

# **PROGRAMMING FOR PROBLEM SOLVING [R24A0501] LECTURE NOTES**

**B.TECH I YEAR – I SEM(R24)  
(2024-25)**



**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**MALLA REDDY COLLEGE OF  
ENGINEERING & TECHNOLOGY**

**(Autonomous Institution – UGC, Govt. of India)**

Recognized under 2(f) and 12 (B) of UGC ACT 1956

(Affiliated to JNTUH, Hyderabad, Approved by AICTE - Accredited by NBA & NAAC – 'A' Grade - ISO 9001:2015 Certified)

Maisammaguda, Dhulapally (Post Via. Hakimpet), Secunderabad – 500100, Telangana State, India

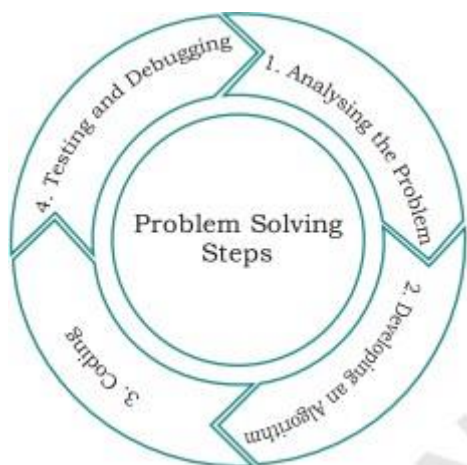
## PREFACE

Today, computers are all around us. We use them for doing various tasks in a faster and more accurate manner. For example, using a computer or smartphone, we can book train tickets online.

We usually use the term computerisation to indicate the use of computer to develop software in order to automate any routine human task efficiently. Computers are used for solving various day-to-day problems and thus problem solving is an essential skill that a computer science student should know. It is pertinent to mention that computers themselves cannot solve a problem. Precise step-by-step instructions should be given by us to solve the problem. Thus, the success of a computer in solving a problem depends on how correctly and precisely we define the problem, design a solution (algorithm) and implement the solution (program) using a programming language. Thus, problem solving is the process of identifying a problem, developing an algorithm for the identified problem and finally implementing the algorithm to develop a computer program.

### Steps for Problem Solving

When problems are straightforward and easy, we can easily find the solution. But a complex problem requires a methodical approach to find the right solution. In other words, we have to apply problem solving techniques. Problem solving begins with the precise identification of the problem and ends with a complete working solution in terms of a program or software.



**Steps for Problem Solving**

**Analysing the problem:** we need to read and analyse the problem statement carefully in order to list the principal components of the problem and decide the core functionalities that our solution should have. By analysing a problem, we would be able to figure out what are the inputs that our program should accept and the outputs that it should produce

**Developing an Algorithm:** It is essential to devise a solution before writing a program code for a given problem. The solution is represented in natural language and is called an algorithm. We start with a tentative solution plan and keep on refining the algorithm until the

algorithm is able to capture all the aspects of the desired solution. For a given problem, more than one algorithm is possible and we have to select the most suitable solution

**Coding:** After finalising the algorithm, we need to convert the algorithm into the format which can be understood by the computer to generate the desired solution. Different high level programming languages can be used for writing a program.

**Testing and Debugging:** The program created should be tested on various parameters. The program should meet the requirements of the user. It must respond within the expected time. It should generate correct output for all possible inputs.

This course introduces Python Language as a coding tool.

Python is a widely used general-purpose, high level programming language. It was created by Guido van Rossum in 1991 and further developed by the Python Software Foundation.

### **Why Python**

It was designed with an emphasis on code readability, and its syntax allows programmers to express their concepts in fewer lines of code. Python is a programming language that lets you work quickly and integrate systems more efficiently.

This digital notes serves as an introductory material for first year students, enabling them to understand problem solving through python.

The content is divided into five units as per syllabus

**Unit I** – Introduction to Computer System, Problem Solving with algorithms and Flowcharts. Basics of Python Language – Tokens, Data types – Basic and Collection types

**Unit II** – Operators and Control structures

**Unit III** – Arrays using numpy

**Unit IV** – Modular Programming using functions

**Unit V** – Files and Exception Handling

We would like to extend our sincere gratitude to Dr. VSK Reddy, Director, Malla Reddy College of Engineering and Technology (autonomous) and Dr. S. Srinivasa Rao, Principal, Malla Reddy College of Engineering and Technology, under whose patronage we were able to write this content. We are also indebted to Dr.S. Shanthi, Head of the Department, Computer Science and Engineering for her constant support and motivation for our academic growth. We would also like to acknowledge our colleagues who have provided their valuable suggestions in preparing this content.

# MALLA REDDY COLLEGE OF ENGINEERING AND TECHNOLOGY

I Year B.Tech-I Sem

L /T/P/C

3/-/-/3

## (R24A0501) PROGRAMMING FOR PROBLEM SOLVING

### COURSE OBJECTIVES:

The students will be able

1. To understand basics of programming.
2. To learn how to use conditional statements and loops.
3. To structure Python programs using arrays.
4. To know the need and usage of functions
5. To learn file operations and exception handling

### UNIT - I

**Introduction to Programming** – Computer Systems, Computer Languages, Algorithms and Flowcharts

**Introduction to Python Language:** Introduction to Python Language, Features of Python, Comments in Python.

**Tokens-** Keywords, Identifiers, Constants, Variables, Python Input and Output Statements

**Basic Data Types:** int, float, boolean, complex and string and its operations.

**Collection Data Types:** List, Tuples, Sets and Dictionaries. Data Type conversions,

### UNIT - II

Operators in Python: Arithmetic operators, Assignment operators, Comparison operators, Logical operators, Identity operators, Membership operators, Bitwise operators, Precedence of operators, Expressions.

**Control Flow and Loops:** Indentation, if statement, if-else statement, nested if else, chained conditional if- elif -else statement, Loops: while loop, for loop using ranges, Loop manipulation using break, continue and pass.

### UNIT- III

**Arrays:** Definition, Advantages of Arrays, Creating an Array, Operations on Arrays, Arrays vs List, Importing the Array Module, Indexing and Slicing on Arrays,

**working with arrays using numPy** - Creating arrays using numpy, numpy Attributes and functions, Matrices in numpy.

### UNIT-IV

**Functions:** Defining a function, Calling a Function, Passing parameters and arguments, Python Function arguments: Positional Arguments, Keyword Arguments, Default Arguments, Variable-length arguments, Scope of the Variables in a Function–Local and Global Variables.

Recursive functions, Anonymous functions, Higher order functions - map(),filter() and reduce() functions in Python, command-line arguments.

## **UNIT-V**

**File Handling in Python:** Introduction to files, Text files and Binary files, Access Modes, Writing Data to a File-write() and writelines(), Reading Data from a File-read(),readline() and readlines(), Random access file operations-seek() and tell().

**Error Handling in Python:** Introduction to Errors and Exceptions: Compile-Time Errors, Logical Errors, Runtime Errors, Types of Exceptions, Python Exception Handling Using try, except and finally statements.

## **COURSE OUTCOMES:**

Upon completion of the course, students will be able to

1. Express proficiency in handling data types in python.
2. Understand the syntax and semantics of python control flow statements
3. Develop programs using arrays
4. Know how to write modular programs using functions.
5. Perform file operations and handle exceptions

## **TEXT BOOKS**

1. “Mastering C”, K R Venugopal, S R Prasad, Tata McGraw Hill Education (India) Private Limited.
2. R.NageswaraRao,“Core Python Programming”, Dreamtech.
3. Allen B. Downey,“Think Python: How to Think Like a Computer Scientist” 2<sup>nd</sup> edition, Updated for Python3, Shroff/O’Reilly Publishers,2016.
4. Python Programming: A Modern Approach, Vamsi Kuramanchi,Pearson.

## **REFERENCEBOOKS:**

1. Core Python Programming,W.Chun,Pearson.
2. Introduction to Python,Kenneth A. Lambert, Cengage.
3. Learning Python, Mark Lutz,Orielly.

## UNIT-1

**Introduction to Programming** – Computer Systems, Computer Languages, Algorithms and Flowcharts

**Introduction to Python Language:** Introduction to Python Language, Features of Python, Comments in Python.

**Tokens-** Keywords, Identifiers, Constants, Variables, Python Input and Output Statements

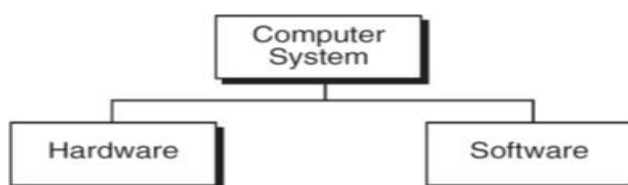
**Basic Data Types:** int, float, boolean, complex and string and its operations.

**Collection Data Types:** List, Tuples, Sets and Dictionaries. Data Type conversions.

### Introduction to Programming

**Computer:** A computer is an electronic device that manipulates information, or data. It has the ability to store, retrieve, and process data.

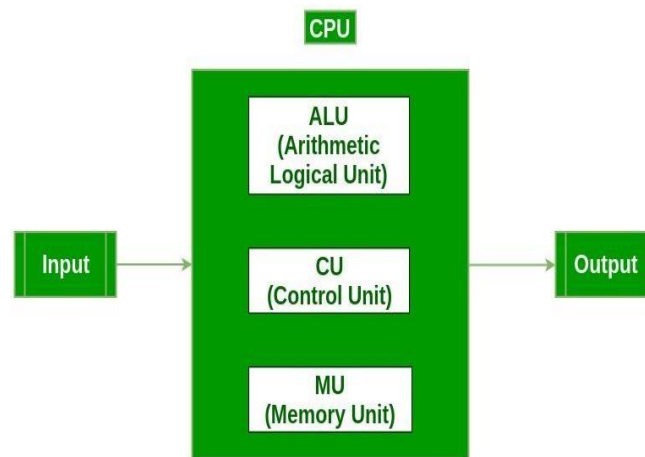
A computer is a combination of **hardware and software** resources which integrate together and provides various functionalities to the user.



Hardware are the physical components of a computer like the processor, memory devices, monitor, keyboard etc. while software is the set of programs or instructions that are required by the hardware resources to function properly. There are a few basic components that aids the working-cycle of a computer i.e. the Input- Process- Output Cycle and these are called as the functional components of a computer. It needs certain input, processes that input and produces the desired output. The input unit takes the input, the central processing unit does the processing of data and the output unit produces the output. The memory unit holds the data and instructions during the processing.

**Digital Computer:** A digital computer can be defined as a programmable machine which reads the binary data passed as instructions, processes this binary data, and displays a calculated digital output. Therefore, Digital computers are those that work on the digital data.

### Functional Components of a Digital Computer

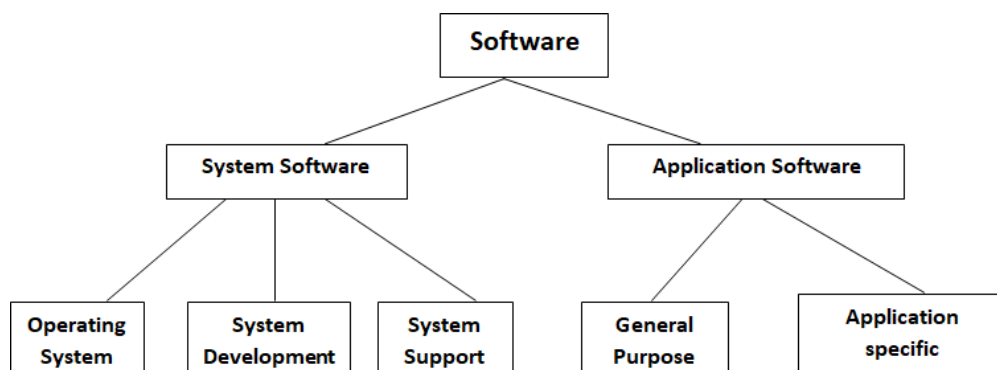


- **Input Unit :** The input unit consists of input devices that are attached to the computer. These devices take input and convert it into binary language that the computer understands. Some of the common input devices are keyboard, mouse, joystick, scanner etc.
- **Central Processing Unit (CPU):** The CPU is called the brain of the computer because it is the control center of the computer. It first fetches instructions from memory and then interprets them so as to know what is to be done. CPU executes or performs the required computation and then either stores the output or displays on the output device.  
The CPU has three main components which are responsible for different functions – Arithmetic Logic Unit (ALU), Control Unit (CU) and Memory Unit
- **Arithmetic and Logic Unit (ALU):** The ALU, as its name suggests performs mathematical calculations and takes logical decisions. Arithmetic calculations include addition, subtraction, multiplication and division. Logical decisions involve comparison of two data items to see which one is larger or smaller or equal.
- **Control Unit:** The Control unit coordinates and controls the data flow in and out of CPU and also controls all the operations of ALU, memory and also input/output units. It is also responsible for carrying out all the instructions stored in the program. It decodes the fetched instruction, interprets it and sends control signals to input/output devices until the required operation is done properly by ALU and memory.
- **Memory :** Memory attached to the CPU is used for storage of data and instructions and is called internal memory. When a program is executed, it's data is copied to the internal memory and is stored in the memory till the end of the execution. The internal memory is also called the Primary memory or Main memory. This memory is also called as RAM, i.e. Random Access Memory.
- **Output Unit :** The output unit consists of output devices that are attached with the computer. It converts the binary data coming from CPU to human understandable form. The common output devices are monitor, printer, plotter etc.

**Computer Software:** Software is the collection of programs (instructions) that allow the hardware to do its job.

There are two types of Computer Software.

- A. System Software
- B. Application Software



**A. System Software:** System Software consists of programs that manage the hardware resources of a computer and perform required information processing tasks.

These programs are divided into three classes.

- i. **Operating System Software:** It provides services such as a user interface, files and data base access and interfaces to communication systems such as Internet protocols. The primary purpose of this software is to keep the system operating in an efficient manner while allowing the users access to the system.
- ii. **System Support Software:** It provides system utilities and other operating services. Examples of system utilities are sort programs and disk format programs. Operating services consist of programs that provide performance statistics for the operational staff and security monitors to protect the system and data.
- iii. **System Development Software:** It includes language translators that convert programs in to machine language for execution, debugging tools to ensure that programs are error - free and computer -assisted software engineering (CASE) systems.

**B. Application Software:** It is directly responsible for helping users to solve their problems. Application software is broken into two classes.

- i. General - Purpose Software
- ii. Application - Specific Software

**General Purpose Software:** It is purchased from a software developer and can be used for more than one application. Examples: word processors, database management systems, computer – aided design systems. They are labeled general purpose because they can solve a variety of user computing



problems.

**Application Specific Software:** It can be used only for its intended purpose.

Example: A general ledger system used by accountants.

They can be used only for the task for which they were designed. They cannot be used for other generalized tasks.

## COMPUTER LANGUAGES:

The language used in the communication of computer instructions is known as the Computer programming language. The computer has its own language and any communication with the computer must be in its language or translated into this language. Three levels of programming languages are available. They are:

1. Machine languages (Low level languages)
2. Assembly (or symbolic) languages
3. Procedure-oriented languages (High level languages)

1. **Machine language:** Computers are made of two-state electronic devices. They can understand only pulse and no-pulse (or '1' and '0') conditions. Therefore, all instructions and data should be written using binary codes 1 and 0. This binary code is called the machine code or machine language.

Machine languages are usually referred to as the first generation languages.

2. **Assembly language:** The assembly language, also referred to as the second-generation programming language, is also a low-level language. In an assembly language, the 0s and 1s of machine language are replaced with abbreviations or mnemonic code. An assembly language program consists of a series of instructions and mnemonics that correspond to a stream of executable instructions. During the execution, the assembly language program is converted into the machine code with the help of an assembler.

3. **High-level languages:** High level languages further simplified programming tasks by reducing the number of computer operation details that had to be specified. High level languages like C, Python, java are more abstract, easier to use, and more portable across platforms, as compared to low level programming languages.

