UNIT-1: Arrays, Structure & Union and User defined function in C programming Language:

1.1 Concepts of Two-Dimensional Numeric Array:

1.1.1 Declaring Two-Dimensional numeric array:

The two-dimensional array can be defined as an array of arrays. The 2D array is organized as matrices which can be represented as the collection of rows and columns. However, 2D arrays are created to implement a relational database look alike data structure. It provides ease of holding the bulk of data at once which can be passed to any number of functions wherever required.

Declaration of two dimensional Array in C:

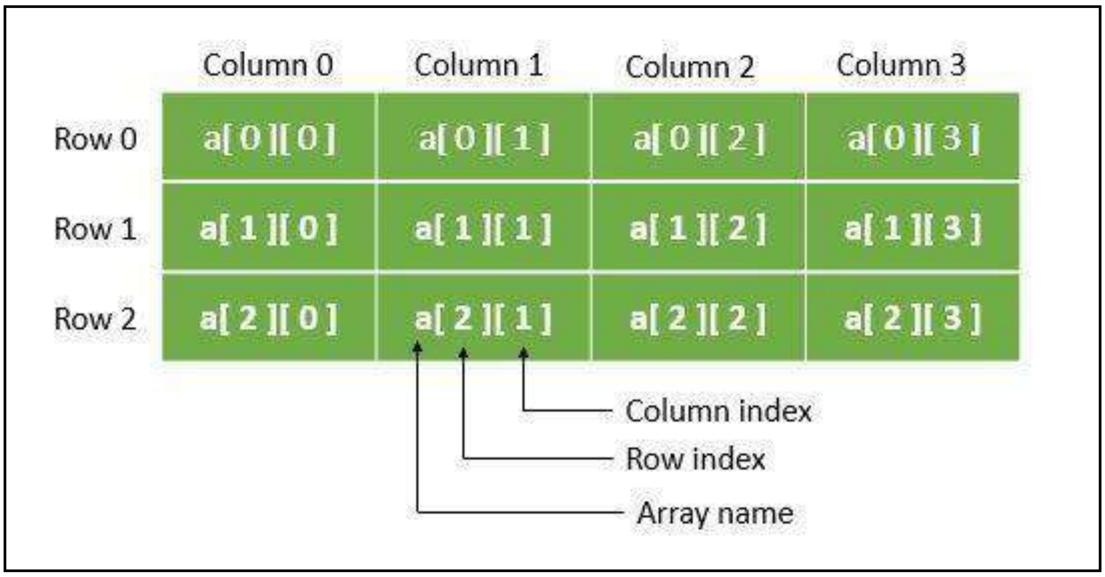
The syntax to declare the 2D array is given below.

```
data_type array_name[rows][columns];
```

Consider the following example.

int x[3][4];

Here, **x** is a two-dimensional (2D) array. The array can hold 12 elements. You can think the array as a table with 3 rows and each row has 4 columns.



Initialization of a 2D array

There are many different ways to initialize two-dimensional array

```
int c[2][3] = {{1, 3, 0}, {-1, 5, 9}};

int c[][3] = {{1, 3, 0}, {-1, 5, 9}};

int c[2][3] = {1, 3, 0, -1, 5, 9};

int arr[4][3]={{1,2,3},{2,3,4},{3,4,5},{4,5,6}};
```

Two-dimensional array example in C

```
#include<stdio.h>
#include<conio.h>
int main()
{
    int i=0,j=0;
    int arr[4][3]={{1,2,3},{2,3,4},{3,4,5},{4,5,6}};    clrscr();
//traversing 2D array
    for(i=0;i<4;i++)      // rows
    {
       for(j=0;j<3;j++)      // cols
       {
          printf("arr[%d] [%d] = %d \n",i , j, arr[i][j]);
        }  //end of j
    }//end of i
  getch(); return 0;
}
```

OUTPUT:

```
arr[0][0] = 1
arr[0][1] = 2
arr[0][2] = 3
arr[1][0] = 2
arr[1][1] = 3
arr[1][2] = 4
arr[2][0] = 3
arr[2][1] = 4
arr[2][2] = 5
arr[3][0] = 4
arr[3][1] = 5
arr[3][2] = 6
```

Example 1: Two-dimensional array to store and print values

```
// C program to store temperature of two cities of a week and display it.
#include <stdio.h>
const int CITY = 2;
const int WEEK = 7;

int main()
{
  int temperature[CITY][WEEK];

  // Using nested loop to store values in a 2d array
  for (int i = 0; i < CITY; ++i)
  {
    for (int j = 0; j < WEEK; ++j)
    {
      printf("City %d, Day %d: ", i + 1, j + 1);
      scanf("%d", &temperature[i][j]);
    }
  }
  printf("\nDisplaying values: \n\n");
  // Using nested loop to display vlues of a 2d array
  for (int i = 0; i < CITY; ++i)
  {
    for (int j = 0; j < WEEK; ++j)
    {
       printf("City %d, Day %d = %d\n", i + 1, j + 1, temperature[i][j]);
    }
  }
    return 0;
}
```

Output:

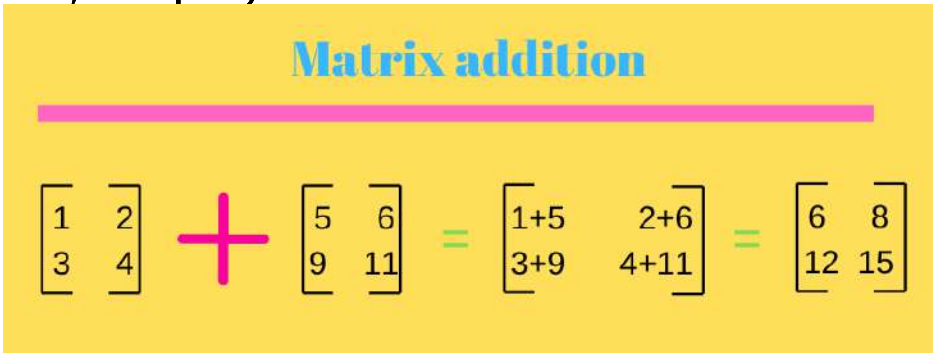
```
City 1, Day 1: 33
City 1, Day 2: 34
City 1, Day 3: 35
City 1, Day 4: 33
City 1, Day 5: 32
City 1, Day 6: 31
City 1, Day 7: 30
City 2, Day 1: 23
City 2, Day 2: 22
City 2, Day 3: 21
City 2, Day 4: 24
City 2, Day 5: 22
City 2, Day 6: 25
City 2, Day 7: 26

Displaying values:

City 1, Day 1 = 33
City 1, Day 2 = 34
City 1, Day 3 = 35
City 1, Day 4 = 33
City 1, Day 5 = 32
City 1, Day 6 = 31
City 1, Day 7 = 30
City 2, Day 1 = 23
City 2, Day 2 = 22
City 2, Day 3 = 21
City 2, Day 4 = 24
City 2, Day 5 = 22
City 2, Day 6 = 25
City 2, Day 7 = 26
```

Example 2: 2D array example:- Storing elements in a matrix and printing it.

```
#include<stdio.h>
#include<conio.h>
void main ()
{
   int arr[3][3],i,j;    clrscr();
   for (i=0;i<3;i++)
   {
      for (j=0;j<3;j++)
      {
         printf("Enter a[%d][%d]: ",i,j);
         scanf("%d",&arr[i][j]);
      }
   }
   printf("\n Printing the elements.... \n");
   for(i=0;i<3;i++)
   {
      printf("\n");
      for (j=0;j<3;j++)
      {
         printf("%d\t",arr[i][j]);
      }
   }     getch();
}
```

Output:
```
Enter a[0][0]: 56
Enter a[0][1]: 10
Enter a[0][2]: 30
Enter a[1][0]: 34
Enter a[1][1]: 21
Enter a[1][2]: 34
Enter a[2][0]: 45
Enter a[2][1]: 56
Enter a[2][2]: 78

Printing the elements ....

56      10      30
34      21      34
45      56      78
```

1.1.2 Two-Dimensional numeric Array operations (Addition, Subtraction, Multiplication, Transpose)



$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 5 & 6 \\ 9 & 11 \end{bmatrix} = \begin{bmatrix} 1+5 & 2+6 \\ 3+9 & 4+11 \end{bmatrix} = \begin{bmatrix} 6 & 8 \\ 12 & 15 \end{bmatrix}$$

Matrix addition in C language to add two matrices, i.e., compute their sum and print it. A user inputs their orders (number of rows and columns) and the matrices. For example, if the order is 2, 2, i.e., two rows and two columns and the matrices are:
First matrix:
1 2
3 4
Second matrix:
4 5
-1 5
The output is:
5 7
2 9

1. Two-Dimensional numeric Array operations: Addition of two Matrix

```
/*Addition of two matrix in C */
#include<stdio.h>
#include<conio.h>
int main()
{
     int a[3][3],b[3][3],sum[3][3],i,j;
     clrscr();

     printf("Enter Matrix A Elements: ");
     for(i=0;i<3;i++)// rows
     {
          for(j=0;j<3;j++)    //cols
          {
            printf("\n Enter Array element A[%d][%d] :",i,j);
          scanf("%d",&a[i][j]);
        }
    }
    printf("Enter Matrix B Elements: ");
    for(i=0;i<3;i++)    // rows
    {
        for(j=0;j<3;j++)    //cols
        {
          printf("\n Enter Array element B[%d][%d] :",i,j);
          scanf("%d",&b[i][j]);
        }
    }
     printf("\n Matrix A : \n");
     printf("~~~~~~~~~~~~~~~~~~~~\n");
     for(i=0;i<3;i++)
     {
        for(j=0;j<3;j++)
        {
          printf(" %d ",a[i][j]);
        }
        printf("\n");
     }

     printf("\n Matrix B : \n");
     printf("~~~~~~~~~~~~~~~~~~~~\n");
     for(i=0;i<3;i++)
     {
        for(j=0;j<3;j++)
        {
          printf(" %d ",b[i][j]);
        }
        printf("\n");
     }
     printf("\n Sum of two matrices : \n");
     printf("~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");
     for(i=0;i<3;i++)
     {
        for(j=0;j<3;j++)
        {
           sum[i][j]=a[i][j]+b[i][j];
           printf("%d\t",sum[i][j]);
         }
            printf("\n");
     }
     getch();     return 0;
}
```

