstrong Number or Not
user_number = input("Enter a positive number to check for Armstrong number"))
n = len(user_number)

def check_armstrong_number(number):
    result = 0
    temp = number
    while temp != 0:
        last_digit = temp % 10
        result += pow(last_digit, n)
        temp = temp // 10
    if number == result:
        print(f"Entered number {number} is a Armstrong number")
    else:
        print(f"Entered number {number} is not a Armstrong number")
```

```
user_number = int(user_number)
check_armstrong_number(user_number)
```

The arguments passed in the function calling statement and the parameters used to receive the values in the function definition may have the same variable names. However, they are entirely independent variables as they exist in different scope.

Scope and Lifetime of Variables

Scope: It refers to the visibility/availability of variables. In other words, which parts of your program can see or use the variable.

Lifetime: The lifetime of a variable refers to the duration of its existence in the main memory.

Python programs have two types of variable scopes:

- Global
- Local

Global Scope: A variable has global scope if its value is accessible and modifiable throughout your program. Global variables have a global scope.

Variables declared outside of any method are called global variables and they are accessible from their declaration line to the end of the program.

Global variables are accessible inside the function provided if the same named variable is not declared inside the function. A local variable can have the same name as a global variable, but they are totally different so changing the value of the local variable has no effect on the global variable.

Example:

```
x = 10   #Global Variable
def sample():
        x = 5 #Local variable
        print(x)   #Output: 5

sample()
print(x) #Output: 10
```

Local Scope: A variable that is defined inside a function definition is called as local variable. Local variables inside a function definition have local scope and exist as long as the function is executing.

The local variable is created and destroyed every time the function is executed, and it cannot be accessed by any code outside the function definition.

To access global variables inside the function, if the function doesn't contain the same named variable, you can directly use the variable.

Example:

```
y =  10
def sample():
  print("Y value inside function ", y)
sample()
```

To modify global variable inside a function:

If the function doesn't contain a variable that is same as global variable, to modify the global variable inside the function definition, use

syntax:

global varibale_name

then we can modify the global variable in the function definition

Example:

```
y =  10
def sample():
  global y
  print("Y value inside function before modification", y)
  y = y + 15
  print("Y value inside function after modification", y)
sample()
print("Y value outside function after modification", y)
```

If the function contains the same named variable as global variable, then **globals()** is used to access global variable inside function.

- The globals() function is used to check if a variable exists at the top-level scope (global scope). The globals() function returns a dictionary of all global variables in the current module.
- The globals() function provides access to the global symbol table as a dictionary. Variables are accessed using the keys in dictionary.

Example:

```
x = 5
def sample():
  x = 10
  print("Local x : ", x)
  print("Global x : ", globals()['x'])
sample()
```

Differences between Global and Local variables

Feature	Local Variables	Global Variables
Where created	Inside functions	Outside functions
Accessibility	Only within function	Everywhere in program
Lifetime	Until function ends	Entire program
Memory Usage	Freed automatically	Stays in memory
Best For	Temporary calculations	Settings, constants
Safety	Cannot interfere with other functions	Can be changed by any function

Parameter Passing ways

In python, a function may or may not have parameters. If the python function definition contains parameters, they are called a formal parameter. User has to send the arguments from the function calling statement to called function. Argument/parameters used in function calling statement are called as actual arguments/parameters.

Different Ways to Pass Parameters to Python Functions:

- Positional Parameters
- Default Parameters
- Keyword Arguments
- Optional Arguments
- Variable Length Parameters
- List, Tuple and Dictionary as Parameters

1. Positional argument

The most basic way to pass parameters to a function definition. The order of arguments matters and must match with the position of actual and formal arguments.

Example:

```
def greetings(name, age):
  print(f"Hello {name}, you are {age} years old.")

greetings("Bob", 25)
```

In the above example, 0^{th} arguments will always be passed to 0^{th} parameter in function definition and vice versa.

2. Default Values

In the function definition, default values can be provided for parameters. So, if arguments are not sent from the function call, default parameters are used.

Note: Defaults arguments must be placed at the end in the parameters list in function definition.
Example:

```
def addition(a, b=20, c=30):
  # variables b and c have default values
  return (a+b+c)

print("addition by passing only 'a' :", addition(10)) # b and c are defaults
print("addition by passing only 'a,b' :", addition(10,40)) # c is default
print("addition by passing only 'a,b,c' :", addition(10,40,50))
```

3. Keyword argument

Arguments can also be passed as 'key=value' pairs to function definition. The order in which arguments are passed doesn't really matter. But the total number of arguments should be equivalent to parameters that are defined in function's definition.

Note: Parameter names must be known to the caller

Example:

```
def greetings(name, age):
    print(f"Hello {name}, you are {age} years old.")

greetings(age=25, name="Bob")
```

4. Optional argument

In function definition, we can make few parameters as optional. We may or may not pass the value for those parameters. For optional parameters we must give empty value using the following syntax:

```
parameter_name = ' '
```

Note: optional parameters must be kept at the end of parameters list in function definition.

Example:

