

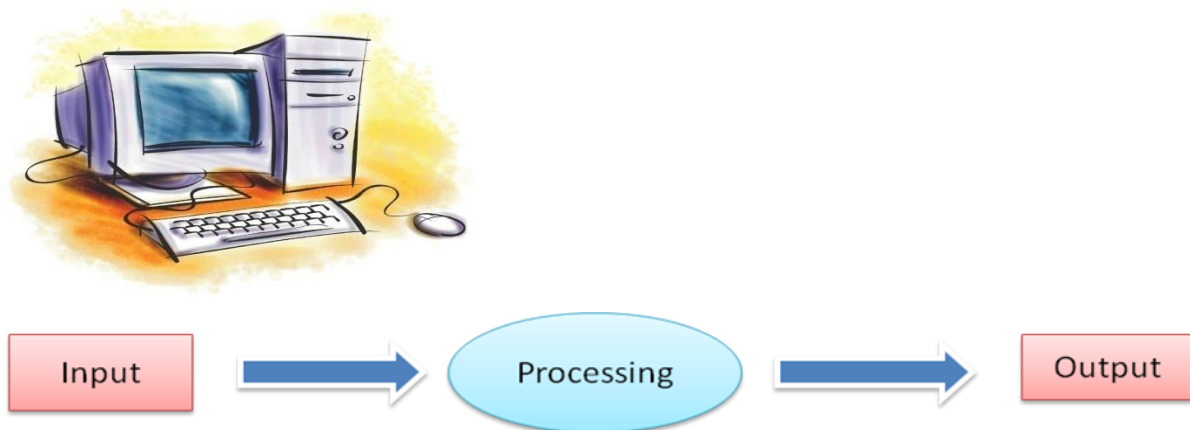
## Unit - 1: Knowing the Computer

**Definition and Block Diagram of a Computer.** Basic parts of a computer (Memory, CPU, Input, and Output), Memory hierarchy, Circuits and Logic, Hardware vs Software, Representation of Data in memory (integer (including negative), floating points etc. to text, images, audio and video), Principle of Abstraction, Language Hierarchy - Machine Language to High Level Language, Compiler, Interpreter, The Command Line Interface (basic linux commands)

### Computer:

A computer is an electronic device which accepts data as input, performs operations on the data based on the instructions stored in the memory and produces output.

General representation of a computer is as shown below:



- **Data** is collection of raw facts and figures.
- **Information** comprises processed data to provide answers to *who, what, where* and *when* type of questions.
- Early computers used vacuum tubes. Technological advances led to new generation of computers that were considerably smaller, faster and less expensive than previous ones.
- The elements of a computer fall into two major categories
  1. Hardware
  2. Software
- **Hardware** is the equipment used to perform the necessary computations and includes central processing unit, monitor, keyboard, mouse, printer and speakers.
- The **software** is the collection of programs that allow the hardware to serve its purpose. The purpose of the software is to use the underlying hardware components.

- **Program** – a list of instructions that enable a computer to perform a specific task
- **Instructions** are the commands given to the computer that tells what it has to do.

### **Characteristics of a computer:**

- |             |             |              |
|-------------|-------------|--------------|
| ✓ Speed     | ✓ Diligence | ✓ No IQ      |
| ✓ Accuracy  | ✓ Versatile | ✓ Economical |
| ✓ Automatic | ✓ Memory    |              |

### **Basic applications of computer:**

- |                      |                         |                            |
|----------------------|-------------------------|----------------------------|
| ✓ Communication      | ✓ Movies                | ✓ Weather forecasting      |
| ✓ Desktop Publishing | ✓ Travel and tourism    | ✓ Education                |
| ✓ Government         | ✓ Business and industry | ✓ Online banking           |
| ✓ Traffic control    | ✓ Hospitals             | ✓ Industry and Engineering |
| ✓ Legal systems      | ✓ Simulation            | ✓ Robots                   |
| ✓ Retail business    | ✓ Geology               | ✓ Sports                   |
| ✓ Music              | ✓ Astronomy             |                            |

---

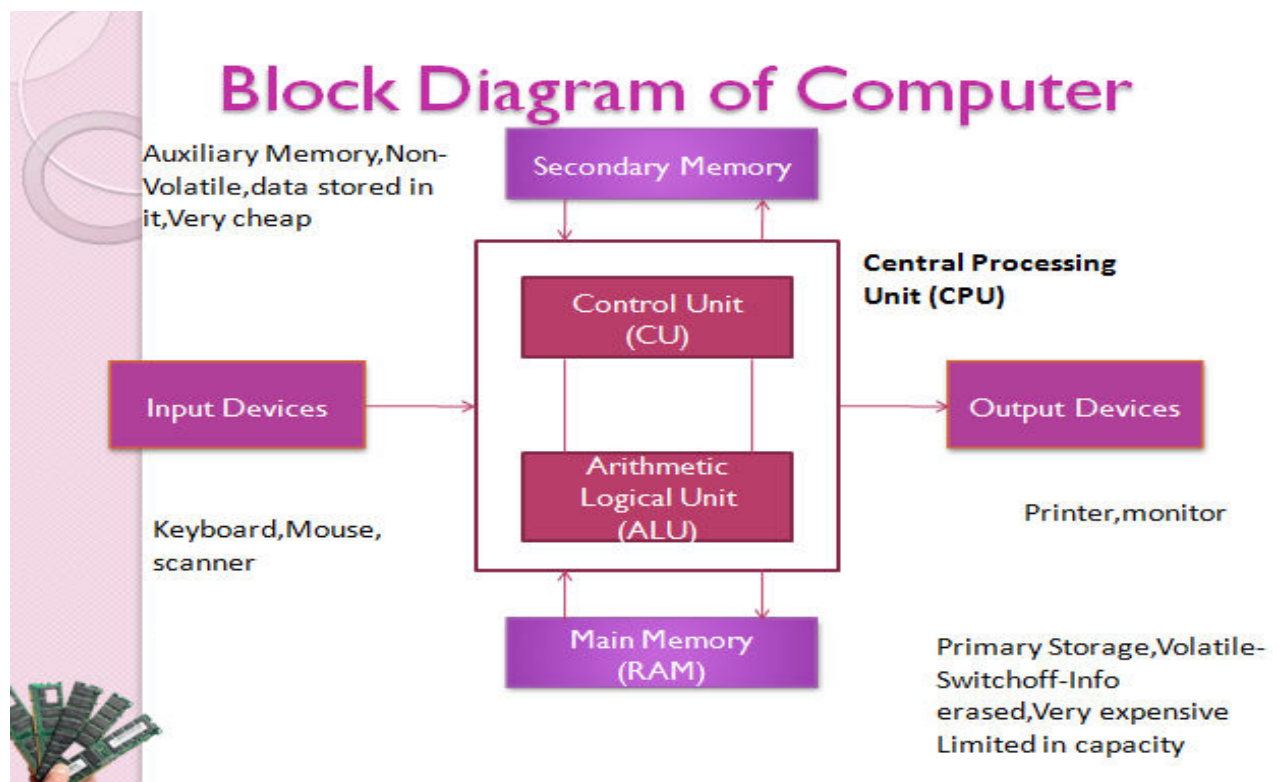
### **Block Diagram of a Computer**

The five major operations performed by a computer are

1. Accepting data or instructions (input)
  2. Storing data
  3. Processing data
  4. Displaying results (output)
  5. Controlling and coordinating all operations inside a computer.
- 
- ✓ Before execution of the program, it should be transferred from secondary storage to main memory.
  - ✓ Data need to be processed is entered through an input device and stored in main memory, where it can be accessed and manipulated by the CPU. Results are stored back in main memory.

- ✓ The information in main memory can be displayed through an output device.
- ✓ **Input:** is the process of entering data and instructions into the computer. Various input devices used are keyboard, mouse, scanner, trackball etc.
- ✓ **Storage:** is the process of storing data and instructions in the computer. Computer has two types of storage areas.

The architecture or structure of a computer is as shown below:



### 1. Primary storage :

- ✓ Is also known as main memory.
- ✓ Directly accessible to the CPU at a very fast speed.
- ✓ Used to store data, programs, intermediate and final results.
- ✓ Very expensive therefore limited in capacity.
- ✓ Volatile – as soon as computer is switched off, the information stored in it gets erased.

### 2. Secondary storage:

- ✓ Also known as secondary memory or auxiliary memory.
- ✓ It is cheaper, non-volatile and used to permanently store data and programs

**Processing:** The process of performing operations on the data as per the instructions specified by the user (program). Main aim of this is to transform data into information. Data and instructions are taken from the primary memory and are transferred to the Arithmetic and Logic Unit, which performs all sorts of calculations. When the processing completes, the final result is transferred to the main memory.

**Output:** is the process of giving the result of data processing to the outside world. Various output devices are monitor, printer etc.

**Controlling :** The function of managing, coordinating, and controlling all the components of the computer system is handled by the control unit. The control unit decides the manner in which the instructions will be executed and the operations will be performed.

---

### Computer Hardware

The essential hardware components of a computer are:

1. CPU (Central Processing Unit)
2. Memory
3. I/O (Input/Output) Devices

#### Central Processing Unit:

- Coordinates all computer operations.
- Performs arithmetic and logical operations on data.
- CPU follows the instructions contained in a computer program to determine which operations should be carried out and in what order.
- The CPU can perform arithmetic operations such as addition, subtraction, multiplication and division.
- The CPU can also compare the contents of two memory cells and make decisions based on the results of that comparison.
- The circuitry of modern CPU is housed in a single integrated circuit or chip.
- An IC that is full CPU is called a microprocessor.
- CPU contains special high speed memory locations called registers.

#### Control Unit(CU):

- ✓ It is responsible for controlling the transfer of data and instructions among other units of a computer.
- ✓ It manages and coordinates all the units of the computer.

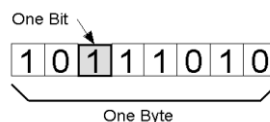
- ✓ It communicates with Input/Output devices for transfer of data or results from storage.
- ✓ It does not process or store data.

### Arithmetic Logical Unit(ALU):

- ✓ It performs arithmetic operations like addition, subtraction, multiplication, and division.
- ✓ It performs logic operations such as comparing, selecting, matching, and merging of data.
- ✓ Examples of logic operations are comparisons of values such as NOT, AND, and OR.

### Memory:

- Memory is an essential component in any computer.
- Memory cell is an individual storage location in memory.
- Address of memory cell is the relative position of a memory cell in the computers main memory.
- The data stored in the memory cell are called contents of the cell.
- The ability to store programs as well as data is called stored program concept. A program's instructions must be stored in main memory before they can be executed.
- A memory cell is actually a grouping of smaller units called bytes. A byte is the amount of storage required to store a single character. A byte is even composed of smaller units of storage called bit. Bit refers **B**inary **d**igit.
- Binary refers to a number system based on two numbers, 0 and 1. Therefore a bit is either 0 or 1.
- There are eight bits in a byte.



Random Access Memory(RAM)

- Main Memory: stores programs, data and results.

- RAM:
  - Offers temporary storage of programs and data while the programs are temporarily being executed by the computer.
  - RAM is usually volatile memory, which means that everything in RAM will be lost when the computer is switched off.
  - Accessible to the programmer.



### Secondary Memory

Read-Only Memory (ROM)

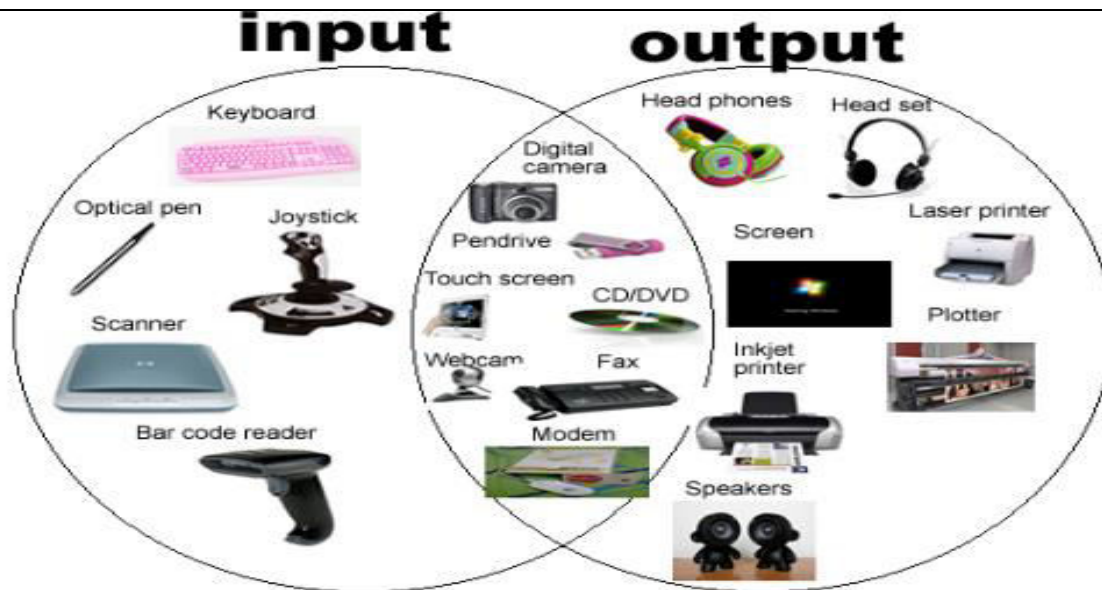
- ROM:
  - Stores information permanently in the computer.
  - Its name is read-only, because computer can read but cannot write information in ROM.
  - ROM is non-volatile. Data stored in ROM do not disappear when the computer is switched off.
- Secondary storage devices : these devices are needed for the following reasons.



- ✓ Computers need permanent / semi permanent storage, so that information can be retained during a power-loss or when the computer is turned off.
- ✓ Systems typically store more information than will fit in memory.
- ✓ Some of the most frequently used secondary storage devices : compact disk, magnetic tape, floppy disk, hard disk, zip disk, digital video disk.

### Input / output devices:

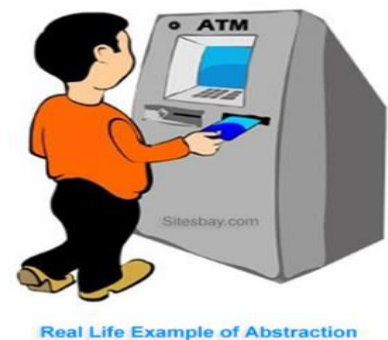
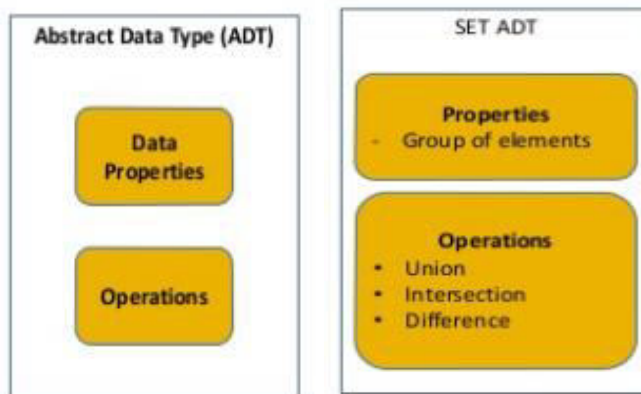
- Input / output devices are used to communicate with the computer.
- They allow us to enter data for a computation and to observe the results of that computation.



### Principle of Abstraction:

Abstraction is the process of trying to identify the most important or inherent qualities of an object or model, and ignoring or omitting the unimportant aspects.(information hiding)

It consists of two parts: data and Operations



Real Life Example of Abstraction

### Memory Hierarchy

- ✓ Memory hierarchy is the hierarchy of memory and storage devices found in a computer system.
- ✓ It ranges from the slowest but high capacity auxiliary memory to the fastest but low capacity cache memory.

### Need-

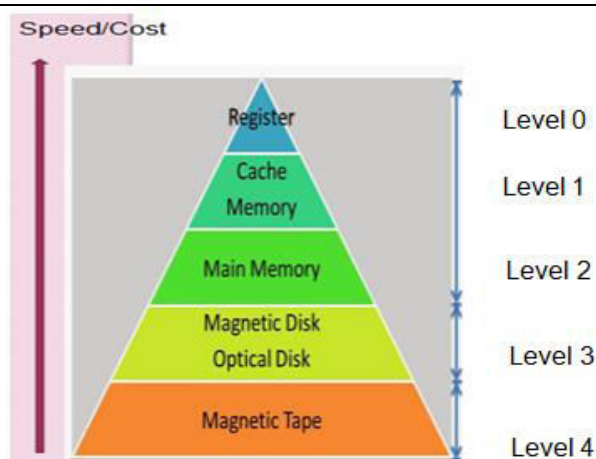
There is a trade-off among the three key characteristics of memory namely-



- ✓ Cost
- ✓ Capacity
- ✓ Access time

### Level-0:

- ✓ At level-0, registers are present which are contained inside the CPU.
- ✓ Since they are present inside the CPU, they have least access time.
- ✓ They are most expensive and therefore smallest in size (in KB).
- ✓ Registers are implemented using **Flip-Flops**.



### Level-1:

- ✓ At level-1, **Cache Memory** is present.
- ✓ It stores the segments of program that are frequently accessed by the processor.
- ✓ It is expensive and therefore smaller in size (in MB).
- ✓ Cache memory is implemented using static RAM.

### Level-2:

- ✓ At level-2, main memory is present.
- ✓ It can communicate directly with the CPU and with auxiliary memory devices through an I/O processor.
- ✓ It is less expensive than cache memory and therefore larger in size (in few GB).
- ✓ Main memory is implemented using dynamic RAM.

### Level-3:

- ✓ At level-3, secondary storage devices like **Magnetic Disk** are present.
- ✓ They are used as back up storage.
- ✓ They are cheaper than main memory and therefore much larger in size (in few TB).

### Level-4:

- ✓ At level-4, tertiary storage devices like magnetic tape are present.
- ✓ They are used to store removable files.
- ✓ They are cheapest and largest in size (1-20 TB).

### Observations-

The following observations can be made when going down in the memory hierarchy-



- ✓ Cost / bit decreases
- ✓ Frequency of access decreases
- ✓ Capacity increases
- ✓ Access time increases

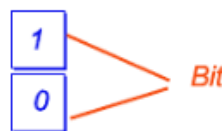
### Units of storage in Computer

Computers are electronic machines which operate using binary logic. These devices use two different values (0 and 1) to represent data. This corresponding number system is referred as Binary number system.

Differences between binary and decimal number systems.

|               | Decimal | binary |
|---------------|---------|--------|
| <b>base</b>   | 10      | 2      |
| <b>Digits</b> | 0 to 9  | 0, 1   |

A **bit** is the smallest unit of information that can be stored or manipulated on a computer; it consists of either zero or one. We can also call a *bit* a *binary* digit. A single bit can store two different values (either 0 or 1)

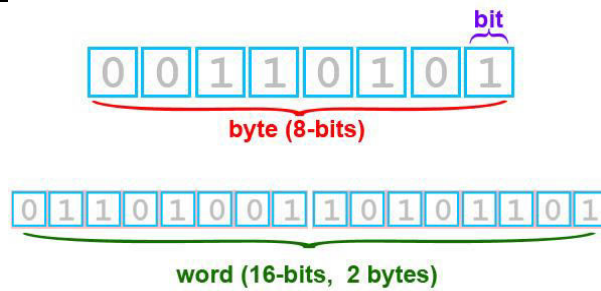


A **Nibble** is a group of four bits.

**Byte** : is a group of eight bits.

- A byte is a simply a fixed-length sequence of bits. Modern computers organize data into bytes to increase the data processing efficiency of network equipment, disks and memory. Computers by convention set one byte to equal eight (8) bits.
- A byte can store  $2^8$  or 256 different values.

A **word** is a group of 16 bits.



Besides bytes, data is also specified using the units shown in the table

| Unit     | Abbreviation | Equals to  | Bytes          |
|----------|--------------|------------|----------------|
| Byte     | Byte         | 8 bits     | $2^0$ bytes    |
| Kilobyte | KB           | 1024 bytes | $2^{10}$ bytes |
| Megabyte | MB           | 1024 KB    | $2^{20}$ bytes |
| Gigabyte | GB           | 1024 MB    | $2^{30}$ bytes |
| Terabyte | TB           | 1024 GB    | $2^{40}$ bytes |

## Circuits and Logic:

The basic digital electronic circuit that has one or more inputs and single output is known as **Logic gate**.

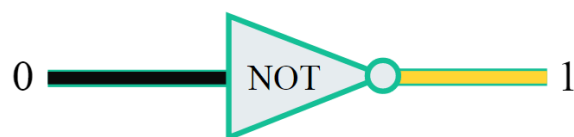
Logic gates are divided into three types. They are

1. Basic Gates: NOT, AND, OR
2. Universal Gates : NAND, NOR
3. Arithmetic Gates: EXOR, EXNOR

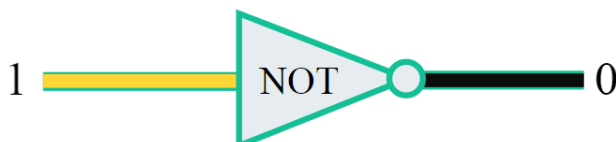
### NOT Gate:

The simplest gate is the NOT gate, also known as an inverter. It accepts a single input and outputs the opposite value.

If the input is 0, the output is 1:



If the input is 1, the output is 0:



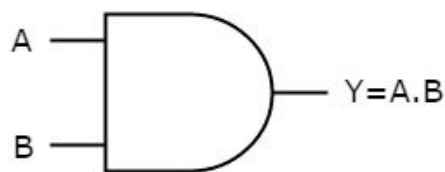
### Truth table for NOT

| Input | Output |
|-------|--------|
| A     | Y      |
| 0     | 1      |
| 1     | 0      |

### AND Gate:

If both inputs are '1', then only the output, Y is '1'. For remaining combinations of inputs, the output, Y is '0'. It is optional to represent the **Logical AND** with the symbol '·'.

The following figure shows the **symbol** of an AND gate, which is having two inputs A, B and one output, Y.



The following table shows the **truth table** of 2-input AND gate.

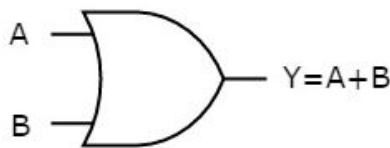
| A | B | Y=A.B |
|---|---|-------|
| 0 | 0 | 0     |
| 0 | 1 | 0     |
| 1 | 0 | 0     |
| 1 | 1 | 1     |

### OR Gate:

This **logical OR** is represented with the symbol '+'.

If both inputs are '0', then only the output, Y is '0'. For remaining combinations of inputs, the output, Y is '1'.

The following figure shows the **symbol** of an OR gate, which is having two inputs A, B and one output, Y.



The following table shows the **truth table** of 2-input OR gate.

| A | B | Y=A+B |
|---|---|-------|
| 0 | 0 | 0     |
| 0 | 1 | 1     |
| 1 | 0 | 1     |
| 1 | 1 | 1     |

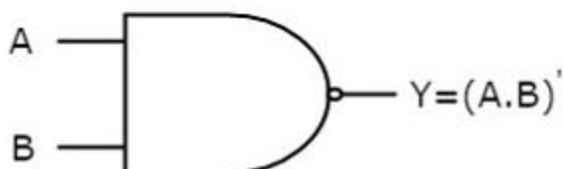
### Universal gates

NAND & NOR gates are called as **universal gates**. Because we can implement any Boolean function, which is in sum of products form by using NAND gates alone. Similarly, we can implement any Boolean function, which is in product of sums form by using NOR gates alone.

### NAND gate

NAND gate is a digital circuit that has two or more inputs and produces an output, which is the **inversion of logical AND** of all those inputs.

The following image shows the **symbol** of NAND gate, which is having two inputs A, B and one output, Y.



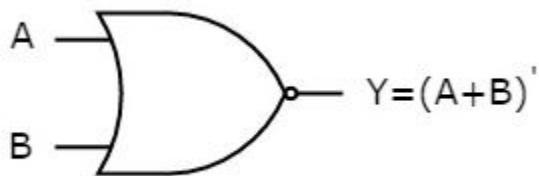
The following table shows the **truth table** of 2-input NAND gate.

| A | B | $Y=A.B$ | $(A.B)^I$ |
|---|---|---------|-----------|
| 0 | 0 | 0       | 1         |
| 0 | 1 | 0       | 1         |
| 1 | 0 | 0       | 1         |
| 1 | 1 | 1       | 0         |

### NOR Gate:

NOR gate is a digital circuit that has two or more inputs and produces an output, which is the **inversion of logical OR** of all those inputs.

The following figure shows the **symbol** of NOR gate, which is having two inputs A, B and one output, Y.



The following table shows the **truth table** of 2-input NOR gate

| A | B | $Y=A+B$ | $Y^I$ |
|---|---|---------|-------|
| 0 | 0 | 0       | 1     |
| 0 | 1 | 1       | 0     |
| 1 | 0 | 1       | 0     |
| 1 | 1 | 1       | 0     |

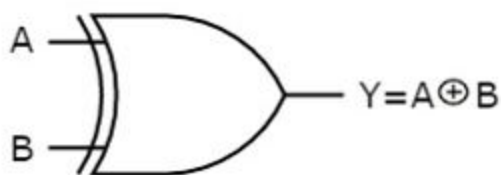
### Arithmetic Gates: EXOR, EXNOR

Ex-OR & Ex-NOR gates are called as Arithmetic gates. Because, these two gates are special cases of OR & NOR gates.

#### Ex-OR gate

The full form of Ex-OR gate is **Exclusive-OR** gate. If both the inputs are 1 or both the inputs are 0, then obtained output is 0. And for the remaining cases the output is taken as 1.

Below figure shows the **symbol** of Ex-OR gate, which is having two inputs A, B and one output, Y.