### Algorithm:

An algorithm is a finite step-by-step procedure for solving a particular problem. It is a finite sequence of steps which when followed will give desired result.

### The following are the features or characteristics of an algorithm

**Finiteness:** An algorithm terminates after a fixed number of steps

**Definiteness:** Each step of the algorithm is precisely defined. It should be clear and unambiguous

**Effectiveness:** All the operations used in the algorithm can be performed exactly in a fixed duration of time. Every step should be basic and essential

**Input:** An algorithm may have 0 or more inputs

**Output:** An algorithm must have one or more outputs

**Example:** Write an algorithm to find the sum of two numbers

Step 1: Read two numbers n1 and n2

Step 2: Add n1 and n2 and store the result in sum

Step 3: write sum



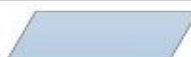


Step 4: Stop

### Flowchart:

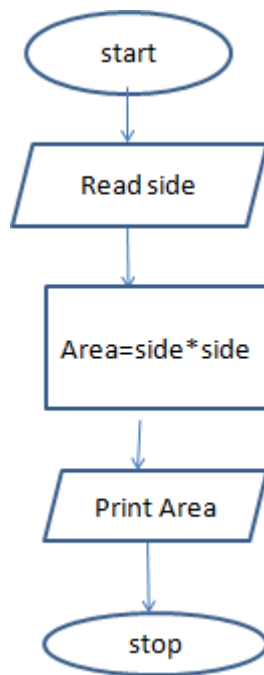
A flowchart can also be defined as a diagrammatic representation of an algorithm.

A flowchart is a type of diagram that represents a workflow or process.

### Symbols used in Flowchart

Symbol	Name	Function
	Start/end	An oval represents a start or end point
	Arrows	A line is a connector that shows relationships between the representative shapes
	Input/Output	A parallelogram represents input or output
	Process	A rectangle represents a process
	Decision	A diamond indicates a decision

**Example:** Flowchart to find the area of a square



## Introduction to Python Language

### Python Basics:

- Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language.
- Python is programming language as well as scripting language.
- Python is also called as Interpreted language
- So it is an uncomplicated and robust programming language that delivers both the power and complexity of traditional compiled languages along with the ease-of-use (and then some) of simpler scripting and interpreted languages.

### Difference between Scripting Language and Programming Language

Factor	Scripting Language	Programming Language
Type of language	Interpreter based	Compiler based
Usage	To combine existing components	To develop from scratch
Running	Inside other program (dependent)	Independent of a parent program
Conversion	High level instructions converted to machine language	Full program converted to machine language in one time
Design	Makes coding simple and fast	Gives full use of language
Compilation	No need to compile	Needs to compile first
File type	Does not create a file type	Creates a .exe file
Coding type	It is a small piece of code	It is a full code of a program
Time to develop	Less time as required less code	More time as you need to write the full code
Complexity	Easy to write and use	Difficult
Interpretation	It is interpreted in another program	Stand-alone compile result, no need to be interpreted by another program

**Why Python?**

- **Python** is a high-level, interpreted, interactive and object-oriented scripting language.
- Python is designed to be highly readable. It uses English keywords frequently where as other languages use punctuation, and it has fewer syntactical constructions than other languages.
- **Python** is a **MUST** for students and working professionals to become a great Software Engineer especially when they are working in Web Development Domain.

**History of Python:**

- Invented in the Netherlands, early 90s by Guido van Rossum
- Python was conceived in the late 1980s and its implementation was started in December 1989
- Guido Van Rossum is fan of '**Monty Python's Flying Circus**', this is a famous TV show in Netherlands
- Named after Monty Python
- Open sourced from the beginning

**Who uses python today...**

- Python is being applied in real revenue-generating products by real companies.

**For instance:**

- Google makes extensive use of Python in its web search system, and employs Python's creator.
- Intel, Cisco, Hewlett-Packard, Seagate, Qualcomm, and IBM use Python for hardware testing.
- The YouTube video sharing service is largely written in Python

**Features of Python:****1. Easy**

When we say the word 'easy', we mean it in different contexts.

**a. Easy to Code**

- Python is very **easy to code** as compared to other popular languages like Java and C++.
- Anyone can learn **Basic Python syntax** in just a few hours. Thus, it is programmer-friendly.

**b. Easy to Read**

- Being a high-level language, Python code is quite like English. Looking at it, you can tell what the code is supposed to do.
- Also, since it is **dynamically-typed**, it mandates indentation. This aids readability.

**2. Expressive**

- First, let's learn what is expressiveness. Suppose we have two languages A and B, and all programs that can be made in A can be made in B using local transformations.
- However, there are some programs that can be made in B, but not in A, using local transformations. Then, B is said to be more expressive than A.
- **Python** provides us with a myriad of constructs that help us focus on the solution rather than on the syntax.
- This is one of the outstanding python features that tell you why you should learn Python.

### 3. Free and Open-Source

- Firstly, Python is **freely available**. You can download it from the Python Official Website.
- Secondly, it is **open-source**. This means that its source code is available to the public. You can download it, change it, use it, and distribute it.
- This is called **FLOSS(Free/Libre and Open Source Software)**. As the Python community, we're all headed toward one goal- an ever-bettering Python.

### 4. High-Level

- Python is a high-level language. This means that as programmers, we don't need to remember the system architecture.
- Also, we need not manage memory. This makes it more **programmer-friendly** and is one of the key python features.

### 5. Portable

- Let's assume you've written a Python code for your Windows machine. Now, if you want to run it on a Mac, you don't need to make changes to it for the same.
- In other words, you can take one code and run it on any machine. This makes Python a **portable language**.
- However, you must avoid any system-dependent features in this case.

### 6. Interpreted

- If you're familiar with any languages like C++ or Java, you must first compile it, and then run it. But in Python, there is no need to compile it.
- Internally, its source code is converted into an immediate form called **bytecode**.
- So, all you need to do is to run your Python code without worrying about linking to libraries, and a few other things.
- By interpreted, we mean the source code is executed line by line, and not all at once. Because of this, it is **easier to debug your code**.
- Also, interpreting makes it just slightly slower than Java, but that does not matter compared to the benefits it offers.
- If you have any doubt in DataFlair's features of python programming language article, drop a comment below and we will get back to you

### 7. Object-Oriented

- A programming language that can model the real world is said to be object-oriented. It focuses on objects and combines data and functions.
- Contrarily, a procedure-oriented language revolves around functions, which are code that can be reused.
- Python supports both **procedure-oriented** and **object-oriented programming** which is one of the key python features.
- It also supports multiple inheritance, unlike Java.
- A class is a blueprint for such an object. It is an abstract data type and holds no values.

### 8. Extensible

- If needed, you can write some of your Python code in other languages like C++.
- This makes Python an extensible language, meaning that it can be extended to other languages.

### 9. Embeddable

- We just saw that we can put code in other languages in our Python source code.
- However, it is also possible to put our Python code in a source code in a different language like C++.
- This allows us to integrate scripting capabilities into our program of the other language.

### 10. Large Standard Library

- Python downloads with a large library that you can use so you don't have to write your own code for every single thing.
- There are libraries for regular expressions, documentation-generation, unit-testing, web browsers, threading, databases, CGI, email, image manipulation, and a lot of other functionality.

### 11. GUI Programming

- Software is not user-friendly until its GUI is made. A user can easily interact with the software with a GUI.
- Python offers various libraries for making Graphical user interface for your applications.
- For this, you can use Tkinter, wxPython or JPython. These toolkits allow you for easy and fast development of GUI.

### 12. Dynamically Typed

- Python is dynamically-typed. This means that the type for a value is decided at runtime, not in advance. This is why we don't need to specify the type of data while declaring it

## Python Applications

Python is known for its general-purpose nature that makes it applicable in almost every domain of software development. Python makes its presence in every emerging field. It is the fastest-growing programming language and can develop any application.

Application areas where Python can be applied.



## GETTING STARTED

### Python Versions

- 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.

### Finding an Interpreter for Python Programming

**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/>



**Python Installation:**

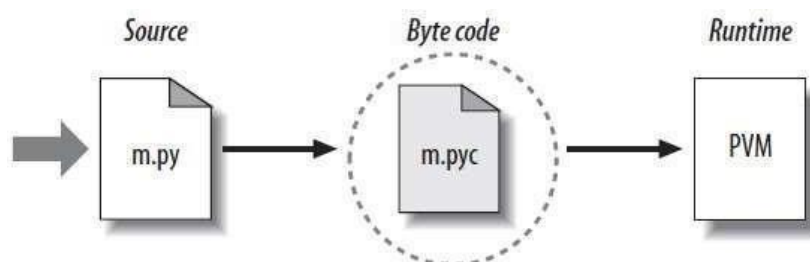
There are many interpreters available freely to run Python scripts like IDLE (Integrated Development Environment) which is installed when you install the python software from <http://python.org/downloads/>

**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)

**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.
- Your code is automatically compiled, but then it is interpreted.



Source code extension is **.py**

Byte code extension is **.pyc** (compiled python code)

### Modes for using Python interpreter:

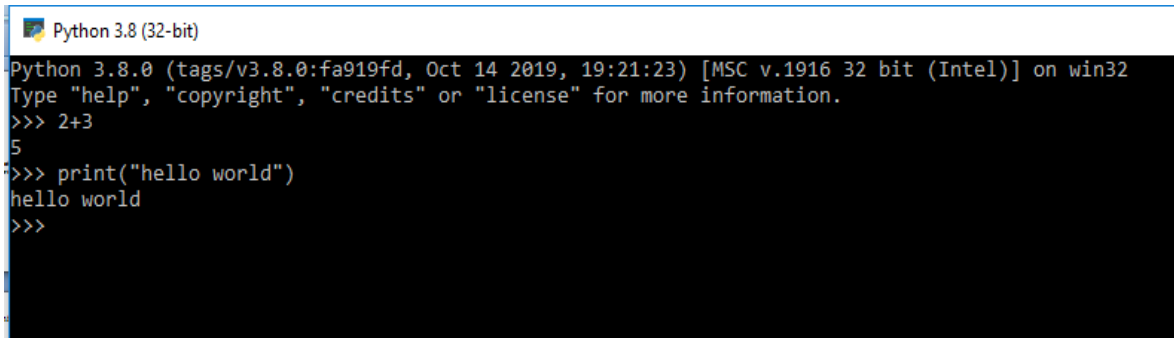
There are two modes for using the Python interpreter:

- Interactive Mode
- Script Mode

#### 1. 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.

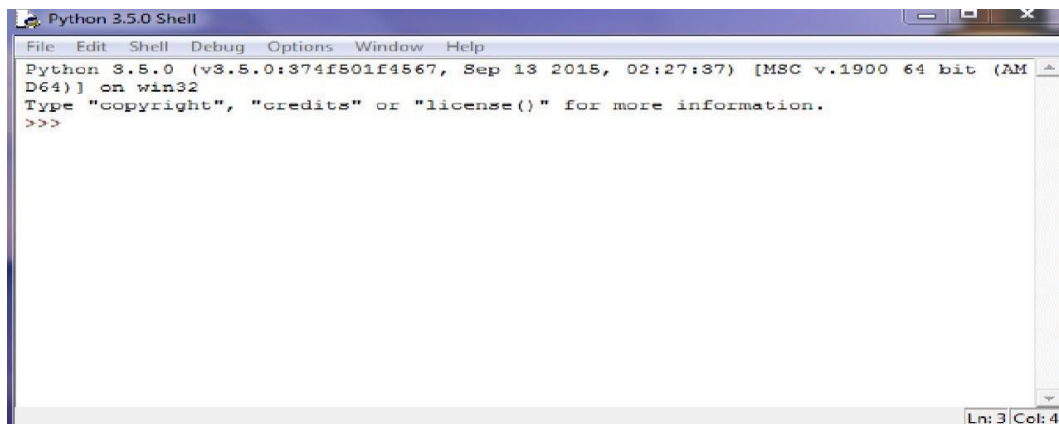
```
>>> print("hello world")
hello world
# Relevant output is displayed on subsequent lines without the >>> symbol
>>> x=[0,1,2]
# Quantities stored in memory are not displayed by default.
>>> x
#If a quantity is stored in memory, typing its name will display it. [0, 1, 2]
>>> 2+3
```



- 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.

#### 2. 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. Working with the interactive mode is better when Python programmers deal with small pieces of code as you can type and execute the 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.



### Python input() Function

Python input() function is used to get input from the user. It prompts for the user input and reads a line. After reading data, it converts it into a string and returns that. It throws an error EOFError if EOF is read.

#### Signature

```
var = input ([prompt])
```

#### Parameters

**prompt:** It is a string message which prompts for the user input.

#### Return

It returns user input after converting into a string.

#### Python input() Function Example 1

Here, we are using this function get user input and display to the user as well.

1. `# Python input() function example`
2. `# Calling function`
3. `val = input("Enter a value: ")`
4. `# Displaying result`
5. `print("You entered:",val)`

#### Output:

```
Enter a value: 45
You entered: 45
```

### Python input() Function Example 2

The input() method returns string value. So, if we want to perform arithmetic operations, we need to cast the value first. See the example below.

1. `# Python input() function example`
2. `# Calling function`
3. `val = input("Enter an integer: ")`
4. `# Displaying result`
5. `val = int(val) # casting into string`
6. `sqr = (val*val) # getting square`
7. `print("Square of the value:",sqr)`

### Output:

```
Enter an integer: 12
Square of the value: 144
```

### Python print() Function

Python **print()** function prints the given object on the screen or other standard output devices.

#### Signature

**print**(object(s), sep=separator, end=end, file=file, flush=flush)

#### Parameters

**object(s):** It is an object to be printed. The Symbol \* indicates that there may be more than one object.

**sep='separator' (optional):** The objects are separated by sep. The default value of sep is ' '.

**end='end' (optional):** it determines which object should be print at last.

**file (optional):** - The file must be an object with write(string) method. If it is omitted, sys.stdout will be used which prints objects on the screen.

**flush (optional):** If True, the stream is forcibly flushed. The default value of flush is False

#### Return

It does not return any value. **Python**

### print() Function Example 1

1. `print("Python is programming language.")`
2. `x = 7`
4. `# Two objects passed`
5. `print("x =", x)`
6.
7. `y = x`
8. `# Three objects passed`
9. `print('x =', x, '= y')`

**Output:**

```
Python is programming language.  
x = 7  
x = 7 = y
```

**Python print() Function Example 2**

The below example use print() with separator and end parameters.

```
1. x = 7  
2. print("x =", x, sep='00000', end='\n\n\n')3.  
print("x =", x, sep='0', end="")
```

**Output:**

```
a =000007  
a =07
```

**Comments:**

- **Comments** begins with a hash(#) symbol and is useful in mentioning that the whole line should be considered as a comment until the end of line.
- It is used to basically provide the description of particular line of code.
- So it will ignore from execution by Python Interpreter.

**Example:**

```
# To print Hello World (It is comment)  
>>> print ("Hello World")  
• So it executes only one line of code.
```

**Python Identifiers**

- A Python identifier is a name used to identify a variable, function, class, module or other object. An identifier starts with a letter A to Z or a to z or an underscore (\_) followed by zero or more letters, underscores and digits (0 to 9).
- Python does not allow punctuation characters such as @, \$, and % within identifiers. Python is a case sensitive programming language. Thus, Manpower and manpower are two different identifiers in Python.

**Python Keywords:**

The following list shows the Python keywords. These are reserved words and you cannot use them as constant or variable or any other identifier names. Mostly Python keywords contain lowercase letters only.