2. Two-Dimensional numeric Array operations: Subtraction of two Matrix.

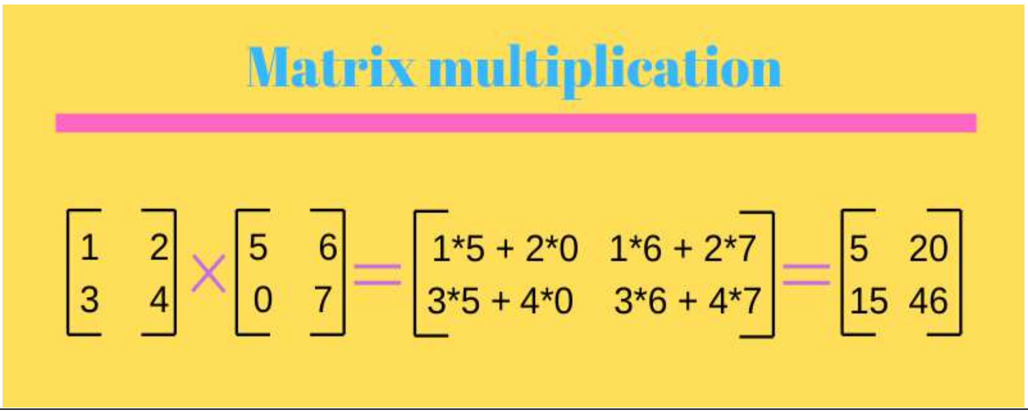
```
/* Subtraction of two matrix in C */
#include<stdio.h>
#include<conio.h>

int main()
{
    int a[3][3],b[3][3],sub[3][3],i,j;
    clrscr();
```

```
    printf("Enter Matrix A Elements: ");
    for(i=0;i<3;i++)// rows
    {
        for(j=0;j<3;j++)    //cols
        {
          printf("\n Enter Array element A[%d][%d] :",i,j);
          scanf("%d",&a[i][j]);
        }
    }
    printf("Enter Matrix B Elements: ");
    for(i=0;i<3;i++)    // rows
    {
        for(j=0;j<3;j++)    //cols
        {
          printf("\n Enter Array element B[%d][%d] :",i,j);
          scanf("%d",&b[i][j]);
        }
    }
     printf("\n Matrix A : \n");
     printf("~~~~~~~~~~~~~~~~~~~~~~~\n");
     for(i=0;i<3;i++)
     {
        for(j=0;j<3;j++)
        {
          printf(" %d ",a[i][j]);
        }
        printf("\n");
     }
     printf("\n Matrix B : \n");
     printf("~~~~~~~~~~~~~~~~~~~~~~~\n");
     for(i=0;i<3;i++)
     {
        for(j=0;j<3;j++)
        {
          printf(" %d ",b[i][j]);
        }
        printf("\n");
     }

     printf("\n Subtraction of two matrices : \n");
     printf("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");
     for(i=0;i<3;i++)
     {
        for(j=0;j<3;j++)
        {
          sub[i][j]=a[i][j]-b[i][j];
          printf("%d\t", sub[i][j]);
        }
        printf("\n");
     }
      getch();        return 0;
}
```

3. Two-Dimensional numeric Array operations: Multiplication of two Matrix.
Matrix multiplication in C language to calculate the product of two matrices (two-dimensional arrays). A user inputs the orders and elements of the matrices. If the multiplication isn't possible, an error message is displayed.

Some points important points for matrix multiplication:

- This program asks the user to enter the size (rows and columns) of two matrices.
- To multiply two matrices, the number of columns of the first matrix should be equal to the number of rows of the second matrix.
- The program below asks for the number of rows and columns of two matrices until the above condition is satisfied.
- For matrix multiplication, the number of columns in the first matrix must be equal to the number of rows in the second matrix.
- The result matrix has the number of rows of the first and the number of columns of the second matrix.

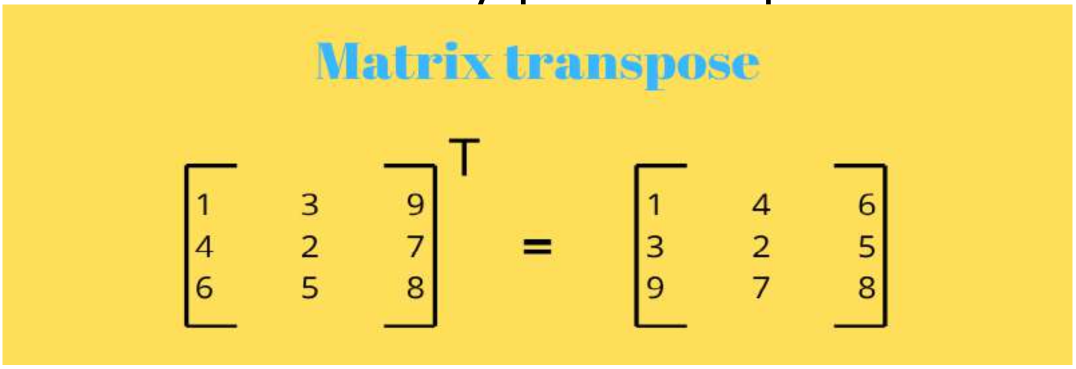


$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 5 & 6 \\ 0 & 7 \end{bmatrix} = \begin{bmatrix} 1*5 + 2*0 & 1*6 + 2*7 \\ 3*5 + 4*0 & 3*6 + 4*7 \end{bmatrix} = \begin{bmatrix} 5 & 20 \\ 15 & 46 \end{bmatrix}$$

```
      /* C program to find multiplication of two matrices */
#include<stdio.h>
#include<conio.h>
int main()
{
   int m1[10][10],m2[10][10],mul[10][10],r1,c1,r2,c2,i,j,k;
   clrscr();

   printf("Enter rows and column for the first matrix: ");
   scanf("%d %d", &r1, &c1);
   printf("Enter rows and column for the second matrix: ");
   scanf("%d %d", &r2, &c2);

  // Taking input until 1st matrix columns is not equal to 2nd matrix row
  while(c1 != r2)
  {
      printf("Error!!!! Enter rows and columns again.\n");
      printf("Enter rows and columns for the first matrix: ");
      scanf("%d %d", &r1, &c1);
      printf("Enter rows and columns for the second matrix: ");
      scanf("%d %d", &r2, &c2);
  }
 printf("\nEnter elements for First Matrix: \n");
 for(i=0;i<r1;i++)
 {
   for(j=0;j<c1;j++)
   {
```

```
            printf("Enter m1[%d][%d] : ", i, j);
            scanf("%d", &m1[i][j]);
        }
    }
    printf("\nEnter elements for Second Matrix: \n");
    for(i = 0; i < r2;i++)
    {
        for(j = 0; j < c2; j++)
        {
            printf("Enter m2[%d][%d] : ", i, j);
            scanf("%d", &m2[i][j]);
        }
    }
    for (i = 0; i < r1;i++)      // first matrix row
    {
        for (j = 0; j < c2; j++)   // second matrix column
        {
            mul[i][j] = 0;
            for (k = 0; k < r2; k++)
            {
                mul[i][j] = mul[i][j] + ( m1[i][k] * m2[k][j] );
            }
        }
    }
    printf("\n First Matrix : \n");
    for(i=0;i < r1;i++)
    {
        for(j=0;j < c1;j++)
        {
            printf("%d\t", m1[i][j]);
        }
        printf("\n");
    }
    printf("\n Second Matrix:\n");
    for(i=0;i < r2;i++)
    {
        for(j=0;j < c2;j++)
        {
            printf("%d\t", m2[i][j]);
        }
        printf("\n");
    }
    printf("\nMultiplication of Matrices : \n");
    for(i=0;i < r1;i++)
    {
        for(j=0;j < c2;j++)
        {
            printf("%d\t", mul[i][j]);
        }
        printf("\n");
    }
    getch();
    return 0;
}
```

4. Two-Dimensional numeric Array operations: Transpose of two Matrix.



$$\begin{bmatrix} 1 & 3 & 9 \\ 4 & 2 & 7 \\ 6 & 5 & 8 \end{bmatrix}^T = \begin{bmatrix} 1 & 4 & 6 \\ 3 & 2 & 5 \\ 9 & 7 & 8 \end{bmatrix}$$

Transpose of a matrix in C language: This C program prints transpose of a matrix. To obtain it, we interchange rows and columns of the matrix. For example, consider the following 3 X 2 matrix:
1 2
3 4
5 6
Transpose of the matrix:
1 3 5
2 4 6
When we transpose a matrix, its order changes, but for a square matrix, it remains the same.

```
/* C program to find transpose of 3 X 3 matrix  */
#include<stdio.h>
#include<conio.h>
int main()
{
        int a[3][3],i,j;
       clrscr();
       printf("Enter Matrix A Elements: ");
       for(i=0;i<3;i++)     // rows
       {
           for(j=0;j<3;j++)     //cols
           {
             printf("\n Enter Array element A[%d][%d] :",i,j);
             scanf("%d",&a[i][j]);
           }
       }
       printf("\n Matrix A : \n");
       printf("~~~~~~~~~~~~~~~~~~~~~~~\n");
       for(i=0;i<3;i++)
       {
           for(j=0;j<3;j++)
           {
               printf("%d\t",a[i][j]);
           }
           printf("\n");
       }
       printf("\n Transpose of matrix A : \n");
       printf("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");
       for(i=0;i<3;i++)
       {
```