```
def name(first, last, middle=''):
  ''' middle name is optional here'''
  full_name = f"{first} {middle} {last}"
  return full_name

data = name('Guido', 'Rossum')
data1 = name('Guido', 'Rossum', 'Van')
print(data,"\n",data1)
```

5. Variable Length Parameters

Sometimes we may not be sure that how many number of arguments should be passed to a function definition ahead of time. So, Python allows us to pass variable number of arguments to a function definition. It basically creates an empty tuple or dictionary when an unknown number of parameters are sent.

This is done in two ways:

- **Variable-Length Arguments (*args)**
- **Variable-Length Keyword Arguments (**kwargs)**

*args and **kwargs are mostly used as parameters in function definitions to pass a variable number of arguments to the calling function. We can change the names also, but * and ** are mandatory.

- *args as parameter in function definition allows you to pass variable length argument to the calling function.
- **kwargs as parameter in function definition allows you to pass keyworded, variable length dictionary arguments to the calling function.

Example:

```
def var_argu(*args):  #creates a tuple data structure
  t = args
  print(t)

var_argu(10,20,30,40)
```

Example:

```
def var_kwargu(**kwargs): #creates a dictionary
      t = kwargs
      print(t)

var_kwargu(a=10,b=20,c=30,d=40)
```

6. Passing List, Tuple and Dictionary data structures as arguments

We can also pass a List, Tuple and Dictionary as an argument to a function definition.

Example:

```
#List as an argument to function definition
def list_pass(lt):
  print(lt)
  sum = 0
  for i in lt:
```

```
    sum = sum + i
  print("total sum of list elements are ", sum)

l = [10,20,30,10,40,50] #List allows duplicates
list_pass(l)
```

Example:

```
#Tuple as an argument to function
def tuple_pass(tl):
  print(tl)
  sum = 0
  for i in tl:
    sum = sum + i
  print("total sum of list elements are ", sum)

t = (10,20,30,40,50) #tuple never allow duplicates
tuple_pass(t)
```

Example:

```
#Dictionary as an argument to function
def dict_pass(dt):
  print(dt)
  print(dt.keys()) # displays all the keys
  print(dt.values()) # displays all the values
  for key in dt:
    print(dt[key]) #prints values of each key

dicti = {'A':'Apple', 'B':'Banana'} #Dictionary contains key=value pairs
dict_pass(dicti)
```

Command Line Arguments

A Python program can also accept any number of arguments from the command line. Command line arguments is a methodology in which user will give inputs to the program through the console at the time of executing the program.

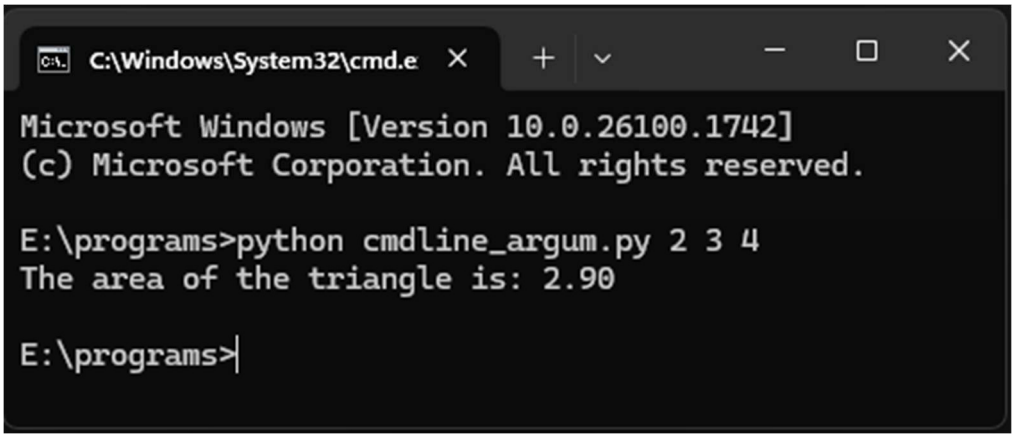
To use command line arguments, program must import sys module. All the command line arguments in Python are stored in a list of strings called 'argv'.

Example: argv=['program_name', 'arg1', 'arg2', ... , 'argn']

Command line arguments are accessed in the program using the index numbers.

Example:

```
#cmdline_argum.py
# Area of a triangle with their sides
import sys, math
#Enter triangle sides through command line
a=float(sys.argv[1])
b=float(sys.argv[2])
c=float(sys.argv[3])
if len(sys.argv) != 4:
  print("Enter 3 values")
  sys.exit(1)
if a+b > c and a+c > b and b+c > a:
  s = (a+b+c)/2
  area = math.sqrt(s*(s-a)*(s-b)*(s-c))
  print(f"The area of the triangle is: {area:.2f}")
else:
  print("The given sides do not form a valid triangle.")
```



Recursive Functions

Recursion is the process of defining something in terms of itself. In Python, a function can call other functions. It is even possible that a function can call itself.

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function.

Syntax:

```
def sample():
        ......
        sample()
        ......

sample()
```

Every recursive function must have a base condition that stops the recursion process, if not the function calls itself infinitely.

Example1:

#Program to find the factorial of a number using recursive function