Table 3.1. Python Keywords			
<code>and</code>	<code>elif</code>	<code>global</code>	<code>or</code>
<code>assert</code>	<code>else</code>	<code>if</code>	<code>pass</code>
<code>break</code>	<code>except</code>	<code>import</code>	<code>print</code>
<code>class</code>	<code>exec</code>	<code>in</code>	<code>raise</code>
<code>continue</code>	<code>finally</code>	<code>is</code>	<code>return</code>
<code>def</code>	<code>for</code>	<code>lambda</code>	<code>try</code>
<code>del</code>	<code>from</code>	<code>not</code>	<code>while</code>

**Lines and Indentation:**

- Python provides no braces to indicate blocks of code for class and function definitions or flow control.
- Blocks of code are denoted by line indentation, which is rigidly enforced.

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

For example –

```
a=10
b=20
if(a>b):
    print("a is greater than b")
else:
    print("a is less than b")
```

However, the following block generates an error –

```
a=10
b=20
if(a>b):
    print("a is greater than b")
else:
    print("a is less than b")
```

Thus, in Python all the continuous lines indented with same number of spaces would form a block.

**Variables in Python:**

- 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= 50 # An integer assignment
b= 7000.0 # A floating point
c = "John" # A string
print (a)
print (b)
print (c)
```

**This produces the following result –**

```
50
7000.0
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,"mrec"
```

- 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.

#### **Example**

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

**Output:** python

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 Identifiers:**

Identifiers are the name given to variables, classes, methods(functions), etc. For example,

```
language = 'Python'
```

Here, `language` is a variable (an identifier) which holds the value `'Python'`.

We cannot use keywords as variable names as they are reserved names that are built-in to Python. For example,



```
continue = 'Python'
```

The above code is wrong because we have used `continue` as a variable name. To learn more about variables, visit [Python Variables](#).

### Rules for Naming an Identifier

- Identifiers cannot be a keyword.
- Identifiers are case-sensitive.
- It can have a sequence of letters and digits. However, it must begin with a letter or `_`. The first letter of an identifier cannot be a digit.
- It's a convention to start an identifier with a letter rather `_`.
- Whitespaces are not allowed.
- We cannot use special symbols like `!`, `@`, `#`, `$`, and so on.

### Some Valid and Invalid Identifiers in Python

#### Valid Identifiers

score

return\_value

highest\_score

name1

convert\_to\_string

#### Invalid Identifiers

@core

return

highest score

1name

convert to\_string

### Things to Remember

Python is a case-sensitive language. This means, `Variable` and `variable` are not the same. Always give the identifiers a name that makes sense. While `c = 10` is a valid name, writing `count = 10` would make more sense, and it would be easier to figure out what it represents when you look at your code after a long gap.

Multiple words can be separated using an underscore, like `this is a long variable`

## Data Types in Python:

- Every value in Python has a datatype.
- Since everything is an object in Python programming, data types are actually classes and variables are instance (object) of these classes.

There are various data types in Python. Some of the important types are listed below.

- Numbers
- Strings
- List
- Tuples
- Sets
- Dictionary

## NUMBERS

## Number data types:

- Number data types store numeric values.
- They are immutable data types, means that changing the value of a number data type results in a newly allocated object.
- Number objects are created when you assign a value to them
- Python has three built-in numeric data types: integers, floating-point numbers, and complex numbers.

### Integers:

- An **integer** is a whole number with no decimal places. For example, 1 is an integer, but 1.0 isn't.
- So in Python, integers are zero, positive or negative whole numbers without a fractional part and having unlimited precision, e.g. 0, 100, -10.

The followings are valid integer literals in Python.

[illegible]



- Scientific notation is used as a short representation to express floats having many digits. For example: 345.56789 is represented as 3.4556789e2 or 3.4556789E2

```
>>> f=1e3
>>> f
1000.0
>>> f=1e5
>>> f
100000.0
>>> f=3.4556789e2
>>> f
345.56789
```

## STRINGS

Strings are amongst the most popular types in Python. We can create them simply by enclosing characters in quotes. Python treats single quotes the same as double quotes. This contrasts with most other scripting languages, which use single quotes for literal strings and double quotes to allow escaping of characters.

Python uses the "raw string" operator to create literal quotes, so no differentiation is necessary. Other languages such as C use single quotes for characters and double quotes for strings.

Python does not have a character type; this is probably another reason why single and double quotes are the same.

Nearly every Python application uses strings in one form or another. Strings are immutable, meaning that changing an element of a string requires creating a new string. Strings are made up of individual characters, and such elements of strings may be accessed sequentially via slicing.

### How to Create and Assign Strings

Creating strings is as simple as assigning a value to a variable:

```
>>> aString = 'Hello World!'
>>> anotherString = "Python is cool!"
>>> print (aString)
Hello World!
>>> print (anotherString)
Python is cool!
>>> aBlankString = ""
>>> print (aBlankString)"
```

### How to Access Values (Characters and Substrings) in Strings

- Python does not support a character type; these are treated as strings of length one, thus also considered a substring.
- To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring:

```
>>> aString = 'Hello World!'
>>> aString[0] # Forward Indexing
'H'
>>> aString[-1] # Backward Indexing
'd'
>>> aString[1:5] #Slicing
'ello'
>>> aString[6:]
'World!'
```

### How to Update Strings

- You can "update" an existing string by (re)assigning a variable to another string.
- The new value can be related to its previous value or to a completely different string altogether.

```
>>> aString = 'Hello World!'
>>> aString = aString[:6] + 'Python!'
>>> aString
'Hello Python!'
>>> aString = 'different string altogether'
>>> aString
'different string altogether'
```

- Like numbers, strings are not mutable, so you cannot change an existing string without creating a new one from scratch.
- That means that you cannot update individual characters or substrings in a string.
- However, as you can see above, there is nothing wrong with piecing together part of your old string and assigning it to a new string.

### How to Remove Characters and Strings

To repeat what we just said, strings are immutable, so you cannot remove individual characters from an existing string. What you can do, however, is to empty the string, or to put together another string which drops the pieces you were not interested in.

Let us say you want to remove one letter from "Hello World!"... the (lowercase) letter "l," for example:

```
>>> aString = 'Hello World!'
>>> aString = aString[:3] #slicing from the start + aString[4:]#slicing to the end
>>> aString
'Helo World!'
```

**Membership Operator(in, not in)**

- The membership question asks whether a character (string of length one) appears in a string.
- True is returned if that character appears in the string and False otherwise
- Here are a few more examples of strings and the membership operators.

```
>>> 'c' in 'abcd'
True
>>> 'n' in 'abcd'
False
>>> 'n' not in 'abcd'
True
```

**Multi Line Strings**

You can assign a multiline string to a variable by using three quotes:

```
>>>a="""This is Python Programming,
Python Language is awesome,
It is very easy to understand and
Easy to implement code"""
```

```
>>>print(a)
This is Python Programming,
Python Language is awesome, It
is very easy to understand and
Easy to implement code
```

**Escape Character**

To insert characters that are illegal in a string, use an escape character. An escape character is a backslash \ followed by the character you want to insert.

An example of an illegal character is a double quote inside a string that is surrounded by double quotes:

**Example:**

The escape character allows you to use double quotes when you normally would not be allowed:

```
>>>txt = "Python is an easy and \"interesting\" language."
>>>print(txt)
Python is an easy and "interesting" language.
```

**Concatenation of Two or More Strings**

- Joining of two or more strings into a single one is called concatenation.
- The + (concatenation) operator does this in Python. Simply writing two string literals together also concatenates them.
- The \* (repetition) operator can be used to repeat the string for a given number of times.

**# Python String Operations**

```
str1 = 'Hello'
str2 = 'World!'
# using +
print('str1 + str2 = ', str1 + str2)
```

```
# using *
print('str1 * 3 =', str1 * 3)
```

- When we run the above program, we get the following output:

```
str1 + str2 = HelloWorld!
str1 * 3 = HelloHelloHello
```

**String Length**

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

```
>>>a= "Hello,World!"
>>> print(len(a))
12
```

**Upper Case**

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

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

**Lower Case**

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

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

**Remove Whitespace**

Whitespace is the space before and/or after the actual text, and very often you want to remove this space. The strip() method removes any whitespace from the beginning or the end:

```
>>>a = " Hello, World! "
>>>print(a.strip()) # returns "Hello, World!"
Hello, World!
```

**Replace String**

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

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

### Split String

The split() method returns a list where the text between the specified separator becomes the list items.

```
a = "Hello, World!"  
print(a.split(",")) # returns ['Hello', ' World!']  
['Hello', ' World!']
```

### String Format

We can combine strings and numbers by using the format() method. The format() method takes the passed arguments, formats them, and places them in the string where the placeholders {} are: Use the format() method to insert numbers into strings:

```
>>>age = 36  
>>>txt = "My name is John, and I am {}"  
>>>print(txt.format(age))  
My name is John, and I am 36
```

### String Methods

Method	Description
capitalize()	Converts the first character to upper case
casefold()	Converts string into lower case
center()	Returns a centered string
count()	Returns the number of times a specified value occurs in a string
encode()	Returns an encoded version of the string
endswith()	Returns true if the string ends with the specified value
expandtabs()	Sets the tab size of the string
find()	Searches the string for a specified value and returns the position of where it was found
format()	Formats specified values in a string
format_map()	Formats specified values in a string



index()	Searches the string for a specified value and returns the position of where it was found
isalnum()	Returns True if all characters in the string are alphanumeric
isalpha()	Returns True if all characters in the string are in the alphabet
isdecimal()	Returns True if all characters in the string are decimals
isdigit()	Returns True if all characters in the string are digits
isidentifier()	Returns True if the string is an identifier
islower()	Returns True if all characters in the string are lower case
isnumeric()	Returns True if all characters in the string are numeric
isprintable()	Returns True if all characters in the string are printable
isspace()	Returns True if all characters in the string are whitespaces
istitle()	Returns True if the string follows the rules of a title
isupper()	Returns True if all characters in the string are upper case
join()	Joins the elements of an iterable to the end of the string
ljust()	Returns a left justified version of the string
lower()	Converts a string into lower case
lstrip()	Returns a left trim version of the string
partition()	Returns a tuple where the string is parted into three parts
replace()	Returns a string where a specified value is replaced with a specified value
rfind()	Searches the string for a specified value and returns the last position of where it was found
rindex()	Searches the string for a specified value and returns the last position of where it was found
rjust()	Returns a right justified version of the string
rpartition()	Returns a tuple where the string is parted into three parts
rsplit()	Splits the string at the specified separator, and returns a list
rstrip()	Returns a right trim version of the string

split()	Splits the string at the specified separator, and returns a list
startswith()	Returns true if the string starts with the specified value
strip()	Returns a trimmed version of the string
swapcase()	Swaps cases, lower case becomes upper case and vice versa
title()	Converts the first character of each word to upper case
translate()	Returns a translated string
upper()	Converts a string into upper case
zfill()	Fills the string with a specified number of 0 values at the beginning

## LIST

- The list is a most versatile Data type available in Python in which objects are mutable in type and can be written as a list of comma-separated values (items) between square brackets.
- Important thing about a list is that items in a list need not be of the same type.
- Strings consist only of characters and are immutable (cannot change individual elements) while lists are flexible container objects which hold an arbitrary number of Python objects.
- Creating lists is simple; adding to lists is easy, too, as we see in the following examples.

### Example:

```
list1 = ['physics', 'chemistry', 1997, 2000];
```

```
list2 = [1, 2, 3, 4, 5];
```

```
mylist = ["apple", "banana", "cherry"]
```

## List

Lists are used to store multiple items in a single variable.

Lists are one of 4 built-in data types in Python used to store collections of data, the other 3 are [Tuple](#), [Set](#), and [Dictionary](#), all with different qualities and usage.

Lists are created using square brackets:

**Example:**

Create a List:

```
thislist=["apple", "banana", "cherry"]  
print(thislist)
```

## List Items

List items are ordered, changeable, and allow duplicate values.

List items are indexed, the first item has index [0], the second item has index [1] etc.

## Ordered

When we say that lists are ordered, it means that the items have a defined order, and that order will not change.

If you add new items to a list, the new items will be placed at the end of the list.

**Note:** There are some [list methods](#) that will change the order, but in general: the order of the items will not change.

## Changeable

The list is changeable, meaning that we can change, add, and remove items in a list after it has been created.

## Allow Duplicates

Since lists are indexed, lists can have items with the same value:

### Example

Lists allow duplicate values:

```
thislist=["apple", "banana", "cherry", "apple", "cherry"]  
print(thislist)
```

## List Length

To determine how many items a list has, use the `len()` function:

### Example

Print the number of items in the list:

```
thislist=["apple", "banana", "cherry"]  
print(len(thislist))
```

### List Items - Data Types

List items can be of any data type:

#### Example

String, int and boolean data types:

```
list1=["apple", "banana", "cherry"]
```

```
list2=[1, 5, 7, 9, 3]
```

```
list3 = [True, False, False]
```

A list can contain different data types:

#### Example

A list with strings, integers and boolean values:

```
list1 = ["abc", 34, True, 40, "male"]
```

```
type()
```

From Python's perspective, lists are defined as objects with the data type 'list':

```
<class 'list'>
```

#### Example

What is the data type of a list?

```
mylist=["apple", "banana", "cherry"]
```

```
print(type(mylist))
```

### How to Create and Assign Lists

- Creating lists is as simple as assigning a value to a variable.
- You handcraft a list (empty or with elements) and perform the assignment.
- Lists are delimited by surrounding square brackets ( [ ] ).

```
>>> aList = [123, 'abc', 4.56, ['inner', 'list'], 7-9j]
```

```
>>> anotherList = [None, 'something to see here']
```

```
>>> print (aList)
[123, 'abc', 4.56, ['inner', 'list'], (7-9j)]
```

```
>>> print (anotherList) [None,
'something to see here']
```

```
>>> aListThatStartedEmpty = []
>>> print (aListThatStartedEmpty)
[]
```

### How to Access Values in Lists

- Slicing works similar to strings; use the square bracket slice operator ( [ ] ) along with the index or indices.

```
>>> aList[0]
123
```

```
>>> aList[1:4]
['abc', 4.56, ['inner', 'list']]
```

```
>>> aList[:3]
[123, 'abc', 4.56]
```

```
>>> aList[3][1]
'list'
```

### How to Update Lists

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

```
>>> aList
[123, 'abc', 4.56, ['inner', 'list'], (7-9j)]
```

```
>>> aList[2]
4.56
```

```
>>> aList[2] = 'float replacer'
>>> aList
[123, 'abc', 'float replacer', ['inner', 'list'], (7-9j)]
```

```
>>>
>>> anotherList.append("hi, i'm new here")
>>> print (anotherList)
```

```
[None, 'something to see here', 'hi, i'm new here']
```

```
>>> aListThatStartedEmpty.append('not empty anymore')
>>> print (aListThatStartedEmpty)
['not empty anymore']
```

### How to insert item in List

- To insert a new list item, without replacing any of the existing values, we can use the insert() method.
- The insert() method inserts an item at the specified index:

```
>>>str_list = ['jack', 'jumped', 'over', 'candlestick']

>>>str_list.insert(2, 'park')

>>>print(str_list)
['jack', 'jumped', 'park', 'over', 'candlestick']
```

### How to Remove List Elements and Lists

- To remove a list element, you can use either the **del** statement if you know exactly which element(s) you are deleting or the remove() method if you do not know.

```
>>> aList
[123, 'abc', 'float replacer', ['inner', 'list'], (7-9j)]

>>> del aList[1]
>>> aList
[123, 'float replacer', ['inner', 'list'], (7-9j)]

>>> aList.remove(123)
>>> aList
['float replacer', ['inner', 'list'], (7-9j)]
```

- You can also use the pop() method to remove and return a specific object from a list.

```
>>> aList.pop()
(7-9j)
```

- To clear the list, clear() method will be used to empties the list. The list still remains, but

it has no content.

```
>>> aList.clear()
```

```
print(aList)
```

```
[]
```

- Normally, removing an entire list is not something application programmers do.
- Rather, they tend to let it go out of scope (i.e., program termination, function call completion, etc.) and be garbage-collected, but if they do want to explicitly remove an entire list, use the **del** statement:

```
>>>del aList
```

### Membership (in, not in)

- With strings, the membership operator determined whether a single character is a member of a string.
- With lists (and tuples), we can check whether an object is a member of a list (or tuple).