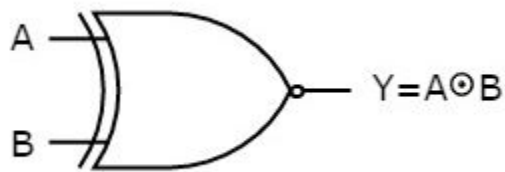
The following table shows the **truth table** of 2-input Ex-OR gate.

| A | B | $Y = A \oplus B$ |
|---|---|------------------|
| 0 | 0 | 0                |
| 0 | 1 | 1                |
| 1 | 0 | 1                |
| 1 | 1 | 0                |

#### Ex-NOR:

The full form of Ex-NOR gate is **Exclusive-NOR** gate. It is an inversion of Ex-OR

The following figure shows the **symbol** of Ex-NOR gate, which is having two inputs A, B and one output, Y.



The following table shows the **truth table** of 2-input Ex-NOR gate.

| A | B | $Y = A \odot B$ |
|---|---|-----------------|
| 0 | 0 | 1               |
| 0 | 1 | 0               |
| 1 | 0 | 0               |
| 1 | 1 | 1               |

### Advantages of Logic Gates

- Logic gates are quick yet use low energy.
- Logic gates don't get overworked.
- Logic gates can lessen the prescribed number of I/O ports needed by a microcontroller.
- Logic gates can bring about straightforward data encryption and decryption.

### Applications of Logic gates

- NAND Gates are used in Burglar alarms and buzzers.
- They are basically used in circuits involving computation and processing.
- They are also used in push button switches. E.g. Door Bell.
- They are used in the functioning of street lights.

### Operating System:

The collection of computer programs that control the interaction of the user and computer hardware is called the operating system.

Its responsibilities are

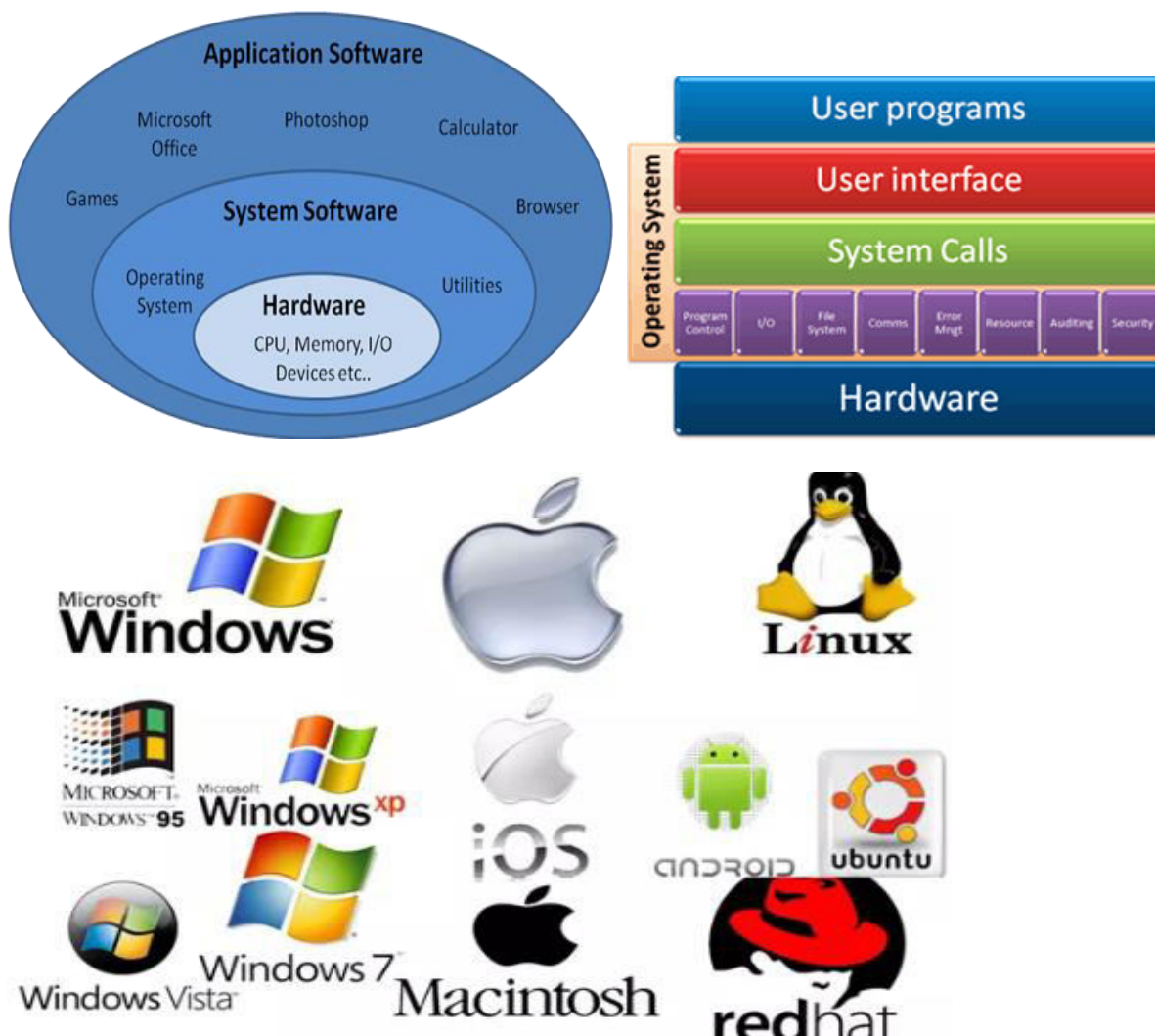
- ✓ Communicating with the user- receiving commands and carrying them out.



- ✓ Managing allocation of memory, processor time, other resources for various tasks.
- ✓ Collecting input from the keyboard, mouse and other input devices, and providing this data to the currently running program.
- ✓ Conveying program output to the screen, printer or other output device.
- ✓ Accessing data from secondary storage.
- ✓ Writing data to secondary storage.

Widely used OS are – windows, linux.

The relationship between hardware, system software and application software is as shown in the below diagram:



## Representation of Data in memory (integer (including negative), floating points etc. to text, images, audio and video):

Computer uses a *fixed number of bits* to represent a piece of data, which could be a number, a character, or others. For example, a 3-bit memory location can hold one of these eight binary patterns: 000, 001, 010, 011, 100, 101, 110, or 111.

Refer to Units of storage in Computer page no:9

### Integer Representation:

The commonly-used bit-lengths for integers are 8-bit, 16-bit, 32-bit or 64-bit. Besides bit-lengths, there are two representation schemes for integers:

**Unsigned Integers:** We can represent zero and positive integers.

#### ***n*-bit Unsigned Integers**

An *n*-bit pattern can represent  $2^n$  distinct integers. An *n*-bit unsigned integer can represent integers from 0 to  $(2^n) - 1$

| n  | Minimum | Maximum   |
|----|---------|---|
| 8  | 0       | $(2^8) - 1$ (=255)  |
| 16 | 0       | $(2^{16}) - 1$ (=65,535)                                  |
| 32 | 0       | $(2^{32}) - 1$ (=4,294,967,295) (9+ digits)               |
| 64 | 0       | $(2^{64}) - 1$ (=18,446,744,073,709,551,615) (19+ digits) |

**Signed Integers:** We can represent zero, positive and negative integers.

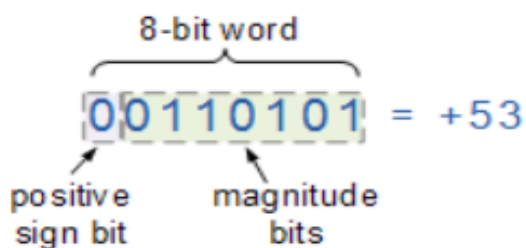
Three representation schemes had been proposed for signed integers:

- ✓ Sign-Magnitude representation
- ✓ 1's Complement representation
- ✓ 2's Complement representation

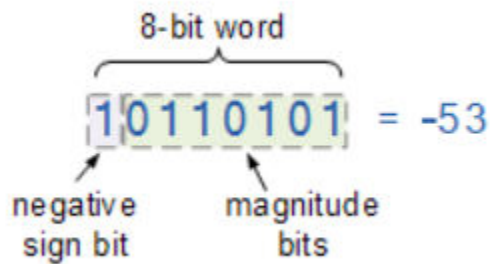
### **Sign-Magnitude representation:**

Signed Binary Numbers use the MSB(most significant bit) as a sign bit to display a range of either positive numbers or negative numbers

**Positive Signed Binary Numbers:**

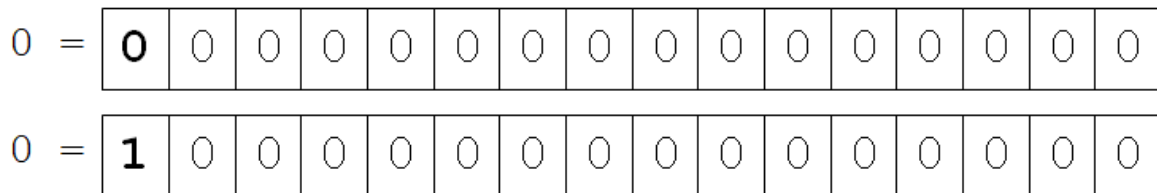


## Negative Signed Binary Numbers



### Disadvantages:

An unfortunate feature of Sign-Value representation is that there are two ways of representing the value zero: all bits set to zero, and all bits except the sign bit set to zero.



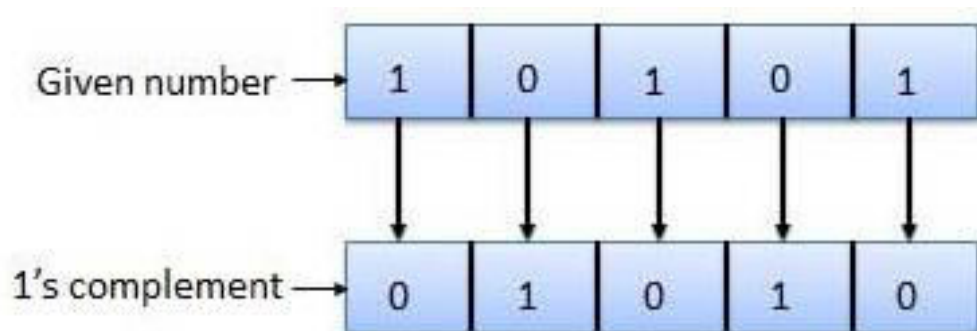
This makes the math a bit confusing.

### One's Complement of a Signed Binary Number

**One's Complement** or **1's Complement** as it is also termed, is another method which we can use to represent negative binary numbers in a signed binary number system. In one's complement, positive numbers (also known as non-complements) remain unchanged as before with the sign-magnitude numbers.

Negative numbers however, are represented by taking the one's complement (inversion, negation) of the unsigned positive number.

### 1's Complement of 21



### Disadvantage:

An unfortunate feature of One's Complement representation is that there are two ways of representing the value zero: all bits set to zero, and all bits set to one.

$$0 = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline \end{array}$$

$$0 = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ \hline \end{array}$$

### Two's Complement of a Signed Binary Number

**Two's Complement** or **2's Complement** as it is also termed, is another method like the previous sign-magnitude and one's complement form, which we can use to represent negative binary numbers in a signed binary number system.

2's complement = 1's complement + 1

#### Example #1

$$\begin{array}{rcl} 5 & = & 00000101 \\ & & \downarrow \downarrow \downarrow \downarrow \downarrow \downarrow \downarrow \\ & & 11111010 \\ & & \quad \quad \quad +1 \\ \hline -5 & = & 11111011 \end{array} \quad \begin{array}{l} \text{Complement Digits} \\ \text{Add 1} \end{array}$$

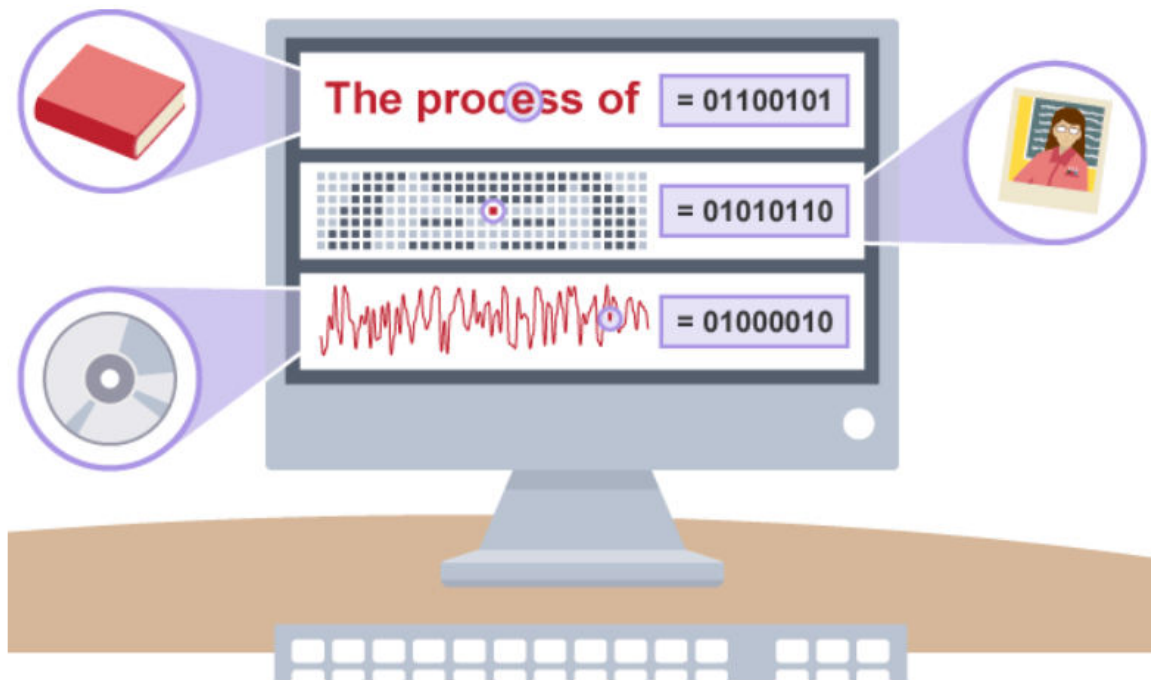
### Advantage:

In 2's complement, there is only one representation for zero – 00000000 (+0) because if we add 1 to 11111111 (-1), we get 00000000 (+0) which is the same as positive zero. This is the reason why 2's complement is generally used.

### Representing text, images and sound:

#### Representing data

All data inside a computer is transmitted as a series of electrical signals that are either **on** or **off**. Therefore, in order for a computer to be able to process any kind of data, including text, images and sound, they must be converted into binary form. If the data is not converted into binary – a series of 1s and 0s – the computer will simply not understand it or be able to process it.



## Representing text

When any key on a keyboard is pressed, it needs to be converted into a binary number so that it can be processed by the computer and the typed character can appear on the screen.



A code where each number represents a character can be used to convert text into binary. One code we can use for this is called ASCII. The ASCII code takes each character on the keyboard and assigns it a binary number. For example:

- the letter 'a' has the binary number 0110 0001 (this is the denary number 97)
- the letter 'b' has the binary number 0110 0010 (this is the denary number 98)
- the letter 'c' has the binary number 0110 0011 (this is the denary number 99)

Text characters start at denary number 0 in the ASCII code, but this covers special characters including punctuation, the return key and control characters as well as the number keys, capital letters and lower case letters.

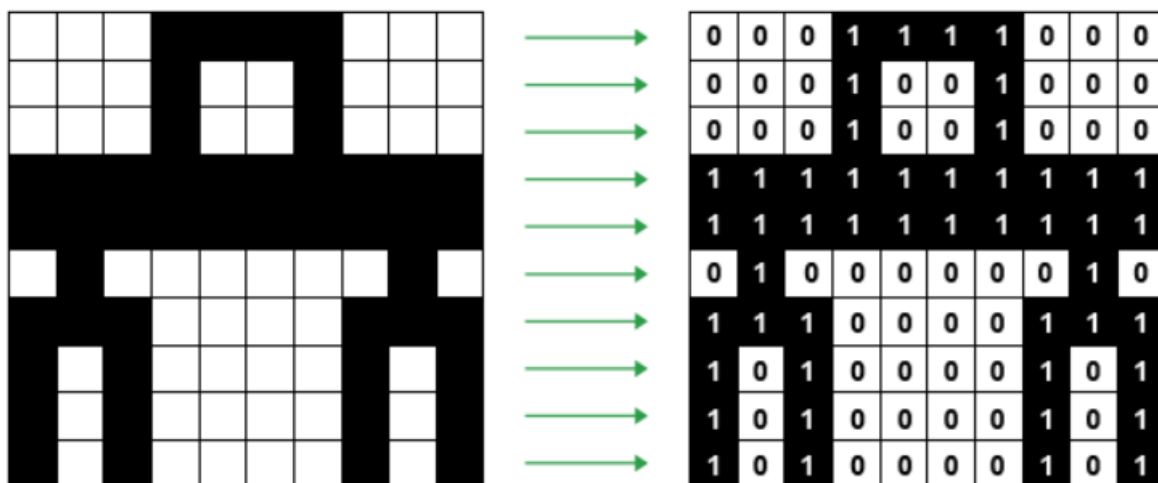
ASCII code can only store 128 characters, which is enough for most words in English but not enough for other languages. If you want to use accents in European languages or larger alphabets such as Cyrillic (the Russian alphabet) and Chinese Mandarin then more characters are needed. Therefore another code, called Unicode, was created. This meant that computers could be used by people using different languages.

## Representing images

Images also need to be converted into binary in order for a computer to process them so that they can be seen on our screen. Digital images are made up of pixels. Each pixel in an image is made up of binary numbers.

If we say that 1 is black (or on) and 0 is white (or off), then a simple black and white picture can be created using binary.

To create the picture, a grid can be set out and the squares coloured (1 – black and 0 – white). But before the grid can be created, the size of the grid needs to be known. This data is called metadata and computers need metadata to know the size of an image. If the metadata for the image to be created is 10x10, this means the picture will be 10 pixels across and 10 pixels down. This example shows an image created in this way:





---

## Adding colour

The system described so far is fine for black and white images, but most images need to use colours as well. Instead of using just 0 and 1, using four possible numbers will allow an image to use four colours. In binary this can be represented using two bits per pixel:

- 00 – white
- 01 – blue
- 10 – green
- 11 – red

While this is still not a very large range of colours, adding another binary digit will double the number of colours that are available:

- 1 bit per pixel (0 or 1): two possible colours
- 2 bits per pixel (00 to 11): four possible colours
- 3 bits per pixel (000 to 111): eight possible colours
- 4 bits per pixel (0000 – 1111): 16 possible colours
- ...
- 16 bits per pixel (0000 0000 0000 0000 – 1111 1111 1111 1111): over 65 000 possible colours

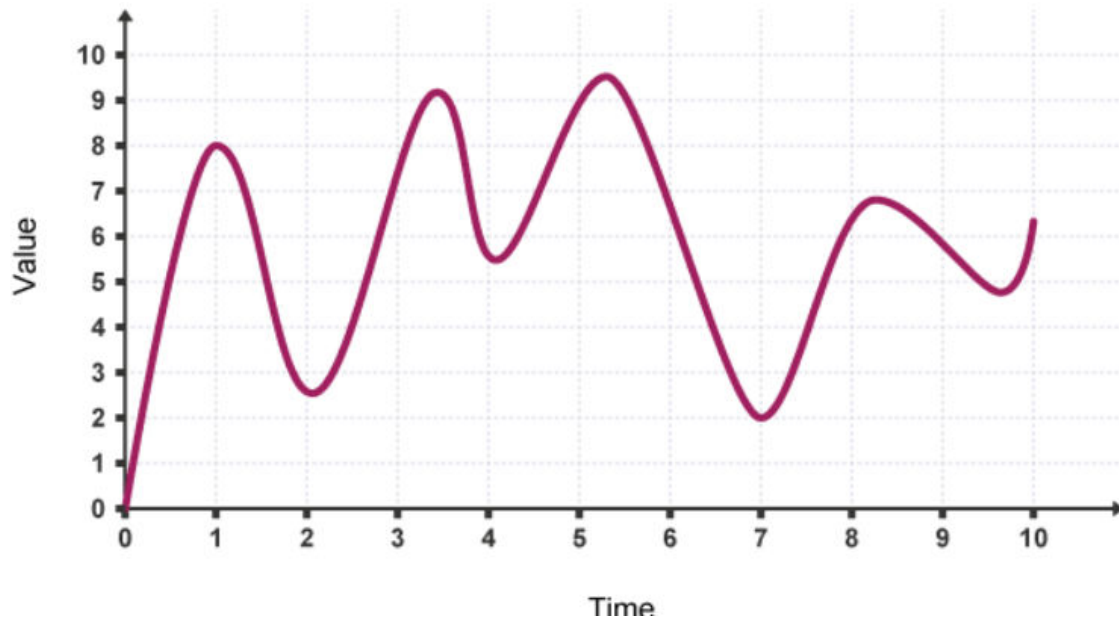
The number of bits used to store each pixel is called the colour depth. Images with more colours need more pixels to store each available colour. This means that images that use lots of colours are stored in larger files.

## Representing sound

Sound needs to be converted into binary for computers to be able to process it. To do this, sound is captured - usually by a microphone - and then converted into a digital signal.

An analogue to digital converter will sample a sound wave at regular time intervals. For example, a sound wave like this can be sampled at each time sample point:



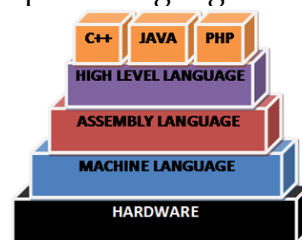


The samples can then be converted to binary. They will be recorded to the nearest whole number.

| Time sample | 1    | 2    | 3    | 4    | 5    | 6    | 7    | 8    | 9    | 10   |
|-------------|------|------|------|------|------|------|------|------|------|------|
| Denary      | 8    | 3    | 7    | 6    | 9    | 7    | 2    | 6    | 6    | 6    |
| Binary      | 1000 | 0011 | 0111 | 0110 | 1001 | 0111 | 0010 | 0100 | 0110 | 0110 |

### Language Hierarchy:

- There are generally three types of computer languages. They are:
  1. Machine Language,
  2. Assembly Language and
  3. High-level languages.



### Machine Language: (1940's)

- It is the lowest-level programming language.
- The fundamental/native language of the computer's processor, also called Low Level Language i.e., each computer has its own machine language.
- All programs are converted into machine language before they can be executed.

- It consists of combination of 0's and 1's that represent high and low electrical voltage.
- Machine language is a sequence of instructions written in the form of binary number to which computer responds directly.
- It is considered as first generation language.
- A machine language instruction generally has three parts as shown below.

| Operation Code | Mode | Operand |
|----------------|------|---------|
|----------------|------|---------|

Part1 : command or operation code, it conveys the computer what function has to be performed by the instruction.

Part2 : it either specifies the operand contains data on which the operation has to be performed or it specifies that the operand contains a location, the contents of which have to be subject to the operation.

Table 9.1 Code in machine language to add two integers

| Hexadecimal          | Code in machine language |      |      |      |
|----------------------|--------------------------|------|------|------|
| (1FEF) <sub>16</sub> | 0001                     | 1111 | 1110 | 1111 |
| (240F) <sub>16</sub> | 0010                     | 0100 | 0000 | 1111 |
| (1FEF) <sub>16</sub> | 0001                     | 1111 | 1110 | 1111 |
| (241F) <sub>16</sub> | 0010                     | 0100 | 0001 | 1111 |
| (1040) <sub>16</sub> | 0001                     | 0000 | 0100 | 0000 |
| (1141) <sub>16</sub> | 0001                     | 0001 | 0100 | 0001 |
| (3201) <sub>16</sub> | 0011                     | 0010 | 0000 | 0001 |
| (2422) <sub>16</sub> | 0010                     | 0100 | 0010 | 0010 |
| (1F42) <sub>16</sub> | 0001                     | 1111 | 0100 | 0010 |
| (2FFF) <sub>16</sub> | 0010                     | 1111 | 1111 | 1111 |
| (0000) <sub>16</sub> | 0000                     | 0000 | 0000 | 0000 |

**Advantages:** Though it is not a human friendly language, it offers following advantages.

1. **Translation Free:** Translation is not required as the CPU directly understands machine instructions.
2. As conversion is not needed, programs developed using these languages takes less execution time.
3. More control on hardware.

**Disadvantages:**

1. Difficult to use: It is difficult to understand and develop a program using machine language as it is very complex to write using binary numbers.
2. Machine dependent: Computer architectures differ from one another. The programmer has to remember machine characteristics while preparing a program.
3. Error Prone: since the programmer has to remember all the opcodes and the memory locations, machine language is bound to be error prone.
4. Difficult to debug and modify.

**Assembly Language (1950's):**

- The next evolution in programming came with the idea of replacing binary code for instruction and addresses with symbols or mnemonics.
- Because they used symbols, these languages were first known as symbolic languages. The set of these mnemonic languages were later referred to as assembly languages.
- This language is also referred as low-level language.
- Every computer has its own assembly language that is dependent upon the internal architecture of the processor.
- An assembler is a translator that takes input in the form of the assembly language program and produces machine language code as its input.

**Table 9.2** Code in assembly language to add two integers

| Code in assembly language |         |          | Description                                     |
|---------------------------|---------|----------|---|
| LOAD                      | RF      | Keyboard | Load from keyboard controller to register F     |
| STORE                     | Number1 | RF       | Store register F into Number1                   |
| LOAD                      | RF      | Keyboard | Load from keyboard controller to register F     |
| STORE                     | Number2 | RF       | Store register F into Number2                   |
| LOAD                      | R0      | Number1  | Load Number1 into register 0                    |
| LOAD                      | R1      | Number2  | Load Number2 into register 1                    |
| ADDI                      | R2      | R0 R1    | Add registers 0 and 1 with result in register 2 |
| STORE                     | Result  | R2       | Store register 2 into Result                    |
| LOAD                      | RF      | Result   | Load Result into register F                     |
| STORE                     | Monitor | RF       | Store register F into monitor controller        |
| HALT                      |         |          | Stop  |

### Assembler

- An assembler is a translator that takes input in the form of the assembly language program and produces machine language code as its input.