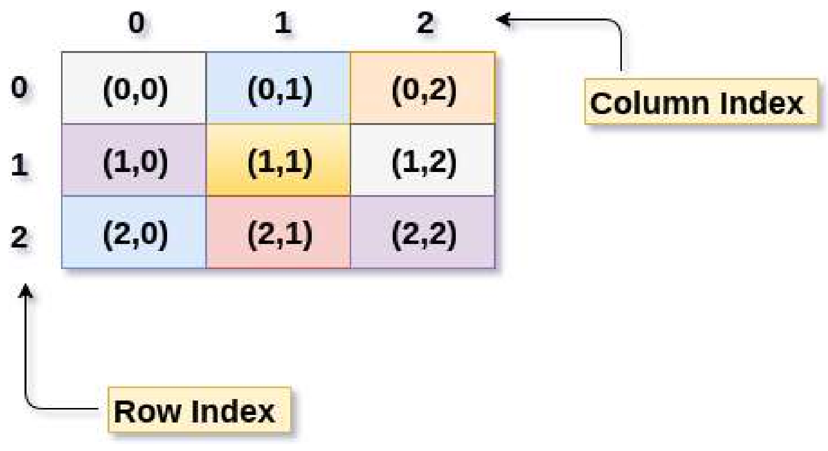
```
            for(j=0;j<3;j++)
            {
                    printf("%d\t",a[j][i]);
            }
            printf("\n");
        }
        getch();  return 0;
}
```

1.1.3 Element Address in array (Row major and Column major)

2D arrays are created to implement a relational database table look alike data structure, in computer memory, the storage technique for 2D array is similar to that of an one dimensional array.

The size of a two dimensional array is equal to the multiplication of number of rows and the number of columns present in the array. We do need to map two dimensional array to the one dimensional array in order to store them in the memory.

A 3 X 3 two dimensional array is shown in the following image. However, this array needs to be mapped to a one dimensional array in order to store it into the memory.



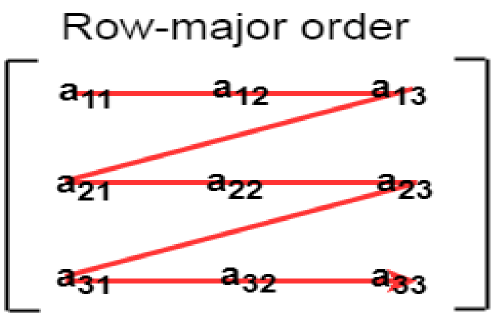
A two dimensional Array A is the collection of 'm X n' elements. Programming language stores the two dimensional array in one dimensional memory in either of two ways - There are two main techniques of storing 2D array elements into memory

1. Row Major ordering

In row major ordering, all the rows of the 2D array are stored into the memory contiguously. Considering the array shown in the above image, its memory allocation according to row major order is shown as follows.

(0,0)	(0,1)	(0,2)	(1,0)	(1,1)	(1,2)	(2,0)	(2,1)	(2,2)

first, the 1st row of the array is stored into the memory completely, then the 2nd row of the array is stored into the memory completely and so on till the last row.



First row of the array occupies the first set of memory locations reserved for the array; Second row occupies the next set, and so forth.
To determine element address A[i,j]:

Location (A[i,j]) =Base Address + (N x (i - 1)) + (j - 1)

For example :

Given an array [1…5,1…7] of integers. Calculate address of element A[4,6], where BA=900.

Solution:- I = 4 , J = 6 ,M= 5 , N= 7   (M=ROW , N=COLUMN)

Location (A [4,6]) = BA + (7 x (4-1)) + (6-1)

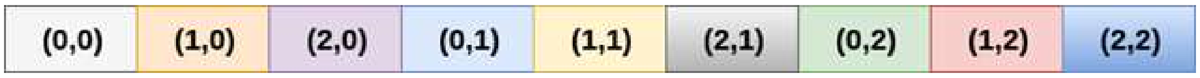
= 900 + (7 x 3) +5

= 900 + 21 + 5

= 926

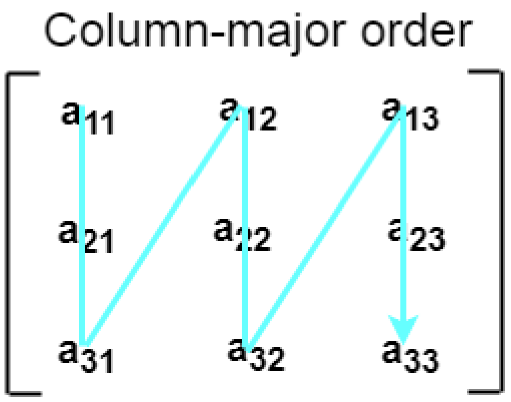
2. Column Major ordering

According to the column major ordering, all the columns of the 2D array are stored into the memory contiguously. The memory allocation of the array which is shown in in the above image is given as follows.



(0,0)	(1,0)	(2,0)	(0,1)	(1,1)	(2,1)	(0,2)	(1,2)	(2,2)

first, the 1st column of the array is stored into the memory completely, then the 2nd row of the array is stored into the memory completely and so on till the last column of the array.



Order elements of first column stored linearly and then comes elements of next column.
To determine element address A[i,j]:
Location (A[i,j]) = Base Address + (M x (j - 1)) + (i - 1)

For example :
Given an array [1…6,1…8] of integers. Calculate address element A[5,7], where BA=300.
Solution:- I = 5 , J = 7, M= 6 , N= 8

Location (A [5,7])= BA + (6 x (7-1)) + (5-1)

= 300 + (6 x 6) + 4

= 300 + 36 + 4

= 340

1.1.4 Two-Dimensional Character Array:

What is String?

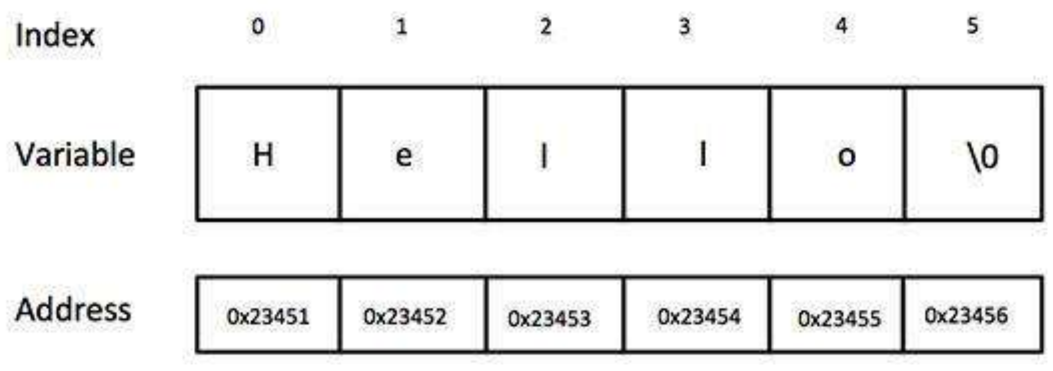
Strings are actually one-dimensional array of characters terminated by a null character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a null.

The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
If you follow the rule of array initialization then you can write the above statement as follows −

char greeting[] = "Hello";

Following is the memory presentation of the above defined string in C −



Actually, you do not place the *null* character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print the above mentioned string –

```
#include <stdio.h>
int main ()
{
   char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
   printf("Greeting message: %s\n", greeting );
   return 0;
}
```

OUTPUT:

```
Greeting message: Hello
```

❖ **1.1.4.1 Declaring & Initializing Two-Dimensional character array or Multidimensional Character Array**

Two-Dimensional character array Syntax:-

char string-array-name[row-size][column-size];

A 2D character array is declared in the following manner:

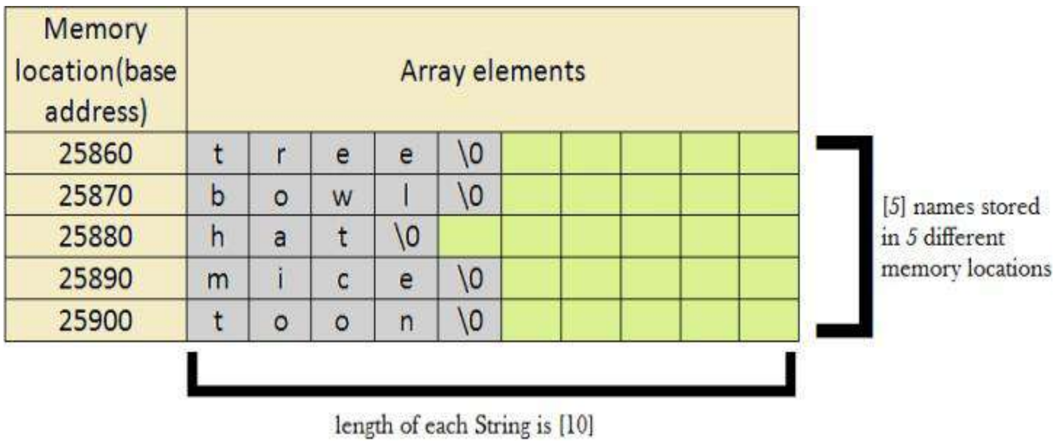
char name[5][10];

The order of the subscripts is kept in mind during declaration. The first subscript [5] represents the number of Strings that we want our array to contain and the second subscript [10] represents the length of each String. This is static memory allocation. We are giving 5*10=50 memory locations for the array elements to be stored in the array.

Initialization of the character array occurs in this manner:

```
char name[5][10]={
                "tree",
                "bowl",
                "hat",
                "mice",
                "toon"   };
```

see the diagram below to understand how the elements are stored in the memory location:



Memory location(base address)	Array elements									
25860	t	r	e	e	\0					
25870	b	o	w	l	\0					
25880	h	a	t	\0						
25890	m	i	c	e	\0					
25900	t	o	o	n	\0					

[5] names stored in 5 different memory locations
length of each String is [10]

The areas marked in green shows the memory locations that are reserved for the array but are not used by the string. Each character occupies 1 byte of storage from the memory.