```
>>> mixup_list = [4.0, [1, 'x'], 'fruits', (-1.9+6j)]
```

```
>>>mixup_list
```

```
[4.0, [1, 'x'], 'fruits', (-1.9+6j)]
```

```
>>> 'fruits' in mixup_list
```

```
True
```

```
>>>
```

```
>>> 'x' in mixup_list
```

```
False
```

```
>>> 'x' in mixup_list[1]
```

```
True
```

### Concatenation(+) Operator

- The concatenation operator allows us to join multiple lists together.

```
>>> num_list = [43, -1.23, -2, 6.19e5]
```

```
>>> str_list = ['jack', 'jumped', 'over', 'candlestick']
```

```
>>> mixup_list = [4.0, [1, 'x'], 'fruits', -1.9+6j]
```

```
>>>
```

```
>>> num_list + mixup_list
```

```
[43, -1.23, -2, 619000.0, 4.0, [1, 'x'], 'fruits', (-1.9+6j)]
```

```
>>>
```

```
>>> str_list + num_list
```

```
['jack', 'jumped', 'over', 'candlestick', 43, -1.23, -2, 619000.0]
```

- We can use the **extend()** method in place of the concatenation operator to append the

contents of a list to another.

- Using `extend()` is advantageous over concatenation because it actually appends the elements of the new list to the original, rather than creating a new list from scratch like `+` does

### **Repetition (\*)**

- Use of the repetition operator may make more sense with strings, but as a sequence type, lists and tuples can also benefit from this operation, if needed:

```
>>> num_list * 2
[43, -1.23, -2, 619000.0, 43, -1.23, -2, 619000.0].
```

### **len()**

- For strings, `len()` gives the total length of the string, as in the number of characters.
- For lists (and tuples), it will not surprise you that `len()` returns the number of elements in the list (or tuple).
- Container objects found within count as a single item.
- Our examples below use some of the lists already defined above in previous sections.

```
>>> len(num_list)4
```

### **Copy a List**

- You cannot copy a list simply by typing `list2 = list1`, because: `list2` will only be a reference to `list1`, and changes made in `list1` will automatically also be made in `list2`.
- There are ways to make a copy, one way is to use the built-in List method `copy()`.

```
>>>str_list = ['jack', 'jumped', 'over', 'candlestick']
>>>mylist = str_list.copy()
>>>print(mylist)
['jack', 'jumped', 'over', 'candlestick']
```

- Another way to make a copy is to use the built-in method `list()`.

```
>>>str_list = ['jack', 'jumped', 'over', 'candlestick']
>>>mylist = list(str_list)
>>>print(mylist)
['jack', 'jumped', 'over', 'candlestick']
```

### **List Methods**



- Python has a set of built-in methods that you can use on lists.

Method	Description
<u>append()</u>	Adds an element at the end of the list
<u>clear()</u>	Removes all the elements from the list
<u>copy()</u>	Returns a copy of the list
<u>count()</u>	Returns the number of elements with the specified value
<u>extend()</u>	Add the elements of a list (or any iterable), to the end of the current list
<u>index()</u>	Returns the index of the first element with the specified value
<u>insert()</u>	Adds an element at the specified position
<u>pop()</u>	Removes the element at the specified position
<u>remove()</u>	Removes the item with the specified value
<u>reverse()</u>	Reverses the order of the list
<u>sort()</u>	Sorts the list

## TUPLES

### Tuples:

- Tuples are another container type extremely similar in nature to lists.
- The only visible difference between tuples and lists is that tuples use parentheses and lists use square brackets.
- Functionally, there is a more significant difference, and that is the fact that tuples are immutable.
- Tuples are used to store multiple items in a single variable.
- Tuple is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Set, and Dictionary, all with different qualities and usage.
- A tuple is a collection which is ordered and **unchangeable**.

### Tuple Items

- Tuple items are ordered, unchangeable, and allow duplicate values.
- Tuple items are indexed, the first item has index [0], the second item has index [1] etc.

1. **Ordered:** When we say that tuples are ordered, it means that the items have a defined order, and that order will not change.

2. **Unchangeable** : Tuples are unchangeable, meaning that we cannot change, add or remove items after the tuple has been created.

3. **Allow Duplicates** : Since tuple are indexed, tuples can have items with the same value

### How to Create and Assign Tuples

Creating and assigning lists are practically identical to lists, with the exception of emptytuples.

- These require a trailing comma ( , ) enclosed in the tuple delimiting parentheses ( ( ) ).

```
>>> aTuple = (123, 'abc', 4.56, ['inner', 'tuple'], 7-9j)
>>> anotherTuple = (None, 'something to see here')
>>> print aTuple
(123, 'abc', 4.56, ['inner', 'tuple'], (7-9j))
```

```
>>> print anotherTuple
(None, 'something to see here')
```

```
>>> emptyTuple = ()
>>> print emptyTuple
()
```

### How to Access Values in Tuples

- Slicing works similar to lists: Use the square bracket slice operator ([ ]) along with the index or indices.

```
>>> aTuple = (123, 'abc', 4.56, ['inner', 'tuple'], 7-9j)
>>> aTuple[1:4]
('abc', 4.56, ['inner', 'tuple'])
>>>
>>> aTuple[:3]
(123, 'abc', 4.56)
>>> aTuple[3][1]
'tuple'
```

### How to Update Tuples

- Like numbers and strings, tuples are immutable which means you cannot update them or change values of tuple elements.

1. In strings, we were able to take portions of an existing string to create a new string. The same applies for tuples.

```
>>> aTuple = aTuple[0], aTuple[1], aTuple[-1]
```

```
>>> aTuple
(123, 'abc', (7-9j))
>>> tup1 = (12, 34.56)
>>> tup2 = ('abc', 'xyz')
>>> tup3 = tup1 + tup2
>>> tup3
(12, 34.56, 'abc', 'xyz')
```

**2. Change Tuple Values:** Once a tuple is created, you cannot change its values. Tuples are unchangeable, or immutable as it also is called.

- But there is a workaround. You can convert the tuple into a list, change the list, and convert the list back into a tuple.

```
>>>aTuple = (123, 'abc', 4.56, ['inner', 'tuple'], 7-9j)
>>>aList = list(aTuple)
>>>aList[1] = "xyz"
>>>aTuple = tuple(aList)

>>>print(aTuple)
(123, 'xyz', 4.56, ['inner', 'tuple'], (7-9j))
```

### How to Remove Tuple Elements and Tuples

- Removing individual tuple elements is not possible.
- There is, of course, nothing wrong with putting together another tuple with the undesired elements discarded.
- To explicitly remove an entire list, just use the **del** statement:

```
>>> del aTuple
```

### Single Element Tuples

- If we take a single element in tuple it will not be considered as a tuple but it will be considered as either int, float or strings based on the type of value which we have taken as a single element.
- But there will not be any problem in list for single element in it.

```
>>> ['abc']
['abc']
>>> type(['abc']) # a list
<class 'list'>
>>>
>>> [123]
[123]
>>> type([123]) # also a list
<class 'list'>
>>>
```

```
>>> ('xyz')
'xyz'
>>> type(('xyz')) # a string, not a tuple
<class 'string'>
>>>
>>> (456)
456
>>> type((456)) # an int, not a tuple
<class 'int'>
```

- It probably does not help your case that the parentheses are also overloaded as the expression grouping operator.
- Parentheses around a single element take on that binding role rather than as a delimiter for tuples.
- The workaround is to place a trailing comma (,) after the first element to indicate that this is a tuple and not a grouping.

```
>>> ('xyz',)
('xyz',)
>>> (456,)
(456,)
```

### Unpacking a Tuple

- When we create a tuple, we normally assign values to it. This is called "packing" a tuple.

### Packing a Tuple:

```
>>> aTuple = (123, 'abc', 4.56, ['inner', 'tuple'], 7-9j)
>>> aTuple
(123, 'abc', 4.56, ['inner', 'tuple'], 7-9j)
```

- But, in Python, we are also allowed to extract the values back into variables. This is called "unpacking"

### Unpacking a tuple:

```
>>> (a,b,c,d,e)=aTuple
>>> a
123
>>> b
'abc'
>>> c
4.56
>>> d
['inner', 'tuple']
>>> e(7-9j)
```

**Note:** The number of variables must match the number of values in the tuple, if not, you must use an asterisk to collect the remaining values as a list.

**Using Asterisk\*:**

- If the number of variables is less than the number of values, you can add an \* to the variable name and the values will be assigned to the variable as a list:

```
>>> (a,b,*c)=aTuple
>>> a
123
>>> b
'abc'
>>> c
[4.56, ['inner', 'tuple'], (7-9j)]
```

**No Enclosing Delimiters:**

- Any set of multiple objects, comma-separated, written without identifying symbols, i.e., brackets for lists, parentheses for tuples, etc., default to tuples, as indicated in these short examples:

```
>>> 'abc', -4.24e93, 18+6.6j, 'xyz'
('abc', -4.24e+093, (18+6.6j), 'xyz')
>>>
>>> x, y = 1, 2
>>> x, y
(1, 2)
```

- Any function returning multiple objects (also no enclosing symbols) is a tuple.

**Join Two Tuples (+)**

- To join two or more tuples you can use the concatenation (+) operator:

```
>>>aTuple = ("a", "b" , "c")
>>>anotherTuple = (1, 2, 3)

>>>joinTuple = aTuple + anotherTuple
>>>print(joinTuple)
('a', 'b', 'c', 1, 2, 3)
```

**Multiply Tuples(\*)**

- If you want to multiply the content of a tuple a given number of times, you can use the repetition (\*) operator:

```
>>>a = ("a", "b" , "c")
>>> b= a * 3
>>>print(b)
('a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c')
```

**Tuple Methods:**

**count():**The count() method returns the number of times a specified value appears in the tuple.

**Example:**

```
>>>atuple = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5)
>>>x = atuple.count(5)
>>>print(x)2
```

**index():**The index() method finds the first occurrence of the specified value. This method raises an exception if the value is not found.

**Example:**

Search for the first occurrence of the value 8, and return its position:

```
>>>aTuple = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5)
>>>x = aTuple.index(8)
>>>print(x)3
```

**SETS****Sets:**

- Sets are used to store multiple items in a single variable.
- A set is a collection which is both unordered and un indexed.
- Sets are written with curly brackets.
- As Sets are unordered, so you cannot be sure in which order the items will appear.
- A set as a whole can be changed, but the elements present in the set are unchangeable.
- Every element in the set must be unique (no duplicate values) and must be immutable (unchangeable).
- These elements can be of the data type.
- However, sets are in itself data structure, which is unordered and mutable (changeable).
- These are used for performing mathematical operations such as union, intersection, etc.

**Creating Sets:**

- You can create set either by placing all the items in the curly braces or by using the set () function. The elements can be of integer, string, float, etc).

**Example1: Using curly braces**

```
>>>My_Print= {10, 'Hi', (7)}
>>>print (My_Print)
{10, 'Hi', 7}
```

**Example2: Using Set () function**

```
>>>a= set ({10, 9.0, "bye"})
>>>print (a)
{9.0, 10, 'bye'}
```

**Example 3: Empty set**

```
>>>a=set()
>>>print(a)
set()
```

**Example 4:**

```
>>>aSet = {'jack', 'jumped', 'over', 'candlestick'}
>>>print(aSet) #printing for first time
{'candlestick', 'jack', 'jumped', 'over'}
>>>print(aSet) #printing for second time
{'jumped', 'over', 'candlestick', 'jack'}
```

**Example 5:**

```
>>> aSet = {'jack', 'jumped', 'over', 'candlestick', 'over'}
>>> aSet # Sets won't print duplicate values
{'jack', 'over', 'jumped', 'candlestick'}
```

**Modifying a Set:**

- If you want to add only one element, then use add () method, and for adding multiple-element you need to use the update() method.

**Example1: For adding single element**

```
>>>a={10,3}
>>>print(a)
{10,3}
>>>a.add(2)
>>>print(a)
{10,3,2}
```

**Example2: Using update() method**

```
>>>a = {10,3}
>>>print(a)
{10,3}
>>> a.update([2,3,4])
>>>print(a)
{10,2,3,4}
```

**Removing elements from a set:**

- To remove particular elements you can use discard() and remove ().
- The remove() method removes the specified element from the set.
- The remove() method takes a single element as an argument and removes it from the set.
- The remove() removes the specified element from the set and updates the set. It doesn't return any value.

- If the element passed to remove() doesn't exist, **KeyError** exception is thrown.

**Example 1: Remove an Element From The Set**

```
>>>language = {'English', 'French', 'German'}
>>>language.remove('German')
>>>print('Updated language set:', language)
Updated language set: {'English', 'French'}
```

**Example 2: Deleting Element That Doesn't Exist**

```
>>>animal = {'cat', 'dog', 'rabbit'}
>>>animal.remove('fish')
>>>print('Updated animal set:', animal)
```

**It generates Key Error**

- You can use the set discard() method if you do not want this error.
- The discard() method removes the specified element from the set.
- However, if the element doesn't exist, the set remains unchanged; you will not get an error.

```
>>>animal = {'cat', 'dog', 'rabbit'}
>>>animal.discard('fish')
>>>print('Updated animal set:', animal)
Updated animal set: {'rabbit', 'dog', 'cat'}
```

**Accessing items in a set:**

- A set element cannot be referenced by an index number since it is unordered.
- However, the items in a set can be looped through by using a for a loop.
- You can also choose to access a particular item in a set by using the “in” keyword. However, note that it is case sensitive.

**Using for loop:**

```
>>>fruits_set = {"mango", "banana", "orange"}
>>> for x in fruits_set:
    print(x)
mango
banana
orange
```

**Using membership “in” keyword:**

```
>>>fruits_set = {"mango", "banana", "orange"}
>>>print("orange" in fruits_set)
True
>>> print("Orange" in fruits_set)
False
```



**Clearing all items in a set:**

- You can remove all items in a set by using the clear() method.

```
>>> fruits_set.clear()
>>> print(fruits_set)
set()
```

**Copying a set**

- An existing set can be copied to a new set by using the copy() method.