- An assembler is a system software which converts an assembly language program to its equivalent object code.
- The input to the assembler is a source code written in assembly language and the output is an object code.
- Basic assembler functions are
  - Translating mnemonic language to its equivalent object code.
  - Assigning machine address to symbolic labels.

### **Advantages:**

- It is easy to write and maintain programs in assembly language than in machine languages

- Improved readability than machine language.
- Less Error prone when compared with machine language.
- Assembly programs can run much faster and use less memory and other resources.
- More control on hardware.

#### Disadvantages:

- Machine dependent: It is specific to particular machine architecture.
- Hard to learn
- Assembly language program is less efficient than machine language program.
- Programming is difficult and time consuming.

#### High-Level Languages:

- Although assembly languages greatly improved programming efficiency, they still required programmers to concentrate on the hardware they were using. Working with symbolic languages was also very tedious, because each machine instruction had to be individually coded. The desire to improve programmer efficiency and to change the focus from the computer to the problem being solved led to the development of high-level languages.
- These languages have instructions that are similar to human languages and have a set grammar that makes it easy for a programmer to write programs and identify correct errors in them.



#### Advantages:

- Readability : programs written in these languages are more readable than those written in assembly and machine languages.
- Portability: High level programming languages can be run on different machines with little or no change.
- Easy debugging: Errors can be easily detected and removed.
- Development of software is easy.

#### Disadvantages:

- Poor control on hardware
- Less efficient

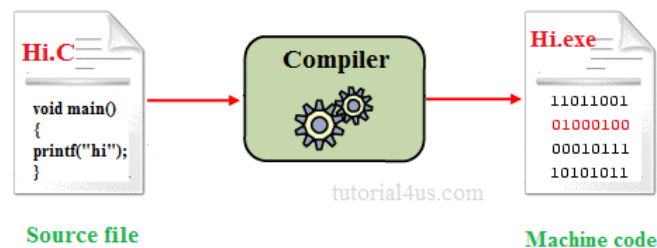
#### Compiler vs interpreter.

##### Translation

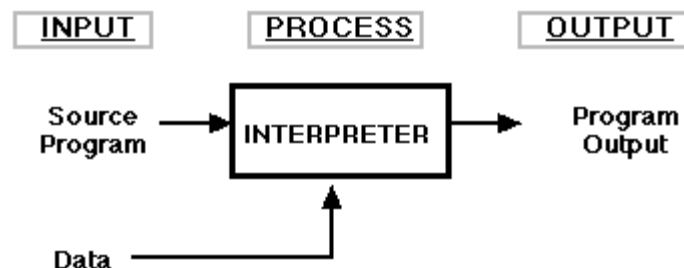
Programs today are normally written in one of the high-level languages. To run the program on a computer, the program needs to be translated into the machine language of the computer on which it will run.

- The program in a high-level language is called the source program.
  - The translated program in machine language is called the object program.
- Two methods are used for translation: compilation and interpretation.

**Compiler:** A compiler is a translator which converts the program written in high-level language into assembly code or into another form of intermediate code or directly into machine code. A compiler converts the whole source code at once into object code and then executes it. So, compiler is faster than a interpreter.



**Interpreter:** An interpreter is a translator which converts the source program into object or machine code. The interpreter converts the source code line-by-line and executes it immediately, which results in less performance. Thus, an interpreter is slower than a compiler.



### Difference between compiler and interpreter

| <u>Comparison</u> | <u>Compiler</u>  | <u>Interpreter</u>  |
|-------------------|--|---|
| <b>Input</b>      | It takes an entire program at a time.                          | It takes a single line of code  |
| <b>Speed</b>      | Comparatively fast   | Slower  |
| <b>Memory</b>     | Memory requirement is more due to the creation of object code. | It requires less memory as it does not create intermediate object code. |
| <b>Errors</b>     | Displays all errors after compilation ,all at the same time    | Displays error of each line one by one                                  |

|                        |                        |                            |
|------------------------|------------------------|----------------------------|
| <b>Error Detection</b> | Difficult              | Easy                       |
| <b>Examples</b>        | C,C++,Java,Scala,etc., | Python,Perl,PHP,Ruby,etc., |

## Command Line Interface

The command line is a text interface for your computer. It's a program that takes in commands, which it passes on to the computer's operating system to run.

From the command line, you can navigate through files and folders on your computer, just as you would with Windows Explorer on Windows or Finder on Mac OS. The difference is that the command line is fully text-based.

Today, with graphical user interfaces (GUI), most users never use command-line interfaces (CLI).

However, CLI is still used by software developers and system administrators to configure computers, install software, and access features that are not available in the graphical interface.

## Basic Linux CLI Commands

| Command            | Description  |
|--------------------|--|
| ls                 | List the directory (folder) system.                |
| cd <i>pathname</i> | Change directory (folder) in the file system.      |
| cd ..              | Move one level up (one folder) in the file system. |
| cp                 | Copy a file to another folder.                     |
| mv                 | Move a file to another folder.                     |
| mkdir              | Creates a new directory (folder).                  |
| rmdir              | Remove a directory (folder).                       |
| clear              | Clears the CLI window.                             |
| exit               | Closes the CLI window.                             |
| man <i>command</i> | Shows the manual for a given command.              |

## Basic Windows CLI Commands

| Command                           | Description  |
|-----------------------------------|--|
| <code>dir</code>                  | List the directory (folder) system.                |
| <code>cd <i>pathname</i></code>   | Change directory (folder) in the file system.      |
| <code>cd \</code>                 | Move to the root folder of the file system.        |
| <code>cd ..</code>                | Move one level up (one folder) in the file system. |
| <code>copy</code>                 | Copy a file to another folder.                     |
| <code>move</code>                 | Move a file to another folder.                     |
| <code>type <i>filename</i></code> | Type a file.                                       |
| <code>mkdir or md</code>          | Creates a new directory (folder).                  |
| <code>rmdir or rd</code>          | Removes a directory (folder).                      |
| <code>cls</code>                  | Clears the CLI window.                             |
| <code>exit</code>                 | Closes the CLI window.                             |
| <code>help <i>command</i></code>  | Shows the manual for a given command.              |

## Linux Commands

1. **pwd** — When you first open the terminal, you are in the home directory of your user. To know which directory you are in, you can use the “pwd” command. It gives us the absolute path, which means the path that starts from the root. The root is the base of the Linux file system. It is denoted by a forward slash( / ). The user directory is usually something like “/home/username”.

```
nayso@Alok-Aspire:~$ pwd
/home/nayso
```

2. **ls** — Use the “ls” command to know what files are in the directory you are in. You can see all the hidden files by using the command “ls -a”.



```
nayso@Alok-Aspire:~$ ls
Desktop      itsuserguide.desktop  reset-settings  VCD_Copy
Documents    Music                  School_Resources  Videos
Downloads    Pictures               Students_Works_10
examples.desktop  Public                 Templates
GplatesProject  Qgis Projects         TuxPaint-Pictures
```

3. **cd** — Use the "cd" command to go to a directory. For example, if you are in the home folder, and you want to go to the downloads folder, then you can type in "cd Downloads". Remember, this command is case sensitive, and you have to type in the name of the folder exactly as it is.

```
nayso@Alok-Aspire:~$ cd Downloads
nayso@Alok-Aspire:~/Downloads$ cd
```

4. **mkdir&rmdir** — Use the mkdir command when you need to create a folder or a directory. For example, if you want to make a directory called "DIY", then you can type "mkdir DIY". Remember, as told before, if you want to create a directory named "DIY Hacking", then you can type "mkdir DIY\Hacking". Use rmdir to delete a directory. But rmdir can only be used to delete an empty directory. To delete a directory containing files, use rm.

```
nayso@Alok-Aspire:~/Desktop$ ls
nayso@Alok-Aspire:~/Desktop$ mkdir DIY
nayso@Alok-Aspire:~/Desktop$ ls
DIY
nayso@Alok-Aspire:~/Desktop$ rmdir DIY
nayso@Alok-Aspire:~/Desktop$ ls
nayso@Alok-Aspire:~/Desktop$
```

5. **rm** - Use the rm command to delete files and directories. Use "rm -r" to delete just the directory. It deletes both the folder and the files it contains when using only the rm command.

```
nayso@Alok-Aspire:~/Desktop$ ls
newer.py  New Folder
nayso@Alok-Aspire:~/Desktop$ rm newer.py
nayso@Alok-Aspire:~/Desktop$ ls
New Folder
nayso@Alok-Aspire:~/Desktop$ rm -r New\ Folder
nayso@Alok-Aspire:~/Desktop$ ls
nayso@Alok-Aspire:~/Desktop$
```

6. **man& --help** — To know more about a command and how to use it, use the man command. It shows the manual pages of the command. For example, "man cd" shows the manual pages of the cd command. Typing in the

command name and the argument helps it show which ways the command can be used (e.g., `cd -help`).

```
TOUCH(1)                                User Commands                                TOUCH(1)

NAME
    touch - change file timestamps

SYNOPSIS
    touch [OPTION]... FILE...

DESCRIPTION
    Update the access and modification times of each FILE to the current
    time.

    A FILE argument that does not exist is created empty, unless -c or -h
    is supplied.

    A FILE argument string of - is handled specially and causes touch to
    change the times of the file associated with standard output.

    Mandatory arguments to long options are mandatory for short options
    too.

    -a      change only the access time

Manual page touch(1) line 1 (press h for help or q to quit)
```

7. **cp** — Use the `cp` command to copy files through the command line. It takes two arguments: The first is the location of the file to be copied, the second is where to copy.

```
nayso@Alok-Aspire:~/Desktop$ ls /home/nayso/Music/
nayso@Alok-Aspire:~/Desktop$ cp new.txt /home/nayso/Music/
nayso@Alok-Aspire:~/Desktop$ ls /home/nayso/Music/
new.txt
```

8. **mv** — Use the `mv` command to move files through the command line. We can also use the `mv` command to rename a file. For example, if we want to rename the file “text” to “new”, we can use “`mv text new`”. It takes the two arguments, just like the `cp` command.

```
nayso@Alok-Aspire:~/Desktop$ ls
new.txt
nayso@Alok-Aspire:~/Desktop$ mv new.txt newer.txt
nayso@Alok-Aspire:~/Desktop$ ls
newer.txt
```

## UNIT - II: Computational Thinking and Introduction to Python







**Simple logic building through flowcharting:** Flowchart symbols, conditional and repetition blocks. Computational Thinking, Algorithm, Pseudo code, Time/Space complexity. Only Big O notation. Basic structure of a Python program, **Elements of Python programming Language:** token, literals, identifiers, keywords, expression, type conversions, Numbers, Variables, Input/output statements, basic data types. Operators and their types and precedence, expressions. Control structures in Python - conditionals and loops

### Flowchart

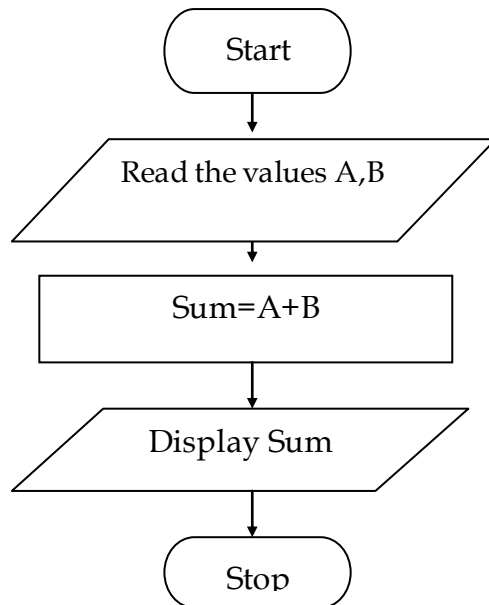
**Flowchart** is a diagrammatic representation of sequence of logical steps of a program. Flowcharts use simple geometric shapes to depict processes and arrows to show relationships and process/data flow.

### Flowchart Symbols

The common symbols used in drawing flowcharts are

| Symbol  | Symbol Name       | Purpose  |
|---|-------------------|--|
|  | Start/Stop        | Used at the beginning and end of the algorithm to show start and end of the program. |
|  | Process           | Indicates processes like mathematical operations.                                    |
|  | Input/ Output     | Used for denoting program inputs and outputs.  |
|  | Decision          | Stands for decision statements in a program, where answer is usually Yes or No.      |
|  | Arrow             | Shows relationships between different shapes.  |
|  | On-page Connector | Connects two or more parts of a flowchart, which are on the same page.               |

### Flow Chart for adding of two numbers



### Advantages of flowchart:

1. The Flowchart is an excellent way of communicating the logic of a program.
2. It is easy and efficient to analyze problem using flowchart.
3. During program development cycle, the flowchart plays the role of a guide or a blueprint. Which makes program development process easier.
4. After successful development of a program, it needs continuous timely maintenance during the course of its operation. The flowchart makes program or system maintenance easier.
5. It helps the programmer to write the program code.
6. It is easy to convert the flowchart into any programming language code as it does not use any specific programming language concept.

### Disadvantage of flowchart

1. The flowchart can be complex when the logic of a program is quite complicated.
2. Drawing flowchart is a time-consuming task.
3. Difficult to alter the flowchart. Sometimes, the designer needs to redraw the complete flowchart to change the logic of the flowchart or to alter the flowchart.
4. In the case of a complex flowchart, other programmers might have a difficult time understanding the logic and process of the flowchart.

---

### Algorithm Complexity

Suppose X is treated as an algorithm and N is treated as the size of input data, the time and space implemented by the Algorithm X are the two main factors which determine the efficiency of X.

**Time Factor** – The time is calculated or measured by counting the number of key operations such as comparisons in sorting algorithm.

**Space Factor** – The space is calculated or measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm  $f(N)$  provides the running time and / or storage space needed by the algorithm with respect of N as the size of input data.

### Space Complexity

Space complexity of an algorithm represents the amount of memory space needed the algorithm in its life cycle.

Space needed by an algorithm is equal to the sum of the following two components

A fixed part that is a space required to store certain data and variables (i.e. simple variables and constants, program size etc.), that are not dependent of the size of the problem.

A variable part is a space required by variables, whose size is totally dependent on the size of the problem. For example, recursion stack space, dynamic memory allocation etc.

Space complexity  $S(p)$  of any algorithm  $p$  is  $S(p) = A + S_p(I)$  Where  $A$  is treated as the fixed part and  $S(I)$  is treated as the variable part of the algorithm which depends on instance characteristic  $I$ . Following is a simple example that tries to explain the concept

### Algorithm

```
SUM(P, Q)
Step 1 - START
Step 2 -  $R \leftarrow P + Q + 10$ 
Step 3 - Stop
```

Here we have three variables  $P$ ,  $Q$  and  $R$  and one constant. Hence  $S(p) = 1+3$ . Now space is dependent on data types of given constant types and variables and it will be multiplied accordingly.

### Time Complexity

Time Complexity of an algorithm is the representation of the amount of time required by the algorithm to execute to completion. Time requirements can be denoted or defined as a numerical function  $t(N)$ , where  $t(N)$  can be measured as the number of steps, provided each step takes constant time.

For example, in case of addition of two  $n$ -bit integers,  $N$  steps are taken. Consequently, the total computational time is  $t(N) = c \cdot n$ , where  $c$  is the time consumed for addition of two bits. Here, we observe that  $t(N)$  grows linearly as input size increases.

### Asymptotic Notations

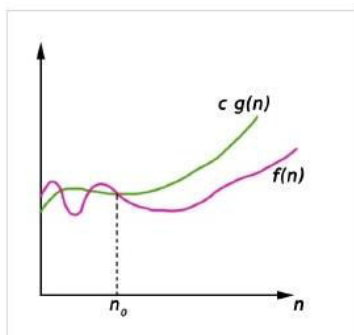
Asymptotic notations are used to represent the complexities of algorithms for asymptotic analysis. These notations are mathematical tools to represent the complexities. There are three notations that are commonly used.

#### Big Oh Notation

Big-Oh ( $O$ ) notation gives an upper bound for a function  $f(n)$  to within a constant factor.

We write  $f(n) = O(g(n))$ , If there are positive constants  $n_0$  and  $c$  such that, to the right of  $n_0$  the  $f(n)$  always lies on or below  $c \cdot g(n)$ .

$O(g(n)) = \{ f(n) : \text{There exist positive constant } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq c \cdot g(n), \text{ for all } n \geq n_0 \}$



---

**conditional and repetition blocks:** Check the flowcharts in the topic “Control structures in Python - conditionals and loops”

---

### Computational Thinking:

It follows 4 steps

1. **Decomposition:** Divide the problem into smaller problems, usually problems that we have solved before.
2. **Pattern Recognition:** Find similar repeated patterns, identify the differences. Note patterns may come from other domains that you have dealt with.
3. **Abstraction:** Define the bigger problem as combination of solution of smaller problems, each of which is solved the same way. Hide all the details and focus on the bigger picture.
4. **Algorithm Design:** Write step by step procedure

---

### Algorithm:

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

#### Characteristic or features of an algorithm :

1. **Input**, means it has zero or more inputs, i.e., an algorithm can run without taking any input.
2. **Output**, means it has one or more outputs, i.e., an algorithm must produce atleast one output.
3. **Finiteness**, means it must always terminate after a finite number of steps.
4. **Definiteness**, means each step must be precisely defined and clear.
5. **Effectiveness**, means it is also generally expected to be effective.

An algorithm should be and **unambiguous** and independent of any programming code, i.e., **language independent**.

#### Algorithm to add two numbers entered by the user

Step 1: Start

Step 2: Read values num1 and num2.

Step 3: Add num1 and num2 and assign the result to sum.

sum ← num1 + num2

Step 4: Display sum

Step 5: Stop

#### Algorithm to find greatest of three numbers

```
Step 1: Start
Step 2: Read variables a,b and c.
Step 3: If a >= b and a>=c
    Assign a to the largest.
    Elif b>=a and b>=c
    Assign b to the largest.
    Else
    Assign c to the largest.
Step 4: Display largest
Step 5: Stop
```

---

### Pseudo code:

- 1)Pseudo code is one of the method that is used to represent an algorithm
- 2)It is not written using specific rules, so it will not execute in computer
- 3)There are lots of formats for writing pseudo code and most of them follow languages as C,FORTRAN
- 4)Pseudo code can be read ,understood by programmers who are familiar with different programming languages.

### Pseudo code for adding of two numbers

```
BEGIN
NUMBER s1, s2, sum
OUTPUT("Input number1:")
INPUT s1
OUTPUT("Input number2:")
INPUT s2
sum=s1+s2
OUTPUT sum
END
```

### Difference between Algorithm and pseudo code

| Algorithm  | Pseudocode  |
|--|---|
| 1)Well defined sequence of steps that provide solution for a problem | 1)One of the method that can be used to represent algorithm             |
| 2)It is written in natural language.                                 | 2)It is written in informal way,closely related to programming language |
| 3)Any user can understand  | 3)Only programmer familiar with language can understand                 |

---

### Elements of a programming Language Python:

#### Token:

Tokens are the smallest unit of the program.

They are **Reserved words or Keywords,Identifiers,Literals,Operators**



## Literals:

Literal is a raw data given in a variable or constant. In Python, there are various types of literals they are as follows:

- 1) Numeric Literals
- 2) String literals
- 3) Boolean literals
- 4) Literal Collections

### Numeric Literals

Binary literals  
Decimal Literals  
Octal Literal  
Hexadecimal Literal  
Float Literal  
Complex Literal

**# program to demonstrate Numeric Literals**

```
a = 0b1010 #Binary Literals
b = 100 #Decimal Literal
c = 0o310 #Octal Literal
d = 0x12c #Hexadecimal Literal
print(a, b, c, d)
```

**#Float Literal**

```
float_1 = 10.5
float_2 = 1.5
print(float_1, float_2)
```

**#Complex Literal**

```
x = 2+3.14j
print(x, x.imag,x.real)
```

### String literals

A string literal is a sequence of characters surrounded by quotes. We can use both single, double, or triple quotes for a string.

```
S1='This is Python'
S2="This is Python"
S3=""""This is Python"""
print(S1)
print(S2)
print(S3)
```

**Character Literals:** Character literal is a single character surrounded by single or double quotes .

```
c1='a'
c2="a"
c3=""""a"""
print(c1)
```

```
print(c2)
print(c3)
```

### Boolean literals:

A Boolean literal can have any of the two values: `True` or `False`.

```
x = True
y = False
a = True + 4
b = False + 10
print("x is", x)
print("y is", y)
print("a:", a)
print("b:", b)
```

### Literal Collections

There are four different literal collections List literals, Tuple literals, Dict literals, and Set literals.

```
L=[1,2,3]
T=(1,2,3)
S={1,2,3}
d={'1':'v1','2':'v2'}
print("List:",L)
print("Tuple:",T)
print("Set:",S)
print("Dictionary:",d)
```

### Python Identifiers

An identifier is a name given to entities like class, functions, variables, etc. It helps to differentiate one entity from another.

#### Rules for writing identifiers

- ✓ Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore \_.
- ✓ An identifier cannot start with a digit.
- ✓ Keywords cannot be used as identifiers
- ✓ We cannot use special symbols like !, @, #, \$, % etc. in our identifier

---

### Keywords :

- ✓ Keywords are reserved words in python
- ✓ We cannot use keyword as variable names, function names and any other identifier.
- ✓ They are used to define syntax and structure of python language.
- ✓ Keywords are case sensitive
- ✓ There are 33 keywords in python 3.7

Following are keywords available in Python:

|              |          |         |          |        |
|--------------|----------|---------|----------|--------|
| <b>False</b> | class    | finally | is       | return |
| <b>None</b>  | continue | for     | lambda   | try    |
| <b>True</b>  | def      | from    | nonlocal | while  |
| and          | del      | global  | not      | with   |
| as           | elif     | if      | or       | yield  |
| assert       | else     | import  | pass     |        |
| break        | except   | in      | raise    |        |

---

## Type Conversion

The process of converting the value of one data type (integer, string, float, etc.) to another data type is called type conversion. Python has two types of type conversion.

1)Implicit Type Conversion

2)Explicit Type Conversion

### 1)Implicit Type Conversion

In this method Python automatically converts one data type(lower) to another data type(higher). This process doesn't need any user involvement.

Let's see an example where Python promotes the conversion of the lower data type (integer) to the higher data type (float) to avoid data loss.

#### #Program to demonstrate Implicit Conversion

```
a=10
b=5.5
c=a+b
print("The sum of a and b is ",c)
print("The datatype of c is",type(c))
```

### 2)Explicit Type Conversion

In Explicit Type Conversion, users convert the data type of an object to required data type.

We use the predefined functions like int(), float(), str(), etc to perform explicit type conversion.

This type of conversion is also called **typecasting** because the user casts (changes) the data type of the objects.

In Type Casting, loss of data may occur as we enforce the object to a specific data type.

Syntax :

```
<required_datatype>(expression)
```

#### # Write a program to demonstrate explicit conversion

```
a=5.5
b=10
c=a+b
print("Before explicit conversion",c)
print(type(c))
```

```
c=int(c)
print("After explicit conversion",c)
print(type(c))
```

### Output:

Before explicit conversion 15.5

<class 'float'>

After explicit conversion 15

<class 'int'>

---

## Variables

Variables are just names of memory locations. Variables are used to store and retrieve values from memory. Type of the variable is based on the value we assign to the variable.

Variables are containers for storing the data value.

But there are a few rules to follow when choosing name of variable

- ✓ Names must start with a letter or underscore and can be followed by any number of letters, digits, or underscores.
- ✓ Cannot start with a number
- ✓ It contain alpha-numeric characters & underscore.
- ✓ Variable names are case sensitive(age, Age, AGE are different).
- ✓ Names cannot be reserved words or keywords

### Syntax :

Variablename = value

### Eg:

- ✓ x = 10
  - ✓ x='python'
  - ✓ z=6.1
- Assign value to multiple variable
- ✓ x,y,z=10,20,30  
print(x)  
print(y)  
print(z)
  - ✓ x,y,z=10.0,'Ayaan',6  
print(x)  
print(y)  
print(z)

Identify the following are valid or invalid???

A=10 --- valid

\_a=11 --- valid

1a=12 --- Invalid

a@=13 --- Invalid

a1\_2=14 --- valid

---

## Input/Output Statements:

**input():** The `input()` function allows user to give input.

Syntax

`input(prompt)`

*prompt*- It can be empty or A String.

Eg:

- ❖ `a=int(input())`
- ❖ `b=float(input())`
- ❖ `c=input()`
- ❖ `a=int(input("enter the value of a"))`
- ❖ `print('Enter your name:')`  
`x = input()`  
`print('Hello, ' + x)`
- ❖ `x = input('Enter your name:')`  
`print('Hello, ' + x)`

Multiple input at a time:

```
a,b=input("enter firstname"),input("enter lastname")
```

```
print(a)
```

```
print(b)
```

**Output():** This function is used to output the data or print the result on monitor.

Eg:

- ❖ `print('Welcome')`
- ❖ `print("Python")`
- ❖ `print("""Hello""")`
- ❖ `print(1,2,3,4)`
- ❖ `print(1,2,3,sep='*')`
- ❖ `print(1,2,3,4,sep='*',end='&')`
- ❖ **Output formatting:**

Sometimes we would like to format our output to make it look attractive.

```
x = 5
```

```
y = 10
print('The value of x is {} and y is {}'.format(x,y))
```

**Output:**

The value of x is 5 and y is 10

---

## Basic Data Types(Numbers also included)

Every value in Python has a datatype.