```
def facto(n):
  if n == 0 | n == 1:   #base condition
    return n
  else:
    return n * facto(n-1)

n = int(input("enter n value"))
print(f" Factorial of {n} is {facto(n)}")
```

Example2:

Recursive function to display Fibonacci series

```
def fibonacci_recursive(n):
  if n <= 1: # Base cases
    return n
  else:
    return fibonacci_recursive(n-1) + fibonacci_recursive(n-2)

# Number of terms to display
n = int(input("Enter n value"))
print("Fibonacci sequence:")
for i in range(n):
  print(fibonacci_recursive(i), end=" ")
```

Advantages of Recursion

- Recursive functions make the code look clean and elegant.
- A complex task can be broken down into simpler sub-problems using recursion.
- Sequence generation is easier with recursion than using some nested iteration.

Disadvantages of Recursion

- Sometimes the logic behind recursion is hard to follow through.
- Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
- Recursive functions are hard to debug.

Strings: Creating and Storing Strings, Basic String Operations, Accessing Characters in String by Index Number, String Slicing and Joining, String Methods, Formatting Strings.

Introduction

String is basic data type available in Python. A string in python is a sequence of characters, which includes letters, numbers, punctuation marks and spaces.

In python strings are immutable, i.e. one a string value is assigned to a variable, it cannot be modified. But we can create new strings from the original string by manipulating the content of the string.

Creating and Storing Strings

Single quote, double quotes or triple quotes are used to create string constants in python.
To store a string value inside a variable, you just need to assign a string to a variable.

- Single character is also called as string. str1=”c”
- Empty strings are allowed in python. str1=""

Python strings are of str data type. To display the data type of a string variable, use type() and pass string variable as an argument to it.

Basic String Operations

The str() Function

The str() function returns a string version of the object. If the object is not provided, then it returns an empty string.

```
>>> str(10)  # Output: '10'
>>> str1 = str()
>>> type(str1)  # Output: <class 'str'>
```

String Concatenation

In Python, strings are concatenated by using '+' operator
Example:

```
>>>str1="Hello "
>>>str2="Everyone"
>>>result = str1+str2
>>>print(result) # Output: Hello Everyone
```

Note: don't use '+' operator to concatenate different types of data.

String Repetition

'*' operator is used to create a repeated sequence of strings
Example:

```
>>>str1="wow"
>>>result=str1*3
>>>print(result)  #Output: wowwowwow
```

The presence of a string in another string is checked by using 'in' and 'not in' (membership) operators. It returns a Boolean True or False. The 'in' operator evaluates to True if the string value in the left operand appears in the sequence of characters of string value in right operand. The 'not in' operator evaluates to True if the string value in the left operand does not appear in the sequence of characters of string value in right operand.

String Comparison

Comparison of two string returns a Boolean value. You can use (>, <, <=, >=, ==, !=) to compare two strings. Python compares strings using ASCII value of the characters.

Accessing Characters in String by Index Number

When string variable is stored in the memory, each character in the string has an index number. First character in the string is always at index 0, next character is at index 1, and so on. The length of a string is the number of characters in it.

0	1	2	3	4	5	6	7	8	9	10
H	E	L	L	O		W	O	R	L	D

Each character in a string is accessed by using a subscript operator i.e., a square bracket []. Square brackets are used to perform indexing in a string to get the value at a specific index.

The syntax for accessing an individual character in a string is as shown below.

string_name[index]

where index is usually in the range of 0 to one less than the length of the string.

The value of index should always be an integer and indicates the character to be accessed.
If the specified index number is more than the number of characters in the string, then it results in IndexError: string index out of range error.

Python also supports negative indexing. So, individual elements of a string are also accessed by using negative indexes.

H	E	L	L	O		W	O	R	L	D
-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Example: from a large sting constant, if you want to access the last character, simply use '-1' as index number.

String traversing using for loop

String in python is a sequence of characters. So, any sequence can be iterated by using for loop.
Example:

```
str1 = "Hello World"
for i in str1:
    print(i)
```

Example:

'''Write Python Program to Count the Total Number of Vowels, Consonants and Blanks in a String'''

```
vowels = "aeiou"
def vcbcounts():
  user_string = input("Enter a string: ")
  vcount = 0
  ccount = 0
  blanks = 0
  for i in user_string:
    if i in vowels:
      vcount+=1
    elif "a" <= i <= "z":
      ccount+=1
    else:
      blanks+=1
  print(f"Number of Vowels = {vcount}")
  print(f"Number of Consonants = {ccount}")
  print(f"Number of Blanks = {blanks}")

vcbcounts()
```

Example:

#Program to Print the Characters Which Are Common in Two Strings