Example of Multidimensional Character Array

```
#include <stdio.h>
void main()
{         char text[10][80];
          int i;
          clrscr();
       for(i = 0; i < 10; i++)
       {
              printf("\n Enter Some string for index %d: ", i + 1);
              gets(text[i]);
       }
       for(i = 0; i < 10; i++)
       {
              printf("\n Some string for index %d: ", i + 1);
              puts(text[i]);
       }
  getch();
}
```

1.1.4.2 Two-Dimensional character Array operations (Searching elements, copying, merging, finding length of given string)

Example1: Program to search for a string in the string array

```
#include<stdio.h>
#include<string.h>
#include<conio.h>
int main()
{
    char name[5][10],item[10];
    int i,x;        clrscr();
    for(i=0;i<5;i++ )
    {
         printf("Enter %d strings: ", i+1);
         scanf("%s", &name[i][0]);
    }
        /* entering the item to be found in the string array*/
   printf("Enter the string to be searched: ");
   scanf("%s", &item);
         /* process for finding the item in the string array*/
    for(i=0; i<5 ;i++ )
    {
          x=strcmp(&name[i][0],item);
          if(x == 0)
          {
             printf("\n Searched string %s is at position %d ",item,i+1);
             break;
          }
    }
    if(i > 4)
    {
        printf("\n The searched string does not match any name in the list...");
    }
    getch();     return 0;
}
```

Output:-

```
Enter 1 strings: one
Enter 2 strings: two
Enter 3 strings: three
Enter 4 strings: four
Enter 5 strings: five
Enter the string to be searched: four
Searched string four is at position : 4
```

Example2: Lexicographic order is the way of ordering words based on the alphabetical order of their component letters. It is also known as lexical order, dictionary order, and alphabetical order. It is similar to the way we search for any word in the dictionary. We start our search by simply searching for the first letter of the word. Then we try to find the second letter and so on. The words in the dictionary are arranged in lexicographic order.

```
/*  Write a program to sort elements in lexicographical order in C language
     (dictionary order). */
#include<stdio.h>
```

```
#include<string.h>
#include<conio.h>
int main()
{
     char str[10][50],temp[50];
     int i,j;   clrscr();

     printf("Enter 10 Words:\n");
     for(i=0;i<10;i++)
     {
           printf("\n Enter String %d : ",i+1);
           scanf("%s[^\n]",str[i]);
     }
     for(i=0;i<9;i++)
     {
         for(j=i+1;j<10;j++)
         {
           if( strcmp(str[i],str[j])  > 0 )
           {
              strcpy(temp,str[i]);
              strcpy(str[i],str[j]);
              strcpy(str[j],temp);
           }
         }
      }
      printf("\n In Lexicographical order: \n");
      for(i=0;i<10;i++)
              puts(str[i]);
  getch();
  return 0;
}
```

Example3: Write a C program String Copy without using string manipulation library functions.

```
#include<stdio.h>
#include<conio.h>
int main()
{
  char str1[100], str2[100];
  int i;   clrscr();
  printf("Enter first String: ");
  gets(str1);
  // str2 is empty or having garbage value
  for(i=0;str1[i]!='\0';i++)
  {
    str2[i]=str1[i];
  }
  str2[i]='\0';
  printf("Second string is: ");
  puts(str2);
  getch();   return 0;
}
```

OUTPUT:

```
Enter first String: Adjust Everywhere

Second string is: Adjust Everywhere
```

Example 4: Write a C program to Concatenate two strings without using string manipulation library functions.(merging of two string in one array)

```
#include<stdio.h>
#include<conio.h>
int main()
{
   char str1[100], str2[50];
   int i, j; clrscr();

   printf("Enter first string: ");
   gets(str1);

   printf("Enter second string: ");
   gets(str2);

   // calculate the length of string str1 & store it in the variable i
   for(i=0; str1[i]!='\0'; i++);

   for(j=0; str2[j] != '\0'; j++, i++)
   {
         str1[i] = str2[j];
   }
   str1[i] = '\0';
   printf("After Concatenation first string becomes : \n");
   puts(str1);
   getch();     return 0;
 }
```

Output:

```
Enter first string: PROGRAMMING
Enter second string: IS FUN!!!!
After Concatenation first string
becomes : PROGRAMMING IS FUN!!!!
```

Example5: Write a C program to Find the length of the string without using string manipulation library functions.

```
#include<stdio.h>
#include<conio.h>
int main()
{
    char str[200];
    int i;   clrscr();
    printf("Enter string: ");
    scanf("%[^\n]",str);
    for(i=0;str[i]!='\0';i++);
    printf("length of entered string is=%d",i);
    getch();       return 0;
}
```

OUTPUT:

```
Enter string: sdj international college
length of entered string is=25
```

1.2 Concepts of Structure and Union:

1.2.1 Defining, declaring and Initializing structure and Union

Why we use structure?

In C, there are cases where we need to store multiple attributes of an entity. It is not necessary that an entity has all the information of one type only. It can have different attributes of different data types. For example, an entity Student may have its name (string), roll number (int), marks (float). To store such type of information regarding an entity student, we have the following approaches:

- Construct individual arrays for storing names, roll numbers, and marks.
- Use a special data structure to store the collection of different data types.
- Let's look at the first approach in detail.

```
#include<stdio.h>
void main ()
{
  char names[3][10],dummy;
  int roll_numbers[3],i;
  float marks[3];
  clrscr();
  for (i=0;i<3;i++)
  {
    printf("Enter the name, roll number, and marks of the student %d : ",i+1);
    scanf("%s %d %f",&names[i],&roll_numbers[i],&marks[i]);
    scanf("%c",&dummy); // enter will be stored into dummy character at each iteration
  }
  printf("Printing the Student details...\n");
  for (i=0;i<3;i++)
  {
       printf("%s  %d  %.2f \n",names[i],roll_numbers[i],marks[i]);
  }    getch();
}
```

Output

```
Enter the name, roll number, and marks of the student   1 : Arun 90 91
Enter the name, roll number, and marks of the student   2 : Varun 91 56
Enter the name, roll number, and marks of the student   3 : Sham 89 69

Printing the Student details...
Arun 90 91.000000
Varun 91 56.000000
Sham 89 69.000000
```

The above program may fulfil our requirement of storing the information of an entity student. However, the program is very complex, and the complexity increase with the amount of the input. The elements of each of the array are stored contiguously, but all the arrays may not be stored contiguously in the memory. C provides you with an additional and simpler approach where you can use a special data structure, i.e., structure, in which, you can group all the information of different data type regarding an entity.

❖ What is Structure?

Structure in c is a user-defined data type that enables us to store the collection of different data types. Each element of a structure is called a member. Structures can simulate the use of classes and templates as it can store different information. The **struct** keyword is used to define the structure.

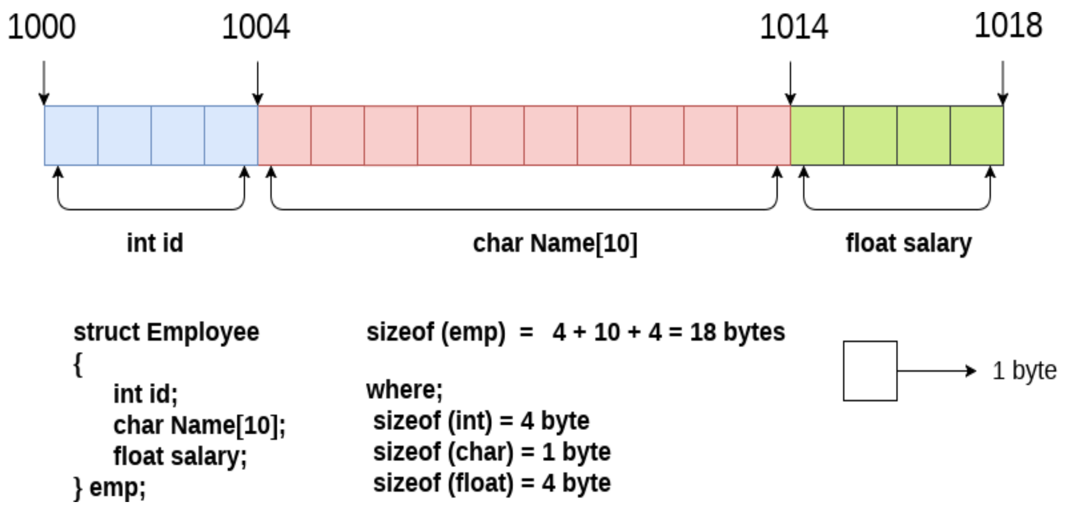
Let's see the syntax to define the structure in c.

```
struct structure_name
{
   data_type member1;
   data_type member2;
   .
   .
   data_type memeberN;
};
```

Example to define a structure for an entity employee in c.

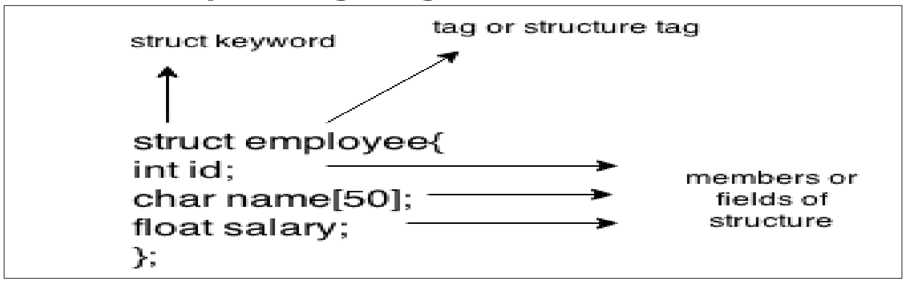
```
struct employee
{
     int id;
     char name[10];
     float salary;
};
```

The following image shows the memory allocation of the structure employee that is defined in the above example.