Python has the following data types/ objects built-in by default, in these categories:

|                        |  |
|------------------------|--|
| <u>Text Type:</u>      | <code>str</code>   |
| <u>Numeric Types:</u>  | <code>int</code> , <code>float</code> , <code>complex</code> |
| <u>Sequence Types:</u> | <code>list</code> , <code>tuple</code> , <code>range</code>  |
| <u>Mapping Type:</u>   | <code>dict</code>  |
| <u>Set Types:</u>      | <code>set</code> , <code>frozenset</code>                    |
| <u>Boolean Type:</u>   | <code>bool</code>  |

**type():** To know the data type of any object by using the `type()` function

```
✓ x=5
  print(type(x))
✓ x=10.0
  print(type(x))
✓ x='hello'
  print(type(x))
```

## Python Numbers:

They are categorized into mainly three types. They are `int`, `float` and `complex`.

**Int:** It consists of positive and negative numbers without decimals. They are of unlimited length.

```
✓ x=1
  print(x)
  print(type(x))
✓ y=123456789012345678901234567890
  print(y)
  print(type(y))
✓ z=-234
  print(z)
  print(type(z))
```

**float:** It consists of positive and negative numbers containing decimals numbers

```
✓ x=1.10
  print(x)
  print(type(x))
✓ y=0.123
  print(y)
  print(type(y))
✓ z=-0.123
  print(z)
  print(type(z))
```

**complex numbers:** Complex numbers in python are in the form of a + bj

```
✓ x=3+5j
  print(x)
  print(type(x))
✓ y=5j
  print(y)
  print(type(y))
✓ z=-5j
  print(z)
  print(type(z))
```

### Booleans:

Python provides Boolean datatype called bool with values True & False. True represents integer 1 and False represents integer 0. bool is a sub class of integer class. Following are examples on Booleans:

Eg:

```
✓ print(True+1)
✓ print(False*99)
✓ print(True+22)
✓ print(False*2)
✓ a=True
  print(type(a))
✓ b=False
  print(type(b))
✓ c=true
  print(type(c))
```

### Strings:

A string is a sequence of characters.

Strings are used to store textual information. It is represented by single quote or double quote.

Eg:

- ✓ 'B.V.Prasanthi'
- ✓ "A"
- ✓ "123"
- ✓ a='python'  
print(a)  
print(type(a))
- ✓ b='12.345'  
print(b)  
print(type(b))
- ✓ c=str(input("enter c"))  
print(c)  
print(type(c))
- ✓ d=input()  
print(d)  
print(type(d))

---

## Python 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

## Python Arithmetic Operators

Arithmetic operators are used with numeric values to perform common mathematical operations:

| Operator | Name           | Example |
|----------|----------------|---------|
| +        | Addition       | x + y   |
| -        | Subtraction    | x - y   |
| *        | Multiplication | x * y   |
| /        | Division       | x / y   |
| %        | Modulus        | x % y   |
| **       | Exponentiation | x ** y  |
| //       | Floor division | x // y  |



Examples:

### #Arithmetic Operators

```
x=int(input("Enter the value of x"))
y=int(input("Enter the values of y"))
print("Sum=",x+y)
print("Sub=",x-y)
print("Product=",x*y)
print("Division",x/y)
print("Modulus=",x%y)
print("Exponentiation=",x**y)
print("Floor division=",x//y)
```

```
guest-gUh7cf@slave03:~$ gedit arithmetic.py
guest-gUh7cf@slave03:~$ python3 arithmetic.py
Enter the value of x3
enter the values of y2
Sum= 5
Sub= 1
Product= 6
Division 1.5
Modulus= 1
Exponentiation= 9
Floor division= 1
```

### Python Comparison Operators

Comparison operators are used to compare two values:

| Operator | Name                     | Example |
|----------|--------------------------|---------|
| ==       | Equal                    | x == y  |
| !=       | Not equal                | x != y  |
| >        | Greater than             | x > y   |
| <        | Less than                | x < y   |
| >=       | Greater than or equal to | x >= y  |
| <=       | Less than or equal to    | x <= y  |

### #Comparison Operators

```
x=int(input("Enter the values of x:"))
y=int(input("Enter the values of y:"))
```

```

print(x==y)
print(x!=y)
print(x>y)
print(x<y)
print(x>=y)
print(x<=y)

```

```

guest-gUh7cf@slave03:~$ gedit comparison.py
guest-gUh7cf@slave03:~$ python3 comparison.py
Enter the values of x:3
Enter the values of y:6
False
True
False
True
False
True

```

## Python Logical Operators

Logical operators are used to combine conditional statements:

| Operator | Description   | Example                                  |
|----------|---|--|
| and      | Returns True if both statements are true                | <code>x &lt; 5 and x &lt; 10</code>      |
| or       | Returns True if one of the statements is true           | <code>x &lt; 5 or x &lt; 4</code>        |
| not      | Reverse the result, returns False if the result is true | <code>not(x &lt; 5 and x &lt; 10)</code> |

| p | q | $p \wedge q$ | p | q | $p \vee q$ |
|---|---|--------------|---|---|------------|
| T | T | T            | T | T | T          |
| T | F | F            | T | F | T          |
| F | T | F            | F | T | T          |
| F | F | F            | F | F | F          |

## #Logical operators

```

x=int(input("Enter the value of x:"))
print(x<5 and x<10)
print(x<5 or x<4)
print(not(x>10))

```

```

guest-gUh7cf@slave03:~$ gedit logical.py

```

```
guest-gUh7cf@slave03:~$ python3 logical.py
```

```
Enter the value of x:6
```

```
False
```

```
False
```

```
True
```

## Python Identity Operators

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

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

### #identity operators

```
x=int(input("Enter the value of x:"))
```

```
y=int(input("Enter the value of y:"))
```

```
z=x
```

```
print(x is y)
```

```
print(x is not y)
```

```
print(x is z)
```

```
guest-gUh7cf@slave03:~$ gedit identity.py
```

```
guest-gUh7cf@slave03:~$ python3 identity.py
```

```
Enter the value of x:5
```

```
Enter the value of y:9
```

```
False
```

```
True
```

```
True
```

## Python Membership Operators

Membership operators are used to test if a sequence is presented in an object:

| Operator | Description  | Example    |
|----------|--|------------|
| in       | Returns True if a sequence with the specified value is present in the object     | x in y     |
| not in   | Returns True if a sequence with the specified value is not present in the object | x not in y |

### #Membership operators

```
x=[1,2,3,4,5]
y=int(input("enter the value of y:"))
print(y in x)
print(y not in x)
```

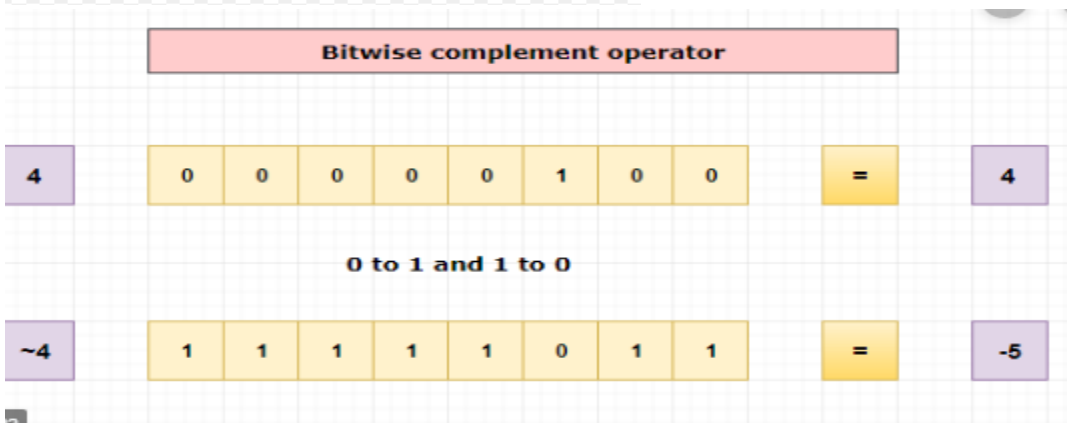
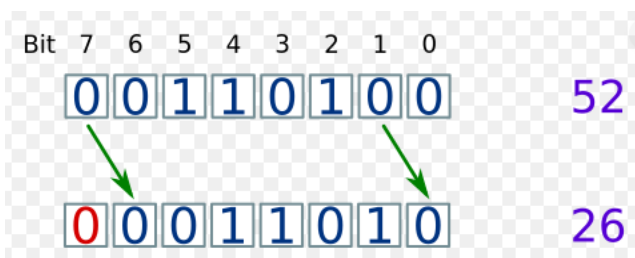
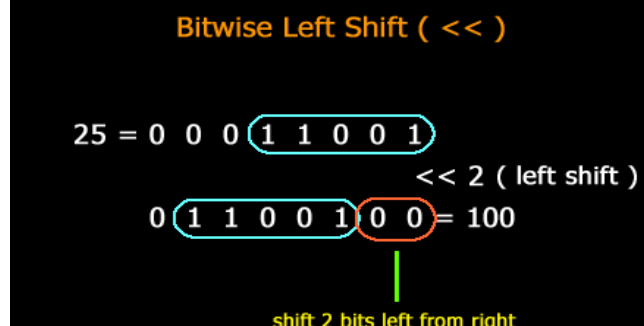
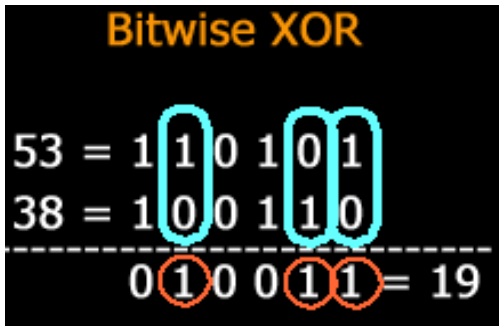
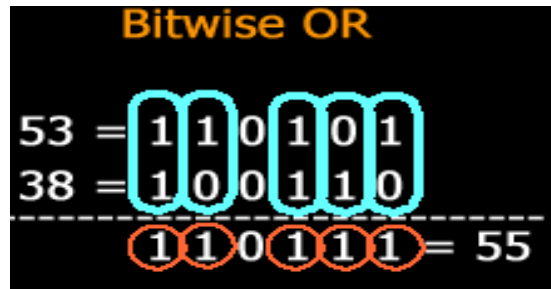
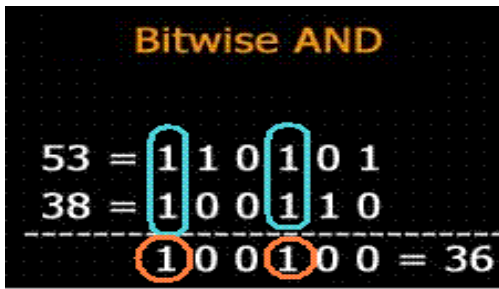
```
guest-gUh7cf@slave03:~$ gedit membership.py
guest-gUh7cf@slave03:~$ python3 membership.py
enter the value of y:5
True
False
```

## Python Bitwise Operators

Bitwise operators are used to compare (binary) numbers:

| 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 |

| a | b | a&b | a b | a^b | ~a |
|---|---|-----|-----|-----|----|
| 0 | 0 | 0   | 0   | 0   | 1  |
| 0 | 1 | 0   | 1   | 1   | 1  |
| 1 | 0 | 0   | 1   | 1   | 0  |
| 1 | 1 | 1   | 1   | 0   | 0  |



## #program on Bitwise Operator

```
x=int(input("enter the value of x:"))
y=int(input("enter the value of y:"))
print(x&y)
print(x | y)
print(x ^ y)
print(~x)
print(x<<1)
print(x>>1)
```

```
guest-gUh7cf@slave03:~$ gedit bitwise.py
guest-gUh7cf@slave03:~$ python3 bitwise.py
enter the value of x:5
enter the value of y:3
1
7
```

6  
-6  
10  
2

## Python Assignment Operators

Assignment operators are used to assign values to variables:

| 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  |
| %=       | x %= 3  | x = x % 3  |
| //=      | x //= 3 | x = x // 3 |
| **=      | x **= 3 | x = x ** 3 |
| &=       | x &= 3  | x = x & 3  |
| =        | x  = 3  | x = x   3  |
| ^=       | x ^= 3  | x = x ^ 3  |
| >>=      | x >>= 3 | x = x >> 3 |
| <<=      | x <<= 3 | x = x << 3 |

### # write a program to demonstrate assignment operators

```
x=int(input("enter the value of x:"))
x += 3          #same as x=x+3
print("x=",x)
x -= 3          #same as x=x-3
print("x=",x)
x *= 3          #same as x=x*3
print("x=",x)
x /= 3          #same as x=x/3
print("x=",x)
x %= 3          #same as x=x%3
print("x=",x)
x //= 3         #same as x=x//3
print("x=",x)
x **= 3         #same as x=x**3
print("x=",x)
```

guest-gUh7cf@slave03:~\$ gedit assignment.py

```
guest-gUh7cf@slave03:~$ python3 assignment.py
```

```
enter the value of x:5
```

```
x= 8
```

```
x= 5
```

```
x= 15
```

```
x= 5.0
```

```
x= 2.0
```

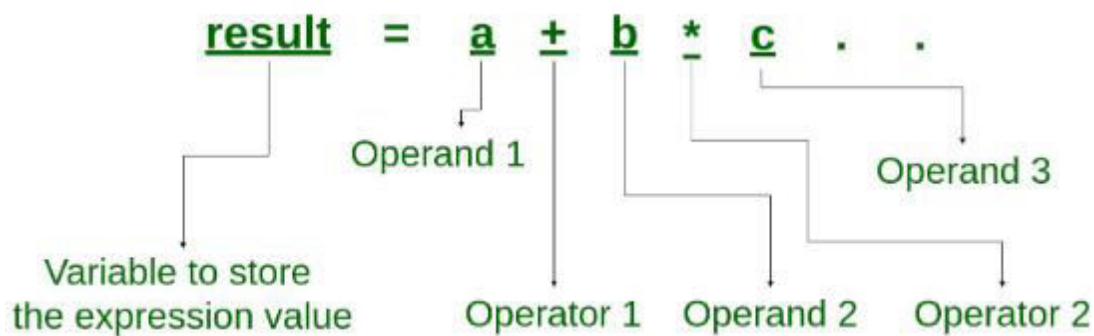
```
x= 0.0
```

```
x= 0.0
```

## Expressions and order of evaluations(precedence)

**Expression:** An expression is a combination of operators, constants and variables. An expression may consist of one or more operands, and zero or more operators to produce a value.

### What is an Expression?



Eg:

✓ `10 - 4 * 2`

✓ `a*b/c`

✓ `a*b+2`

**Python** has well-defined rules for specifying the **order** in which the operators in an **expression** are **evaluated** when the **expression** has several operators.

The operator precedence in Python are listed in the following table. It is in descending order, upper group has higher precedence than the lower ones.

| Operators | Meaning     |
|-----------|-------------|
| ()        | Parentheses |
| **        | Exponent    |

|  |   |
|--|---|
| +x, -x, ~x                                   | Unary plus, Unary minus, Bitwise NOT              |
| *, /, //, %                                  | Multiplication, Division, Floor division, Modulus |
| +, -   | Addition, Subtraction                             |
| <<, >>                                       | Bitwise shift operators                           |
| &  | Bitwise AND                                       |
| ^  | Bitwise XOR                                       |
|  | Bitwise OR  |
| ==, !=, >, >=, <, <=, is, is not, in, not in | Comparisons, Identity, Membership operators       |
| not  | Logical NOT                                       |
| and  | Logical AND                                       |
| or   | Logical OR  |

### Operator precedence rule in Python

Eg:

- ✓ `print(10 - 4 * 2)`
- ✓ `print((10 - 4) * 2)`
- ✓ `print(3+2-1)`

### Associativity of Python Operators

We can see in the above table that more than one operator exists in the same group. These operators have the same precedence.

When two operators have the same precedence, associativity helps to determine which the order of operations.

Associativity is the order in which an expression is evaluated that has multiple operator of the same precedence. Almost all the operators have left-to-right associativity.

For example, multiplication and floor division have the same precedence. Hence, if both of them are present in an expression, left one is evaluates first.

- ✓ `print(5 * 2 // 3)` L to R

Exponent operator `**` has right-to-left associativity in Python.



- ✓ # Right-left associativity of \*\* exponent operator  
# Output: 512  
print(2 \*\* 3 \*\* 2)
- ✓ # Shows the right-left associativity of \*\*  
# Output: 64  
print((2 \*\* 3) \*\* 2)

---

## Control structures in Python - conditionals and loops

### Conditionals:

#### Decision Making Statements/Selection/Conditional Statements

- ❖ if statement
- ❖ if else statements
- ❖ elif ladder
- ❖ Nested if statements

#### if statement

Decision making is required when we want to execute a code only if a certain condition is satisfied.

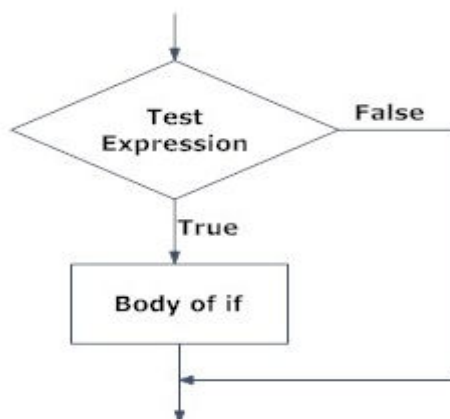
#### Python if Statement Syntax

```
If testexpression:  
    statement(s)
```

Here, the program evaluates the **test expression** and will execute **statement(s)** only if the text expression is **True**.

If the text expression is **False**, the **statement(s)** is not executed.

#### Python if Statement Flowchart



Eg

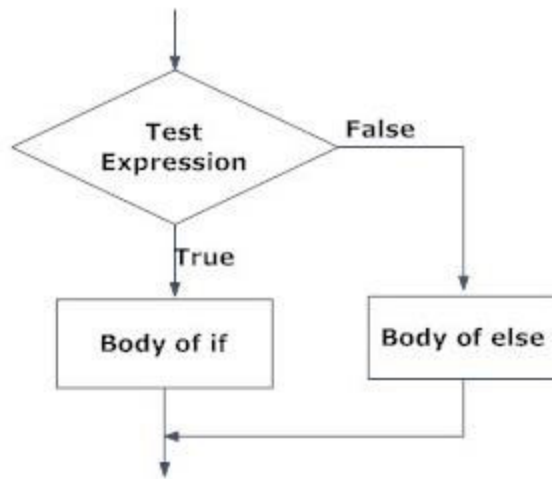
```
num = int(input("Enter the value of num"))  
if num > 0:  
    print(num, "is a positive number.")
```

### if...else Statement

## Python if..else Flowchart

### Syntax of if...else

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



The if..else statement evaluates **test expression** and will execute body of if only when test condition is True.

If the condition is False, body of else is executed. Indentation is used to separate the blocks.

### **# Program checks if the number is positive or negative**

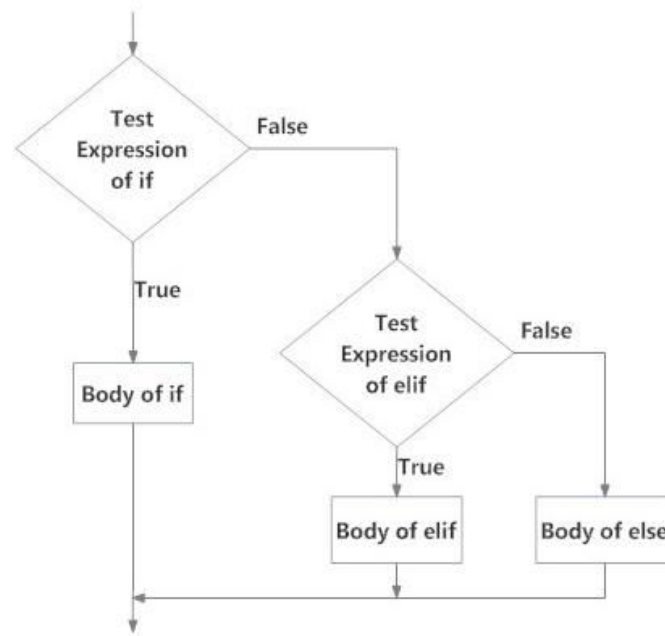
```
num = int(input("Enter the value of num"))  
if num >= 0:  
    print("Positive or Zero")  
else:  
    print("Negative number")
```

### Elif ladder

## Flowchart of if...elif...else

### Syntax of if...elif...else

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



The elif is short for else if. It allows us to check for multiple expressions. If the condition for if is False, it checks the condition of the next elif block and so on. If all the conditions are False, body of else is executed.

#### #program on elif ladder

```
num = float(input("Enter the value of num"))  
if num > 0:  
    print("Positive number")  
elif num == 0:  
    print("Zero")  
else:  
    print("Negative number")
```

#### # Python program to find the largest number among the three input numbers

```
a = int(input("Enter the value of a"))  
b = int(input("Enter the value of b"))  
c = int(input("Enter the value of c"))  
if (a >= b) and (a >= c):  
    largest = a  
elif (b >= a) and (b >= c):  
    largest = b  
else:  
    largest = c  
print("The largest number is",largest)
```

### Nested if Statement

If statement in another if is called as nested if statements.

#### Syntax:

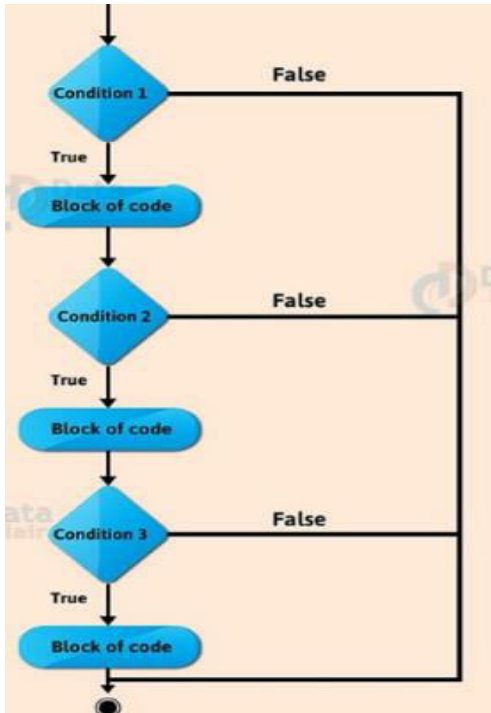
```
if cond1:  
    if cond2:  
        if cond3:  
            statements
```

or

```

if cond1:
    if cond2:
        statements
    else:
        statements
else:
    if cond3:
        statements
    else:
        statements

```



#### #program on Nested if

```

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

```

### Repetition Statements(loops):

- ❖ for loop
- ❖ while loop

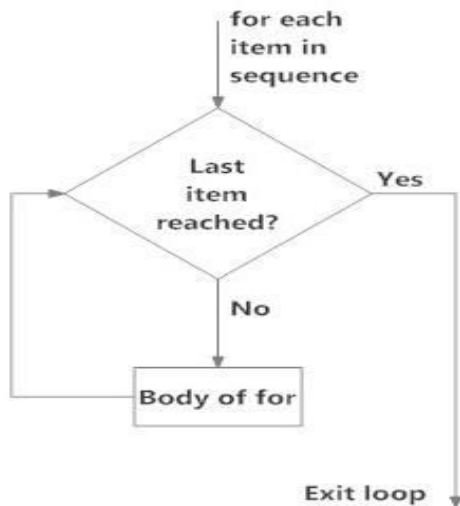
#### for loop

The for loop in Python is used to iterate over a sequence ([list](#), [tuple](#), [string](#)) or other iterable objects. Iterating over a sequence is called traversal.

# Syntax of for Loop

```
for val in sequence:  
    Body of for
```

## Flow chart



Here, `val` is the variable that takes the value of the item inside the sequence on each iteration. Loop continues until we reach the last item in the sequence. The body of for loop is separated from the rest of the code using indentation.

Eg:

```
for i in range(3):  
    print(i)
```

Syntax for range:

```
range(end)           #0 to end-1  
range(start,end)     #start to end-1  
range(start,end,step)
```

Few more example on for loop:

```
for i in range(0,3):  
    print(i)
```

```
for i in range(1,3):  
    print(i)
```

```
#print odd numbers  
for i in range(1,10,2):  
    print(i)
```

```
#print even numbers  
for i in range(0,11,2):  
    print(i)
```

## Else in For Loop

```
for x in range(6):  
    print(x)
```

```
else:  
    print("Finally finished!")
```

### Looping Through a String

```
for i in "ayaan":  
    print(i)
```

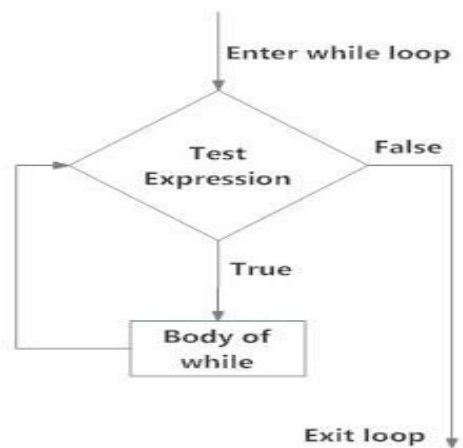
### while Loop

The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true.

### Syntax of while Loop

```
while test_expression:  
    Body of while
```

### Flow chart



Eg:

```
#print numbers from 1 to 5
```

```
i = 1
```

```
while i < 6:
```

```
    print(i)
```

```
    i += 1
```

```
#print odd numbers 1,3,5,7,9
```

```
i=1
```

```
while i<10:
```

```
    print(i)
```

```
    i=i+2
```

```
#print even numbers 0,2,4,6,8,10
```

```
i=0
```

```
while i<=10:
```

```
    print(i)
```

```
    i=i+2
```

## Break and continue:

In Python, break and continue statements can alter the flow of a normal loop.

Loops iterate over a block of code until test expression is false, but 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.