```
def common_characters(string_1, string_2):
        for letter in string_1:
                if letter in string_2:
                        print(f"Character '{letter}' is found in both the strings")

common_characters('rose', 'goose')
```

String Slicing

To retrieve a substring or sub-part of a sequence, slice operator is used in python.

The syntax for string slicing is,

string_name[start:end[:step]]

With string slicing, a sequence of characters is accessed by specifying a range of index numbers separated by a colon. String slicing returns a sequence of characters beginning at start index and extending up to end-1 index. The start and end indexing values must be integers.

- If start index is not mentioned in the slice operator, slicing starts from 0^{th} index.

- If the end index is not mentioned in the slice operator, slicing ends at last index.
- If both the start and end index values are missing, then the entire string is displayed.
- If the start index is equal to or higher than the end index, then it results in an empty string.
- If the end index number is beyond the end of the string, it stops at the end of the string.

In the slice operation, third argument 'step' which is an optional can be specified along with the start and end index numbers. This 'step' refers to the number of characters that can be skipped after the start indexing character in the string. The default value of step is one.

String slicing can also be done by using negative indexing or both positive and negative indexes. When using negative indexing for substring, specify the lowest negative integer number in the start index position. You can also combine positive and negative indexing numbers (positive number for start and negative indexes for end).

'''Write Python Code to Determine Whether the Given String is a Palindrome or Not Using Slicing operator'''

```
def palindrome():
    user_string = input("Enter string: ")
    if user_string == user_string[::-1]:
        print(f"User entered string is palindrome")
    else:
        print(f"User entered string is not a palindrome")
palindrome()
```

String Joining

Strings can be joined by using join() method. The syntax of join() method is:

string_name.join(sequence)

sequence can be a string or list.

If the sequence is a string, then join() function inserts string_name between each character of the string sequence and returns the concatenated string.

```
>>> numbers = "123"
>>> characters = "amy"
>>> password = numbers.join(characters)
>>> password #Output: 'a123m123y'
```

If the sequence is a list, then join() function inserts string_name between each item of list sequence and returns the concatenated string.

```
>>> date_of_birth = ["17", "09", "1950"]
>>> ":".join(date_of_birth) #Output: '17:09:1950'
```

Splitting String Using split() Method

A given string is divided into a list of strings based on the specified separator/delimiter.
The split() method returns a list of string items by breaking the string using the delimiter. The syntax of split() method is:

string_name.split([delimiter])

Here the delimiter string is optional. A given string is divided into list of strings based on the specified delimiter. If the delimiter is not specified then whitespace is considered as the delimiter string to split the string.

String Methods

To get the list of methods associated with string, dir(str) is used.
>>>dir(str)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'removeprefix', 'removesuffix', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']

In Python, names with double underscores before and after (e.g., __init__, __str__) has a specific purpose. These names are referred to as "dunder" (double underscore) methods or magic methods. Python uses them internally to implement certain behaviours or functionalities.

They are special because:

- Double underscore methods are not meant to be called directly by the user. Instead, they are invoked implicitly by Python in specific situations.
- These methods are reserved for Python's internal use, ensuring that user-defined methods or attributes do not conflict with Python's built-in functionality.
- Avoid creating user defined methods with double underscores. but we can override or extends them.

len(): It calculates the number of characters in a string. The white space characters are also counted.
Syntax: ***len(string_variable)***
max(): max() function returns a character having highest ASCII value.
Syntax: ***max(string_variable)***
min(): min() function returns character having lowest ASCII value.

Syntax: ***min(string_variable)***
replace(old,new[,max]): used to replace the occurrences of old with new upto max number of times.
upper(): converts the string into uppercase letters
lower(): converts the string into lowercase letters
title(): converts the string into titlecased (all the words begin with capital letter)
swapcase(): converts lowercase to uppercase and uppercase to lowercase.
capitalize(): Used to make the first letter of the entire string as capital letter and the rest lowercased.
Syntax: ***string_variable.capitalize()***
center(width[,fillchar): Used to make the string_variable as center based on the width value. Default filler is space, or depends on fillchar value.

```
a = 'hello'
a.center(15)
'    hello     '
```

count(substring[,start[,end]]): used to count the occurrences of substring in the range of start and end. Start and End are optional.

startswith(prefix[,start[,end]]): returns true, if the string starts with specified prefix in the range of start to end, otherwise false.
endswith(suffix[,start[,end]]): returns true if the string ends with specified suffix in the range of start to end. Start and end are optional.

find(substring[,start[,end]]): searches for substring, and returns the position of the first character of substring in the given string in the range of start and end indexes. if not returns -1.

isalnum(): If all characters in a string are alphanumeric, it returns True, otherwise, returns False**.**
isalpha(): returns true, if all the characters are alphabetics, otherwise false
isdecimal(): returns true, if all the characters are decimals, otherwise false
isdigit(): returns true, if all the characters are digits, otherwise false
isidentifier(): returns true, if the string is valid identifier, otherwise false
islower(): returns true, if all the characters in the string are lowercase, otherwise false
isupper(): returns true, if all the characters in the string are uppercase, otherwise false
isspace(): returns true, if there are only whitespace characters in the string, otherwise false
isnumeric(): returns true, if all the characters in the string are numeric (digits and Unicode numeric value), otherwise false.
istitle(): returns true, if the string is a title cased string, otherwise false.

ljust(): retruns the string in left justification
rjust(): returns the string in right justification
strip(): used to remove leading and trailing spaces from a string. You can also specify a set of characters to remove from the beginning and end of the string.
lstrip(): remove leading (beginning of the string) spaces from a string.
rstrip(): remove trailing (end of the string) spaces from a string.
zfill(length): pads a string with zeros on the left until it reaches a specified length. This method does not modify the original string, it returns a new string with the padding applied.

Example:
'''Write Python Program That Accepts a Sentence and Calculate the Number of Words, Digits, Uppercase Letters and Lowercase Letters'''