Here, **struct** is the keyword; **employee** is the name of the structure; **id**, **name**, and **salary** are the members or fields of the structure.

Let's understand it by the diagram given below:



Points to remember while defining a Structure

1. Key Word **struct** is required for defining structure.
2. Structure_name can be any valid identifier.
3. Set of variables will be declared inside curly braces "{ }" is known as structure body.
4. Different types of variable or array or structure can be declared inside structure body.
5. Only declaration is allowed inside structure body, we cannot initialize variable inside structure body.
6. The closing curly brace in the structure type declaration must be followed by a semicolon (;)
7. Defining structure will not declare any variable it acts as a template which means defining structure will not allocate any memory to the variables it is just represents information (set of variables).
8. Structure name will be used for creating instance of a structure (declaring a variable of type structure).
9. Structure Definition is mostly written before starting of main function which can be used globally. (Globally means structure variables can be created anywhere in the program either in main function or User Defined Function).

❖ Declaring structure variable

We can declare a variable for the structure so that we can access the member of the structure easily. Structure can be declared by two ways.

1. Tagged Declaration
2. Typedef Declaration

There are two ways to declare Tagged Declaration structure variable:

1. By struct keyword within main() function
2. By declaring a variable at the time of defining the structure.

1st way:

Let's see the example to declare the structure variable by struct keyword. It should be declared within the main function.

```
struct employee
{
    int id;
    char name[50];
    float salary;
};
```

Now write given code inside the main() function.

```
struct employee e1, e2;
```

The variables e1 and e2 can be used to access the values stored in the structure. Here, e1 and e2 can be treated in the same way as the objects in C++ and Java.

2nd way:

Let's see another way to declare variable at the time of defining the structure.

```
struct employee
{
    int id;
    char name[50];
    float salary;
}e1,e2;
```

❖ Which approach is good?

1. If number of variable are not fixed, use the 1st approach. It provides you the flexibility to declare the structure variable many times.
2. If number of variable are fixed, use 2nd approach. It saves your code to declare a variable in main() function.

❖ Structure Initialization

Like a variable of any other datatype, structure variable can also be initialized at compile time.

```
struct Patient
{
   float height;
   int weight;
   int age;
};
struct Patient p1 = { 180.75 , 73, 23 };       //initialization
```

OR

```
struct Patient p1;

p1.height = 180.75;     //initialization of each member separately
p1.weight = 73;
p1.age = 23;
```

1.2.2 typedef and accessing structure member

Type Definitions (*typedef*)

The C programming language provides a keyword called **typedef**, which you can use to give a type a new name. Typedef is a keyword that is used to give a new symbolic name for the existing name in a C program. This is same like defining alias for the commands.

```
  /* example of typedef */

#include<stdio.h>
#include<conio.h>
void main()
{
        typedef int LENGTH;
        LENGTH l, w, h;
        l=5,w=10,h=45;
        clrscr();
        printf("\n The value of Length is %d",l);
        printf("\n The value of Width is %d",w);
        printf("\n The value of Height is %d",h);
        getch();
}
```

Using typedef with structures

Syntax:

```
typedef struct Structure_name Structure_variable
```

Example:

```
typedef struct student status;
```

- ➢ When we use "typedef" keyword before struct <tag_name> like above, after that we can simply use type definition "status" in the C program to declare structure variable.
- ➢ Now, structure variable declaration will be, "status record".

- This is equal to “struct student record”. Type definition for “struct student” is status. i.e. status = “struct student”

Structure declaration using typedef in c:

```
typedef struct student
{
        int mark [2];
        char name [10];
        float average;
} status;
```

To declare structure variable, we can use the below statements.

```
status record1;             /* record 1 is structure variable */
status record2;             /* record 2 is structure variable */
```

Example program for C Structure using typedef

```
// Structure using typedef
#include <stdio.h>
#include <string.h>

typedef struct student
{
    int id;
    char name[20];
    float percentage;
}status;
int main()
{
      status record;

      record.id=1;
      strcpy(record.name, "Raju");
      record.percentage = 86.5;

      printf(" Id is: %d \n", record.id);
      printf(" Name is: %s \n", record.name);
      printf(" Percentage is: %f \n", record.percentage);
      getch();            return 0;
}
```

OUTPUT:

```
Id is: 1
Name is:  Raju
Percentage is: 86.500000
```

❖ Accessing members of the structure

There are two ways to access structure members:

1. By **.** (member or dot operator)
2. By **->** (structure pointer operator)

Let's see the code to access the *id* member of *p1* variable by . (member) operator.

```
p1.id
```

Example of structure in C language.

```
#include<stdio.h>
#include<string.h>
struct employee
```

```
{
   int id;
   char name[50];
}e1;   //declaring e1 variable for structure
int main( )
{
   e1.id=101;                                //store first employee information
  strcpy(e1.name, "Bhumika Patel");          //copying string into char array
  printf( "employee 1 id : %d\n", e1.id);    //printing first employee information
  printf( "employee 1 name : %s\n", e1.name);
  return 0;
}
```

OUTPUT :

```
employee 1 id : 101
employee 1 name : Bhumika Patel
```

Example of the structure in C language to store many employees information.

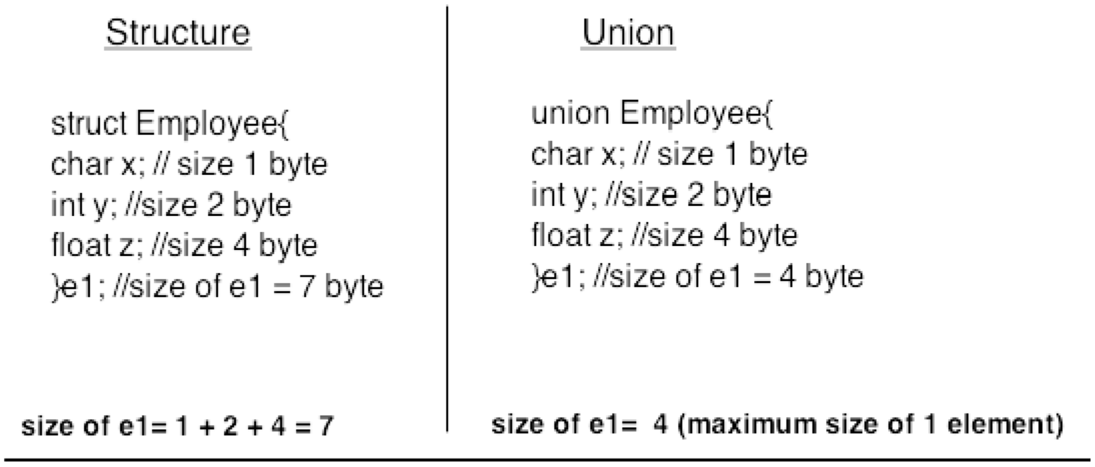
```
#include<stdio.h>
#include<string.h>
struct employee
{
   int id;
   char name[50];
   float salary;
}e1,e2;                          // Declaring e1 and e2 variables for structure
int main( )
{
  e1.id=101;                              // Initialize first employee information
  strcpy(e1.name, "Tony Stark");    // copying string into char array
  e1.salary=56000;
  e2.id=102;                               // Initialize second employee information
  strcpy(e2.name, "James Bond");
  e2.salary=126000;

      //printing first employee information
  printf( "employee 1 id : %d\n", e1.id);
  printf( "employee 1 name : %s\n", e1.name);
  printf( "employee 1 salary : %f\n", e1.salary);

      //printing second employee information
  printf( "employee 2 id : %d\n", e2.id);
  printf( "employee 2 name : %s\n", e2.name);
  printf( "employee 2 salary : %f\n", e2.salary);
  return 0;
}
```

OUTPUT:

```
employee 1 id : 101
employee 1 name : Tony Stark
employee 1 salary : 56000.000000
employee 2 id : 102
employee 2 name : James Bond
employee 2 salary : 126000.000000
```

Defining Union

- Like structure, **Union in c language** is *a user-defined data type* that is used to store the different type of elements.
- At once, only one member of the union can occupy the memory. In other words, we can say that the size of the union in any instance is equal to the size of its largest element.



Structure

```
struct Employee{
char x; // size 1 byte
int y; //size 2 byte
float z; //size 4 byte
}e1; //size of e1 = 7 byte
```

size of e1= 1 + 2 + 4 = 7

Union

```
union Employee{
char x; // size 1 byte
int y; //size 2 byte
float z; //size 4 byte
}e1; //size of e1 = 4 byte
```

size of e1= 4 (maximum size of 1 element)

- **Advantage of union over structure**
  1. It occupies less memory because it occupies the size of the largest member only.
- **Disadvantage of union over structure**
  1. Only the last entered data can be stored in the union. It overwrites the data previously stored in the union.

The **union** keyword is used to define the union.

Syntax to define union in c.

```
union union_name
{
   data_type member1;
   data_type member2;
   .
   .
   data_type memeberN;
};
```

Example to define union for an employee in c language

```
union employee
{
    int id;
    char name[50];
    float salary;
};
```

Example of union in C language.