### Python break statement

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

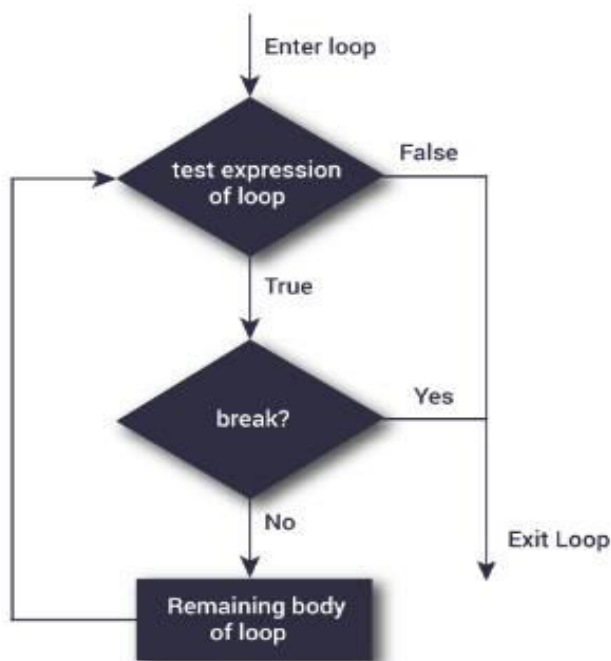
If break statement is inside a nested loop (loop inside another loop), break will terminate the innermost loop.

## Syntax of break

break

```
for var in sequence:
    # codes inside for loop
    if condition:
        break
    # codes inside for loop
# codes outside for loop
```

```
while test expression:
    # codes inside while loop
    if condition:
        break
    # codes inside while loop
# codes outside while loop
```



#using of break in for loop(output: 1 2 )

```
for i in range(1,5):
```

```
    if i==3:
```

```
        break
```

```
    print(i)
```

#using break in while loop (output: 1 2)

i=1

while(i<5):

if i==3:

break

print(i)

i=i+1

## Python continue statement

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.

## Syntax of Continue

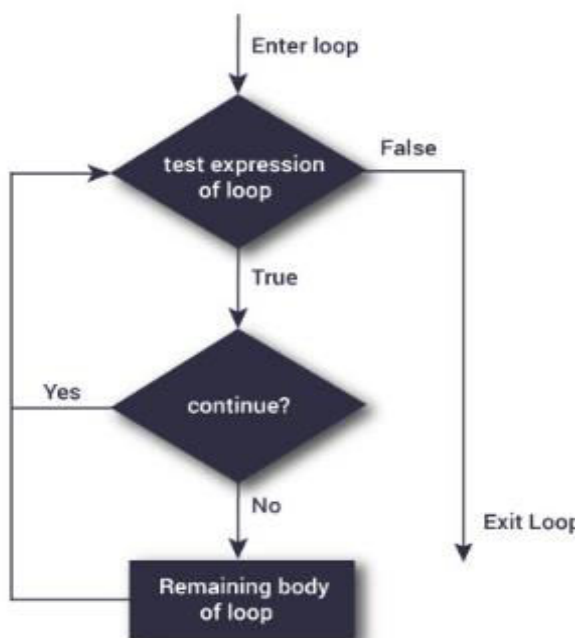
```
continue
```

```
for var in sequence:
    # codes inside for loop
    if condition:
        continue
    # codes inside for loop

# codes outside for loop
```

```
while test expression:
    # codes inside while loop
    if condition:
        continue
    # codes inside while loop

# codes outside while loop
```



#using continue in for loop (output:1 2 4 5)

for i in range(1,5):

if i==3:

continue

print(i)



#using continue in while loop (output:1 2 4 5)

i = 0

while i < 4:

    i=i+1

    if i == 3:

        continue

    print(i)

**UNIT – III: Python Data Structures and Modularization**

List and List Operations, Using Lists to represent Matrices, Strings, String operations, Tuples, Dictionaries, Sets, Iterators and generators, comprehensions. Basic math functions, User defined Functions, parameters to functions, positional, keyword and default arguments, Lambda Functions, recursion. Packages, modules and namespaces.

**Lists:**

- List is a data structure and is a collection of ordered & mutable(changeable) elements.
- List allows duplicate members.
- A list is created by placing all the items (elements) inside a square bracket [], separated by commas.
- List is dynamic. They grow and shrink on demand.
- It can have elements of same or different data types

Ex:

```
✓ a=[]
  print(a)
```

```
✓ a=[1,2,3]
  print(a)

  print(*a)
```

```
#List of integers
a=[1,2,3]
print(a)
```

```
#List with mixed datatypes
a=[1,"hai",3.4]
print(a)
```

**Operations:****1.concatenation:**

We can also use + operator to combine two lists. This is also called concatenation.

❖ **Syntax:**<list1>+<list2>

```
✓ l = [1, 3, 5]
  print(l + [9, 7, 5])
  # Output: [1, 3, 5, 9, 7, 5]
```

**2.multiplication or repetition:**

- ❖ **We can also use** \* operator repeats a list for the given number of times. This is also called Repetition

**syntax:** <list>\*<integer-value>

✓ print(["re"] \* 3)

#Output: ["re", "re", "re"]

### 3. List Index:

We can use the index operator [] to access an item in a list. Index starts from 0. So, a list having 5 elements will have index from 0 to 4.

✓ a=[1,2,3,4,5]

print(a[1])

print(a[3])

### 4. slicing:

Slicing is used to extract part of a list

We can access a range of elements in a list by using the slicing operator (colon).

```
l = ['a','b','c','d','e','f','g']
```

```
print(l)
```

```
print(l[1])
```

```
print(l[1:3])
```

```
print(l[1:-1])
```

```
print(l[:3])
```

```
print(l[2:])
```

```
print(l[:-1])
```

```
print(l[1::2])
```

```
print(l[::-1])
```

### 5. Membership Operator: we can find element exists in a list or not. it returns Boolean values

Ex: a=[1,2,3]

```
print(1 in a)
```

```
print(4 not in a)
```

### 6. Iteration: we can iterate each element in a list using for loop

**Syntax:** for i in list:

Ex: for i in [1,2,3]:

```
print(i)
```

### 7. list Length: It is used to find the length of the list.

**Syntax:** len(list)

Ex: l=[10,20,30]

```
print(len(l))
```

### list methods:

#### 1.append():

The append() method adds an element to the end of the list.

The syntax of the `append()` method is:

**syntax:** `listname.append(element)`

```
✓ l=[2,4,6,10]
  l.append(12)
  print("updated list=",l)
  #output:[2,4,6,10,12]
✓ l1=['a','b','c']
  l1.append('d')
  print("updated list=",l1)
  #output:['a','b','c','d']
```

**2.insert():** The `insert()` method insert one element at a desired location .

**syntax:** `listname.insert(index,element)`

```
✓ l=[1,3,5]
  l.insert(1,7)
  print("updated list=",l)
  #output:[1,7,3,5]
```

**3.sort():** Sort elements in a list in ascending order

**syntax:** `listname.sort()`

```
Ex: l=[2,3,4,5,1]
     l.sort()
     print(l)
```

o/p: **[1,2,3,4,5]**

**4.count():** This method Returns the count of occurrence of an element in a list

**syntax:** `list.count(x)`

```
Ex: l=[1,2,3,3,4,5]
     print(l.count(3))
```

o/p:2

**5.index():** This method returns the index of the element.

**Syntax:** `listname.index(element)`

```
Ex: l=[1,2,3,4,7,10,13,11,12]
     print(l.index(13))
```

o/p: 6

**6.pop():** This method removes and returns last object from the list.

**Syntax:** `list.pop(element)`

```
Ex: l=[12,14,13,10]
     print(l.pop())      #10 is removed
     print(l)            o/p:[12,14,13]
     print(l.pop(1))     #14 is removed
     print(l)            o/p:[12,13]
```

**7.remove():**This method removes the first instance of element in list.

Syntax: **list.remove(element)**

Ex: `l=[1,3,5,7,8,3]`  
`l.remove(3)`  
`print(l)`

o/p:[1,5,7,8,3]

**8.reverse():**This method reverses the list elements.

Syntax: **list.reverse()**

Ex: `l=[1,4,7,8,11,23]`  
`l.reverse()`  
`print(l)`

o/p:[23,11,8,7,4,1]

**9.extend():**

The extend() extends the list by adding all items of a list (passed as an argument) to the end.

Syntax: `list1.extend(list2)`

Ex: `l1=[2,3,4]`  
`l2=['a','b','c']`  
`l1.extend(l2)`  
`print(l1)`  
`print(l2)`

**10.copy():**make a copy of a list with the copy() method.

Syntax: **listname.copy()**

Ex: `l1=[3,17,19,2,13]`  
`l2=l1.copy()`  
`print(l2)`

### Using Lists to represent Matrices

In Python, we can implement a matrix as a nested list (list inside a list). We can treat each element as a row of the matrix.

For example `X = [[1, 2], [4, 5], [3, 6]]` would represent a 3x2 matrix. First row can be selected as `X[0]` and the element in first row, first column can be selected as `X[0][0]`.

**1)#write a program to display given matrix**

```
X = [[12,7,3],
      [4 ,5,6],
      [7 ,8,9]]
print(X)
```

```

# iterate through rows
for i in range(len(X)):
    # iterate through columns
    for j in range(len(X[0])):
        print(X[i][j],end='\t')
    for r in result:
        print()

```

2)#Write a program to add two matrices

```

X = [[12,7,3], [4 ,5,6],[7 ,8,9]]
Y = [[5,8,1],[6,7,3],[4,5,9]]
result = [[0,0,0],[0,0,0], [0,0,0]]
# iterate through rows
for i in range(len(X)):
    # iterate through columns
    for j in range(len(X[0])):
        result[i][j] = X[i][j] + Y[i][j]
        print(result[i][j],end='\t')
    print()

```

**# program to find norm of matrix**

```

x=[[1,2,3],[4,5,6],[7,8,9]]
sum=0
for i in range(len(x)):
    for j in range(len(x[i])):
        if(i==j):
            sum=sum+x[i][j]*x[i][j]
print(sum)

```

### Tuples:

Tuple is an ordered collection of elements.

Tuples are similar to list, but it is a sequence of immutable objects.

Tuples are fixed in length, heterogeneous, and can be nested.

Individual elements can be accessed through offset (index).

They are represented by ().

```

Ex1:  t1=()
      print(type(t1))

```

```

Ex2:  t2=(1,2,3,4,5)
      print(t2)

```

```
Ex3:  t3=(1,1.2,'abc')
      print(t3)
```

### Basic tuple operations:

**1.concatenation:**we can use + operator to combine the two tuples.this is also called concatenation.

**Syntax:** tuplename1+tuplename2

```
Ex:    t1=(1,2,3)
        t2=(4,5,6)
        print(t1+t2)
```

**o/p:**(1,2,3,4,5,6)

**2.Repition:**we can also repeat elements in a tuple for a given number of times using \* operator.

**Syntax:** tuple\*int-value

```
t=(1,2,3)
print(t*3)
```

**o/p:**(1,2,3,1,2,3,1,2,3)

**3.Tuple Length:**It is used to find the length of the tuple.

**Syntax:** len(tuple)

```
Ex:    t=(10,20,30)
        print(len(t))
```

**o/p:**3

**4. Index:**We can use the index operator [] to access an item in a tuple. Index starts from 0.

**Syntax:** tuplename[index ]

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

**5.Membership Operator:**we can find element exists in a tuple or not.it returns Boolean values

```
Ex:    a=(1,2,3)
        print(1 in a)
        print(4 not in a)
```

**6.Iteration:** we can iterate each element in a tuple using for loop

**Syntax:** for i in tuple:

```
Ex:    for i in (1,2,3):
        print(i)
```

### 7.slicing:

Slicing is used to extract part of a tuple

We can access a range of elements in a list by using the slicing operator (colon).

```
t = ('a','b','c','d','e','f','g')
print(t)
print(t[1])
```

```
print(t[1:3])
print(t[1:-1])
print(t[:3])
print(t[2:])
print(t[:-1])
print(t[1::2])
print(t[::-1])
```

### Methods:

**Tuple does not support remove(),pop(),append(),sort(),reverse(),insert() ,extend()**

**1) index():** This method returns the index of the element.

Syntax: tuplename.index(element)

Ex: t=(1,2,3,4,7,10,13,11,12)  
print(t.index(13))

o/p: 6

**2) count():** This method Returns the count of occurrence of an element in a list

**syntax:** tuplename.count(element)

Ex: t=(1,2,3,3,4,5)  
print(t.count(3))

o/p:2

**3) copy:** Copying one tuple to another tuple is done by using assignment operator

t1=(1,2,3)

t2=t1

print(t2)

**4) zip():**

It takes two or more sequence(list,tuple) and zip them into list of tuples or tuple of tuples.

✓ t1=(1,2,3)

t2=(5,7,8,9,10)

t3=tuple(zip(t1,t2))

print(t3) #output: ((1, 5), (2, 7), (3, 8))

✓ l1=[1,2,3]

l2=[5,7,8,9,10]

l3=list(zip(l1,l2))

print(t3) #output: [(1, 5), (2, 7), (3, 8)]

### **Compare List and Tuple.**

| S.NO | LIST              | TUPLE               |
|------|-------------------|---------------------|
| 1    | Lists are mutable | Tuple are immutable |



| S.NO | LIST  | TUPLE   |
|------|---|---|
| 2    | Implication of iterations is Time-consuming                                   | Implication of iterations is comparatively Faster         |
| 3    | The list is better for performing operations, such as insertion and deletion. | Tuple data type is appropriate for accessing the elements |
| 4    | Lists consume more memory   | Tuple consume less memory as compared to the list         |
| 5    | Lists have several built-in methods   | Tuple does not have many built-in methods.                |
| 6    | The unexpected changes and errors are more likely to occur                    | In tuple, it is hard to take place.                       |

### Sets:

- ✓ **Set:** A set is unordered collection of elements and it is mutable
- ✓ Every element is unique and must be immutable.
- ✓ Sets are represented by { } or by built in function set()

Ex:

- ✓ `s={1,2,3}`  
`print(s)`
- ✓ `s1={"python",(1,2,3)}`  
`print(s1)`
- ✓ `s2={[1,2,3],1.0,'hai'}`  
`print(s2)`     #output:error because list is mutable, but set should have immutable elements
- ✓ `s={3,2,3,1}`  
`print(s)`
- ✓ `s=set([10,20,30])`  
`print(s)`

### **Set operations:**

**1) Union(|):** It contains all the elements of set a and set b

```
a={1,2,3,4,5}
b={4,5,6,7,8}
print(a | b)
```

**2) Intersection(&):** It contains common elements in both sets

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

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

**3)Difference(-):**Difference of A & B contain only elements that are in A ,but not in B

```
a={1,2,3,4,5}
```

```
b={4,5,6,7,8}
```

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

**4)Set Symmetric difference(^):**Symmetric difference of a and b is set of elements in both a and b except that are common in both

```
a={1,2,3,4,5}
```

```
b={4,5,6,7,8}
```

```
print(a^b)          #output: {1, 2, 3, 6, 7, 8}
```

**5)Membership operators:**It return Boolean values

```
A={'a',1,2.5,3}
```

```
Print('a' in A)
```

```
Print(1 not in A)
```

**Set Methods:**

**1)add():**This method is used to add elements in a set

Syntax:      setname.add(element)

Ex:      s={'c','java','cpp'}

```
s.add('python')
```

```
print(s)
```

**2)clear():**This method is used to remove the elements from a set

Syntax:      setname.clear()

Ex:      v={1,2,3,4}

```
v.clear()
```

```
print(v)
```

**3)copy():**This method is used to copy the elements from one to another set

Syntax:      setname.copy()

Ex:      s={1,2,3}

```
s1=s.copy()
```

```
print(s1)
```

**4)difference():**Difference of A & B contain only elements that are in A ,but not in B

Syntax:      set1.difference(set2)

Ex:      a={1,2,3,4,5}

```
b={4,5,6,7,8}
```

```
print(a.difference(b))
```

**5)intersection()**It contains common elements in both sets

**Syntax:**        `set1.intersection(set2)`

**Ex:**            `a={1,2,3,4,5}`  
                  `b={4,5,6,7,8}`  
                  `print(a.intersection(b))`

**6)union():**It contains all the elements of set a and set b

**Syntax:**        `set1.union(set2)`

**Ex:**            `a={1,2,3,4,5}`  
                  `b={4,5,6,7,8}`  
                  `print(a.union(b))`

**7)symmetric\_difference():**Symmetric difference of a and b is set of elements in both a and b except that are common in both

**Syntax:**        `set1.symmetric_difference(set2)`

**Ex:**            `a={1,2,3,4,5}`  
                  `b={4,5,6,7,8}`  
                  `print(a.symmetric_difference(b))`

**8)discard():**It takes single element and removes it from set

**Syntax:**        `setname.discard(element)`

**Ex:**            `n={1,2,3,4}`  
                  `n.discard(3)`  
                  `print(n)`

**9)remove():**This method searches for given element in the set and removes it

**Syntax:**        `setname.remove(element)`

**Ex:**            `n={1,2,3,4}`  
                  `n.remove(3)`  
                  `print(n)`  
                  ✓ `n={1,2}`  
                  `n.remove(3)`        `#keyerror`

**Note:**If we try to remove the element in the set which is not present ,it gives **KeyError**

**10)pop():**It removes random element in the set

**Syntax:**        `setname.pop()`

**Ex:**            `s={'a','b','c'}`  
                  `s.pop()`  
                  `print(s)`

**11)isdisjoint():**

Two sets are said to be disjoint,if they have no common elements(True/False)

**Syntax:**        `set1.isdisjoint(set2)`

`a={1,2,3}`  
`b={4,5,6}`

```
print(a.isdisjoint(b))
```

### 12)issubset():

This method returns true if all elements of one set is present in second set.

Syntax:        set1.issubset(set2)

```
a={1,2,3}
```

```
b={1,2,3,4}
```

```
print(a.issubset(b))
```

---

## Dictionaries:

- ✓ Python dictionary is an unordered collection of elements and it is mutable.
- ✓ We can store values as a pair of keys and values  
Represented by dict() or {}
- ✓ Keys in dictionary are unique and they are immutable(strings,numbers,tuples)

### Creating a dictionary

- ✓ d={}
 

```
print(d)
```
- ✓ d1={1:'Ay',2:'aa',3:'n'}
 

```
print(d1)
```
- ✓ d2={'a':1,'b':2,'c':3}
 

```
print(d2)
```
- ✓ d3={1:1.1,2:'a',3:[1,2,3],4:(1,2,3)}    #mixed dictoinary
 

```
print(d3)
```
- ✓ d4=[[1,2,3]:1,'a':2]
 

```
print(d4)
```

    #Error-keys are of immutable nature
- ✓ d=dict([('a',1),('b',2),('c',3)])
 

```
print(d)
```

### Dictionary Operations:

1)**index:**It is used to find the value at particular index

Syntax:    dictname[index/key]

Ex:    d={'a':1,'b':2,'c':3}
 

```
print(d['b'])
```

    #output:2

2)**Len:**It is used to find length of a dictionary

Syntax:    len(dictname)

Ex:    d={'a':1,'b':2,'c':3}
 

```
print(len(d))
```

    #output:3

3)**Iteration:**Iteration is used to access the elements.

Syntax:    for i in dictname:

Ex:    s={1:1,3:9,5:25,7:49}
 

```
for i in s:
```

```
print(s[i])
```

**output:**1

```
9
25
49
```

**4)Membership operator:** This operator returns Boolean values, to check whether the keys are present in dictionary or not

**Ex:**     s={1:1,3:9,5:25,7:49}  
           print(1 in s)  
           print(48 not in s)

### Dictionary Methods

**1)keys():**It returns keys from the dictionary

Syntax: dictname.keys()

**Ex:**     d2={'a':1,'b':2,'c':3}  
           print(d2.keys())

**2)values():**It returns values from the dictionary

Syntax: dictname.values()

**Ex:**     d2={'a':1,'b':2,'c':3}  
           print(d2.values())

**3)get():**This method is used to read the value at particular index

**Ex:**             d2={'a':1,'b':2,'c':3}  
                   print(d2.get('a'))

**4)update():**This method is used to change the values in a dictionary

Syntax: olddict.update(newdict)

**Ex:**

```
d={'o':1,'t':2,'th':3}
d1={'tw':2}
d.update(d1)
print(d)
```

**5)copy():**This method is used to copy a dictionary.

**Syntax:**       dictname.copy()

**Ex:**             d2={'a':1,'b':2,'c':3}  
                   d1=d2.copy()  
                   print(d2)

**6)clear():**It remove all the elements from a dictionary

Syntax: dictname.clear()

```
Ex:    d={1:'one',2:'two'}  
       d.clear()  
       print(d)
```

**7)pop():**This method is used to remove the keys from dictionary

Syntax: dictname.pop(key)

```
Ex:    f={1:'app',2:'man',3:'stra'}  
       f.pop(1)  
       print(f)  
       #output: {2:'man',3:'stra'}
```

---

## Iterators and generators

Note:For clear explanation refer to

<https://www.hackerearth.com/practice/python/iterators-and-generators/iterators-and-generators-1/tutorial/>

Iterators are containers for objects so that you can loop over the objects. In other words, you can run the "for" loop over the object. There are many iterators in the Python standard library. For example, list is an iterator and you can run a for loop over a list.

Python generator gives us an easier way to create python iterators. This is done by defining a function but instead of the return statement returning from the function, use the "yield" keyword. For example, see how you can get a simple vowel generator below.

## Comprehensions

Comprehensions in Python provide us with a short and concise way to construct new sequences (such as lists, set, dictionary etc.) using sequences which have been already defined. Comprehensions are of three types:

- list comprehensions
- set comprehensions
- dict comprehensions

List comprehensions were introduced in Python 2.0; while set and dict comprehensions have been introduced in Python 2.7.

## List Comprehensions

List comprehension is the most popular Python comprehension. It allows us to create a new list of elements that satisfy a condition from an iterable. An iterable is any Python construct that can be looped over like lists, strings, tuples, sets. In list comprehensions we use square brackets.

General syntax of a list comprehension is as follows:

```
[expression for ele1 in iterable1 if cond1
    for ele2 in iterable2 if cond2
    ...
    for eleN in iterableN if condN]
```

Following is an example of list comprehension for creating a list of squares of numbers in range 1-10:

```
✓ s=[x*x for x in range(1,11)]
  print(s)
```

In the previous example, we can also use if condition to generate only squares of even numbers:

```
✓ s=[x*x for x in range(1,11) if x%2==0]
```

Result of above comprehension will be:

```
[4, 16, 36, 64, 100]
```

### Set Comprehensions

Set comprehensions were added to python in version 2.7. In set comprehensions, we use the braces rather than square brackets. For example, to create the set of the squares of all numbers between 0 and 10 the following set comprehension can be used:

```
x = {i**2 for i in range(10)}
print(x)
output: {0, 1, 64, 4, 36, 9, 16, 49, 81, 25}
```

### Dict Comprehensions

Just like set comprehensions, dict comprehensions were added to python in version 2.7. Below we create a mapping of a number to its square using dict comprehension:

```
x = {i:i**2 for i in range(10)}
print(x)
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

## Introduction to Functions

- ✓ A function is a block of organized and reusable program code that performs a specific, single, and well-defined task.
- ✓ A function provides an interface for communication in terms of how information is transferred to it and how results are generated.

## Need for Functions

- ✓ Simplifies program development by making it easy to test separate functions.
- ✓ Understanding programs becomes easier.
- ✓ Libraries contain several functions which can be used in our programs to increase the productivity.
- ✓ By dividing a large program into smaller functions, different programmers can work on different functions.
- ✓ Users can create their own functions and use them in various locations in the main program.

There are two types of functions:

- ✓ Built-in-functions
- ✓ User defined functions

### Built-in functions:

Python interpreter has number of functions that are available for use. They are called as Built-in functions.

Eg: `input()`, `print()`, `range()`, `abs()`, `len()`, `type()`, `sum()`, `sorted()`, etc.,

### User defined functions

## Defining Functions

A function definition consists of a function header that identifies the function, followed by the body of the function. The body of a function contains the code that is to be executed when a function is called. To define a function, we have to remember following points:

- Function definition starts with the keyword `def`
- The keyword `def` is followed by the function name and parentheses.
- After parentheses, a colon (`:`) should be placed.
- Parameters or arguments that the function accepts should be placed inside the parentheses.
- A function might have a return statement.

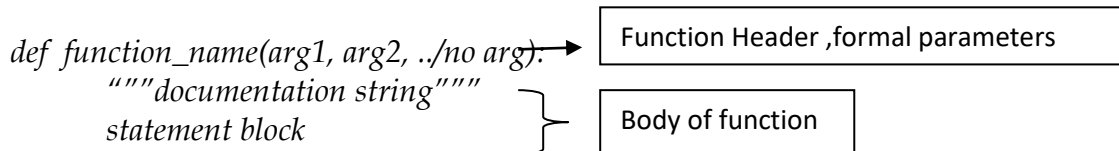


- The function code should be indented properly.

A function definition contains two parts:

- Function header
- Function body

The syntax of a function definition is as follows:



Example for defining a function without arguments is as follows:

```
✓ def sample():
    """ This is sample program created by B.V.Prashanthi"""
    print("hello")
    print("python")
```

Example for defining a function with arguments is as follows:

```
✓ def add(a,b):
    """ This is sample program for addition"""
    print(a+b)
```

## Calling a Function

Before calling a function, it should be defined. We call a function to execute the code inside the function.

Syntax for a function call is as follows:

```
function_name( )
function_name( [arg1, arg2, ...] ) → Function Call,Actual parameters
```

Eg:

```
sample()
add(a,b)
```

The arguments or parameters passed in a function call are called actual parameters. The arguments used in the function header of a function definition are called formal parameters. When a function is called, control moves to the first line inside the function definition. After the body of function definition is executed, control returns back to the next statement after the function call. Points to remember while calling a function:

- The function name and number of parameters must be same in the function call and function definition.
- When the number parameters passed doesn't match with the parameters in the function definition, error is generated.
- Names of arguments in the function call and function definition can be different.
- Arguments can be passed as expressions. The expression will get executed first and then the value is passed to the formal parameter.
- The parameter list must be separated by commas.
- If the function returns a value it must be assigned to some variable in the calling function.

Let's consider the following example which demonstrates function definition and function call:

#program without parameters