```
def string_processing(user_string):
  word_count = 0
  digit_count = 0
  upper_case_count = 0
  lower_case_count = 0
  for each_char in user_string:
    if each_char.isdigit():
      digit_count += 1
    elif each_char.isspace():
      word_count += 1
    elif each_char.isupper():
      upper_case_count += 1
    elif each_char.islower():
      lower_case_count += 1
    else:
      pass
  print(f"Number of digits in sentence is {digit_count}")
  print(f"Number of words in sentence is {word_count + 1}")
  print(f"Number of upper case letters in sentence is {upper_case_count}")
  print(f"Number of lower case letters in sentence is {lower_case_count}")

user_input = input("Enter a sentence ")
string_processing(user_input)
```

Formatting Strings

Python supports multiple ways to format text strings.

Few of them are:

- %-formatting
- string.format()
- f-strings

%-formatting

The old-style formatting uses the % operator and it resembles C-style formatting.

Syntax:

"format specifier" % (variable/value)
format specifiers: %s, %d.....

Example:
>>>name="Bob"
>>>age=30

>>>print("Name = %s, Age = %d" %(name, age)) #Output: Name = Bob, Age = 30

string.format()

This style of formatting is used to insert value of a variable, expression or an object in the placeholder {} of the string.

The syntax for *format()* method is,

string.format(p0, p1, ...)

string.format(k0=v0, k1=v1, ...)

where

- p0, p1,...pn are called as positional arguments, 0,1,.....n are indexes
- k0, k1,...kn are keyword arguments with their assigned values are v0, v1,...vn. Keys are replaced by values.

Positional arguments are a list of arguments that are accessed with an index of argument inside curly braces like {index}. Index value starts from zero.

Keyword arguments are a list of arguments of type *keyword* = *value*, that can be accessed with the name of the argument inside curly braces like {keyword}.

Example:

```
>>>name = “Bob”
>>>age = 30
>>>print("Name = {0} and Age = {1}".format(name, age))
    Name = Bob and Age = 30
```

Example:

```
>>>name = “Bob”
>>>age = 30
>>>print("Name = {K2} and Age = {K1}".format(K1=age, K2=name))
    Name = Bob and Age = 30
```

f-strings

Formatted strings or f-strings were introduced in Python 3.6. A *f-string* is a string literal that is prefixed with “f”. These strings may contain placeholders, which are expressions/variables enclosed within curly braces {}. At the time of displaying output on the console/screen expressions/variables are replaced by their respective values.

Syntax:

print(f"string {} string”)

Example: Given the Radius, Write Python Program to Find the Area and Circumference of a Circle

```
radius = int(input("Enter the radius of a circle"))
area_of_a_circle = 3.1415 * radius * radius
```

circumference_of_a_circle = 2 * 3.1415 * radius
print(f"Area = {area_of_a_circle} and Circumference = {circumference_of_a_circle}")

f-string format specifiers may also contain evaluated expressions.

The syntax for f-string formatting operation is,

f'string_statements {variable_name [: {width}.{precision}]}'

Specifying width and precision values are optional. If they are specified, they should be included within curly braces along with variable name separated by a colon.

Escape Sequences

Escape Sequences are a combination of a backslash (\) followed by either a letter or a combination of letters and digits. Escape sequences are also called as control sequences.

The backslash (\) character is used to escape the meaning of characters that follow it by substituting their special meaning with an alternate interpretation. So, all escape sequences consist of two or more characters.

Escape Sequence	**Meaning**
\	Break a line into multiple lines while ensuring the continuation of the line
\\	Inserts a backslash character in the string
\'	Inserts a single quote character in the string
\"	Inserts a double quote character in the string
\n	Inserts a new line in the string
\t	Inserts a tab in the string
\r	Inserts a carriage return in the string
\b	Inserts a backspace in the string
\u	Inserts a Unicode character in the string
\0o	Inserts a character in the string based on its octal value
\xhh	Inserts a character in the string based on its hex value

Raw Strings

In python, a string which ignores all type of string formatting including escape characters, that strings are called as raw strings.
A raw string is created by prefixing the character 'r' to the string.

Example:

```
str1=r"Example for \n raw string"
```

Unicode

It is a standard that provides unique number for every character in the world languages. Computers are only capable of storing numbers. So, to store or to transmit every character across the globe onto every platform, a standard coding is required for all the characters in the world languages.

Unicode was adopted by all modern software providers and now allows data to be transported through many different platforms, devices and applications.

Regular Python strings are not Unicode: they are just plain bytes. To create a Unicode string, use the 'u' prefix on the string literal.

Example:

```
str1 = u"A Unicode \u018e string \xf1"
```

Lists: Creating Lists, Basic List operations, Indexing and slicing in Lists, Built in Function used in List, List Methods, Advanced List processing - list comprehension.
Illustrative programs: selection sort, insertion sort, merge sort.

Introduction

In general, a variable can hold only a single value. If it is required to store more values, a greater number of variables are needed for holding those values. Declaring more number of variables may not make much sense. So, to hold multiple values that are related to each other a special variable is needed.

List is a data structure in python in which we can store multiple values that are related together.

Lists in Python are powerful data structures that are used to store and manipulate collections of items. They are dynamic, mutable, and ordered, making them suitable for various tasks such as organizing data, managing tasks, and solving complex problems.

Lists are similar to arrays (dynamic arrays) that allow us to store items of different data types.

Few characteristics of Lists:

- List is an ordered collection of elements.
- List is mutable (elements can be updated).
- List can have elements of different types.
- List elements are stored in sequential order, so they are indexed (index starts from '0')
- Duplicates are allowed in list.

List can store any item like string, number, object, another variable and even another list. You can have a mix of different item types.

Creating Lists

List is created by placing elements inside square brackets [], separated by commas.
The syntax for creating list is:

list_name = [item1, item2, item3, ... , itemn]

Empty List

```
>>> empty_list = []
>>>print(empty_list)   #Output: []
```

List with elements

```
>>>l = [4, 5, 4, 6, 7]
>>>print(l)  #Output: [4, 5, 6, 7]
```

List with different types of data

```
>>>student = ['Bob', 32, 'Computer Science', [2, 4]]
>>>print(student)
```

List can also be created by using **list()** function. it accepts only one argument.

The syntax for list() function:

list([sequence])

where the sequence can be a string, tuple or list itself. If the optional sequence is not specified then an empty list is created.

list() method is also used to convert other sequence types into list type.

```
>>> l = list()
>>> print(l)    # Output: []
>>> l = list((1,2,3,5))
>>> print(l)  #Output: [1,2,3,5]
>>> str1= "hello world"
>>> l = list(str1)
>>> print(l)   #Output: ['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd',]
>>> t= (1, 2, 3, 4)  #tuple
>>> l = list(t)
>>> print(l)  #Output: [1, 2, 3, 4]
```

Basic List operations

In Python, lists are concatenated using the **'+'** sign, and the **'*'** operator is used to create a repeated sequence of list items.

Example:

List Concatenation

```
>>> list_1 = [1, 3, 5, 7]
>>> list_2 = [2, 4, 6, 8]
>>> list_1 + list_2
[1, 3, 5, 7, 2, 4, 6, 8]
```

List Repetition

```
>>> list_1 * 3
[1, 3, 5, 7, 1, 3, 5, 7, 1, 3, 5, 7]
```

List Comparison

```
>>> list_1 == list_2
False
```

Membership operator (in, not in) is used to check for the presence of an item in the list. It returns a Boolean True or False.

Example:

```
>>> list_items = [1,3,5,7]
>>> 5 in list_items
True
```

```
>>> 10 in list_items
False
>>> 10 not in list_items
True
```

Indexing and slicing in Lists

Indexing

List is an ordered sequence of elements, each item in a list can be accessed individually through indexing.

In a list, first item is always at index 0, the second item at index 1 and so on. Lists use square brackets [ ] to access individual elements, and the number inside the bracket is called index.

The syntax for accessing an item in a list is:

list_name[index]

where index must be an integer value and it should be within the range of length of the list. If index is not available in the list, "list index out of range" exception is raised by interpreter

List also supports negative indexing. Indexes start with '-1' from the end of the list to starting element.

Beside accessing individual elements of a list, we can also modify the content at the index position.

```
>>> l = [1,2,3,4]
>>> print(l)   #Output: [1, 2, 3, 4]
>>> l[2] = 5
>>>print(l)   #Output: [1, 2, 5, 4]
```

Slicing of list

Python allows us to access a part of the list by specifying index range along with the colon (:) operator.

The syntax for list slicing is:

list_name[start:stop[:step]]

where start, stop and step are integer number within the range of length of the list.

The slicing operation helps us to retrieve the sub list from **start** to **end-1** index from the original list.

- start indicates the starting index of the slice
- stop indicates the end index of the slice
- step indicates the increment (optional) default is 1.

Negative indexes also used to retrieve sub list of the original list.

Example:

```
>>> fruits = ["grapefruit", "pineapple", "blueberries", "mango", "banana"]
>>> fruits[-3:-1]    # Output: ['blueberries', 'mango']
```

[: : -1] is used to print the list in reverse order

```
>>> print(fruits[: :-1])  #Output: ['banana', 'mango', 'blueberries', 'pineapple', 'grapefruit']
```

Built in Function used in List

To perform few operations on list data structure, python provides few built-in functions, in which list variable must be passed as an argument.

len(): It returns the numbers of items in a list.

```
>>> l = list([5,4,3,2,1])
>>>print(len(l))  #Output: 5
```

sum(): It returns the sum of numbers in the list.

```
>>>print(sum(l)) #Output: 15
```

any(): This function returns True if any of the Boolean values in the list is True.

```
>>> l = [1, 1, 1, 0, 1]
>>>print(any(l))  #Output: True
```

all(): This function returns True if all the Boolean values in the list are True, else returns False.

```
>>> l = [1,1,1,1,1]
>>> print(all(l)) #Output: True
>>> l = [0,1,0,1]
>>>print(all(l))  #Output: False
```

sorted(): This function returns a sorted copy of the list in ascending order.