```
>>> fruits_set = {"mango", "banana", "orange"}
>>> new_fruits_set = fruits_set.copy()
>>> print("new set: ", new_fruits_set)
new set: {"mango", "banana", "orange"}
```

**Sorting a set:**

- The values in a set can be sorted in ascending or descending order using the sorted() method.
- By passing the set variable inside the sorted() parameter, the items in a set will be printed in ascending order by default. The sorted() method takes in three parameter-iterable, key and reverse.
- The iterable parameter is required in which you need to specify the variable name of the set.
- The key and reverse parameter is optional.
- You can use the reverse parameter to sort the items in ascending order (reverse = False) or descending order (reverse = True).

**Example:**

```
>>> vowel_set = {"e", "a", "u", "o", "i"}
>>> print("Default sort: ", sorted(vowel_set))
Default sort: ['a', 'e', 'i', 'o', 'u']
>>> print("Ascending order: ", sorted(vowel_set, reverse=False))
Ascending order: ['a', 'e', 'i', 'o', 'u']
>>> print("Descending order: ", sorted(vowel_set, reverse=True))
Descending order: ['u', 'o', 'i', 'e', 'a']
```

**Python Set Operations:**

- Sets can be used to carry out mathematical set operations like union, intersection, difference and symmetric difference. We can do this with operators or methods.
- Let us consider the following two sets for the following operations.

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

**Set Union:**

- Union of A and B is a set of all elements from both sets.
- Union is performed using | operator. Same can be accomplished using the union() method.

**a) Using | operator :**

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

**b) Use Union method:**

```
# use union function
>>> A.union(B)
{1, 2, 3, 4, 5, 6, 7, 8}
# use union function on B
>>> B.union(A)
{1, 2, 3, 4, 5, 6, 7, 8}
```

**Set Intersection:**

- Intersection of A and B is a set of elements that are common in both the sets.
- Intersection is performed using & operator. Same can be accomplished using the intersection() method.

**a) Using & operator :**

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

**b) Use Intersection method:**

```
# use intersection function on A
>>> A.intersection(B)
{4, 5}
# use intersection function on B
>>> B.intersection(A)
{4, 5}
```

**Set Methods:**

Python Sets Methods	
Method	Description
add()	Adds an element to the set
clear()	Removes all the elements from the set
copy()	Returns a copy of the set
difference()	Returns a set containing the difference between two or more sets
difference_update()	Removes the items in this set that are also included in another, specified set
discard()	Remove the specified item
intersection()	Returns a set, that is the intersection of two other sets
intersection_update()	Removes the items in this set that are not present in other, specified set(s)
isdisjoint()	Returns whether two sets have a intersection or not
issubset()	Returns whether another set contains this set or not
issuperset()	Returns whether this set contains another set or not
pop()	Removes an element from the set
remove()	Removes the specified element
symmetric_difference()	Returns a set with the symmetric differences of two sets
symmetric_difference_update()	inserts the symmetric differences from this set and another
union()	Return a set containing the union of sets
update()	Update the set with the union of this set and others

**PYTHON DICTIONARIES****Dictionary:**

- The last standard type to add to our repertoire is the dictionary, the sole mapping type in Python.
- A dictionary is mutable and is another container type that can store any number of Python objects, including other container types.
- A dictionary is a collection which is ordered\*, changeable and does not allow duplicates.
- So they are implemented as resizable hash tables.
- Dictionaries are used to store data values in key:value pairs.
- As of Python version 3.7, dictionaries are ordered. In Python 3.6 and earlier, dictionaries are unordered.
- Dictionaries are written with curly brackets, and have keys and values
- Creating a dictionary is as simple as placing items inside curly braces { } separated by commas.
- An item has a key and a corresponding value that is expressed as a pair (key: value).
- While the values can be of any data type and can repeat, keys must be of immutable type (string, number or tuple with immutable elements) and must be unique.
- We can also create a dictionary using the built-in dict() function.

```
# empty dictionary
```

```
>>> my_dict = {}
```

```
# dictionary with integer keys
```

```
>>> my_dict = {1: 'apple', 2: 'ball'}
```

```
# dictionary with mixed keys
```

```
>>> my_dict = {'name': 'John', 1: [2, 4, 3]}
```

```
# using dict()
>>>my_dict = dict({1:'apple', 2:'ball'})
```

**Accessing values in Dictionary:**

- While indexing is used with other data types to access values, a dictionary uses keys.
- Keys can be used either inside square brackets [] or with the get() method.
- If we use the square brackets [], KeyError is raised in case a key is not found in the dictionary.
- On the other hand, the get() method returns None if the key is not found.

**a) Accessing values using square bracket:**

```
>>> my_dict = {'name': 'Jack', 'age': 26}
>>> print(my_dict['name'])
Jack
>>> print(my_dict['address'])
Traceback (most recent call last):
File "<pyshell#3>", line 1, in <module>
    print(my_dict['address'])
KeyError: 'address'
```

**b) Accessing values using get() method:**

```
>>> my_dict = {'name': 'Jack', 'age': 26}
>>> print(my_dict.get('age'))
26
>>> print(my_dict.get('address'))
None
>>>
```

**Get Values:**

- The values() method will return a list of all the values in the dictionary.
- ```
>>> x=my_dict.values()
>>> x
dict_values(['Jack', 26])
>>>
```

**Updating Dictionaries (Changing and adding dictionary elements):**

- Dictionaries are mutable. We can add new items or change the value of existing items using an assignment operator.
- If the key is already present, then the existing value gets updated.
- In case the key is not present, a new (key: value) pair is added to the dictionary.

```
>>> my_dict = {'name': 'Jack', 'age': 26}
>>> my_dict['age'] = 27 #updating value in dictionary
```

```
>>> print(my_dict)
{'name': 'Jack', 'age': 27}

# Adding Item in dictionary
>>> my_dict['address'] = 'Downtown'
>>> print(my_dict)
{'name': 'Jack', 'age': 27, 'address': 'Downtown'}
```

**Update Method:**

- The `update()` method will update the dictionary with the items from a given argument. If the item does not exist, the item will be added.
- The argument must be a dictionary, or an iterable object with key:value pairs.

```
>>> my_dict = {'name': 'Jack', 'age': 27, 'address': 'Downtown'}
>>> my_dict.update({'Gender': 'Male'})
>>> my_dict
{'name': 'Jack', 'age': 27, 'address': 'Downtown', 'Gender': 'Male'}
>>>
```

**Removing Dictionary Elements and Dictionaries:**

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

**`pop()` method:**

```
>>> squares = {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
>>> print(squares.pop(4))
16
>>> print(squares)
{1: 1, 2: 4, 3: 9, 5: 25}
```

**`popitem()` method:**

```
>>> print(squares)
{1: 1, 2: 4, 3: 9, 5: 25}
>>> print(squares.popitem())
(5, 25)
>>> print(squares)
{1: 1, 2: 4, 3: 9}
```

**`clear()` method:**

```
>>> squares
{1:1,2: 4, 3: 9}
>>> squares.clear()
>>> print(squares)
{}
```

**del keyword:**

- The del keyword removes the item with the specified key name or it can also delete the dictionary completely.

```
>>> del squares
>>> print(squares)
#this will cause an error because "squares" no longer exists.
```

**Copy a Dictionary:**

- You cannot copy a dictionary simply by typing dict2 = dict1, because: dict2 will only be a *reference* to dict1, and changes made in dict1 will automatically also be made in dict2.
- There are ways to make a copy, one way is to use the built-in dictionary method copy().

**Example:****Make a copy of a dictionary with the copy() method:**

```
>>> my_dict = {'name': 'Jack', 'age': 26}
>>> new_my_dict=my_dict.copy()
>>> print(new_my_dict)
{'name': 'Jack', 'age': 26}
```

- Another way to make a copy is to use the built-in function dict().

**Example:****Make a copy of a dictionary with the dict() function:**

```
>>> new_my_dict=dict(my_dict)
>>> print(new_my_dict)
{'name': 'Jack', 'age': 26}
```

**Dictionary Membership Test**

- We can test if a key is in a dictionary or not using the keyword in. Notice that the membership test is only for the keys and not for the values.

```
# Membership Test for Dictionary Keys
>>>squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
>>>print(1 in squares)
True
>>>print(2 not in squares)
True
```

```
# membership tests for key only not value#
```

```
Output: False
```

```
>>>print(49 in squares)
```

```
False
```

### Dictionary Methods

| Method                    | Description                                                                                                 |
|---------------------------|-------------------------------------------------------------------------------------------------------------|
| <code>clear()</code>      | Removes all the elements from the dictionary                                                                |
| <code>copy()</code>       | Returns a copy of the dictionary                                                                            |
| <code>fromkeys()</code>   | Returns a dictionary with the specified keys and value                                                      |
| <code>get()</code>        | Returns the value of the specified key                                                                      |
| <code>items()</code>      | Returns a list containing a tuple for each key value pair                                                   |
| <code>keys()</code>       | Returns a list containing the dictionary's keys                                                             |
| <code>pop()</code>        | Removes the element with the specified key                                                                  |
| <code>popitem()</code>    | Removes the last inserted key-value pair                                                                    |
| <code>setdefault()</code> | Returns the value of the specified key. If the key does not exist: insert the key, with the specified value |
| <code>update()</code>     | Updates the dictionary with the specified key-value pairs                                                   |
| <code>values()</code>     | Returns a list of all the values in the dictionary                                                          |

### Number Type Conversion:

- Python converts numbers internally in an expression containing mixed types to a common type for evaluation.
- But sometimes, you need to coerce a number explicitly from one type to another to satisfy the requirements of an operator or function parameter.
- Type **int(x)** to convert x to a integer.
- Type **float(x)** to convert x to a floating-point number.
- Type **complex(x)** to convert x to a complex number with real part x and imaginary part zero.
- Type **complex(x, y)** to convert x and y to a complex number with real part x and imaginary part y. x and y are numeric expressions

**Note:** long (long int) will be used in Python 2.x but not in Python 3.x

Python defines type conversion functions to directly convert one data type to another which is useful in day-to-day and competitive programming. This article is aimed at providing information about certain conversion functions.

There are two types of Type Conversion in Python:

1. Python Implicit Type Conversion
2. Python Explicit Type Conversion

### **Type Conversion in Python**

The act of changing an object's data type is known as type conversion. The Python interpreter automatically performs Implicit Type Conversion. Python prevents Implicit Type Conversion from losing data.

The user converts the data types of objects using specified functions in explicit type conversion, sometimes referred to as type casting. When type casting, data loss could happen if the object is forced to conform to a particular data type.

### **Implicit Type Conversion in Python**

In Implicit type conversion of data types in Python, the Python interpreter automatically converts one data type to another without any user involvement. To get a more clear view of the topic see the below examples.

#### **Example**

```
x = 10
print("x is of type:",type(x))
y = 10.6
print("y is of type:",type(y))
z = x + y

print(z)
print("z is of type:",type(z))
```

#### **Output**

```
x is of type: <class 'int'>
y is of type: <class 'float'>
20.6
z is of type: <class 'float'>
```

### **Explicit Type Conversion in Python**

In Explicit Type Conversion in Python, the data type is manually changed by the user as per their requirement. With explicit type conversion, there is a risk of data loss since we are forcing an expression to be changed in some specific data type. Various forms of explicit type conversion are explained below:



**Converting integer to float**

**int(a, base):** This function converts **any data type to an integer**. 'Base' specifies the **base in which the string is** if the data type is a string.

**float():** This function is used to convert **any data type to a floating-point number**.

**# initializing string**

```
s = "10010"
```

```
# printing string converting to int base 2
```

```
c = int(s,2)
```

```
print ("After converting to integer base 2 : ", end="")
```

```
print (c)
```

```
# printing string converting to float
```

```
e = float(s)
```

```
print ("After converting to float : ", end="")
```

```
print (e)
```

**Output:**

```
After converting to integer base 2 : 18
```

```
After converting to float : 10010.0
```

**UNIT II**

**Operators in Python:** Arithmetic operators, Assignment operators, Comparison operators, Logical operators, Identity operators, Membership operators, Bitwise operators, Precedence of operators, Expressions.

**Control Flow and Loops:** Indentation, if statement, if-else statement, chained conditional if-elif-else statement, Loops: While loop, for loop using ranges, Loop manipulation using break, continue and pass.

**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 | Description    | Example  |
|----------|----------------|----------|
| +        | Addition       | $x + y$  |
| -        | Subtraction    | $x - y$  |
| *        | Multiplication | $x * y$  |
| /        | Division       | $x / y$  |
| //       | Floor Division | $x // y$ |
| %        | Remainder      | $x \% y$ |
| **       | Exponentiation | $x ** y$ |

**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 |

**Precedence of Operators:**

Operator precedence affects how an expression is evaluated.

For example,  $x = 7 + 3 * 2$ ; here,  $x$  is assigned 13, not 20 because operator  $*$  has higher precedence than  $+$ , so it first multiplies  $3 * 2$  and then adds into 7.

**Example 1:**

```
>>> 3 + 4 * 2
```

```
11
```

Multiplication gets evaluated before the addition operation

```
>>> (10 + 10) * 2
```

```
40
```

Parentheses () overriding the precedence of the arithmetic operators

**Example 2:**

```
a = 20
```

```
b = 10
```

```
c = 15
```

```
d = 5
```

```
e = 0
```

```
e = (a + b) * c / d
```

```
 #(30 * 15) / 5
```

```
print("Value of (a + b) * c / d is ", e)
```

```
e = ((a + b) * c) / d
# (30 * 15) / 5
print("Value of ((a + b) * c) / d is ", e)
```

```
e = (a + b) * (c / d);
# (30) * (15/5)
print("Value of (a + b) * (c / d) is ", e)
```

```
e = a + (b * c) / d;
# 20 + (150/5)
print("Value of a + (b * c) / d is ", e)
```

**Output:**

```
Value of (a + b) * c / d is 90.0
Value of ((a + b) * c) / d is 90.0
Value of (a + b) * (c / d) is 90.0
Value of a + (b * c) / d is 50.0
```

**Expressions:**

An expression is a combination of values, variables, and operators. An expression is evaluated using assignment operator.

**Examples:**

```
Y=x + 17
>>> x=10
>>> z=x+20
>>> z
30
>>> x=10
>>> y=20
>>> c=x+y
>>> c
30
```

A value all by itself is a simple expression, and so is a variable.

```
>>> y=20
>>> y
20
```

Python also defines expressions only contain identifiers, literals, and operators. So,

**Identifiers:** Any name that is used to define a class, function, variable module, or object is an identifier.

**Literals:** These are language-independent terms in Python and should exist independently in any programming language. In Python, there are the string literals, byte literals, integer literals, floating point literals, and imaginary literals.

### Some of the python expressions are:

#### Generator expression:

**Syntax:** ( compute(var) for var in iterable )

```
>>> 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.

#### Conditional expression:

**Syntax:** true\_value if condition else false\_value

```
>>> x = "1" if True else "2"
>>> x
>>> '1'
```

**Control Flow and Loops**

conditional (if), alternative (if-else), chained conditional (if- elif -else),

Loops: while loop, for loop using ranges, Loop manipulation using pass, continue and break

**Conditional (if):**

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

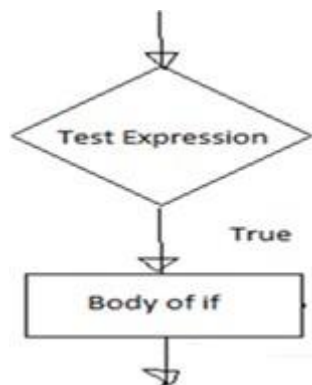
**Syntax:**

if expression:

statement(s)

If the boolean expression evaluates to TRUE, then the block of statement(s) inside the if statement is executed. If boolean expression evaluates to FALSE, then the first set of code after the end of the if statement(s) is executed.

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, "is smaller")
```

```
print("Finish")
```

**Output:**

```
3 is greater
```

done

-1 is smaller

Finish

### Alternative if(If-else):

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

The else statement is an optional statement and there could be at most only one else statement following if.

### Syntax of if - else :

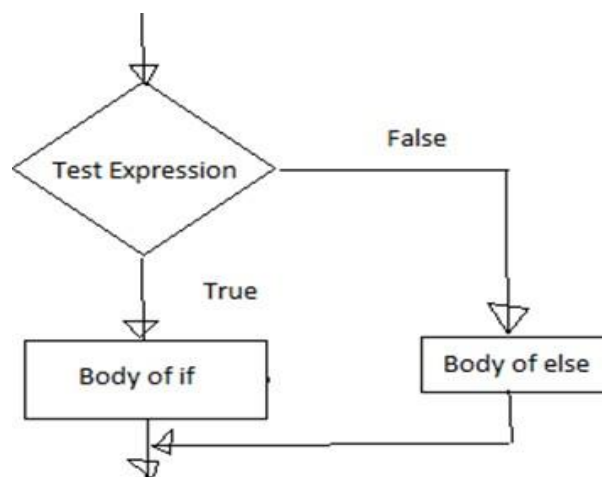
```
if test_expression:
```

```
    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 thanthe input given")
```



```
a=10
b=20
if a>b:
    print("A is Greater than B")
else:
    print("B is Greater than A")
```

### Chained Conditional: (If-elif-else):

The elif statement allows us to check multiple expressions for TRUE and execute a block of code as soon as one of the conditions evaluates to TRUE. Similar to the else, the elif statement is optional. However, unlike else, for which there can be at most one statement, there can be an arbitrary number of elif statements following an if.

### 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:

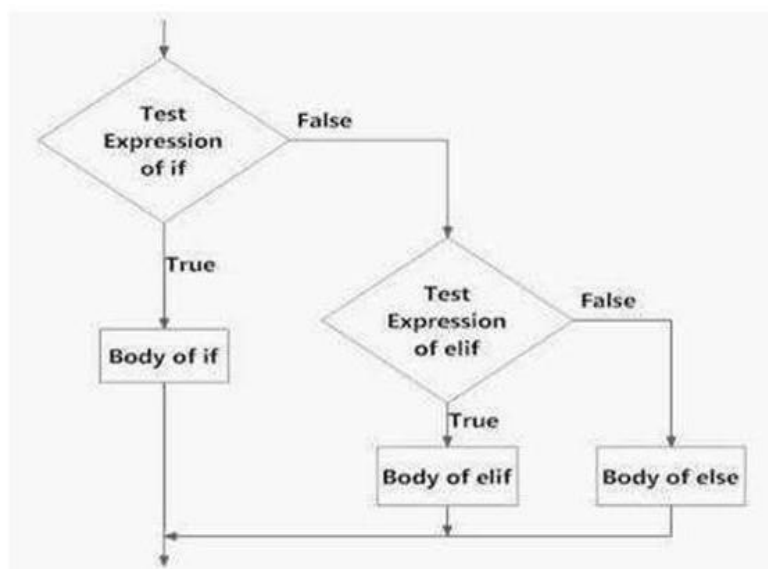


Fig: Operation of if – elif - else statement

**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 and a>c:
    print("a is greater")
elif b>c:
    print("b is greater")
else:
    print("c is greater")
```

**Iteration:**

A loop statement allows us to execute a statement or group of statements multiple times as long as the condition is true. Repeated execution of a set of statements with the help of loops is called iteration.