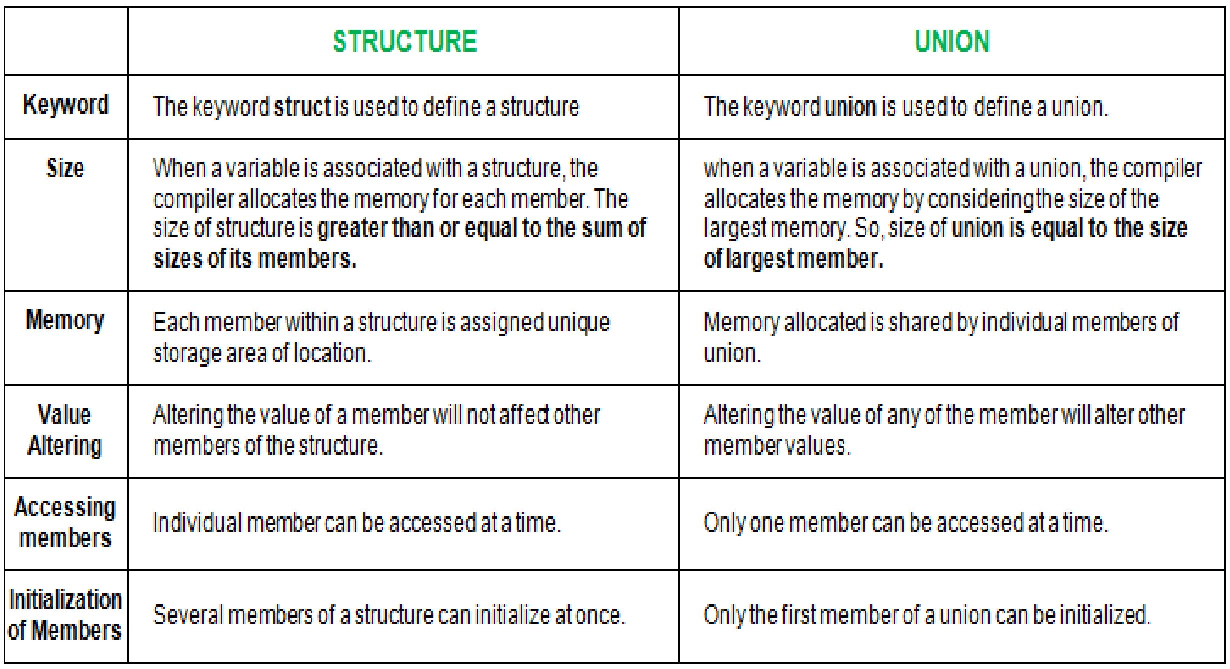
```
#include <stdio.h>
#include <string.h>
union employee
{
    int id;
    char name[50];
}e1;                    //declaring e1 variable for union
int main( )
{
  e1.id=101;                                 //store first employee information
  strcpy(e1.name, "Bhumika Patel");          //copying string into char array
  printf( "employee 1 id : %d\n", e1.id);    //printing first employee information
  printf( "employee 1 name : %s\n", e1.name);
  return 0;
}
```

Output:

```
employee 1 id : 1869508435
employee 1 name : Bhumika Patel
```

Note: As you can see, id gets garbage value because name has large memory size. So only name will have actual value.

1.2.3 Difference between structure and union



	STRUCTURE	UNION
Keyword	The keyword **struct** is used to define a structure	The keyword **union** is used to define a union.
Size	When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is **greater than or equal to the sum of sizes of its members.**	when a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of **union is equal to the size of largest member.**
Memory	Each member within a structure is assigned unique storage area of location.	Memory allocated is shared by individual members of union.
Value Altering	Altering the value of a member will not affect other members of the structure.	Altering the value of any of the member will alter other member values.
Accessing members	Individual member can be accessed at a time.	Only one member can be accessed at a time.
Initialization of Members	Several members of a structure can initialize at once.	Only the first member of a union can be initialized.

Difference: Array v/s Structure

	Array	**Structure**
1.	An array is a derived data type.	A structure is a user-defined data type.
2.	An array is a collection of similar data type items.	A structure is a collection of different data type items.
3.	Memory for the array will be allocated at the time of its declaration.	Memory is allocated only at the time of structure variable declaration.

4.	Array elements are referred by array index.	Structure elements are referred by its variable name using period sign with structure variable.
4.	It is difficult to represent complex related data using array.	It is easy to represent complex and related data using structure.
5.	Array does not have any special keyword to declare it instead it used [] bracket to define array size along with data type.	“struct” keyword is used in structure declaration.

6.	Syntax:
	data type arr_name [size];
		Sytanx :
		struct struct_name
		{
		    Type member1 ;
		    Type member2 ;
		};

1.3 User defined functions :

What is C function?

A large C program is divided into basic building blocks called C function. C function contains set of instructions enclosed by “{ }” which performs specific operation in a C program. Actually, Collection of these functions creates a C program.

A function is a block of statements that performs a specific task. Suppose you are building an application in C language and in one of your program, you need to perform a same task more than once. In such case you have two options –
a) Use the same set of statements every time you want to perform the task
b) Create a function to perform that task, and just call it every time you need to perform that task.
Using option (b) is a good practice and a good programmer always uses functions while writing codes in C.

❖ **Benefits of Using the Function in C**

➢ The function provides modularity.
➢ The function provides reusable code.
➢ In large programs, debugging and editing tasks is easy with the use of functions.
➢ The program can be modularized into smaller parts.
➢ Separate function independently can be developed according to the needs.

❖ **Uses of C functions:**

1. C functions are used to avoid rewriting same logic/code again and again in a program.
2. There is no limit in calling C functions to make use of same functionality wherever required.
3. We can call functions any number of times in a program and from any place in a program.
4. A large C program can easily be tracked when it is divided into functions.

5. The core concept of C functions are, re-usability, dividing a big task into small pieces to achieve the functionality and to improve understand ability of very large C programs.

❖ Types of functions

1. **Built-in (Library) Functions:** The system provided these functions and stored in the library. Therefore it is also called Library Functions. e.g. scanf(), printf(), strcpy(), strlwr(), strcmp(), strlen(), strcat() etc. To use these functions, you just need to include the appropriate C header files.
2. **User Defined Functions:** These functions are defined by the user at the time of writing the program.

1.3.1 Function return type, parameter list, local function variables

1.3.2 Passing arguments to function

❖ **User-defined function(UDF) :** You can also create functions as per your need. Such functions created by the user are known as user-defined functions.

(Note: function names are identifiers and should be unique)

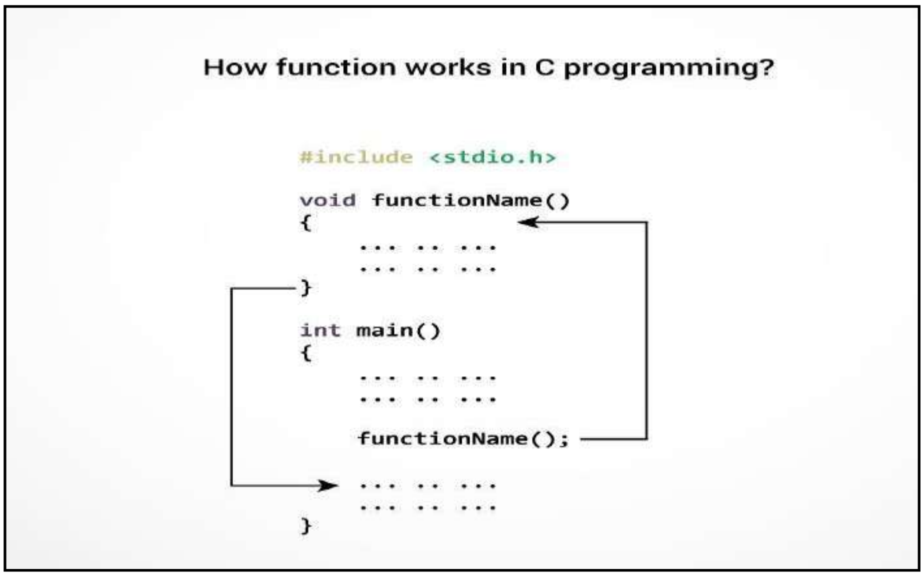
How user-defined function works?

```
#include <stdio.h>
void functionName()      // UDF
{
   ... .. ...
   ... .. ...
}
int main()
{
   ... .. ...
   functionName();
   ... .. ...
}
```

The execution of a C program begins from the main() function.
When the compiler encounters functionName();, control of the program jumps to

```
 void functionName()
```

And, the compiler starts executing the codes inside functionName().
The control of the program jumps back to the main() function once code inside the function definition is executed.



How function works in C programming?

```
#include <stdio.h>

void functionName()
{
    ... .. ...
    ... .. ...
}

int main()
{
    ... .. ...
    ... .. ...

    functionName();

    ... .. ...
    ... .. ...
}
```

❖ **Advantages of user-defined function**
1. The program will be easier to understand, maintain and debug.
2. Reusable codes that can be used in other programs
3. A large program can be divided into smaller modules. Hence, a large project can be divided among many programmers.

❖ **Disadvantage of User Define Function**
1. It will take lots of extra time for program execution.

❖ **Parts of User Define Function**

There are three parts of User Define Function:
1. Function declaration or prototype – This informs compiler about the function name, function parameters and return value's data type.
2. Function call – This calls the actual function
3. Function definition – This contains all the statements to be executed.

1. Function declaration or prototype:

A function prototype is simply the declaration of a function that specifies function's name, parameters and return type. It doesn't contain function body.
A function prototype gives information to the compiler that the function may later be used in the program.

Syntax of function prototype:

```
        returnType functionName(type1 argument1, type2 argument2, ...);
```

In the above example, int addNumbers(int a, int b); is the function prototype which provides the following information to the compiler:
1) name of the function is addNumbers()
2) return type of the function is int
3) two arguments of type int are passed to the function

Note: The function prototype is not needed if the user-defined function is defined before the main() function.

2. Calling a function

Control of the program is transferred to the user-defined function by calling it.
Syntax of function call:

```
functionName(argument1, argument2, ...);
```

In the above example, the function call is made using addNumbers(n1, n2); statement inside the main() function.

3. Function definition

Function definition contains the block of code to perform a specific task. In our example, adding two numbers and returning it.
Syntax of function definition:

```
returnType  functionName(datatype1 argument1, datatype2 argument2, ...)
{
          //body of the function
}
```

When a function is called, the control of the program is transferred to the function definition. And, the compiler starts executing the codes inside the body of a function.