```
✓ def sample():
    """ This is sample program created by B.V.Prashanthi"""
    print("hello")
    print("python")
sample()
```

#program with parameters

```
✓ def add(a,b):
    """ This is sample program for addition"""
    print(a+b)
a=10
b=20
add(a,b)
```

#different parameters in function header and function call

```
✓ def add(x,y):
    """ This is sample program for addition"""
    print(x+y)
a=10
b=20
add(a,b)
```

## Functions Returning Value(Fruitful Functions)

A function may or may not return a value. To return a value, we must use *return* statement in the function definition. Every function by default contains an implicit *return* statement as the last line which returns *None* object to the calling function. The functions which returns a value is called fruitful functions.

A return statement is used for two reasons:

- ✓ Return a value to the caller.
- ✓ To end the execution of a function and give control to caller.

The syntax of *return* statement is as follows:

```
return()
```

*return (expression)*

Example of a function returning a value is as follows:

```
✓ def cube(x):
    return x*x*x
a=int(input("enter the value of a"))
b=cube(a)
print("Cube=",b)

✓ def sq(x):
    return x*x
x=int(input("enter the value of x"))
b=sq(x)
print("Square=",b)
```

## Passing Arguments

If a function definition contains arguments, then in a function call there is a need to pass arguments. For example, in the previous function definition of *cube*, there is a single argument. So, while calling this function we need to pass a single parameter. The *cube* function can be called as follows:

*cube(10)*

In the above function call, 10 is the actual parameter.

## Positional Arguments

Positional arguments are arguments that can be called by their position in the function definition. First positional argument always needed to be listed first when the function is called. Second positional argument always needed to be listed second when the function is called and so on

Eg:

```
✓ def f(a,b,c):
    print(a,b,c)    #output:1 2 3
f(1,2,3)

✓ def f(a,b,c):
    print(a*b+c)    #output:5
f(1,2,3)

✓ def f(a,b,c):
    print(a+b/c)    #output:2
f(1,1,1)

✓ def add(a,b):
    print(a+b)
a=10
```

```
b=20
add(a,b)
```

## Default Arguments

The formal parameters in a function definition can be assigned a default value. Such parameters to which default values are assigned are called default arguments.

- ✓ Default value can be assigned to a parameter by using the assignment ( = ) operator.
- ✓ A function definition can have one or more default arguments.
- ✓ Default arguments allow a function call to pass less parameters than the number of parameters in the function definition. (no.of parameters in function call and function header need not be same)

Following example demonstrates a function with default arguments and its usage:

```
✓ def f(a,b=2,c=3):
    print(a,b,c)    #output:1 2 3
f(1)
```

```
✓ def f(a,b=2,c=3):
    print(a,b,c)    #output:1 2 3
f(a=1)
```

```
✓ def f(a,b=2,c=3):
    print(a,b,c)    #output:1 3 3
f(1,3)
```

```
✓ def f(a,b=2,c=3):
    print(a,b,c)    #output:1 2 6
f(1,c=6)
```

```
✓ def f(a,b=2,c=3):
    print(a,b,c)    #output:2 3 4
f(2,3,4)
```

## Keyword Arguments

In general, when parameters are passed in a function call, they are assigned to formal parameters based on their position. Such parameters are called positional parameters. We can also pass the parameters based on the name (keyword) of the parameter in the function definition. Such parameters are called keyword arguments.

Points to remember when using keyword arguments:

- All keyword parameters must match one of the parameters in the function definition.
- Order of keyword arguments is not important.
- A value should not be passed more than once to a keyword parameter.

Following example demonstrates keyword arguments and their use:

```
✓ def f(a,b,c):
    print(a,b,c)    #output:1 2 3
    f(1,2,3)

✓ def f(a,b,c):
    print(a,b,c)    #output:1 1 1
    f(a=1,b=1,c=1)

✓ def f(a,b,c):
    print(a,b,c)    #output:1 2 3
    f(c=3,b=2,a=1)

✓ def f(a,b,c):
    print(a,b,c)    #output:1 2 8
    f(1,c=8,b=2)

✓ def f(a,b,c):
    print(a,b,c)    #output:1 2 9
    f(1,2,c=9)

✓ def f(a,b,c):
    print(a,b,c)    #output:Error
    f(1,b=2)

✓ def f(a,b,c):
    print(a,b,c)    #output:Error
    f(a=1,b=2,a=3)
```

## Variable-length Arguments

In some cases we cannot exactly tell how many parameters are needed in a function definition. In such cases we can use variable length arguments. Variable length arguments allows to pass random number of arguments in a function call. While creating a variable-length argument, the argument name must be preceded with \* symbol. Points to remember when working with variable-length arguments:

- The random arguments passed to the function forms a tuple.
- A for loop can be used to iterate over and access the elements in the variable-length argument.
- A variable-length argument should be at the end of the parameters list in the function definition.
- Other formal parameters written after variable-length argument must be keyword arguments only.

Following example demonstrates variable-length argument:

```
def fun(name, *friendslist):
    print("Friends of",name,"are: ")
    for x in friendslist:
        print(x, end = ' ')

fun("Shaa", "Ayaan", "Avi", "Velan")
```

Output of the above example is as follows:

```
Friends of Shaa are:
Ayaan Avi Velan
```

---

## Scope of Variables

Variables in a program has two things:

- Scope: Parts of the program in which it is accessible.
- Lifetime: How long a variable stays in the memory.

Based on scope of a variable, there are two types of variables:

1. Global variables
2. Local variables

Following are the differences between a global variable and local variable:

| Global Variable   | Local Variable  |
|---|---|
| Variable which is defined in the main body of the program file. | Variable which is defined inside a function.                                      |
| Accessible throughout the program file.                         | Accessible from the point it is defined to the end of the block it is defined in. |
| Accessible to all functions in the program.                     | They are not related in any way to other variables outside the function.          |

Following example demonstrates local and global variables:

```
x = 20
def fun():
    x = 10
    print("Local x =",x)
fun()
print("Global x =",x)
```

Output of above code is as follows:

```
Local x = 10
Global x = 20
```

### Global Statement

A local variable inside a function can be made *global* by using the *global* keyword. If a local variable which is made global using the *global* statement have the same name as another global variable, then changes on the variable will be reflected everywhere in the program.

Syntax of global statement is as follows:

```
global variable-name
```

Following example demonstrates global keyword:

```
x = 20
def fun():
    global x
    x = 10
    print("Local x =",x)
fun()
print("Global x =",x)
```

Output of the above program is:

```
Local x = 10
Global x = 10
```

In case of nested functions:

- Inner function can access variables in both outer and inner functions.
- Outer function can access variables defined only in the outer function.

### Anonymous Functions or LAMBDA Functions

Functions that don't have any name are known as lambda or anonymous functions. Lambda functions are created using the keyword *lambda*. Lambda functions are one line functions that can be created and used anywhere a function is required. Lambda is simply the name of letter 'L' in the Greek alphabet. A lambda function returns a function object.

A lambda function can be created as follows:

```
lambda arguments-list : expression
```

The arguments-list contains comma separated list of arguments. Expression is an arithmetic expression that uses the arguments in the arguments-list. A lambda function can be assigned to a variable to give it a name.

Following is an example for creating lambda function and using it:

```
✓ power = lambda x : x*x*x
  print(power(3))           #output:27

✓ x = lambda a : a + 10
  print(x(5))               #output:15

✓ x = lambda a, b : a * b
  print(x(5, 6))            #output:30

✓ x = lambda a, b, c : a + b + c
  print(x(5, 6, 2))        #output:13
```

In the above example 1 the lambda function is assigned to power variable. We can call the lambda function by writing power(3), where 3 is the argument that goes in to formal parameter x. Points to remember when creating lambda functions:

- Lambda functions doesn't have any name.
- Lambda functions can take multiple arguments.
- Lambda functions can returns only one value, the value of expression.
- Lambda function does not have any return statement.
- Lambda functions are one line functions.
- Lambda functions are not equivalent to inline functions in C and C++.
- They cannot access variables other than those in the arguments.
- Lambda functions cannot even access global variables.
- Lambda functions can be passed as arguments in other functions.
- Lambda functions can be used in the body of regular functions.
- Lambda functions can be used without assigning it to a variable.
- We can pass lambda arguments to a function.
- We can create a lambda function without any arguments.
- We can nest lambda functions.
- Time taken by lambda function to execute is almost similar to regular functions.

---

## Recursive Functions

Recursion is another way to repeat code in our programs. A recursive functions is a function which calls itself. A recursive functions should contain two major parts:

- base condition part: A condition which terminates recursion and returns the result.
- recursive part: Code which calls the function itself with reduced data or information.



Recursion uses divide and conquer strategy to solve problems.

```
if n>0      fact(n)=n*fact(n-1)
```

```
if n=0      fact(n)=1
```

```
5!=fact(5)=5*fact(4)
```

```
    5*4*fact(3)
```

```
    5*4*3*fact(2)
```

```
    5*4*3*2*fact(1)
```

```
    5*4*3*2*1*fact(0)
```

```
    5*4*3*2*1*1=120
```

### Advantages:

- 1) Recursive functions make the code look clean and easy
- 2) Complex task can be broken into sub problem using recursion
- 3) Sequence generation is easy

### Disadvantages:

- 1) Sometimes logic behind recursion is hard to follow
- 2) Recursive calls are expensive ,as they take a lot of memory and time.
- 3) Recursive functions are hard to debug

Following is an example for recursive function which calculates factorial of a number:

```
def fact(n):
    """Returns the factorial of a number
    using recursion."""
    if n==0 or n==1:
        return 1
    else:
        return n*fact(n-1)
```

```
n=int(input("enter n value"))
print("factorial=",fact(n))
```

In the above example text inside triple quotes is called docstring or documentation string. The docstring of function can be printed by writing `function_name.__doc__`

Following is an example for recursive function which calculates GCD of two numbers:

```
def gcd(x,y):
    rem = x%y
    if rem == 0:
        return y
    else:
        return gcd(y, rem)
```

```
x=int(input("enter the value of x"))
y=int(input("enter the value of y"))
print("Gcd of two numbers =",gcd(x,y))
```

---

## Modules: Creating modules, import statement, from. Import statement, name spacing.

### Modules

- A function allows to reuse a piece of code.
- A module on the other hand contains multiple functions, variables, and other elements which can be reused.
- A module is a Python file with .py extension. Each .py file can be treated as a module.

### Creation of Module:

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

### Use of Module:

```
import mymodule
mymodule.greeting("Ayaan")
```

Output:Hello,Ayaan

### Variables in Module

```
Mymodule1.py
person1 = {
    "name": "python",
    "age": 36,
    "country": "India"
}
```

### Import Statement:

Import keyword allows to use functions or variables available in module.

```
import mymodule1
a = mymodule1.person1["age"]
print(a)
```

### Renaming module:

```
import mymodule1 as mx
a = mx.person1["age"]
print(a)
```

### Built-in Modules

There are several built-in modules in Python

- import platform  
x = platform.system()  
print(x)
- import sys  
print(sys.path)
- import random  
print(random.choice(range(1,101)))

### **from...import Statement**

A module contains several variables and functions.

#### **Syntax:**

```
from module import variable/function
from math import pi
```

```
Ex:circle.py
from math import pi
def area(r):
    return pi*r*r*r
def peri(r):
    return 2*pi*r
```

```
test.py
import circle
r = int(input())
print("Area of circle is:", circle.area(r))
```

### **Namespace**

A namespace is a logical collection of names. Namespaces are used to eliminate name collisions. Python does not allow programmers to create multiple identifiers with same name. But, in some case we need identifiers having same name.

For example, consider two modules *module1* and *module* have the same function *sample()* as follows:

```
#Module 1
def sample():
    print("sample in module 1")
```

```
#Module 2
def sample():
    print("sample in module 2")
```

When we import both modules into same program and try to refer *sample()*, error will be generated:

```
import module1
import module2
sample()
```

output:Error

To avoid the error, we have to use fully qualified name as follows:

```
import module1
import module2
module1.sample()
module2.sample()
```

Each module has its own namespace.

### Built-in, Global, Local Namespaces

The *built-in namespace* contains all the built-in functions and constants from the `__builtin__` library. The *global namespace* contains all the elements in the current module. The *local namespace* contains all the elements in the current function or block. Runtime will search for the identifier first in the local, then in the global, then finally in the built-in namespace.

**UNIT – III: Python Data Structures and Modularization**

List and List Operations, Using Lists to represent Matrices, Strings, String operations, Tuples, Dictionaries, Sets, Iterators and generators, comprehensions. Basic math functions, User defined Functions, parameters to functions, positional, keyword and default arguments, Lambda Functions, recursion. Packages, modules and namespaces.

**Lists:**

- List is a data structure and is a collection of ordered & mutable(changeable)elements.
- List allows duplicate members.
- A list is created by placing all the items (elements) inside a square bracket [], separated by commas.
- List is dynamic. They grow and shrink on demand.
- It can have elements of same or different data types

Ex:

```
✓ a=[]
  print(a)
```

```
✓ a=[1,2,3]
  print(a)

  print(*a)
```

```
#List of integers
  a=[1,2,3]
  print(a)
```

```
#List with mixed datatypes
  a=[1,"hai",3.4]
  print(a)
```

**Operations:****1.concatenation:**

We can also use + operator to combine two lists. This is also called concatenation.

❖ **Syntax:**<list1>+<list2>

```
✓ l = [1, 3, 5]
print(l + [9, 7, 5])
```

# Output: [1, 3, 5, 9, 7, 5]

**2.multiplication or repitition:**

- ❖ **We can also use** \* operator repeats a list for the given number of times. This is also called Repetition

**syntax:** <list>\* <integer-value>

✓ print(["re"] \* 3)

#Output: ["re", "re", "re"]

### 3. List Index:

We can use the index operator [] to access an item in a list. Index starts from 0. So, a list having 5 elements will have index from 0 to 4.

✓ a=[1,2,3,4,5]

print(a[1])

print(a[3])

### 4. slicing:

Slicing is used to extract part of a list

We can access a range of elements in a list by using the slicing operator (colon).

l = ['a','b','c','d','e','f','g']

print(l)

print(l[1])

print(l[1:3])

print(l[1:-1])

print(l[:3])

print(l[2:])

print(l[:-1])

print(l[1::2])

print(l[::-1])

### 5. Membership Operator: we can find element exists in a list or not. it returns Boolean values Ex:

a=[1,2,3]

print(1 in a)

print(4 not in a)

### 6. Iteration: we can iterate each element in a list using for loop

**Syntax:** for i in list:

Ex: for i in [1,2,3]:

print(i)

### 7. list Length: It is used to find the length of the list.

**Syntax:** len(list)

Ex: l=[10,20,30]

print(len(l))

### list methods:

### 1.append():

The `append()` method adds a element to the end of the list.

The syntax of the `append()` method is:

**syntax:** `listname.append(element)`

```
✓ l=[2,4,6,10]
  l.append(12)
  print("updated list=",l)
  #output:[2,4,6,10,12]
✓ l1=['a','b','c']
  l1.append('d')
  print("updated list=",l1)
  #output:['a','b','c','d']
```

**2.insert():**The `insert()` method insert one element at a desired location .

**syntax:** `listname.insert(index,element)`

```
✓ l=[1,3,5]
  l.insert(1,7)
  print("updated list=",l)
  #output:[1,7,3,5]
```

**3.sort():**Sort elements in a list in ascending order

**syntax:** `listname.sort()`

```
Ex: l=[2,3,4,5,1]
    l.sort()
    print(l)
o/p: [1,2,3,4,5]
```

**4.count():** This method Returns the count of occurrence of an element in a list

**syntax:** `list.count(x)`

```
Ex: l=[1,2,3,3,4,5]
    print(l.count(3))
```

o/p:2

**5.index():**This method returns the index of the element.

Syntax: `listname.index(element)`

```
Ex: l=[1,2,3,4,7,10,13,11,12]
    print(l.index(13))
```

o/p: 6

**6.pop():**This method removes and returns last object from the list.

Syntax: `list.pop(element)`

```
Ex: l=[12,14,13,10]
```

```
print(l.pop()) #10 is removed
print(l) o/p:[12,14,13]
print(l.pop(1)) #14 is removed
print(l) o/p:[12,13]
```

**7.remove():**This method removes the first instance of element in list.

Syntax: **list.remove(element)**

```
Ex: l=[1,3,5,7,8,3]
     l.remove(3)
     print(l)
```

o/p:[1,5,7,8,3]

**8.reverse():**This method reverses the list elements.

Syntax: **list.reverse()**

```
Ex: l=[1,4,7,8,11,23]
     l.reverse()
     print(l)
```

o/p:[23,11,8,7,4,1]

**9.extend():**

The extend() extends the list by adding all items of a list (passed as an argument) to the end.

Syntax: **list1.extend(list2)**

Ex:

```
l1=[2,3,4]
l2=['a','b','c']
l1.extend(l2)
print(l1)
print(l2)
```

**10.copy():**make a copy of a list with the copy() method.

Syntax: **listname.copy()**

```
Ex: l1=[3,17,19,2,13]
     l2=l1.copy()
     print(l2)
```

### **Using Lists to represent Matrices**

In Python, we can implement a matrix as a nested list (list inside a list). We can treat each element as a row of the matrix.

For example  $X = [[1, 2], [4, 5], [3, 6]]$  would represent a 3x2 matrix. First row can be selected as  $X[0]$  and the element in first row, first column can be selected as  $X[0][0]$ . **1)#write a program to display given matrix**



```
X = [[12,7,3],
     [4 ,5,6],
     [7 ,8,9]]
print(X)
```

```
# iterate through rows
for i in range(len(X)):
    # iterate through columns
    for j in range(len(X[0])):
        print(X[i][j],end='\t')
    for r in result:
        print( )
```

2)#Write a program to add two matrices

```
X = [[12,7,3], [4 ,5,6],[7 ,8,9]]
Y = [[5,8,1],[6,7,3],[4,5,9]]
result = [[0,0,0],[0,0,0], [0,0,0]]
# iterate through rows
for i in range(len(X)):
    # iterate through columns
    for j in range(len(X[0])):
        result[i][j] = X[i][j] + Y[i][j]
        print(result[i][j],end='\t')
    print()
```

**# program to find norm of matrix**

```
x=[[1,2,3],[4,5,6],[7,8,9]]
sum=0
for i in range(len(x)):
    for j in range(len(x[i])):
        if(i==j):
            sum=sum+x[i][j]*x[i][j]
print(sum)
```

---

### Tuples:

Tuple is an ordered collection of elements.

Tuples are similar to list, but it is a sequence of immutable objects.

Tuples are fixed in length, heterogeneous, and can be nested.

Individual elements can be accessed through offset (index).

They are represented by ().

```
Ex1: t1=()
```

```
print(type(t1))
```

```
Ex2: t2=(1,2,3,4,5)
```

```
print(t2)
```

BV Prasanthi , Dept of CSE Page 5

**UNIT - 3 | 6**

```
Ex3: t3=(1,1.2,'abc')
```

```
print(t3)
```

### Basic tuple operations:

**1.concatenation:**we can use + operator to combine the two tuples.this is also called concatenation. **Syntax:** tuplename1+tuplename2

**Ex:** t1=(1,2,3)

```
t2=(4,5,6)
```

```
print(t1+t2)
```

**o/p:**(1,2,3,4,5,6)

**2.Repetition:**we can also repeat elements in a tuple for a given number of times using \* operator.

**Syntax:** tuple\*int-value

```
t=(1,2,3)
```

```
print(t*3)
```

**o/p:**(1,2,3,1,2,3,1,2,3)

**3.Tuple Length:**It is used to find the length of the tuple.

**Syntax:** len(tuple)

**Ex:** t=(10,20,30)

```
print(len(t))
```

**o/p:**3

**4. Index:**We can use the index operator [] to access an item in a tuple. Index starts from 0.

**Syntax:** tuplename[index ]

**Ex:** t=(1,2,3,4,5)

```
print(t[1]) #Output:2
```

```
print(t[3]) #Output:4
```

**5.Membership Operator:**we can find element exists in a tuple or not.it returns Boolean values

```
Ex: a=(1,2,3)
```

```
print(1 in a)
```

```
print(4 not in a)
```

**6.Iteration:** we can iterate each element in a tuple using for loop

**Syntax:** for i in tuple:

**Ex:** for i in (1,2,3):

```
print(i)
```

### 7.slicing:

Slicing is used to extract part of a tuple

We can access a range of elements in a list by using the slicing operator (colon).

```
t = ('a','b','c','d','e','f','g')
```

```
print(t)
```

```
print(t[1])
```

```
print(t[1:3])
```

```
print(t[1:-1])
```

```
print(t[:3])
```

```
print(t[2:])
```

```
print(t[:-1])
```

```
print(t[1::2])
```

```
print(t[::-1])
```

### Methods:

**Tuple does not support remove(),pop(),append(),sort(),reverse(),insert() ,extend()**

1) **index()**: This method returns the index of the element.

Syntax: tuplename.index(element)

Ex: t=(1,2,3,4,7,10,13,11,12)

```
print(t.index(13))
```

o/p: 6

2) **count()**: This method Returns the count of occurrence of an element in a list

**syntax:** tuplename.count(element)

Ex: t=(1,2,3,3,4,5)

```
print(t.count(3))
```

o/p:2

3) **copy**: Copying one tuple to another tuple is done by using assignment operator

```
t1=(1,2,3)
```

```
t2=t1
```

```
print(t2)
```

4) **zip()**:

It takes two or more sequence(list,tuple) and zip them into list of tuples or tuple of tuples.

```
✓ t1=(1,2,3)
```

```
t2=(5,7,8,9,10)
```

```
t3=tuple(zip(t1,t2))
```

```
print(t3) #output: ((1, 5), (2, 7), (3, 8))
```

```
✓ l1=[1,2,3]
```

```
l2=[5,7,8,9,10]
```

```
l3=list(zip(l1,l2))
```

```
print(t3) #output: [(1, 5), (2, 7), (3, 8)]
```

## Compare List and Tuple.

S.NO LIST TUPLE 1 Lists are mutable Tuple are immutable

BV Prasanthi , Dept of CSE Page 7  
UNIT - 3 | 8

S.NO LIST TUPLE

- 2 Implication of iterations is Time-consuming Implication of iterations is comparatively Faster
- 3 The list is better for performing operations, such as insertion and deletion. Tuple data type is appropriate for accessing the elements
- 4 Lists consume more memory Tuple consume less memory as compared to the list
- 5 Lists have several built-in methods Tuple does not have many built-in methods.
- 6 The unexpected changes and errors are more likely to occur In tuple, it is hard to take place.

---

## Sets:

- ✓ **Set:** A set is unordered collection of elements and it is mutable
- ✓ Every element is unique and must be immutable.
- ✓ Sets are represented by { } or by built in function set()

Ex:

```
✓ s={1,2,3}  
print(s)
```

```
✓ s1={"python",(1,2,3)}  
print(s1)
```

```
✓ s2=[[1,2,3],1.0,'hai']  
print(s2) #output:error because list is mutable,but set should have  
immutable elements
```

```
✓ s={3,2,3,1}  
print(s)
```

```
✓ s=set([10,20,30])
```

```
print(s)
```

### Set operations:

**1)Union(|):**It contains all the elements of set a and set b

```
a={1,2,3,4,5}
```

```
b={4,5,6,7,8}
```

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

**2)Intersection(&):**It contains common elements in both sets

```
a={1,2,3,4,5}
```

```
b={4,5,6,7,8}
```

BV Prasanthi , Dept of CSE Page 8

UNIT - 3 | 9

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

**3)Difference(-):**Difference of A & B contain only elements that are in A ,but not in B

```
a={1,2,3,4,5}
```

```
b={4,5,6,7,8}
```

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

**4)Set Symmetric difference(^):**Symmetric difference of a and b is set of elements in both a and b except that are common in both

```
a={1,2,3,4,5}
```

```
b={4,5,6,7,8}
```

```
print(a^b) #output: {1, 2, 3, 6, 7, 8}
```

**5)Membership operators:**It return Boolean values

```
A={'a',1,2.5,3}
```

```
Print('a' in A)
```

```
Print(1 not in A)
```

### Set Methods:

**1)add():**This method is used to add elements in a set

Syntax: setname.add(element)

```
Ex: s={'c','java','cpp'}
```

```
s.add('python')
```

```
print(s)
```

**2)clear():**This method is used to remove the elements from a set

Syntax: setname.clear()

```
Ex: v={1,2,3,4}
```

```
v.clear()
```

```
print(v)
```

**3)copy():**This method is used to copy the elements from one to another set

Syntax: setname.copy()

```
Ex: s={1,2,3}
      s1=s.copy()
      print(s1)
```

**4)difference():**Difference of A & B contain only elements that are in A ,but not in B

Syntax: set1.difference(set2)

```
Ex: a={1,2,3,4,5}
      b={4,5,6,7,8}
      print(a.difference(b))
```

**5)intersection()**It contains common elements in both sets

BV Prasanthi , Dept of CSE Page 9

UNIT - 3 | 10

Syntax: set1.intersection(set2)

```
Ex: a={1,2,3,4,5}
      b={4,5,6,7,8}
      print(a.intersection(b))
```

**6)union():**It contains all the elements of set a and set b

Syntax: set1.union(set2)

```
Ex: a={1,2,3,4,5}
      b={4,5,6,7,8}
      print(a.union(b))
```

**7)symmetric\_difference():**Symmetric difference of a and b is set of elements in both a and b except that are common in both

Syntax: set1.symmetric\_difference(set2)

```
Ex: a={1,2,3,4,5}
      b={4,5,6,7,8}
      print(a.symmetric_difference(b))
```

**8)discard():**It takes single element and removes it from set

Syntax: setname.discard(element)

```
Ex: n={1,2,3,4}
      n.discard(3)
      print(n)
```

**9)remove():**This method searches for given element in the set and removes it

Syntax: setname.remove(element)