Loops statements are used when we need to run same code again and again, each time with a different value.

**Statements:**

In Python Iteration (Loops) statements are of three types:

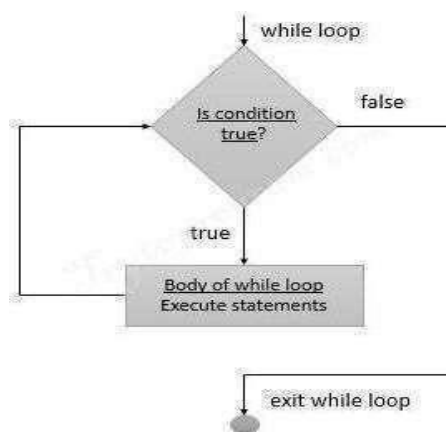
1. While Loop
2. For Loop
3. Nested For Loops

**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**

```
i=1
while i<=5:
    print("Mrcet college")
    i=i+1
```

**output:**

```
Mrcet college
Mrcet college
Mrcet college
Mrcet college
Mrcet college
```

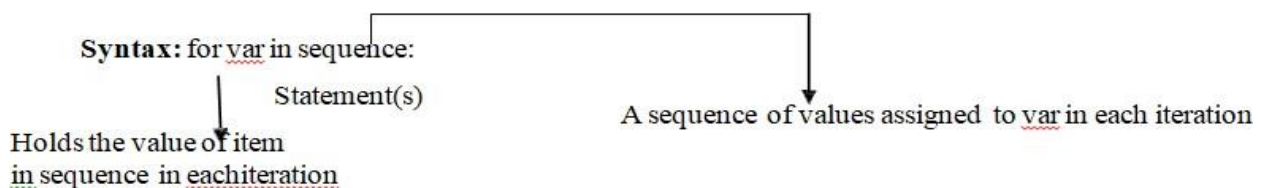
```
i = 1
while (i<=10):
    print(i)
    i += 1
```

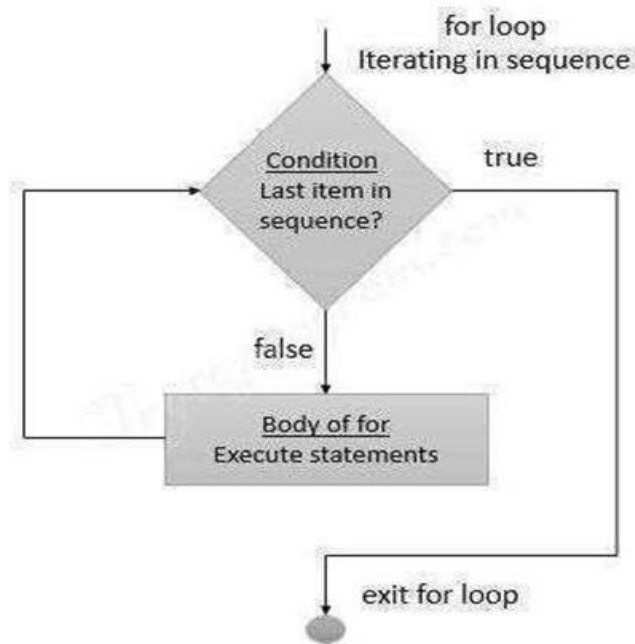
**output:**

```
1
2
3
4
5
6
7
8
9
10
```

**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



**Flowchart:****Example:**

```
numbers = [1, 2, 4, 6, 11, 20]
```

```
seq=0
```

```
for val in numbers:
```

```
    seq=val*val
```

```
    print(seq)
```

**Output:**

```
1
4
16
36
121
400
```

**Iterating over a list:**

```
#list of items
```

```
list = ['Apple', 'Banana', 'Orange', 'Grapes']
```

```
i = 1
```

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

**Output:**

```
Item 1 is Apple
Item 2 is Banana
Item 3 is Orange
Item 4 is Grapes
```

**Iterating over a dictionary:**

```
#creating a dictionary
college = {"CSE":"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:**

```
Keys are:
CSE
IT
ECE
Values are:
block1
block2
block3
```

**Nested For loop:**

When one Loop defined within another Loop is called Nested Loops.

**Syntax:**

```
for val in sequence:
    for val in sequence:
        statements

statements
```

**# Example 1 of Nested For Loops (Pattern Programs)**

```
for i in range(1,6):  
    for j in range(0,i):  
        print(i, end=" ")  
    print("")
```

**Output:**

```
1  
2 2  
3 3 3  
4 4 4 4  
5 5 5 5 5
```

**# Example 2 of Nested For Loops (Pattern Programs)**

```
for i in range(1,6):  
    for j in range(5,i-1,-1):  
        print(i, end=" ")  
    print("")
```

**Output:**

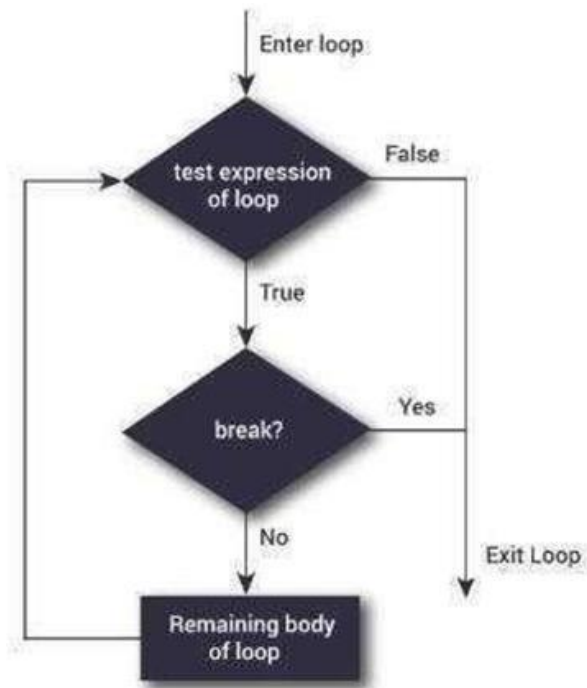
```
1 1 1 1 1  
2 2 2 2  
3 3 3  
4 4
```

**break and continue:**

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 breakcondition 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 upto number 88**

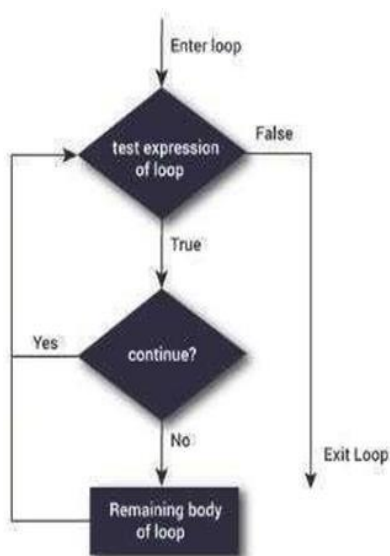
```
for num in [11, 9, 88, 10, 90, 3, 19]:  
    print(num)  
    if(num==88):  
        print("The number 88 is found")  
        print("Terminating the loop")  
        break
```

**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:**

```
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:**

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:**

>>>

## Unit III

**Arrays:** Definition, Advantages of Arrays, Creating an Array, Operations on Arrays, Arrays vs List, Importing the Array Module, Indexing and Slicing on Arrays,

**working with arrays using numPy** - Creating arrays using numpy, numpy Attributes and functions, Matrices in numpy.

### Arrays

Arrays are an ordered collection of elements of the same data type. Python arrays can only hold a sequence of multiple items that are of the same type.

#### Difference between Python Lists and Python Arrays

Lists and arrays behave similarly.

Just like arrays, lists are an ordered sequence of elements.

They are also mutable and not fixed in size, which means they can grow and shrink throughout the life of the program. Items can be added and removed, making them very flexible to work with.

However, lists and arrays are **not** the same thing.

**Lists** store items that are of **various data types**. This means that a list can contain integers, floating point numbers, strings, or any other Python data type, at the same time. That is not the case with arrays.

**Arrays** store only items that are of the **same single data type**. There are arrays that contain only integers, or only floating point numbers, or only any other Python data type you want to use.

#### Advantages of Python Arrays

Lists are built into the Python programming language, whereas arrays aren't. Arrays are not a built-in data structure, and therefore need to be imported via the array module in order to be used.

Arrays of the array module are useful when you want to work with homogeneous data.

They are also more compact and take up less memory and space which makes them more size efficient compared to lists.

#### Creating Python arrays

In order to create Python arrays, the array module has to be imported which contains all the necessary functions.

There are three ways you can import the array module:

1. By using import array

```
import array
```

```
#how you would create an array  
array.array()
```

2. import array as arr

```
#how you would create an array  
arr.array()
```

3. from array import \*, with \* importing all the functionalities available.

```
from array import *
```

```
#how you would create an array  
array()
```

### Define Arrays in Python

Once the array module is imported, we can then go on to define a Python array.

The **general syntax** for creating an array looks like this:

```
variable_name = array(typecode,[elements])
```

- variable\_name would be the name of the array.
- The typecode specifies what kind of elements would be stored in the array. Whether it would be an array of integers, an array of floats or an array of any other Python data type. Remember that all elements should be of the same data type.
- Inside square brackets you mention the elements that would be stored in the array, with each element being separated by a comma. You can also create an *empty* array by just writing variable\_name = array(typecode) alone, without any elements.

The different type codes that can be used with the different data types when defining Python arrays:

| TYPECODE | C TYPE             | PYTHON TYPE       | SIZE |
|----------|--------------------|-------------------|------|
| 'b'      | signed char        | int               | 1    |
| 'B'      | unsigned char      | int               | 1    |
| 'u'      | wchar_t            | Unicode character | 2    |
| 'h'      | signed short       | int               | 2    |
| 'H'      | unsigned short     | int               | 2    |
| 'i'      | signed int         | int               | 2    |
| 'I'      | unsigned int       | int               | 2    |
| 'l'      | signed long        | int               | 4    |
| 'L'      | unsigned long      | int               | 4    |
| 'q'      | signed long long   | int               | 8    |
| 'Q'      | unsigned long long | int               | 8    |
| 'f'      | float              | float             | 4    |
| 'd'      | double             | float             | 8    |

#### Example of how to define an array in Python:

```
import array as arr
numbers = arr.array('i',[10,20,30])
print(numbers)
```

```
#output
#array('i', [10, 20, 30])
```

**Example of how to create an array numbers of float data type.**

```
from array import *  
#an array of floating point values  
numbers = array('d',[10.0,20.0,30.0])  
print(numbers)
```

```
#output  
#array('d', [10.0, 20.0, 30.0])
```

**Array Indexing and How to Access Individual Items in an Array in Python**

Each item in an array has a specific address. Individual items are accessed by referencing their *index number*.

Indexing in Python, and in all programming languages and computing in general, starts at 0.

To access an element, we first write the name of the array followed by square brackets. Inside the square brackets you include the item's index number.

The index value of the last element of an array is always one less than the length of the array.

Where n is the length of the array, n - 1 will be the index value of the last item.

The general syntax would look something like this:

```
array_name[index_value_of_item]
```

**Example:**

```
import array as arr  
numbers = arr.array('i',[10,20,30])  
print(numbers[0]) # gets the 1st element  
print(numbers[1]) # gets the 2nd element  
print(numbers[2]) # gets the 3rd element
```

```
#output  
#10  
#20  
#30
```

We can also access each individual element using negative indexing.

With negative indexing, the last element would have an index of -1, the second to last element would have an index of -2, and so on.

**Example**

```
import array as arr
numbers = arr.array('i',[10,20,30])
print(numbers[-1]) #gets last item
print(numbers[-2]) #gets second to last item
print(numbers[-3]) #gets first item
```

```
#output
#30
#20
#10
```

**How to Slice an Array in Python**

To access a specific range of values inside the array, use the slicing operator, which is a colon :. When using the slicing operator and you only include one value, the counting starts from 0 by default. It gets the first item, and goes up to but not including the index number you specify.

```
import array as arr
#original array
numbers = arr.array('i',[10,20,30])
#get the values 10 and 20 only
print(numbers[:2]) #first to second position
```

```
#output
#array('i', [10, 20])
```

When you pass two numbers as arguments, you specify a range of numbers. In this case, the counting starts at the position of the first number in the range, and up to but not including the second one:

```
import array as arr
#original array
numbers = arr.array('i',[10,20,30])
#get the values 20 and 30 only
print(numbers[1:3]) #second to third position
```

```
#output
#array('i', [20, 30])
```

## Array Methods

Below are some operations that can be performed in an array:

1. **append():** This method is used to add the value mentioned in its arguments at the end of the array
2. **clear():** This method removes all the elements from the array
3. **copy():** This method creates and returns an identical copy of the array
4. **insert():** This method is used to add the value(x) at the ith position specified in its argument.
5. **pop():** This method removes the element at the position mentioned in its argument and returns it
6. **remove():** This method is used to remove the first occurrence of the value mentioned in its arguments
7. **index():** This method returns the index of the first occurrence of the value mentioned in the arguments.
8. **reverse():** This method reverses the array. In this example, we are reversing the array by using reverse()

### Example showing array methods

```
import array
arr1 = array.array('i', [1,2,3,4,5])
print('Elements of array are :')
for i in arr1:
    print(i, end=' ')
#methods of array module
arr1.append(6)
arr1.append(2)
print('Array elements after append operation : ')
for i in arr1:
    print(i, end=' ')
print('No. of occurrences of element 2 :', arr1.count(2))
print('Index of element 5 :', arr1.index(5))
print('Element removed from array :', arr1.pop(3))
print('Array elements after remove operation :')
for i in arr1:
    print(i, end=' ')
arr1.reverse()
print('Array elements after reverse operation :')
for i in arr1:
    print(i, end=' ')
```



### Array Types in Python

When talking about arrays, any programming language like C or Java offers two types of arrays. They are:

**Single dimensional arrays:** These arrays represent only one row or one column of elements. For example, marks obtained by a student in 5 subjects can be written as 'marks' array, as:

```
marks = array('i', [50, 60, 70, 66, 72])
```

The above array contains only one row of elements. Hence it is called single dimensional array or **one dimensional array**.