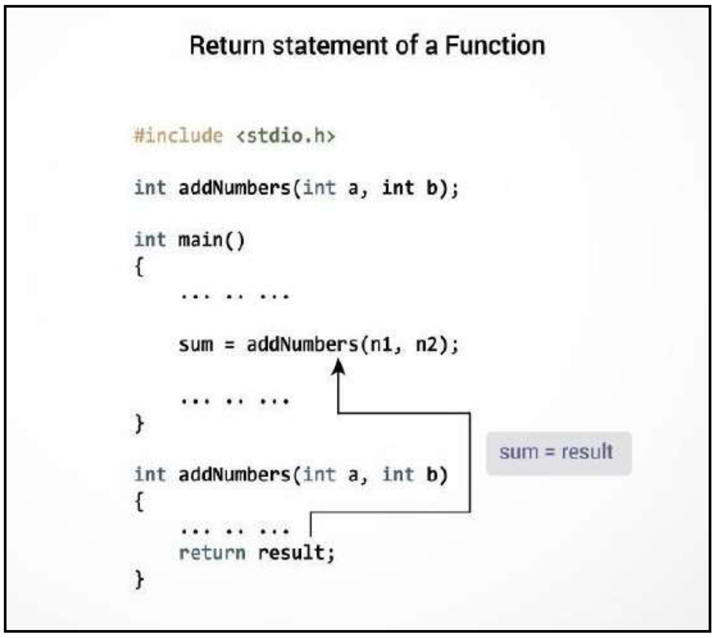
Example1: Creating a user defined function addnumbers()

```
#include <stdio.h>
int addnumbers(int num1, int num2);        // Function declaration or prototype
int main()
{
    int var1, var2,sum;   clrscr();
    printf("Enter number 1: ");
    scanf("%d",&var1);
    printf("Enter number 2: ");
    scanf("%d",&var2);
    sum = addnumbers (var1, var2);             // function calling
    printf("Output: %d", sum);    getch();
    return 0;
}
int addnumbers (int num1, int num2)    // Function definition
{
    int result;
    result = num1+num2;           // Arguments are used here
    return result;
}
```
Output:
```
Enter number 1: 100
Enter number 2: 120
Output: 220
```

❖ Return Statement

The return statement terminates the execution of a function and returns a value to the calling function. The program control is transferred to the calling function after the return statement.

In the above example, the value of the **result** variable is returned to the main function. The **sum** variable in the main() function is assigned this value.


Return statement of a Function

```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... .. ...

    sum = addNumbers(n1, n2);

    ... .. ...
}

int addNumbers(int a, int b)
{
    ... .. ...
    return result;
}
```

Example2: Creating a void user defined function that doesn't return anything

```
#include <stdio.h>
  /* function return type is void and it doesn't have parameters*/
void introduction()
{
   printf("Hi I am Function \n");
   printf("U can call me any number of time.\n");
   printf("How are you?");
}
int main()
```

```
{
        introduction();    /*calling function*/
        return 0;
}
```

Output:

```
Hi I am Function
U can call me any number of time.
How are you?
```

❖ **Few Points to Note regarding functions in C:**

1) main() in C program is also a function.
2) Each C program must have at least one function, which is main().
3) There is no limit on number of functions; A C program can have any number of functions.
4) return type: Data type of returned value. It can be void also, in such case function doesn't return any value.
5) A function can call itself and it is known as "Recursion".

1.3.3 Calling function from main() function or from other function.

Function call by value is the default way of calling a function in C programming. Before we discuss function call by value, lets understand the terminologies that we will use while explaining this:

Actual parameters: The parameters that appear in function calls.

Formal parameters: The parameters that appear in function declarations.

```
#include<stdio.h>
#include<conio.h>

     /* function declaration */
int max(int num1, int num2);          // Formal parameters in function call

int main ()
{
     /* local variable definition */
   int a = 100;
   int b = 200;
   int ret;    clrscr();
     /* calling a function to get max value */
   ret = max(a, b);          // Actual parameters in function call
   printf( "Max value is : %d\n", ret);
     getch();     return 0;
}
      /* function returning the max between two numbers */
int max(int num1, int num2)
{
      /* local variable declaration */
   int result;

   if (num1 > num2)
        result = num1;
   else
        result = num2;
```

```
  return result;
}
```

Example : **Calling function from other function.**

```
#include<stdio.h>
#include<conio.h>
 /* function declarations */
float pi();
float cal_area(float rad);
int main ()
{
   /* local variable definition */
   float r,area; clrscr();
   printf("Enter Radius value : ");
   scanf("%f",&r);

   area = cal_area(r);
   printf( "Area of circle is : %f \n", area );
   getch();    return 0;
}
float pi()
{
   const float pi=3.14;
   return pi;
}
float cal_area(float rad)
{    /* local variable declaration */
    float result;
    result = pi() * rad * rad ;           // calling function from other function
    return result;
}
```

1.3.4 Function with No arguments and no return value, No arguments and a return value, with arguments and no return value, with arguments and a return value. **OR** **Types of User-defined Functions in C Programming**

These 4 programs below check whether the integer entered by the user is an even number or not.

The output of all these programs below is the same, and we have created a user-defined function in each example. However, the approach we have taken in each example is different.
1: No arguments passed and no return value
2: No arguments passed but a return value
3: Argument passed but no return value
4: Argument passed and a return value

Example 1: No arguments passed and no return value

```
#include<stdio.h>
#include<conio.h>

void checkIsEven();      //Function declaration or prototype
int main()
{
    clrscr();
    checkIsEven();              // argument is not passed
    getch();       return 0;
}
void checkIsEven()    // return type is void meaning doesn't return any value
{
      int n;
      printf("Enter a positive integer: ");
      scanf("%d",&n);
      if(n%2 == 0)
            printf("\n %d is an even number.", n);
      else
           printf("\n %d is not an even number.", n);
}
```

Example 2: No arguments passed but a return value

```
#include<stdio.h>
#include<conio.h>
int GetInteger();        //Function declaration or prototype
int main()
{
     int num;
     clrscr();
     num=GetInteger();    // argument is not passed
     if(num%2 == 0)
                printf("\n %d is an even number.", num);
     else
             printf("\n %d is not an even number.", num);
    getch();
    return 0;
}
int GetInteger()    // returns integer entered by the user
{
        int n;
        printf("Enter a positive integer: ");
        scanf("%d",&n);
        return n;
}
```

Example 3: Argument passed but no return value

```
#include<stdio.h>
#include<conio.h>
void GetEven(int n);            //Function declaration or prototype
void main()
{
    int num;
    clrscr();
    printf("Enter a positive integer: ");
    scanf("%d",&num);
    GetEven(num);       // num(argument) is passed to the function
    getch();
}
void GetEven(int n)    // function return type is void meaning doesn't return any value
{
    if(n%2 == 0)
            printf("\n %d is an even number.", n);
    else
            printf("\n %d is not an even number.", n);
}
```

Example 4: Argument passed and a return value

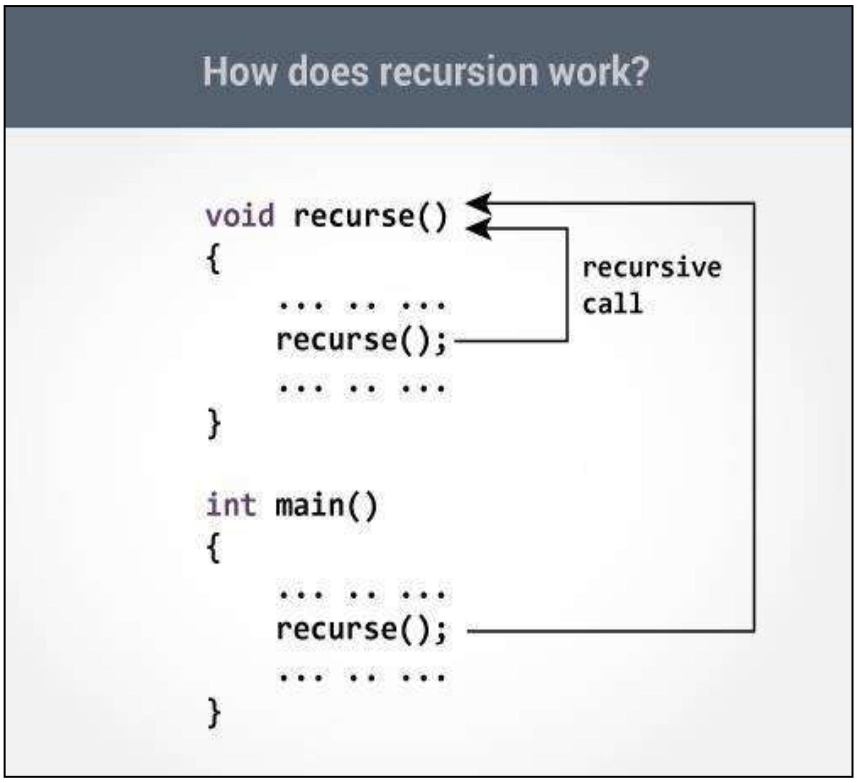
```
#include<stdio.h>
#include<conio.h>
int addnumbers(int num1, int num2);      //Function declaration or prototype
int main()
{
    int var1, var2,sum;
    clrscr();
    printf("Enter number 1: ");
    scanf("%d",&var1);
    printf("Enter number 2: ");
    scanf("%d",&var2);
    sum = addnumbers (var1, var2);    // function calling
    printf("Output: %d", sum);
    getch();
    return 0;
}
int addnumbers (int num1, int num2)  // Function definition
{
    int result;
    result = num1+num2;        // Arguments are used here
    return result;            // function  return int value
}
```

Which approach is better?

Well, it depends on the problem you are trying to solve. In this case, passing argument and returning a value from the function (**example 4**) is better.