```
Ex: n={1,2,3,4}
      n.remove(3)
      print(n)
```

✓ n={1,2}

n.remove(3) #keyerror

Note: If we try to remove the element in the set which is not present, it gives KeyError

**10)pop():** It removes random element in the set

Syntax: setname.pop()

Ex: s={'a','b','c'}

s.pop()

print(s)

**11)isdisjoint():**

Two sets are said to be disjoint, if they have no common elements (True/False)

Syntax: set1.isdisjoint(set2)

a={1,2,3}

b={4,5,6}

BV Prasanthi , Dept of CSE Page 10

UNIT - 3 | 11

print(a.isdisjoint(b))

**12)issubset():**

This method returns true if all elements of one set are present in second set.

Syntax: set1.issubset(set2)

a={1,2,3}

b={1,2,3,4}

print(a.issubset(b))

---

## **Dictionaries:**

- ✓ Python dictionary is an unordered collection of elements and it is mutable.
- ✓ We can store values as a pair of keys and values  
Represented by dict() or {}
- ✓ Keys in dictionary are unique and they are immutable (strings, numbers, tuples)

### **Creating a dictionary**

✓ d={}

print(d)

✓ d1={1:'Ay', 2:'aa', 3:'n'}

print(d1)

✓ d2={'a':1, 'b':2, 'c':3}

print(d2)

✓ d3={1:1.1, 2:'a', 3:[1,2,3], 4:(1,2,3)} #mixed dictionary

print(d3)

✓ d4=[[1,2,3]:1, 'a':2]

```
print(d4) #Error-keys are of immutable nature
✓ d=dict([('a',1),('b',2),('c',3)])
print(d)
```

### Dictionary Operations:

**1)index:**It is used to find the value at particular index

**Syntax:** dictname[index/key]

**Ex:** d={'a':1,'b':2,'c':3}

```
print(d['b']) #output:2
```

**2)Len:**It is used to find length of a dictionary

**Syntax:** len(dictname)

**Ex:** d={'a':1,'b':2,'c':3}

```
print(len(d)) #output:3
```

**3)Iteration:**Iteration is used to access the elements.

**Syntax:** for i in dictname:

**Ex:** s={1:1,3:9,5:25,7:49}

```
for i in s:
```

```
    print(s[i])
```

**output:1**

```
9
```

```
25
```

```
49
```

**4)Membership operator:** This operator returns Boolean values, to check whether the keys are present in dictionary or not

**Ex:** s={1:1,3:9,5:25,7:49}

```
print(1 in s)
```

```
print(48 not in s)
```

### Dictionary Methods

**1)keys():**It returns keys from the dictionary

**Syntax:** dictname.keys()

**Ex:** d2={'a':1,'b':2,'c':3}

```
print(d2.keys())
```

**2)values():**It returns values from the dictionary

**Syntax:** dictname.values()

**Ex:** d2={'a':1,'b':2,'c':3}

```
print(d2.values())
```

**3)get():**This method is used to read the value at particular index



Ex: `d2={'a':1,'b':2,'c':3}`  
`print(d2.get('a'))`

**4)update():**This method is used to change the values in a dictionary

**Syntax:** `olddict.update(newdict)`

Ex:

```
d={'o':1,'t':2,'th':3}
d1={'tw':2}
d.update(d1)
print(d)
```

**5)copy():**This method is used to copy a dictionary.

**Syntax:** `dictname.copy()`

Ex: `d2={'a':1,'b':2,'c':3}`  
`d1=d2.copy()`  
`print(d2)`

**6)clear():**It remove all the elements from a dictionary

**Syntax:** `dictname.clear()`

BV Prasanthi , Dept of CSE Page 12

**UNIT - 3 | 13**

Ex: `d={1:'one',2:'two'}`  
`d.clear()`  
`print(d)`

**7)pop():**This method is used to remove the keys from dictionary

**Syntax:** `dictname.pop(key)`

Ex: `f={1:'app',2:'man',3:'stra'}`  
`f.pop(1)`  
`print(f)`  
#output: `{2:'man',3:'stra'}`

---

## Iterators and generators

Note:For clear explanation refer to

<https://www.hackerearth.com/practice/python/iterators-and-generators/iterators-and-generators-1/tutorial/>

Iterators are containers for objects so that you can loop over the objects. In other words, you can run the "for" loop over the object. There are many iterators in the Python standard library. For example, list is an iterator and you can run a for loop over a list.

Python generator gives us an easier way to create python iterators. This is done by defining a function but instead of the return statement returning from the function, use the "yield" keyword.

For example, see how you can get a simple vowel generator below.

## Comprehensions

Comprehensions in Python provide us with a short and concise way to construct new sequences (such as lists, set, dictionary etc.) using sequences which have been already defined. Comprehensions are of three types:

- list comprehensions
- set comprehensions
- dict comprehensions

List comprehensions were introduced in Python 2.0; while set and dict comprehensions have been introduced in Python 2.7.

### List Comprehensions

BV Prasanthi , Dept of CSE Page 13

**UNIT - 3 | 14**

List comprehension is the most popular Python comprehension. It allows us to create a new list of elements that satisfy a condition from an iterable. An iterable is any Python construct that can be looped over like lists, strings, tuples, sets. In list comprehensions we use square brackets.

General syntax of a list comprehension is as follows:

```
[expression for ele1 in iterable1 if cond1
  for ele2 in iterable2 if cond2
  ...
  for eleN in iterableN if condN]
```

Following is an example of list comprehension for creating a list of squares of numbers in range 1-10:

```
✓ s=[x*x for x in range(1,11)]
  print(s)
```

In the previous example, we can also use if condition to generate only squares of even numbers:

```
✓ s=[x*x for x in range(1,11) if x%2==0]
```

Result of above comprehension will be:

```
[4, 16, 36, 64, 100]
```

## Set Comprehensions

Set comprehensions were added to python in version 2.7. In set comprehensions, we use the braces rather than square brackets. For example, to create the set of the squares of all numbers between 0 and 10 the following set comprehension can be used:

```
x = {i**2 for i in range(10)}
```

```
print(x)
```

```
output: {0, 1, 64, 4, 36, 9, 16, 49, 81, 25}
```

## Dict Comprehensions

Just like set comprehensions, dict comprehensions were added to python in version 2.7. Below we create a mapping of a number to its square using dict comprehension:

```
x = {i:i**2 for i in range(10)}
```

```
print(x)
```

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

## Introduction to Functions

- ✓ A function is a block of organized and reusable program code that performs a specific, single, and well-defined task.
- ✓ A function provides an interface for communication in terms of how information is transferred to it and how results are generated.

## Need for Functions

- ✓ Simplifies program development by making it easy to test separate functions.
  - ✓ Understanding programs becomes easier.
  - ✓ Libraries contain several functions which can be used in our programs to increase the productivity.
  - ✓ By dividing a large program into smaller functions, different programmers can work on different functions.
  - ✓ Users can create their own functions and use them in various locations in the main program.
- There are two types of functions:
- ✓ Built-in-functions

## ✓ User defined functions

### Built-in functions:

Python interpreter has number of functions that are available for use. They are called as Built-in functions.

Eg: input(), print(), range(), abs(), len(), type(), sum(), sorted(), etc.,

### User defined functions

## Defining Functions

A function definition consists of a function header that identifies the function, followed by the body of the function. The body of a function contains the code that is to be executed when a function is called. To define a function, we have to remember following points:

- Function definition starts with the keyword def
  - The keyword def is followed by the function name and parentheses.
  - After parentheses, a colon (:) should be placed.
  - Parameters or arguments that the function accepts should be placed inside the parentheses.
- A function might have a return statement.

BV Prasanthi , Dept of CSE Page 15

UNIT - 3 | 16

- The function code should be indented properly.

A function definition contains two parts:

- Function header
- Function body

The syntax of a function definition is as follows:

```
def function_name(arg1, arg2, ..no  
arg): """documentation string"""  
    statement block
```

Function Header , formal      parameters      Body of function

Example for defining a function without arguments is as follows:

```
✓ def sample():  
    """ This is sample program created by B.V.Prashanthi """  
    print("hello")  
    print("python")
```

Example for defining a function with arguments is as follows:

```
✓ def add(a,b):  
    """ This is sample program for addition """  
    print(a+b)
```

## Calling a Function

Before calling a function, it should be defined. We call a function to execute the code inside the function.

Syntax for a function call is as follows:

```
function_name( )           sample()  
function_name( )           add(a,b)  
function_name( [arg1, arg2, ...] )  
                                Function Call, Actual parameters
```

Eg:

The arguments or parameters passed in a function call are called actual parameters. The arguments used in the function header of a function definition are called formal parameters. When a function is called, control moves to the first line inside the function definition. After the body of function definition is executed, control returns back to the next statement after the function call. Points to remember while calling a function:

BV Prasanthi , Dept of CSE Page 16

**UNIT - 3 | 17**

- The function name and number of parameters must be same in the function call and function definition.
- When the number parameters passed doesn't match with the parameters in the function definition, error is generated.
- Names of arguments in the function call and function definition can be different. • Arguments can be passed as expressions. The expression will get executed first and then the value is passed to the formal parameter.
- The parameter list must be separated by commas.
  - If the function returns a value it must be assigned to some variable in the calling function.

Let's consider the following example which demonstrates function definition and function call: #program without parameters

```
✓ def sample():  
    """ This is sample program created by B.V.Prashanthi """  
    print("hello")  
    print("python")
```

```

sample()
#program with parameters
✓ def add(a,b):
    """ This is sample program for addition"""
    print(a+b)

a=10
b=20
add(a,b)
#different parameters in function header and function call
✓ def add(x,y):
    """ This is sample program for addition"""
    print(x+y)

a=10
b=20
add(a,b)

```

## Functions Returning Value(Fruitful Functions)

A function may or may not return a value. To return a value, we must use *return* statement in the function definition. Every function by default contains an implicit *return* statement as the last line which returns *None* object to the calling function. The functions which returns a value is called fruitful functions.

A return statement is used for two reasons:

- ✓ Return a value to the caller.
- ✓ To end the execution of a function and give control to caller.

The syntax of *return* statement is as follows:

```
return()
```

BV Prasanthi , Dept of CSE Page 17  
UNIT - 3 | 18

```
return (expression)
```

Example of a function returning a value is as follows:

```

✓ def cube(x):
    return x*x*x
a=int(input("enter the value of a"))
b=cube(a)
print("Cube=",b)

✓ def sq(x):
    return x*x
x=int(input("enter the value of x"))
b=sq(x)
print("Square=",b)

```

## Passing Arguments

If a function definition contains arguments, then in a function call there is a need to pass arguments. For example, in the previous function definition of *cube*, there is a single argument. So, while calling this function we need to pass a single parameter. The *cube* function can be called as follows:

```
cube(10)
```

In the above function call, 10 is the actual parameter.

---

## Positional Arguments

Positional arguments are arguments that can be called by their position in the function definition. First positional argument always needed to be listed first when the function is called. Second positional argument always needed to be listed second when the function is called and so on. Eg:

```
✓ def f(a,b,c):  
    print(a,b,c) #output:1 2 3  
f(1,2,3)
```

```
✓ def f(a,b,c):  
    print(a*b+c) #output:5  
f(1,2,3)
```

```
✓ def f(a,b,c):  
    print(a+b/c) #output:2  
f(1,1,1)
```

```
✓ def add(a,b):  
    print(a+b)
```

```
a=10
```

BV Prasanthi , Dept of CSE Page 18

UNIT - 3 | 19

```
b=20
```

```
add(a,b)
```

## Default Arguments

The formal parameters in a function definition can be assigned a default value. Such parameters to which default values are assigned are called default arguments.

- ✓ Default value can be assigned to a parameter by using the assignment ( = ) operator.
- ✓ A function definition can have one or more default arguments.
- ✓ Default arguments allow a function call to pass less parameters than the number of parameters in the function definition. (no.of parameters in function call and function

header need not be same)

Following example demonstrates a function with default arguments and its usage:

```
✓ def f(a,b=2,c=3):  
    print(a,b,c) #output:1 2 3  
f(1)
```

```
✓ def f(a,b=2,c=3):  
    print(a,b,c) #output:1 2 3  
f(a=1)
```

```
✓ def f(a,b=2,c=3):  
    print(a,b,c) #output:1 3 3  
f(1,3)
```

```
✓ def f(a,b=2,c=3):  
    print(a,b,c) #output:1 2 6  
f(1,c=6)
```

```
✓ def f(a,b=2,c=3):  
    print(a,b,c) #output:2 3 4  
f(2,3,4)
```

## Keyword Arguments

In general, when parameters are passed in a function call, they are assigned to formal parameters based on their position. Such parameters are called positional parameters. We can also pass the parameters based on the name (keyword) of the parameter in the function definition. Such parameters are called keyword arguments.

Points to remember when using keyword arguments:

- All keyword parameters must match one of the parameters in the function definition.
- Order of keyword arguments is not important.
- A value should not be passed more than once to a keyword parameter.

Following example demonstrates keyword arguments and their use:

```
✓ def f(a,b,c):  
    print(a,b,c) #output:1 2 3  
f(1,2,3)
```

```
✓ def f(a,b,c):  
    print(a,b,c) #output:1 1 1  
f(a=1,b=1,c=1)
```

```
✓ def f(a,b,c):  
    print(a,b,c) #output:1 2 3
```



*f(c=3,b=2,a=1)*

✓ *def f(a,b,c):*  
    *print(a,b,c) #output:1 2 8*  
    *f(1,c=8,b=2)*

✓ *def f(a,b,c):*  
    *print(a,b,c) #output:1 2 9*  
    *f(1,2,c=9)*

✓ *def f(a,b,c):*  
    *print(a,b,c) #output:Error*  
    *f(1,b=2)*

✓ *def f(a,b,c):*  
    *print(a,b,c) #output:Error*  
    *f(a=1,b=2,a=3)*

## Variable-length Arguments

In some cases we cannot exactly tell how many parameters are needed in a function definition. In such cases we can use variable length arguments. Variable length arguments allows to pass random number of arguments in a function call. While creating a variable-length argument, the argument name must be preceded with \* symbol. Points to remember when working with variable-length arguments:

- The random arguments passed to the function forms a tuple.
- A for loop can be used to iterate over and access the elements in the variable-length argument.
- A variable-length argument should be at the end of the parameters list in the function definition.
- Other formal parameters written after variable-length argument must be keyword arguments only.

BV Prasanthi , Dept of CSE Page 20

**UNIT - 3 | 21**

Following example demonstrates variable-length argument:

```
def fun(name, *friendslist):  
    print("Friends of",name,"are: ")  
    for x in friendslist:  
        print(x, end = ' ')
```

*fun("Shaa", "Ayaan", "Avi", "Velan")*

Output of the above example is as follows:

## Scope of Variables

Variables in a program has two things:

- Scope: Parts of the program in which it is accessible.
- Lifetime: How long a variable stays in the memory.

Based on scope of a variable, there are two types of variables:

1. Global variables
2. Local variables

Following are the differences between a global variable and local variable:

| Global Variable   | Local Variable  |
|---|---|
| Variable which is defined in the main body of the program file. | Variable which is defined inside a function.                                      |
| Accessible throughout the program file.                         | Accessible from the point it is defined to the end of the block it is defined in. |
| Accessible to all functions in the program.                     | They are not related in any way to other variables outside the function.          |

Following example demonstrates local and global variables:

```
x = 20
def fun():
    x = 10
    print("Local x =",x)
fun()
print("Global x =",x)
```

Output of above code is as follows:

```
Local x = 10
Global x = 20
```

## **Global Statement**

A local variable inside a function can be made *global* by using the *global* keyword. If a local variable which is made global using the *global* statement have the same name as another global variable, then changes on the variable will be reflected everywhere in the program.

Syntax of global statement is as follows:

*global variable-name*

Following example demonstrates global keyword:

```
x = 20
def fun():
    global x
    x = 10
    print("Local x =",x)
fun()
print("Global x =",x)
```

Output of the above program is:

```
Local x = 10
Global x = 10
```

In case of nested functions:

- Inner function can access variables in both outer and inner functions.
- Outer function can access variables defined only in the outer function.

## Anonymous Functions or LAMBDA Functions

Functions that don't have any name are known as lambda or anonymous functions. Lambda functions are created using the keyword *lambda*. Lambda functions are one line functions that can be created and used anywhere a function is required. Lambda is simply the name of letter 'L' in the Greek alphabet. A lambda function returns a function object.

A lambda function can be created as follows:

*lambda arguments-list : expression*

The arguments-list contains comma separated list of arguments. Expression is an arithmetic expression that uses the arguments in the arguments-list. A lambda function can be assigned to a variable to give it a name.

Following is an example for creating lambda function and using it:

```
✓ power = lambda x : x*x*x  
print(power(3)) #output:27
```

```
✓ x = lambda a : a + 10  
print(x(5)) #output:15
```

```
✓ x = lambda a, b : a * b  
print(x(5, 6)) #output:30
```

```
✓ x = lambda a, b, c : a + b + c  
print(x(5, 6, 2)) #output:13
```

In the above example 1 the lambda function is assigned to power variable. We can call the lambda function by writing power(3), where 3 is the argument that goes in to formal parameter x. Points to remember when creating lambda functions:

- Lambda functions doesn't have any name.
- Lambda functions can take multiple arguments.
- Lambda functions can returns only one value, the value of expression.
- Lambda function does not have any return statement.
- Lambda functions are one line functions.
- Lambda functions are not equivalent to inline functions in C and C++.
- They cannot access variables other than those in the arguments.
- Lambda functions cannot even access global variables.
- Lambda functions can be passed as arguments in other functions.
- Lambda functions can be used in the body of regular functions.
- Lambda functions can be used without assigning it to a variable.
- We can pass lambda arguments to a function.
- We can create a lambda function without any arguments.
- We can nest lambda functions.
- Time taken by lambda function to execute is almost similar to regular functions.

—

## Recursive Functions

Recursion is another way to repeat code in our programs. A recursive functions is a function which calls itself. A recursive functions should contain two major parts:

- base condition part: A condition which terminates recursion and returns the result.
- recursive part: Code which calls the function itself with reduced data or information.

Recursion uses divide and conquer strategy to solve problems.

if  $n > 0$   $\text{fact}(n) = n * \text{fact}(n-1)$

if  $n = 0$   $\text{fact}(n) = 1$

$5! = \text{fact}(5) = 5 * \text{fact}(4)$

$5 * 4 * \text{fact}(3)$

$5 * 4 * 3 * \text{fact}(2)$

$5 * 4 * 3 * 2 * \text{fact}(1)$

$5 * 4 * 3 * 2 * 1 * \text{fact}(0)$

$5 * 4 * 3 * 2 * 1 * 1 = 120$

### Advantages:

- 1) Recursive functions make the code look clean and easy
- 2) Complex task can be broken into sub problem using recursion
- 3) Sequence generation is easy

### Disadvantages:

- 1) Sometimes logic behind recursion is hard to follow
- 2) Recursive calls are expensive ,as they take a lot of memory and time.
- 3) Recursive functions are hard to debug

Following is an example for recursive function which calculates factorial of a number:

```
def fact(n):  
    """Returns the factorial of a number  
    using recursion."""  
    if n==0 or n==1:  
        return 1  
    else:  
        return n*fact(n-1)
```

```
n=int(input("enter n value"))  
print("factorial=",fact(n))
```

In the above example test inside triple quotes is called docstring or documentation string. The docstring of function can be printed by writing `function_name.__doc__`

Following is an example for recursive function which calculates GCD of two numbers:

```
def gcd(x,y):  
    rem = x%y  
    if rem == 0:  
        return y  
    else:  
        return gcd(y, rem)
```

```
x=int(input("enter the value of x"))
y=int(input("enter the value of y"))
print("Gcd of two numbers =",gcd(x,y))
```

---

## Modules: Creating modules, import statement, from. Import statement, name spacing.

### Modules

- A function allows to reuse a piece of code.
- A module on the other hand contains multiple functions, variables, and other elements which can be reused.
- A module is a Python file with .py extension. Each .py file can be treated as a module.

### Creation of Module:

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

### Use of Module:

```
import mymodule
mymodule.greeting("Ayaan")
Output:Hello,Ayaan
```

### Variables in Module

```
Mymodule1.py
person1 = {
    "name": "python",
    "age": 36,
    "country": "India"
}
```

### Import Statement:

Import keyword allows to use functions or variables available in module.

```
import mymodule1
a = mymodule1.person1["age"]
print(a)
```

### Renaming module:

```
import mymodule1 as mx
a = mx.person1["age"]
print(a)
```

### Built-in Modules

There are several built-in modules in Python

- import platform  
    x = platform.system()  
    print(x)
- import sys  
    print(sys.path)
- import random  
    print(random.choice(range(1,101)))

### **from...import Statement**

A module contains several variables and functions.

#### **Syntax:**

```
from module import variable/function
from math import pi
```

```
Ex:circle.py
from math import pi
def area(r):
    return pi*r*r*r
def peri(r):
    return 2*pi*r
```

```
test.py
import circle
r = int(input())
print("Area of circle is:", circle.area(r))
```

### **Namespace**

A namespace is a logical collection of names. Namespaces are used to eliminate name collisions. Python does not allow programmers to create multiple identifiers with same name. But, in some case we need identifiers having same name.

For example, consider two modules *module1* and *module* have the same function *sample()* as follows:

```
#Module 1
def sample():
    print("sample in module 1")
```

```
#Module 2
def sample():
    print("sample in module 2")
```

When we import both modules into same program and try to refer *sample()*, error will be generated:

```
import module1
import module2
sample()
output:Error
```

To avoid the error, we have to use fully qualified name as follows:

```
import module1
import module2
module1.sample()
module2.sample()
```

Each module has its own namespace.

### **Built-in, Global, Local Namespaces**

The *built-in namespace* contains all the built-in functions and constants from the `__builtin__` library. The *global namespace* contains all the elements in the current module. The *local namespace* contains all the elements in the current function or block. Runtime will search for the identifier first in the local, then in the global, then finally in the built-in namespace.





**UNIT-IV: File Handling**

Files, Types of Files, Creating and Reading Text Data, File Methods to Read and Write Data, Reading and Writing Binary Files, The Pickle Module, Reading and Writing CSV Files, Python os and os.path Modules

**Files:**

A file is a location on disk that stores related information and has a name. There are different kinds of file such as music files, video files, text files. Python gives easy ways to manipulate these files. Generally we divide files in two categories, text file and binary file. Text files are simple text where as the binary files contain binary data which is only readable by computer.

File Input/Output Operations:

- 1) Open a file
- 2) Close a file
- 3) Read a file
- 4) Write a file

**Creating and Reading of Text Data:**

To read and write to a text file ,we must open it

**1)Opening a file:**

Syntax:

```
filepointer=open(filename,mode)
```

Filename can be text file with extension(.txt)

Modes:

"r" - Read - Opens a file for reading, error if the file does not exist

"a" - Append - Opens a file for appending

"w" - Write - Opens a file for writing/overwriting

"x" - Create - Creates the specified file, returns an error if the file exists

**Ex:**

```
f=open("demo.txt",'r')
f=open("demo.txt",'w')
f=open("demo.txt",'a')
```

**2)Close a file:**

Syntax: filepointer.close()

Ex: f=open('demo.txt','r')

```
print(f.read())
f.close()
```

**b)Read a text file:**read() method is used for reading the content in the file.

**#creating demo.txt file**

```
gedit demo.txt
```

```
Python is interesting
programming language.
```

**#Write a python program to read data in demo.txt**

```
f=open('demo.txt','r')
```

```
print(f.read())
```

```
f.close()
```

**#Read only part of a file:**

```
f=open('demo.txt','r')
```

```
print(f.read(3)) #returns first three characters
```

```
f.close()
```

**#Readlines:**

```
f=open('demo.txt','r')
```

```
print(f.readline()) #read only first line
```

```
f.close()
```

**Read all lines at a time:**

```
f=open('demo.txt','r')
```

```
print(f.readlines())
```

```
f.close()
```

**c)Write a file:**

The write() method is used to Write data to a empty file or existing file.

It has two modes:

**"w"** - Write - Opens a file for writing data into empty file (or)overwriting the data in the existing file

**"a"** - Append - Opens a file for appending the data at end of the file

Syntax:

```
Filepointer.write('Write the data which u need to append or overwrite')
```

```
Ex: gedit empty.txt # creation of empty file
```

**#Write a program to write data into empty file**

```
f=open('empty.txt','w')
```

```
f.write("I am writing data into the empty text file")
```

```
f.close()
```

#To view the content in the empty.txt

```
f=open('empty.txt','r')
print(f.read())
f.close()
```

Output: I am writing data into the empty text file

#Write a program to write data into existing file demo.txt

```
f=open("demo.txt",'w')
f.write("I am overwriting the existing content")
f.close()
```

#To view the content in the demo.txt

```
f=open("demo.txt",'r')
print(f.read())
```

output: I am overwriting the existing content

# write a program to Append data in empty.txt

```
f=open("empty.txt",'a')
f.write("python is very easy")
f.close()
```

#To view the content in the empty.txt

```
f=open("empty.txt",'r')
print(f.read())
```

output: I am writing data into the empty text file. python is very easy

### Methods():

1)tell():Returns current file location

Syntax: filepointer.tell()

Eg:

```
f=open('demo.txt','r')
print(f.read())
print(f.tell())
print(f.read(2))
f.close()
```

2)seek():

changes the filepointer position to offset bytes

Syntax:filepointer.seek(offset,from(optional))

Eg: fseek(10) #filepointer moves by position 10

fseek(10,0) #moving filepointer by 10 position from **begin** of file

fseek(10,1) #moving filepointer by 10 position from **current position**

seek(10,2) #moving filepointer by 10 position from **end** of file

3)read()

4)readline()

## Reading and writing of binary files:

### opening a file:

```
filepointer=open(filename,mode)
```

Filename with extension(.bin)

Modes:

"rb" - Read - Opens a binary file for reading

"wb" - write - Opens a binary file for writing /overwriting

Eg: f=open('demo.bin','rb')

### #creating demo.bin file

```
gedit demo.bin
```

```
10
```

```
20
```

```
a
```

```
b
```

```
c
```

### #Write a python program to read the data in demo.bin file

```
f=open('demo.bin','rb')
```

```
print(f.read())
```

```
f.close()
```

### # creating empty.bin file

```
gedit empty.bin
```

### # write a python program to write the data into empty.bin

```
f=open('empty.bin','wb')
```

```
l=[5,6,7]
```

```
l1=bytearray(l)
```

```
f.write(l1)
```

```
f.close()
```

### #to view the contents in empty.bin

```
f=open('empty.bin','rb')
```

```
print(f.read())
```

```
f.close()
```

### #To overwrite the content in existing file demo.bin

```
f=open('demo.bin','wb')
```

```
l=[5,6,7]
```

```
l1=bytearray(l)
```

```
f.write(l1)
```

```
f.close()
#to view the contents in demo.bin
f=open('demo.bin','rb')
print(f.read())
f.close()
```

### CSV (comma Separated value) files

A CSV (comma-separated values) is a file containing text that is separated with a comma. It is a type of plain text that uses specific structuring to arrange tabular data. CSV files are used to handle large amount of data. Eg: Spreadsheets

#### Syntax:

```
col1,          col2,          col3
first row data1, first row data2, first row data3
second row data1, second row data2, second row data3
```

#### Reading and writing of CSV files:

```
#create demo.csv file
```

```
gedit demo.csv
```

```
SN,ROLLNO,NAME
```

```
1,401,ayaan
```

```
2,402,riyaan
```

#### #Write a python program to read the data in demo.csv

```
import csv
```

```
with open('demo.csv', 'r') as file:
```

```
    reader = csv.reader(file)
```

```
    for row in reader:
```

```
        print(row)
```

#### #output:

```
['SN','ROLLNO','NAME']
```

```
['1','401','ayaan']
```

```
['2','402','riyaan']
```

#### #creating empty.csv file

```
gedit empty.csv
```

#### #write the data into empty.csv file

```
import csv
```

```
with open ("empty.csv","w") as file:
```

```
    f=csv.writer(file)
```

```
    f.writerow(['s.No','rollno','name'])
```

```

        f.writerow([1,101,'aaa'])
        f.writerow([2,102,'bbb'])
#write the data into demo.csv file
import csv
with open ("demo.csv","w") as file:
    f=csv.writer(file)
    f.writerow(['s.No','rollno','name'])
    f.writerow([1,101,'aaa'])
    f.writerow([2,102,'bbb'])

```

## The Pickle Module

Python pickle module is used for serializing and de-serializing a Python object structure. Any object in Python can be pickled so that it can be saved on disk. What pickle does is that it “serializes” the object first before writing it to file. Pickling is a way to convert a python object (list, dict, etc.) into a character stream. The idea is that this character stream contains all the information necessary to reconstruct the object in another python script.