**Multi-dimensional arrays:** These arrays represent more than one row and more than one column of elements. For example, marks obtained by 3 students each one in 5 subjects can be written as 'marks' array as:

```
marks = [[50, 60, 70, 66, 72], [60, 62, 71, 56, 70], [55, 59, 80, 68, 65]]
```

The first student's marks are written in first row. The second student's marks are in second row and the third student's marks are in third row. In each row, the marks in 5 subjects are mentioned. Thus this array contains 3 rows and 5 columns and hence it is called **multi-dimensional array**.

```
marks = [[50, 60, 70, 66, 72],  
         [60, 62, 71, 56, 70],  
         [55, 59, 80, 68, 65]]
```

Each row of the above array can be again represented as a single dimensional array. Thus the above array contains 3 single dimensional arrays. Hence, it is called a two dimensional array. A two dimensional array is a combination of several single dimensional arrays. Similarly, a three dimensional array is a combination of several two dimensional arrays.

In Python, we can create and work with single dimensional arrays only. So far, the examples and methods discussed by us are applicable to single dimensional arrays.

Python does not support multi-dimensional arrays. We can construct multidimensional arrays using third party packages like **numpy (numerical python)**

## Working with numpy arrays

NumPy is a Python library used for working with arrays.

It also has functions for working in domain of linear algebra, fourier transform, and matrices.

NumPy was created in 2005 by Travis Oliphant. It is an open source project and you can use it freely.

NumPy stands for Numerical Python.

### Why Use NumPy?

In Python we have lists that serve the purpose of arrays, but they are slow to process.

NumPy aims to provide an array object that is up to 50x faster than traditional Python lists.

The array object in NumPy is called ndarray, it provides a lot of supporting functions that make working with ndarray very easy.

NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently. This is the main reason why NumPy is faster than lists

NumPy module has to be imported to use it.

The array object in NumPy is called ndarray.

We can create a NumPy ndarray object by using the array() function.

### Example:

```
import numpy as np
arr=np.array([1, 2, 3, 4, 5])
print(arr)
```

### Dimensions- Arrays:

#### 0-D Arrays:

0-D arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array.

The following code will create a zero-dimensional array with a value 36.

```
import numpy as np
arr = np.array(36)
print(arr)
```

**Output:** 36

**1-Dimensional Array:** An array that has 0-D arrays as its elements is called uni-dimensional or 1-D array

The code below creates a 1-D array,

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr)
```

**Output:** [1 2 3 4 5]

### Two Dimensional Arrays:

An array that has 1-D arrays as its elements is called a 2-D array.

These are often used to represent matrix

```
import numpy as np
arr1 = np.array([[1, 2, 3], [4, 5, 6]])
print(arr1)
```

**Output:**

```
[[1 2 3]
 [4 5 6]]
```

### Three Dimensional Arrays:

3-D arrays

An array that has 2-D arrays (matrices) as its elements is called 3-D array.

```
import numpy as np
arr1 = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])
print(arr1)
```

**Output:**

```
[[[1 2 3]
 [4 5 6]]
 [[1 2 3]
 [4 5 6]]]
```

## Higher Dimensional Arrays

An array can have any number of dimensions.

When the array is created, you can define the number of dimensions by using the ndmin argument.

```
import numpy as np
arr=np.array([1, 2, 3, 4],ndmin=5)
print(arr)
```

## Check Number of Dimensions

NumPy arrays provides the ndim attribute that returns an integer that tells us how many dimensions the array has.

```
import numpy as np
a=np.array(42)
b=np.array([1, 2, 3, 4, 5])
c=np.array([[1, 2, 3],[4, 5, 6]])
d=np.array([[[1, 2, 3],[4, 5, 6]],[[1, 2, 3],[4, 5, 6]]])
print(a.ndim)
print(b.ndim)
print(c.ndim)
print(d.ndim)
```

## Access Array Elements

Array indexing is the same as accessing an array element.

You can access an array element by referring to its index number.

The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.

```
import numpy as np
arr=np.array([1, 2, 3, 4])
print(arr[0])
```

To access elements from 2-D arrays we can use comma separated integers representing the dimension and the index of the element.

```
import numpy as np
arr=np.array([[1,2,3,4,5],[6,7,8,9,10]])
print('2nd element on 1st row: ',arr[0,1])
```

### Access 3-D Arrays

To access elements from 3-D arrays we can use comma separated integers representing the dimensions and the index of the element.

```
import numpy as np
arr=np.array([[[1, 2, 3],[4, 5, 6]],[[7, 8, 9],[10, 11, 12]]])
print(arr[0,1,2])
```

### Negative Indexing

Use negative indexing to access an array from the end.

**Print the last element from the 2nd dim:**

```
import numpy as np
arr=np.array([[1,2,3,4,5],[6,7,8,9,10]])
print('Last element from 2nd dim: ', arr[1,-1])
```

### Slicing arrays

Slicing in python means taking elements from one given index to another given index.

We pass slice instead of index : *[start:end:step]*

■ Slice elements from index 1 to index 5(excluding index 5)

```
import numpy as np
arr=np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[1:5])
```

### Negative Slicing

■ Slice from the index 3 from the end to index 1 from the end

```
import numpy as np
arr=np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[-3:-1])
```

**Output:**

[5,6]

**Return every other element from index 1 to index 5: (using step)**

```
import numpy as np
arr=np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[1:5:2])
```

**Slicing 2-D Arrays**

**From the second element, slice elements from index 1 to index 4 (not included):**

```
import numpy as np
arr=np.array([[1, 2, 3, 4, 5],[6, 7, 8, 9, 10]])
print(arr[1, 1:4])
```

**Output:**

[7,8, 9]

**Attributes of ndarray**

| Attributes | Description                                                                                                       |
|------------|-------------------------------------------------------------------------------------------------------------------|
| ndim       | returns number of dimension of the array                                                                          |
| size       | returns number of elements in the array                                                                           |
| dtype      | returns data type of elements in the array                                                                        |
| shape      | returns a tuple of integers indicating the number of elements that are stored along each dimension of the array.. |
| itemsize   | returns the size (in bytes) of each elements in the array                                                         |
| data       | returns the buffer containing actual elements of the array in memory                                              |

```
import numpy as np
arr=np.array([[1, 2, 3, 4, 5],[6, 7, 8, 9, 10]])
print(arr.ndim)
print(arr.size)
print(arr.dtype)
print(arr.shape)
print(arr.itemsize)
print(arr.data)
```

**Output:**

```
2
10
int64
(2, 5)
8
<memory at 0x7fb8a2925ff0>
```

**Array methods:****1. numpy.arange():**

The **arange([start,] stop[, step,][, dtype])** : returns an array with evenly spaced elements as per the interval

**Parameters :**

**start** : [optional] start of interval range. By default start = 0

**stop** : end of interval range

**step** : [optional] step size of interval. By default step size = 1,

For any output out, this is the distance between two adjacent values, out[i+1] - out[i].

**dtype** : type of output array

**Example**

```
import numpy as np
a=np.arange(4)
print(a)
b=arange(4,10)
print(b)
c=np.arange(4,20,3)
print(c)
```

**Output:**

```
[0,1,2,3]
[4,5,6,7,8,9]
[4,7,10,13,17]
```

**2. numpy.reshape()**

The **numpy.reshape()** function allows us to reshape an array in Python. Reshaping basically means, changing the shape of an array. And the shape of an array is determined by the number of elements in each dimension.

**reshape()** function shapes an array without changing the data of the array.

```
numpy.reshape(a, newshape, order='C')
```

**Parameters:**

| Name     | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   | Required / Optional |
|----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------|
| a        | Array to be reshaped.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         | Required            |
| newshape | The new shape should be compatible with the original shape. If an integer, then the result will be a 1-D array of that length. One shape dimension can be -1. In this case, the value is inferred from the length of the array and remaining dimensions.                                                                                                                                                                                                                                                                                                                                                                                                                                                      | Required            |
| order    | Read the elements of a using this index order, and place the elements into the reshaped array using this index order. 'C' means to read / write the elements using C-like index order, with the last axis index changing fastest, back to the first axis index changing slowest. 'F' means to read / write the elements using Fortran-like index order, with the first index changing fastest, and the last index changing slowest. Note that the 'C' and 'F' options take no account of the memory layout of the underlying array, and only refer to the order of indexing. 'A' means to read / write the elements in Fortran-like index order if a is Fortran contiguous in memory, C-like order otherwise. |                     |

**Example:**

```
import numpy as np
x = np.array([[2,3,4], [5,6,7]])
print(np.reshape(x, (3, 2)))
```

**Output:**

```
array([[2, 3],
       [4, 5],
       [6, 7]])
```



**Reshaping a NumPy array using a variable and an inferred dimension**

```
import numpy as np

x = np.array([[2,3,4], [5,6,7]])

print(np.reshape(x, (3, -1)))
```

**Output:**

```
array([[2, 3],
       [4, 5],
       [6, 7]])
```

The -1 argument indicates that we want NumPy to automatically determine the number of columns needed based on the total number of elements in the array.

**3. ndarray.flatten()**

The ndarray.flatten() function is used to get a copy of an given array collapsed into one dimension.

This function is useful when we want to convert a multi-dimensional array into a one-dimensional array

**Syntax:**

```
numpy.ndarray.flatten(order='C')
```

**Parameters:**

| Name  | Description                                                                                                                                                                                                                                                                                                       | Required / Optional |
|-------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------|
| order | ‘C’ means to flatten in row-major (C-style) order. ‘F’ means to flatten in column-major (Fortran- style) order. ‘A’ means to flatten in column-major order if a is Fortran contiguous in memory, row-major order otherwise. ‘K’ means to flatten a in the order the elements occur in memory. The default is ‘C’. |                     |

**Example:**

```
import numpy as np
y = np.array([[2,3], [4,5]])
print(y.flatten())
```

**Output:**

```
array([2, 3, 4, 5])
```

**4. numpy.zeros()**

The `numpy.zeros()` function provide a new array of given shape and type, which is filled with zeros.

```
import numpy as np
a=np.zeros(3)
print(a)
```

```
[0., 0., 0.,]
```

```
import numpy as np
a=np.zeros((6,2), dtype = int)
print(a)
```

```
[[0 0],
 [0 0],
 [0 0]]
```

5. The **numpy.ones()** function returns a new array of given shape and type, with ones.  
(Similar to `numpy.zeros`)

```
import numpy as np
a=np.ones((6,2), dtype = int)
print(a)
```

## 6. `numpy.eye()`

The **`numpy.eye()`** function in Python is used to return a two-dimensional array with ones (1) on the diagonal and zeros (0) elsewhere.

### Parameters

The `numpy.eye()` function takes the following parameter values:

- **N**: This represents the number of rows we want in the output array.
- **M**: This represents the number of columns we want in the output array. This is optional.
- **k**: This represents the index of the diagonal. 0 is the default value and the main diagonal. This is optional.
- **dtype**: This represents the data type of array to be returned. This is optional.
- **order**: This represents whether the output should be stored in C or F order in memory. This is optional.
- **like**: This is the array prototype or array\_like object.

An array with 3 rows with the ones starting at the index i.e from the second column

```
import numpy as np
myarray = np.eye(3, k=1,dtype=int)
print(myarray)
```

### Output:

```
[[0 1 0]
 [0 0 1]
 [0 0 0]]
```

An array with 3 rows, 3 columns and with the ones starting at the index 0 i.e from the first column

```
import numpy as np
myarray = np.eye(3,3 ,dtype=int)
print(myarray)
```

### Output:

```
[[1 0 0]
 [0 1 0]
 [0 0 1]]
```

**Operations on numpy Arrays:****Arithmetic Operations:**

```
# Python code to perform arithmetic operations on NumPy array

import numpy as np

# Operations on 1D array
# Initializing the array
arr1 = [1,2,3,4,5]

print('First array:')
print(arr1)

print('\nSecond array:')
arr2 = [6,7,8,9,10]
print(arr2)

print('\nAdding the two arrays:')
print(np.add(arr1, arr2)) # method to perform addition of two arrays

print(arr1+arr2) # using + operator to perform addition of two arrays

print('\nSubtracting the two arrays:')
print(np.subtract(arr1, arr2))

print(arr1 - arr2)

print('\nMultiplying the two arrays:')
print(np.multiply(arr1, arr2))

print(arr1 * arr2)

print('\nDividing the two arrays:')
print(np.divide(arr1, arr2))

print(arr1 / arr2)

# Operations on 2D array
# Initializing the array

arr1 = np.arange(4).reshape(2, 2)

print('First array:')
print(arr1)

print('\nSecond array:')
```

```
arr2 = np.arange(1,5).reshape(2,2)
print(arr2)

print('\nAdding the two arrays:')
print(np.add(arr1, arr2))    # method to perform addition of two arrays

print(arr1+arr2) # using + operator to perform addition of two arrays

print('\nSubtracting the two arrays:')
print(np.subtract(arr1, arr2))

print(arr1 - arr2)

print('\nMultiplying the two arrays:')
print(np.multiply(arr1, arr2))

print(arr1 * arr2)

print('\nDividing the two arrays:')
print(np.divide(arr1, arr2))

print(arr1 / arr2)

print(np.dot(arr1,arr2))    # method to perform matrix multiplication

print(np.transpose(arr1))    # method to perform transpose of a matrix
```

## UNIT-IV

**Functions:** Defining a function, Calling a Function, Passing parameters and arguments, Python Function arguments: Positional Arguments, Keyword Arguments, Default Arguments, Variable-length arguments, Scope of the Variables in a Function–Local and Global Variables. Recursive functions, Anonymous functions, command-line arguments. Higher order functions - map(), filter() and reduce() functions in Python.

### Functions

Functions and its use: 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

**Parameters and arguments:**

Parameters are passed during the definition of function while Arguments are passed during the function call.

**Example:**

#here a and b are parameters

```
def add(a,b): ##function definition
```

```
    return a+b
```

#12 and 13 are arguments #function call

```
result=add(12,13)
```

```
print(result)
```

**Output:**

25

Some examples on functions:

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

#function definition

```
def display():
```

```
    print("vandemataram")
```

```
    print("i am in main")
```

#function call

```
display()
```

```
print("i am in main")
```

**Output:**

i am in main

vandemataram

i am in main

**#Type1 : No parameters and no return type****Example:**

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

**Output:**

function 1

**#Type 2: with param with out return type**

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

**Output:**

Hello

**#Type 3: without param with return type**

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

**Output:**

welcome to python

**#Type 4: with param with return type**

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

**Output:**

python is better then c



## Python function arguments

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():  
    print("helloworld")  
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

**Example:**

```
def hf():  
    print("hw")  
    print("ghkfjg 66666")  
hf()  
hf()  
hf()
```

**Output:**

hw  
ghkfjg 66666  
hw  
ghkfjg 66666  
hw  
ghkfjg 66666

**Example:**

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

**Output:**

9

**Example:**

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

**Output:**

9

**Example:**

```
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

**Example:**

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

**Output:**

>>>

**Example:**

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

**Output:**

HELLOCOLLEGE

**Passing Arguments**

**Positional arguments:** These are the arguments that need to be included in the proper position or order. The first positional argument always needs to be listed first when the function is called. The second positional argument needs to be listed second and the third positional argument listed third, etc.

**Example:**

```
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'
```

**Example:**

```
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.

**Example:**

```
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.