```
>>>l = [2, 5, 4, 3, 1, 9]
>>>print(sorted(l))  # Output: [1, 2, 3, 4, 5, 9]
```

If all the list elements are numeric, we can find the maximum element and minimum element in the list by using the following methods.

max(): max(list_var)

min(): min(list_var)

List Methods

List is mutable ordered collection of elements. So, we can add or delete elements from the given list.

To get all the methods associated with list, use dir() method.

```
>>>dir(list)
['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__', '__delitem__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getstate__', '__gt__',
'__hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',
```

'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']

append(): this method is used to add the elements at the end of the existing list.

list.append(element's):

insert(): this method is used to insert an element at the specified index, if element already exist at that index, it will be moved towards right side.

list.insert(index, element)

index(): this method is used to know the index of the specified element.

list.index(element)

count(): this method is used to count the number of occurrences of the given element.

list.count(element)

extend(): this method is used to add another list at the end of an existing list

list1.extend(list2)

remove(): this method is used to remove the specified element from the list, after removal, list will be adjusted automatically

list.remove(element)

pop(): this method is used to remove the last element of the list and displays the deleted element.

list.pop()

pop(index): this method removes the element available at the specified index and returns the element.

list.pop(index)

reverse(): this method is used to reverse the elements in the given list

list.reverse()

sort(): this method is used to sort the elements in ascending order.

list.sort()

for descending order, use reverse=True

list.sort(reverse=True)

copy(): this method is used to create a copy of the existing list.

list.copy()

clear(): this method is used to remove all the elements from the list. Empty list exists in the memory.

list.clear()

The *del* Statement
You can remove an item from a list based on its index rather than its value.

The difference between ***del*** statement and ***pop()*** function is that the ***del*** statement does not return any value while the pop() function returns a value.

The del statement can also be used to remove slices from a list or clear the entire list.
Example:

```
>>> l = [2,3,4,6,8,1]
>>> del l[3]  #deletes the 3rd indexed element from the list l.
>>> print(l)
        [2,3,4,8,1]
>>> del l[1:3]   # deletes the elements from 1 to 3-1 indexes in the list l.
        [2,8,1]
>>> del l[:]  # deletes all the elements from the list l.
```

Advanced List processing

Creating a list of elements dynamically

Method-1:
for() loop is used to create a list dynamically by reading elements one by one from the user and appending to an existing list.
The following steps are used to create a list dynamically:

- Step1: Create an empty list
- Step2: Based on the required number of elements, use the for loop upto the range of required number.
- Step3: Read one element at a time and append it to the list.

Example:

```
l1 = []  #empty list
for i in range(10):
        n = int(input("enter a integer value:"))
        l1.append(n)

print(l1)
```

Method-2:
We can also create a list dynamically by using input() and split() methods. If a list is created using only input() and split() methods, list contains a collection of string elements.

Example:

```
l1 = input("enter elements separated by a space").split(" ")
```

Method-3:
We can also create a list dynamically using **list(), map()** and **split()** methods.

Syntax:

l1 = list(map(int, input("enter elements separated by a space").split(" ")))

The above statements read a collection of elements separated by space. Then the collection is divided into individual elements based on the delimiter specified in the split() method. Then each element is mapped with int method and stored in l1.

Accessing elements from a list:
Each element of a list is accessed by using for statement.
Example:

```
l = [4,5,3,6,1,2,7,8]
for i in l:
    print(i)
```

Example:
#Find Mean, Variance and Standard Deviation of List Numbers

```
import math
def statistics(list_items):
  mean = sum(list_items)/len(list_items)
  print(f"Mean is {mean}")
  variance = 0
  for item in list_items:
    variance += (item-mean)**2
  variance /= len(list_items)
  print(f"Variance is {variance}")
  standard_deviation = math.sqrt(variance)
  print(f"Standard Deviation is {standard_deviation}")

l = list(map(int, input("enter integers separated by space:").split(" ")))
statistics(l)
```

Nested Lists

It is possible to place a list inside another list in python. A list inside another list is called as nested list. You can traverse through the items of nested lists using indexes and the for loop.
Syntax of nested list:

```
list_name = [[i1,i2,i3], [i4,i5,i6], [i7,i8,i9].....]
```

from the above syntax, list_name[0] indicates first nested list, list_name[1] indicated the second nested list and so on.....
list_name[0][0] indicates the first element of the first nested list.

So, with the help of indexes we can access the nested list elements using for loop.

Example:

```
#Write Python Program to Add Two Matrices

m1 = [[1,2,3],[4,5,6],[7,8,9]]
m2 = [[1,2,3],[4,5,6],[7,8,9]]
m3 = [[0,0,0],[0,0,0],[0,0,0]]
for r in range(len(m1)):
   for c in range(len(m2[0])):
      m3[r][c] = m1[r][c] + m2[r][c]

print("Addition of two matrices is")
print(m3)
'''for items in m3:
   print(items)'''
```

List comprehension

List comprehension is a programming construct available in python to create a new list form the existing list by applying an expression on each element of the existing list.

The basic syntax of list comprehension is:

new_list = [*expression* ***for item in*** *iterable* ***[if*** *condition****]]***

- **expression:** The operation you want to perform on each iterable item.
- **item:** The current item from the iterable.
- **iterable:** The collection of items you are iterating over (may be an existing list).
- **condition:** (Optional) A filter that determines whether the item should be included in the new list.

Example:

```
>>> l = [1,2,3,4]
>>> l1 = [i*2 for i in l]  # doubles all the elements of l
>>> numbers = [i for i in range(1,20) if i % 2 == 0] # generates a list of all even numbers from 1 to 19
```

Example:

```
# program to create a list of even number from an existing list using list comprehension
```

Comparison between for loop and list comprehension

List comprehension makes the code cleaner and more concise than for loop.