```

import pickle
n = int(input('Enter the number of data : '))
l = []
for i in range(n):
    a = input('Enter data '+str(i)+' : ')
    l.append(a)
file = open('demo1.bin', 'wb')
pickle.dump(l, file)
file.close()

```

### output:

```

Enter the number of data : 3
Enter data 0 : python
Enter data 1 : is
Enter data 2 : interesting

```

### Python Pickle load

To retrieve pickled data, we have to use pickle.load()

```

import pickle
file = open('demo1.bin', 'rb')
l = pickle.load(file)
file.close()
print('Showing the pickled data:')
c = 0

```

```
for i in l:
    print('The data ', c, ' is : ', i)
    c = c + 1
```

#### output:

Showing the pickled data:  
 The data 0 is : t  
 The data 1 is : h  
 The data 2 is : e

**Note:**For clear explanation refer to

<https://www.journaldev.com/15638/python-pickle-example#python-pickle-example>

### Python os and os.path Modules

It is possible to automatically perform many operating system tasks. The OS module in Python provides functions for creating and removing a directory (folder), fetching its contents, changing and identifying the current directory, etc.

We first need to import the os module to interact with the underlying operating system. So, import it using the import os statement before using its functions.

#### 1)Getting Current Working Directory

The getcwd() function used to print the current working directory

Control+Alt+t:A terminal is opened and type python

```
>>> import os
>>> os.getcwd()
output:./tmp/guest-iwWXNr
```

#### 2) Creating a Directory

We can create a new directory using the os.mkdir() function, as shown below.

```
>>> import os
>>> os.mkdir("Shanthi")
output:A directory named Shanthi will be created on the Desktop
```

#### 3) List Files and Sub-directories

The listdir() function returns the list of all files and directories in the specified directory.

Now create the files named file1 and file2 in the directory Shanthi.

```
>>> import os
>>> os.listdir("Shanthi")
output:['File1','File2']
```

#### 4) Removing a Directory

The rmdir() function in the OS module removes the specified directory

```
>>> import os
```



```
>>> os.rmdir("Shanthi")
```

output:Shanthi directory on the Desktop will be removed

### OS Path module in Python

This module contains some useful functions on pathnames.

1. **os.path.basename(path)** : This function basically return the file name from the path given.

```
import os
l = os.path.basename("Shanthi/file1")
print(l)
output: file1
```

2. **os.path.dirname(path)** : It is used to return the directory name from the path given.

```
import os
l = os.path.dirname("Shanthi/file1")
print(l)
output:Shanthi
```

3. **os.path.isdir(path)** : This function specifies whether the path is existing directory or not.

```
import os
l = os.path.isdir("Shanthi/file1")
print(l)
output:True
```

4. **os.path.isfile(path)** : This function specifies whether the path is existing file or not.

```
import os
l = os.path.isfile("Shanthi/file1")
print(l)
output:True
```

# UNIT-5

## Object Oriented Programming In Python:

Like other general purpose languages, python is also an object-oriented language since its beginning. Python is an object-oriented programming language. It allows us to develop applications using an Object Oriented approach. In Python, we can easily create and use classes and objects.

Major principles of object-oriented programming system are given below.

- Object
- Class
- Method
- Inheritance
- Polymorphism
- Encapsulation

### Object

Object is simply a collection of data (variables) and methods (functions) that act on those data.

### Class

class is a blueprint for the object.

Like function definitions begin with the keyword `def`, in Python, we define a class using the keyword `class`.

### Syntax

1. `class` ClassName:
2.     <statement-1>
3.     .
4.     .
5.     <statement-N>

The first string is called docstring and has a brief description about the class.

```
class MyNewClass:
```

```
'''This is a docstring. I have created a new class'''  
pass
```

### Example:-

```
class MyClass:  
    "This is my second class"  
    a = 10  
    def func(self):  
        print('Hello')  
  
print(MyClass.a)  
print(MyClass.func)  
print(MyClass.__doc__)
```

### Output:-

```
10  
  
<function MyClass.func at 0x0000000003079BF8>  
  
'This is my second class'
```

## Creating an Object in Python

We saw that the class object could be used to access different attributes.

It can also be used to create new object instances (instantiation) of that class. The procedure to create an object is similar to a [function](#) call.

```
>>> ob = MyClass()
```

### Example:

```
class MyClass:  
    "This is my second class"  
    a = 10  
    def func(self):
```

```
print('Hello')

ob = MyClass()

print(MyClass.func)

print(ob.func)

ob.func()

Output:

<function MyClass.func at 0x000000000335B0D0>

<bound method MyClass.func of <__main__.MyClass object at
0x000000000332DEF0>>

Hello
```

You may have noticed the `self` parameter in function definition inside the class but, we called the method simply as `ob.func()` without any [arguments](#). It still worked.

This is because, whenever an object calls its method, the object itself is passed as the first argument. So, `ob.func()` translates into `MyClass.func(ob)`

## Methods

Methods are functions defined inside the body of a class. They are used to define the behaviors of an object.

```
class Parrot

    def __init__(self, name, age):

        self.name = name

        self.age = age

    def sing(self, song):
```

```

        return "{} sings {}".format(self.name, song)

    def dance(self):

        return "{} is now dancing".format(self.name)

blu = Parrot("Blu", 10)

print(blu.sing(" 'Happy' "))

print(blu.dance())

```

Output:-

Blu is 'Happy'

Blu is now dancing

## Inheritance

Inheritance is a way of creating new class for using details of existing class without modifying it. The newly formed class is a derived class (or child class). Similarly, the existing class is a base class (or parent class).

## Encapsulation

Using OOP in Python, we can restrict access to methods and variables. This prevent data from direct modification which is called encapsulation. In Python, we denote private attribute using underscore as prefix i.e single “\_” or double “\_\_”.

## Polymorphism

Polymorphism is an ability (in OOP) to use common interface for multiple form (data types).

Suppose, we need to color a shape, there are multiple shape option (rectangle, square, circle). However we could use same method to color any shape. This concept is called Polymorphism.

# Python exec() Function

Example

```
x = 'name = "John"\nprint(name)'\nexec(x)
```

Output:

John

## Definition and Usage

The `exec()` function executes the specified Python code.

The `exec()` function accepts large blocks of code, unlike the `eval()` function which only accepts a single expression

## Syntax:

```
exec(object, globals, locals)
```

## Parameter Values

| Parameter      | Description   |
|----------------|---|
| <i>object</i>  | A String, or a code object                          |
| <i>globals</i> | Optional. A dictionary containing global parameters |
| <i>locals</i>  | Optional. A dictionary containing local parameters  |

# Regular Expression in Python:

Module Regular Expressions(RE) specifies a set of strings(pattern) that matches it.

To understand the RE analogy, MetaCharacters are useful, important and will be used in functions of module re.

There are a total of 14 metacharacters and will be discussed as they follow into functions:

```
\  Used to drop the special meaning of character
    following it (discussed below)
[]  Represent a character class
^  Matches the beginning
$  Matches the end
.  Matches any character except newline
?  Matches zero or one occurrence.
|  Means OR (Matches with any of the characters
    separated by it.
*  Any number of occurrences (including 0 occurrences)
+  One or more occurrences
{}  Indicate number of occurrences of a preceding RE
    to match.
()  Enclose a group of REs
```

- **Function compile()**

Regular expressions are compiled into pattern objects, which have methods for various operations such as searching for pattern matches or performing string substitutions.

```
import re
p = re.compile('[a-e]')
print(p.findall("Aye, said Mr. Gibenson Stark"))
```

Output:

['e', 'a', 'd', 'b', 'e', 'a']

Metacharacter backslash '\ ' has a very important role as it signals various sequences. If the backslash is to be used without its special meaning as metacharacter, use '\\'

```
\d  Matches any decimal digit, this is equivalent
     to the set class [0-9].
\D  Matches any non-digit character.
\s  Matches any whitespace character.
\S  Matches any non-whitespace character
\w  Matches any alphanumeric character, this is
     equivalent to the class [a-zA-Z0-9_].
\W  Matches any non-alphanumeric character.
```

### Example Programs:-

```
1) import re
    p = re.compile('\d')
    print(p.findall("I went to him at 11 A.M. on 4th July 1886"))
    p = re.compile('\d+')
    print(p.findall("I went to him at 11 A.M. on 4th July 1886"))
```

Output:-

```
['1', '1', '4', '1', '8', '8', '6']
```

```
['11', '4', '1886']
```

```
2) import re
    p = re.compile('\w')
    print(p.findall("He said * in some_lang."))
    p = re.compile('\w+')
    print(p.findall("I went to him at 11 A.M., he said *** in
some_language."))
    p = re.compile('\W')
    print(p.findall("he said *** in some_language."))
```

Output:

```
['H', 'e', 's', 'a', 'i', 'd', 'i', 'n', 's', 'o', 'm', 'e', '_', 'l',
'a', 'n', 'g']
```

```
['I', 'went', 'to', 'him', 'at', '11', 'A', 'M', 'he', 'said', 'in',
'some_language']
```

```
[' ', ' ', '*', '*', '*', ' ', ' ', '.']
```

```
3) import re
```



```
p = re.compile('ab*')
print(p.findall("ababbaabbb"))
```

Output:

```
['ab', 'abb', 'a', 'abbb']
```

### Function split()

Split string by the occurrences of a character or a pattern, upon finding that pattern, the remaining characters from the string are returned as part of the resulting list.

**Syntax :**

```
re.split(pattern, string, maxsplit=0, flags=0)
```

**Example Program:**

```
from re import split
print(split('\W+', 'Words, words , Words'))
print(split('\W+', "Word's words Words"))
print(split('\W+', 'On 12th Jan 2016, at 11:02 AM'))
print(split('\d+', 'On 12th Jan 2016, at 11:02 AM'))
```

Output:

```
['Words', 'words', 'Words']
['Word', 's', 'words', 'Words']
['On', '12th', 'Jan', '2016', 'at', '11', '02', 'AM']
['On ', 'th Jan ', ', at ', ':', ' AM']
```

**Example-2:**

```
import re
print(re.split('\d+', 'On 12th Jan 2016, at 11:02 AM', 1))
print(re.split('[a-f]+', 'Aey, Boy oh boy, come here', flags = re.IGNORECASE))
print(re.split('[a-f]+', 'Aey, Boy oh boy, come here'))
```

- **Output:**

- ['On ', 'th Jan 2016, at 11:02 AM']
- ['', 'y, ', 'oy oh ', 'oy, ', 'om', ' h', 'r', '']
- ['A', 'y, Boy oh ', 'oy, ', 'om', ' h', 'r', '']

### Function sub()

**Syntax:**

- `re.sub(pattern, repl, string, count=0, flags=0)`

Example:

```
import re
print(re.sub('ub', '~*', 'Subject has Uber booked already', flags = re.IGNORECASE))
print(re.sub('ub', '~*', 'Subject has Uber booked already'))
print(re.sub('ub', '~*', 'Subject has Uber booked already', count=1, flags = re.IGNORECASE))
print(re.sub(r'\sAND\s', ' & ', 'Baked Beans And Spam', flags=re.IGNORECASE))
```

## Output

```
S~*ject has ~*er booked already
S~*ject has Uber booked already
S~*ject has Uber booked already
Baked Beans & Spam
```

## Function subn()

subn() is similar to sub() in all ways, except in its way to providing output. It returns a tuple with count of total of replacement and the new string rather than just the string.

## Example:

```
import re
print(re.subn('ub', '~*' , 'Subject has Uber booked already'))
t = re.subn('ub', '~*' , 'Subject has Uber booked already', flags =
re.IGNORECASE)
print(t)
print(len(t))
print(t[0])
```

## Output:

```
('S~*ject has Uber booked already', 1)
('S~*ject has ~*er booked already', 2)
```

Length of Tuple is: 2

S~\*ject has ~\*er booked already

## Function escape()

### Syntax:

```
re.escape(string)
```

## Example:

```
import re
print(re.escape("This is Awseome even 1 AM"))
print(re.escape("I Asked what is this [a-9], he said \t ^WoW"))
```

## Output:

This\ is\ Awseome\ even\ 1\ AM

I\ Asked\ what\ is\ this\ \[a\ -9\]\,\ he\ said\ \ \ \ \ ^WoW

# Python Exception

An exception can be defined as an unusual condition in a program resulting in the interruption in the flow of the program.

Whenever an exception occurs, the program stops the execution, and thus the further code is not executed. Therefore, an exception is the run-time errors that are unable to handle to Python script. An exception is a Python object that represents an error

Python provides a way to handle the exception so that the code can be executed without any interruption. If we do not handle the exception, the interpreter doesn't execute all the code that exists after the exception.

Python has many **built-in exceptions** that enable our program to run without interruption and give the output. These exceptions are given below:

## Common Exceptions

Python provides the number of built-in exceptions, but here we are describing the common standard exceptions. A list of common exceptions that can be thrown from a standard Python program is given below.

1. **ZeroDivisionError:** Occurs when a number is divided by zero.
2. **NameError:** It occurs when a name is not found. It may be local or global.
3. **IndentationError:** If incorrect indentation is given.
4. **IOError:** It occurs when Input Output operation fails.
5. **EOFError:** It occurs when the end of the file is reached, and yet operations are being performed.

## The problem without handling exceptions

### Example

```
a = int(input("Enter a:"))  
b = int(input("Enter b:"))
```

```
c = a/b
```

```
print("a/b =",c)
```

```
#other code:
```

```
print("Hi I am other part of the program")
```

Output:

Enter a:10

Enter b:0

**Traceback (most recent call last):**

**File "<string>", line 3, in <module>**

**ZeroDivisionError: division by zero**

The above program is syntactically correct, but it through the error because of unusual input. That kind of programming may not be suitable or recommended for the projects because these projects are required uninterrupted execution. That's why an exception-handling plays an essential role in handling these unexpected exceptions. We can handle these exceptions in the following way.

## Exception handling in python

If the Python program contains suspicious code that may throw the exception, we must place that code in the **try** block.

The **try** block must be followed with the **except** statement, which contains a block of code that will be executed if there is some exception in the try block.



### Syntax

**try:**

**#block of code**

**except Exception1:**

**#block of code**

```
except Exception2:
    #block of code
    #other code
```

## Example1

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    c = a/b
except:
    print("Can't divide with zero")
```

Output

Enter a:10

Enter b:0

Can't divide with zero

## Example2

The `try` block will generate an exception, because `x` is not defined: `try`:

```
print(x)
except:
    print("An exception occurred")
```

Output:

An exception occurred

**Since the try block raises an error, the except block will be executed.**

**Without the try block, the program will crash and raise an error:**

# Many Exceptions

You can define as many exception blocks as you want, e.g. if you want to execute a special block of code for a special kind of error:

The Python allows us to declare the multiple exceptions with the except clause. Declaring multiple exceptions is useful in the cases where a try block throws multiple exceptions. The syntax is given below.

## Example1

Print one message if the try block raises a **NameError** and another for other errors:

```
try:
    print(x)
except NameError:
    print("Variable x is not defined")
except:
    print("Something else went wrong")
```

Output:

```
Variable x is not defined
```

## Example2

```
try:
    #print(x)
    a=10/0;
except NameError:
    print("Name Error")
except ArithmeticError:
    print("Arithmetic Exception")
```

Output:

```
Arithmetic Exception
```

### Example3

```
try:
    a=10/0;
except(ArithmeticError, IOError):
    print("Arithmetic Exception")
```

**Output:**  
**Arithmetic Exception**  
**Else**

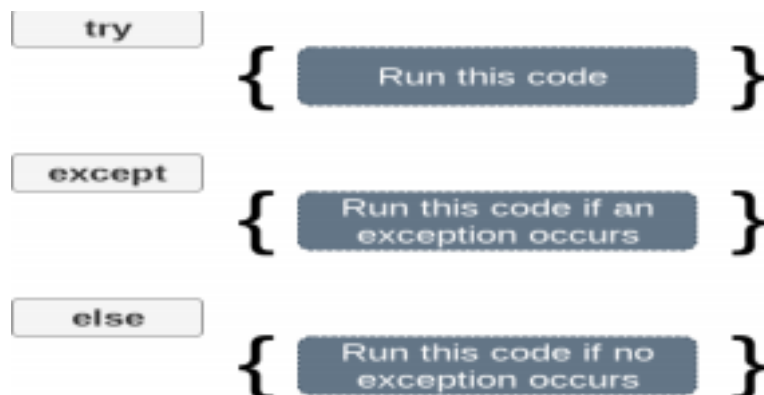
You can use the `else` keyword to define a block of code to be executed if no errors were raised:

The syntax to use the `else` statement with the `try-except` statement is given below.

```
try:
    #block of code

except Exception1:
    #block of code

else:
    #this code executes if no except block is executed
```



### Example-1

In this example, the `try` block does not generate any error:

```
try:
    print("Hello")
except:
    print("Something went wrong")
else:
    print("Nothing went wrong")
```

Output:

```
Hello
Nothing went wrong
```

Example-2 `try` block generates the error

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    c = a/b
    print("a/b =",c)

except Exception:
    print("can't divide by zero")
    print(Exception)
else:
    print("Hi I am else block No Error")
```

Output:

```
Enter a:10
Enter b:0
can't divide by zero
<class 'Exception'>
```



### Example-3 try block does not generates the error

**try:**

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

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

```
c = a/b
```

```
print("a/b =",c)
```

**except Exception:**

```
print("can't divide by zero")
```

```
print(Exception)
```

**else:**

```
print("Hi I am else block No Error")
```

**Output:**

**Enter a:10**

**Enter b:2**

**a/b = 5.0**

**Hi I am else block No Error**

**Note : The except statement with no exception**

**Python provides the flexibility not to specify the name of exception with the exception statement.**

**try:**

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

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

```
c = a/b;
```

```
print("a/b = %d"%c)
```

**except:**

```
print("can't divide by zero")
```

**else:**

```
print("Hi I am else block")
```

**Output:**

**Enter a:10**

**Enter b:0**

**can't divide by zero**

**Note2:** The except statement using with exception variable

**try:**

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

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

```
c = a/b
```

```
print("a/b =",c)
```

**except Exception as e:**

```
print("can't divide by zero")
```

```
print(e)
```

**else:**

```
print("Hi I am else block")
```

**Output:**

**Enter a:10**

**Enter b:0**

**can't divide by zero**

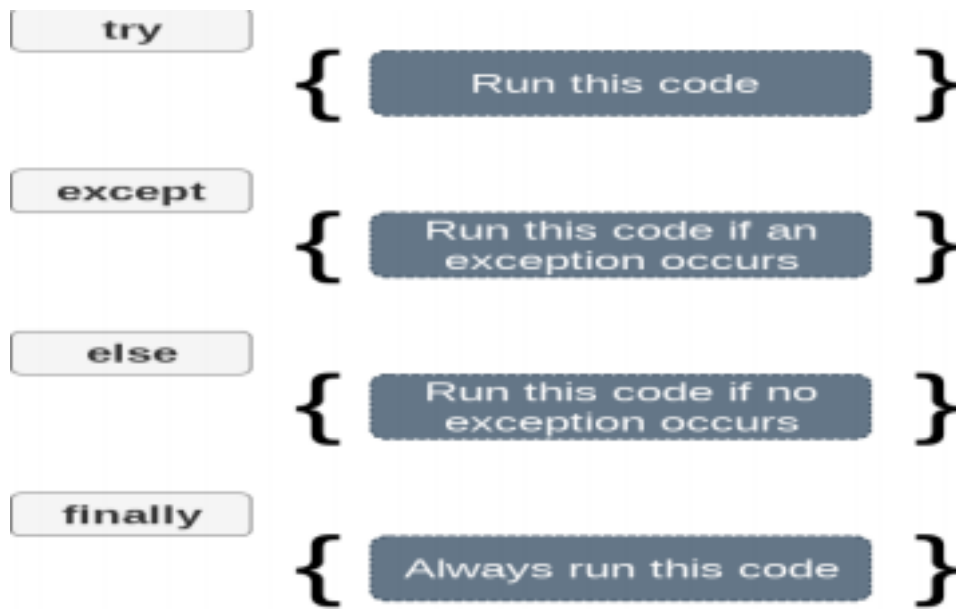
**division by zero**

## Points to remember

1. **Python facilitates us to not specify the exception with the except statement.**
2. **We can declare multiple exceptions in the except statement since the try block may contain the statements which throw the different type of exceptions.**
3. **We can also specify an else block along with the try-except statement, which will be executed if no exception is raised in the try block.**
4. **The statements that don't throw the exception should be placed inside the else block.**

## The try...finally block

- The `finally` block, if specified, will be executed regardless if the try block raises an error or not.
- **Python provides the optional finally statement, which is used with the try statement.**
- **It is executed no matter what exception occurs and used to release the external resource.**
- **The finally block provides a guarantee of the execution. We can use the finally block with the try block in which we can place the necessary code, which must be executed before the try statement throws an exception.**



## Syntax

**try:**

# block of code

# this may throw an exception

**finally:**

# block of code

# this will always be executed

## Example-1

**try:**

```
print(x)
```

**except:**

```
print("Something went wrong")
```

**finally:**

```
print("The 'try except' is finished")
```

Output:

```
Hello
```

```
Something went wrong
```

## Example-2

**Try to open and write to a file that is not writable:**

```
try:
    f = open("demofile.txt")
    f.write("Lorum Ipsum")
except:
    print("Something went wrong when writing to the file")
finally:
    f.close()
```

**Output:**

```
Something went wrong when writing to the file
```

# Raise an exception

you can choose to throw an exception if a condition occurs.

To throw (or raise) an exception, use the `raise` keyword.

## Example-1

Raise an error and stop the program if x is lower than 0:

```
x = -1

if x < 0:

    raise Exception("Sorry, no numbers below zero")
```

Output:

```
Traceback (most recent call last):
  File "demo_ref_keyword_raise.py", line 4, in <module>
    raise Exception("Sorry, no numbers below zero")
Exception: Sorry, no numbers below zero
```

The `raise` keyword is used to raise an exception.

You can define what kind of error to raise, and the text to print to the user.

## Example-2

Raise a `TypeError` if x is not an integer:

```
x = "hello"

if not type(x) is int:
    raise TypeError("Only integers are allowed")
```

Output:

```
Traceback (most recent call last):
  File "demo_ref_keyword_raise2.py", line 4, in <module>
    raise TypeError("Only integers are allowed")
TypeError: Only integers are allowed
```

## Example-3

`try:`

```
age = int(input("Enter the age:"))  
  
if(age<18):  
    raise ValueError  
  
else:  
    print("the age is valid")  
  
except ValueError:  
    print("The age is not valid")
```

**Output:**

```
Enter the age:17  
The age is not valid
```

**Example 4 Raise the exception with message**  
**try:**

```
num = int(input("Enter a positive integer: "))  
  
if(num <= 0):  
  
# we can pass the message in the raise statement  
raise ValueError("That is a negative number!")  
  
except ValueError as e:  
  
    print(e)
```

**Output:**

```
Enter a positive integer: -5  
That is a negative number!
```

**Example 5**

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    if b is 0:
        raise ArithmeticError
    else:
        print("a/b = ",a/b)
except ArithmeticError:
    print("The value of b can't be 0")
```

**Output:**

**Enter a:10**

**Enter b:0**

**The value of b can't be 0**