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
```

**Example:**

```
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 Aare 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","cse"))             #hello("good morning","cse")
```

**Output:**

good morning,cse

**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:

**Example:**

```
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:

**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“.

**Example:**

```
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

**Local and Global scope:****Local Scope:**

A variable which is defined inside a function is local to that function. It is accessible from the point at which it is defined until the end of the function, and exists for as long as the function is executing

**Global Scope:**

A variable which is defined in the main body of a file is called a global variable. It will be visible throughout the file, and also inside any file which imports that file.

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
```

**Example:**

```
x = "global"  
def f():  
    print("x inside :", x)  
f()  
print("x outside:", x)
```

**Output:**

```
x inside : global  
x outside: global
```

**Example:**

```
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,

**Example:**

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

Then when we try to run it shows an error,



**Output:**

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.

**Example:**

# 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:**

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.

**Example:**

# 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:**

local x: 10

global x: 5

**Example:**

#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:**

```
Enter radius of circle 3
28.259999999999998
```

#Program to write sum different product and function.

**Example:**

```
def calculate(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 bvalue"))#using arguments with return value
#function call
s,d,p,q,m = calculate(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:**

```
Enter a value 5
Enter b value 6
Sum= 11
diff= -1 mul= 30
div= 0.8333333333333334
mod= 5
```

**Example:**

#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:**

Enter a value 5  
Enter b value-2 big number= 5

**Example:**

#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:**

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

**Example:**

#Write 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:**

Enter one subject marks 35  
fail

**Example:**

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

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

**Output:**

mrcetcse dept 1  
mrcetcse dept 2  
mrcetcse dept 3  
mrcetcse dept 4  
mrcetcse dept 5  
mrcetcse dept 6  
mrcetcse dept 7  
mrcetcse dept 8  
mrcetcse dept 9

**Recursion in Python:**

The term Recursion can be defined as the process of defining something in terms of itself. In simple words, it is a process in which a function calls itself directly or indirectly.

```
def func(): <--  
    |  
    | (recursive call)  
    |  
func() ----
```

**Example 1:** A Fibonacci sequence is the integer sequence of 0, 1, 1, 2, 3, 5, 8....

```
# Program to print the fibonacci series upto n_terms  
# Recursive function  
def recursive_fibonacci(n):  
    if n <= 1:  
        return n  
    else:  
        return(recursive_fibonacci(n-1) + recursive_fibonacci(n-2))  
  
n_terms = 10  
# check if the number of terms is valid  
if n_terms <= 0:  
    print("Invalid input ! Please input a positive value")  
else:  
    print("Fibonacci series:")  
    for i in range(n_terms):  
        print(recursive_fibonacci(i))
```

### Output:

Fibonacci series:

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

**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....

**Example:**

# Write a program to double a given number

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

**Output:**

10

**Example:**

#Write a program to sum of two numbers

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

**Output:**

9

**Example:**

#Write a program to find biggest of two numbers

```
biggest = lambda x,y: a if x>y else y  
print(biggest(20,30))
```

**Output:**

30

**Command Line Arguments:**

The arguments that are given after the name of the program in the command line shell of the operating system are known as Command Line Arguments.

**Example:**

```
# Python program to demonstrate
# command line arguments

import sys
print ('argument list', sys.argv)
name = sys.argv[1]
print ("Hello { }. How are you?".format(name))
```

**Output:**

```
C:\Python311>python hello.py CSE
#argument list ['hello.py', 'CSE']
Hello CSE. How are you?
```

**Higher Order Functions:**

If a function contains other functions as a parameter or returns a function as an output i.e, the functions that operate with another function are known as Higher order Functions. It is worth knowing that this higher order function is applicable for functions and methods as well that takes functions as a parameter or returns a function as a result. Python too supports the concepts of higher order functions.

**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 where the items are filtered through a function to test if the item is accepted or not.

**Syntax:** filter(function, iterable)

**Example:**

```
#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:**

```
[2, 4, 6]
```

**Example:**

```
#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:**

```
[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)
```

**Example:**

```
#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:**

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

**Example:**

```
# 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:**

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



**Example:**

# 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:**

[2, 4, 6, 8]

**Reduce():**

The reduce(fun,seq) function is used to apply a particular function passed in its argument to all of the list elements mentioned in the sequence passed along. This function is defined in “functools” module.

- 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.

**Example:**

#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:**

55

**Example:**

#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:**

120

**Fruitful functions:**

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 expression as a return value."

(or)

Any function that returns a value is called Fruitful function. A Function that does not return a value is called a void function.

The Keyword return is used to return back the value to the called function.

# 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], [1,7,8,9,10]))
print(common_data([1,2,3,4,5], [6,7,8,9,10]))
```

**Output:**

```
True
True
None
```

**Example:**

```
def area(radius):
    b = 3.14159 * radius**2
    return b
```

Brief on other functions like **sort**, **sorted** and **range**:

The sort() method sorts the elements of a given list in a specific ascending or descending order.

The syntax of the sort() method is:

```
list.sort(key=..., reverse=...)
```

**Example:**

```
L1=[2,4,6,8,1,3,5]
L1.sort()
L2=[9,11,13,10,12,15,14]
L2.sort()
```

The **sorted()** function returns a sorted list of the specified iterable object.

You can specify ascending or descending order. Strings are sorted alphabetically, and numbers are sorted numerically.

**Note:** You cannot sort a list that contains BOTH string values AND numeric values. Syntax:

```
sorted(iterable, key=key, reverse=reverse)
```

**Example:**

```
a = (1, 11, 2)
x = sorted(a)
print(x)
```

The built-in function **range()** generates the integer numbers between the given start integer to the stop integer, i.e., It returns a range object. Using for loop, we can iterate over a sequence of numbers produced by the range() function

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]

**Example:**

```
x=10
for i in range(1,x,2):
    print(i)
```

**Output:**

```
1
3
5
7
9
```

## UNIT – V

**File Handling in Python:** Introduction to files, Text files and Binary files, Access Modes, Writing Data to a File-write() and writelines(), Reading Data from a File-read(),readline() and readlines(), Random access file operations-seek() and tell().

**Error Handling in Python:** Introduction to Errors and Exceptions: Compile-Time Errors, Logical Errors, Runtime Errors, Types of Exceptions, Python Exception Handling Using try, except and finally statements.

### File Handling in Python

Files are named locations on disk to store related information. They are used to permanently store data in a non-volatile memory (e.g. hard disk).

Python provides inbuilt functions for creating, writing and reading files. There are two types of files that can be handled in python, normal text files and binary files (written in binary language,0s and 1s).

**Text files:** A text file is simply a sequence of ASCII or Unicode characters. Each line of text is terminated with a special character called EOL (End of Line), which is the new line character ('\n') in python by default. Python programs, contents written in text editors are some of the example of text files. Text files are less prone to get corrupted as any undesired change may just show up once the file is opened and then can easily be removed.

**Binary files:** Binary file store data in the form of sequence of bytes grouped into eight bits or sometimes sixteen bits. These bits represent custom data and such files can store multiple types of data (images, audio, text, etc) under a single file. A binary file stores the data in the same way as stored in the memory. The .exe files, mp3 file, image files, word documents are some of the examples of binary files. There is no terminator for a line and the data is stored after converting it into machine understandable binary language. Since binary files store data in sequential bytes, a small change in the file can corrupt the file and make it unreadable to the supporting application.

### Opening and Closing Files:

Python provides basic functions and methods necessary to manipulate files by default. The file manipulation is done using a **file** object.

### The open() Function:

Before we can read or write a file, it has to be opened using Python's built-in *open()* function. This function creates a **file** object which would be utilized to call other support methods associated with it.

**Syntax:**

```
file object = open(file_name [, access_mode][, buffering])
```

**Parameters**

**file\_name:** The file\_name argument is a string value that contains the name of the file that you want to access.

**access\_mode:** The access\_mode determines the mode in which the file has to be opened ie. read, write append etc. A complete list of possible values is given below in the table. This is optional parameter and the default file access mode is read (r)

**buffering:** If the buffering value is set to 0, no buffering will take place. If the buffering value is 1, line buffering will be performed while accessing a file. If you specify the buffering value as an integer greater than 1, then buffering action will be performed with the indicated buffer size. If negative, the buffer size is the system default(default behavior).

| Modes | Description                                                                                                                                                                                        |
|-------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| r     | Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.                                                                                  |
| rb    | Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.                                                                 |
| r+    | Opens a file for both reading and writing. The file pointer will be at the beginning of the file.                                                                                                  |
| rb+   | Opens a file for both reading and writing in binary format. The file pointer will be at the beginning of the file.                                                                                 |
| w     | Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.                                                                 |
| wb    | Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.                                                |
| w+    | Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.                                |
| wb+   | Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.               |
| a     | Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing. |

|     |                                                                                                                                                                                                                                            |
|-----|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ab  | Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.                        |
| a+  | Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.                  |
| ab+ | Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing. |
| x   | <b>for exclusive creation.</b> If you open a file in mode x , the file is created and opened for writing – but only if it doesn't already exist.                                                                                           |

### The file object attributes:

| Attribute   | Description                                      |
|-------------|--------------------------------------------------|
| file.closed | Returns true if file is closed, false otherwise. |
| file.mode   | Returns access mode with which file was opened.  |
| file.name   | Returns name of the file.                        |

### Example:

```
fo = open("f1.txt", "wb")
print ("Name of the file: ", fo.name )
print ("Closed or not : ", fo.closed )
print ("Opening mode : ", fo.mode )
```

### Output:

```
Name of the file: f1.txt
Closed or not : False
Opening mode : wb
```

### close() Method:

The close() method of a *file* object flushes any unwritten information and closes the file object, after which no more writing can be done.

Python automatically closes a file when the reference object of a file is reassigned to another file.

**Syntax:**

```
fileObject.close();
```

**Example:**

```
fo = open("fl.txt", "wb")
print( "Name of the file: ", fo.name)
fo.close()
```

**Output:**

Name of the file: fl.txt

**Reading and Writing Files:**

The file object provides a set of access methods to read and write files.

**The write() Method:**

The write() method writes any string to an open file. It does not add a newline character ('\n') to the end of the string:

**Syntax:**

```
fileObject.write(string)
```

| Method          | Description                                                                                                                                                 |
|-----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| write(string)   | This writes the string to the file                                                                                                                          |
| writelines(seq) | This writes the sequence to the file. No line endings are appended to each sequence item. The appropriate line ending(s) need to be added based on the need |

**Example using write():**

```
fo = open("fl.txt", "wb")
fo.write( "Python is a great language.\nYeah its great!!\n");
fo.close()
```

**Example using writelines()**

```
lst = ["Python is a great language.", "Yeah its great!!"]
fo=open("fl.txt","w")
fo.writelines(lst)
fo.close()
```



**Python code to read data from user and write it to a file.**

```
f=open("myfile.txt","a+")
str= " "
print("Enter text to append( @ at end): ")
while str!='@':
    str=input()
    if str!='@':
        f.write(str+"\n")
f.close()
```

**The read() Method:**

The read() method read a string from an open file

**Syntax:**

```
fileObject.read([count])
```

| Method                       | Description                                                                                                                                                                                                                         |
|------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| fileobject.read(size=-1)     | This reads from the file based on the number of size bytes. If no argument is passed or None or -1 is passed, then the entire file is read.                                                                                         |
| fileobject.readline(size=-1) | This reads at most size number of characters from the line. This continues to the end of the line and then wraps back around. If no argument is passed or None or -1 is passed, then the entire line (or rest of the line) is read. |
| fileobject.readlines()       | This reads the remaining lines from the file object and returns them as a list                                                                                                                                                      |

**Example:**

Python code to read data from myfile.txt and display it on the screen

```
fo = open('myfile.txt','r')
data = fo.read()      # reads entire content of the file
print(data)
fo.close()
```

```
fo = open('myfile.txt','r')
data = fo.readline()      # reads a line
print(data)
fo.close()
```

```
fo = open('myfile.txt','r')
data = fo.readlines()     # reads all lines of file as elements of a list
print(data)
fo.close()
```

### Write a python program to perform append and read operations on a file

```
f1=open("file2.txt","w")
f1.write("Hello All ")
f1.close()
f1=open("file2.txt","a")
f1.write("Welcome to Mrcet")
f1.close()
f1=open("file2.txt","r")
print(f1.read())
f1.close()
```

### File Positions:

The **tell()** method tells you the current position within the file in other words, the next read or write will occur at that many bytes from the beginning of the file

The **seek(offset[, from])** method changes the current file position. The offset argument indicates the number of bytes to be moved. The from argument specifies the reference position from where the bytes are to be moved. If from is set to 0, it means use the beginning of the file as the reference position and 1 means use the current position as the reference position and if it is set to 2 then the end of the file would be taken as the reference position.

**Python code showing the usage of tell() and seek() methods on file object**

```
fo = open("f1.txt", "r+")
str = fo.read(10);
print "Read String is : ", str
position = fo.tell();
print "Current file position : ", position
position = fo.seek(0, 0);
str = fo.read(10);
print "Again read String is : ", str
fo.close()
```

**Exception Handling**

In python there are three types of errors

- Compile-time Errors
- Runtime Errors
- Logical Errors

**Compile-time or Syntax Errors:** These occur when your code violates the syntactical rules of Python. Such errors are detected by Python compiler and need to be corrected by programmer.

**Example:**

```
count=1
while count<100          # ':' missing
    print(count)
```

**Runtime Errors:** Also known as exceptions, these errors happen during program execution when PVM cannot execute the byte code. Errors such as attempting to divide by zero or accessing a variable that doesn't exist. Exception handling mechanisms can be used to handle these errors

**Logical Errors:** Logical errors occur when there's a flaw in program's logic like using a wrong formula. These errors are not detected by Python compiler or PVM. Programmer is solely responsible for them.

**Exceptions:**

An exception is runtime error, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

In general, when a Python script encounters a situation that it can't cope with, it raises an exception. An exception is a Python object that represents an error. When a Python script raises an exception, it must either handle the exception immediately otherwise it would terminate and come out.

**Syntax**

```

try:
    statements;
.....
except Exception I:
    #if there is exception 1, then execute this block.
    statements

except Exception II:
    #if there is exception 1, then execute this block.
    statements
.....
else:
    #if there is no exception, then execute this block.
    statements
finally:
    #somecode ....(always executed)
    statements

```

Suspicious code that may raise an exception can be handled by placing the suspicious code in a **try:** block. After the try: block, an **except:** statement, followed by a block of code is included which handles the problem. After the except clause(s), an else-clause can be included. The code in the else-block executes if the code in the try: block does not raise an exception.

**Example**

```

try:
    a = int(input("Enter numerator number: "))
    b = int(input("Enter denominator number: "))
    print("Result of Division: ",(a/b))
except ZeroDivisionError:
    print("Denominator cannot be zero..... ")
finally:
    print("Code execution Wrap up!")

```

A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.

**Syntax:**

```

try:
    statements
.....
except(Exception1[,Exception2[,...ExceptionN]]):
    statements
.....
else:
    statements

```

**Example: To print the element at a given position in a list**

```
l1=[1,2,3,4,5]
```

```
try:
```

```
    n=int(input("Enter position to retrieve value: "))
```

```
    print(l1[n])
```

```
except (IndexError, ValueError):
```

```
    print("Cannot continue due to error... try again!!")
```

```
else:
```

```
    print("Element retrieved successfully...")
```

A generic except clause can also be provided, which handles any exception.

**Syntax**

```
try:
```

```
    statements
```

```
    .....
```

```
except:
```

```
    statements
```

```
    .....
```

```
else:
```

```
    statements
```

**Example**

```
try:
```

```
    a = int(input("Enter numerator number: "))
```

```
    b = int(input("Enter denominator number: "))
```

```
    print("Result of Division: ",(a/b))
```

```
except:
```

```
    print("Error occurred....cannot proceed.")
```

```
finally:
```

```
    print("Code execution Wrap up!")
```

**Built-in Exceptions**

| Exception         | Description                                                  |
|-------------------|--------------------------------------------------------------|
| ImportError       | Raised when an imported module does not exist                |
| IndentationError  | Raised when indentation is not correct                       |
| IndexError        | Raised when an index of a sequence does not exist            |
| KeyError          | Raised when a key does not exist in a dictionary             |
| NameError         | Raised when a variable does not exist                        |
| OverflowError     | Raised when the result of a numeric calculation is too large |
| TypeError         | Raised when two different types are combined                 |
| UnboundLocalError | Raised when a local variable is referenced before assignment |
| ValueError        | Raised when there is a wrong value in a specified data type  |
| ZeroDivisionError | Raised when the second operator in a division is zero        |