```
numbers = [1, 2, 3, 4, 5]
squared_numbers = []
# for loop to square each element
for n in numbers:
    squared_numbers.append(n * n)

print(squared_numbers)
# Output: [1, 4, 9, 16, 25]
```

```
numbers = [1, 2, 3, 4, 5]
# Create a new list using list comprehension
squared_numbers = [n * n for n in numbers]
print(squared_numbers)
# Output: [1, 4, 9, 16, 25]
```

Illustrative programs:

Selection sort

Selection Sort is a comparison sorting algorithm that is used to sort a random list of items in ascending order. It is a simple sorting algorithm that repeatedly selects the smallest element from the unsorted portion of an array and swaps it with the first unsorted element.

Procedure:

The given list is divided into two partitions: The first list contains sorted items, while the second list contains unsorted items.

By default, the sorted list is empty, and the unsorted list contains all the elements. The unsorted list is then scanned for the minimum value, which is then placed in the sorted list. This process is repeated until all the values have been compared and sorted.

How does selection sort work?

The first item in the unsorted partition is compared with all the values to the right-hand side to check if it is the minimum value. If it is not the minimum value, then its position is swapped with the minimum value.

```
def selection_sort(arr):
    n = len(arr)
    for i in range(n - 1):
        min_index = i
        for j in range(i + 1, n):
            if arr[j] < arr[min_index]:
```

```
            min_index = j
        # Swap the smallest element with the first unsorted element
        arr[i], arr[min_index] = arr[min_index], arr[i]

arr = list(map(int, input("Enter few elements separated by comma and press enter").split(" ")))
selection_sort(arr)
print("Sorted array:", arr)
```

Selection sort is known as in-place sorting. It has a time complexity of O(n2) where n is the total number of items in the list. The time complexity measures the number of iterations required to sort the list.

Insertion sort

Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.

Working of Insertion Sort:

Step1: The first element in the array is assumed to be sorted. Take the second element and store it in a separate variable called 'key'.
Step2: Compare 'key' with the first element. If the first element is greater than key, then key is placed in front of the first element. Then the first two elements are sorted.
Step3: Take the third element and compare it with the elements on the left of it. Placed it just behind the element smaller than it. If there is no element smaller than it, then place it at the beginning of the array.
Step4: repeat step3 for the remaining elements.

```
def insertion_Sort(arr):
  for step in range(1, len(arr)):
    key = arr[step]
    j = step - 1

    # Compare key with each element on the left of it until an element smaller than it is found
    # For descending order, change key<arr[j] to key>arr[j].
    while j >= 0 and key < arr[j]:
      arr[j + 1] = arr[j]
      j = j - 1

    # Place key at after the element just smaller than it.
    arr[j + 1] = key

data = list(map(int, input("Enter few elements separated by comma and press enter").split(" ")))
insertion_Sort(data)
print('Sorted Array list in Ascending Order:')
```

print(data)

Time complexity of insertion sort in the average and worst case is $O(n^2)$ and in the best case is O(n). Space complexity is O(1).

Merge sort

Merge Sort is one of the most popular sorting algorithms that is based on the principle of Divide and Conquer Algorithm.

In Divide and Conquer technique, a problem is divided into multiple sub-problems. Each sub-problem is solved individually until a base case appears. Finally, the results of the sub-problems are combined to form the final solution.

Merge Sort Algorithm

The Merge Sort Algorithm repeatedly divides the given list of into two halves until we reach a stage where we try to perform Merge Sort on a sub list of size 1.

MergeSort Program in Python

```
def mergeSort(array):
  if len(array) > 1:
    #  r is the point where the array is divided into two subarrays
    r = len(array)//2
    L = array[:r]
    M = array[r:]

    # Sort the two halves
    mergeSort(L)
    mergeSort(M)

    i = j = k = 0

    # Until we reach either end of either L or M, pick larger among
    # elements L and M and place them in the correct position at A[p..r]
    while i < len(L) and j < len(M):
      if L[i] < M[j]:
        array[k] = L[i]
        i += 1
      else:
        array[k] = M[j]
        j += 1
      k += 1

    # When we run out of elements in either L or M,
    # pick up the remaining elements and put in A[p..r]
```

```
        while i < len(L):
            array[k] = L[i]
            i += 1
            k += 1

        while j < len(M):
            array[k] = M[j]
            j += 1
            k += 1

#array = [6, 25, 12, 10, 19, 1]
array = list(map(int, input("Enter integers separated by space:").split(" ")))
mergeSort(array)
print("Sorted array is: ")
for i in range(len(array)):
    print(array[i], end=" ")
```