1.3.5 Recursive Function

A function that calls itself is known as a recursive function. And, this technique is known as recursion.



```
void recurse()
{
    ... .. ...
    recurse();
    ... .. ...
}

int main()
{
    ... .. ...
    recurse();
    ... .. ...
}
```

- The recursion continues until some condition is met to prevent it.
- To prevent infinite recursion, if...else statement (or similar approach) can be used where one branch makes the recursive call, and other doesn't.
- Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc. The following example ***calculates the factorial of a given number using a recursive function***

```
#include<stdio.h>
#include<conio.h>

int factorial(int i)
{
  if(i <= 1)
  {
    return 1;
  }
   return i * factorial(i - 1);
}
int  main()
{
   int i = 5;      clrscr();
   printf("Factorial of %d is %d\n", i, factorial(i));
   getch();      return 0;
}
```

Output:

```
Factorial of 5 is 120
```

Example2: Fibonacci Series using Recursive Function

```
#include<stdio.h>
#include<conio.h>

int fibonacci(int i)
{
  if(i == 0)
  {
    return 0;
  }
  if(i == 1)
  {
    return 1;
  }
     return fibonacci(i-1) + fibonacci(i-2);
}
int  main()
{
    int i;  clrscr();
  for (i = 0; i < 10; i++)
  {
    printf("%d \t", fibonacci(i));
  }
  getch();    return 0;
}
```

Output :

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

❖ **Advantages of Recursion**

1) Recursion is more elegant and requires few variables which make program clean.
2) Recursion can be used to replace complex nesting code by dividing the problem into same problem of its sub-type.

❖ **Disadvantages of Recursion**

1) In other hand, it is hard to think the logic of a recursive function. It is also difficult to debug the code containing recursion.

FYBCA-SEM2-204 - Programming Skills

Unit-1-Practical Problems

Multidimensional Array

1. Write a Menu base C program with following options.
    1. Addition of two Matrices.
    2. Subtraction of two Matrices.
    3. Multiplication of two Matrices.
    4. Transpose of Single Matrix.
    5. Exit.
2. Write C program to count total duplicate elements in an array.
3. Write C program to check matrix is an identity matrix example.
4. Write C program to find sum of each column in a matrix.
5. Write C program to find sum of each row in a matrix.
6. Write C program to check two matrices are equal or not.
7. Write C program to find lower triangle matrix.
8. Write C program to find upper triangle matrix.
9. Write C program to find sum of each row and column of a matrix.

10	20	30	60
40	50	60	150
70	80	90	240
120	150	180	

User defined Function

1. Write C Program to find area of rectangle using User defined Function.
2. Write C Program to find perimeter of rectangle using User defined Function.
3. Write C program to find area of a triangle using User defined Function.
4. Write C Program to find diameter, circumference and area of circle using User defined Function.
5. Write C program to convert centimeter to meter and kilometer using User defined Function.
6. Write C program to convert temperature from degree Celsius to Fahrenheit using UDF.
7. Write C Program to calculate simple interest using User defined Function.
8. Write C Program to find power of a number using User defined Function.(without pow())
9. Write C Program to find cube using function using User defined Function.
10. Write C program to check the given string is palindrome or not using UDF.
11. Write C Program to convert decimal number to binary number using the UDF.
12. Write C Program to get largest element of an array using the UDF.
13. Write C Program to find maximum and minimum element in array using UDF.
14. Write C program to sort numeric array in ascending or descending order using UDF.
15. Write C program to search element in an array using User define Function.
16. Write C program Menu base Calculator using UDF.
17. Write Menu base C Program with following option using UDF.
    1. To check given number is Prime or not
    2. To check given number is Armstrong or not.
    3. To check given number Perfect or not.
    4. Exit
18. Write C Program to find Sum of Series $1^2+2^2+3^2+.....+n^2$ Using Recursion in C.

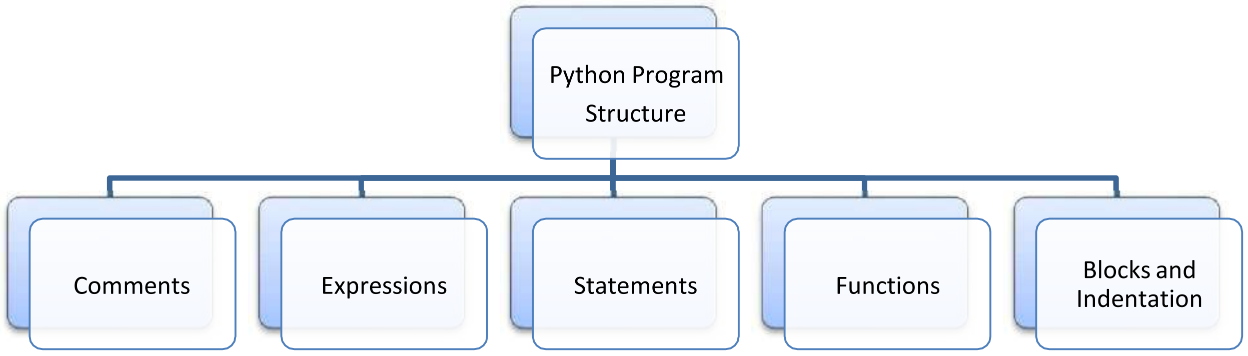
Chapter 2: Python Fundamentals

2.1 Concepts of Interpreter based programming language:

- An Interpreter directly executes instructions line by line written in a programming or scripting language without converting them to an object code or machine code.
- Examples of interpreted languages are Perl, Python and Matlab.

Structure of Python Programming language.

- Basic Structure of python programming includes following components:



Comments

- Comments are the additional readable information to get better understanding about the source code.
- Comments in Python are the non-executable statements.
- Comments are of 2 types:
  - 1)Single Line Comments: which begin with a hash symbol (#).
  - E.g. #This is a sample python program
  - 2)Multiline Comments: which begins with ‘’’ and ends with ‘’’(3 single quotes)
  - E.g. ‘’’ This is a sample python program1

This is a sample python program2 ’’’

Expression:

- An expression is any legal combination of symbols that represents a value.
- An expression represents something which python evaluates and which produces a value.
- E.g. 10, x + 5

Statements:

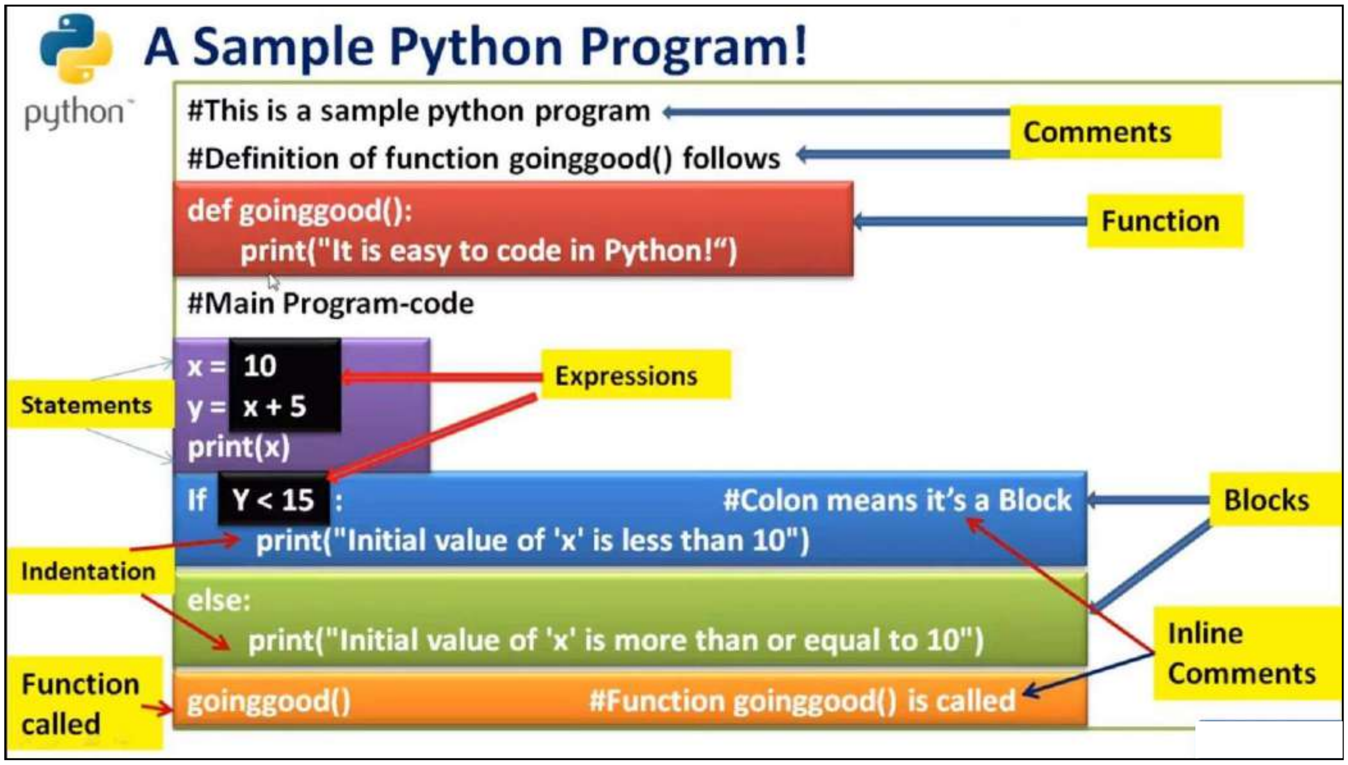
- A statement is a programming instruction that does something i.e. some action takes place.
- A statement executes and may or may not results in a value.
- E.g. print(x + 2), y = x + 5, x = 10

Functions:

- A function is a code that has a name and it can be reused (executed again) by specifying its name in the program, where needed.
- A function begin with ‘def’ statement
- E.g. goinggood( )

Block and Indentation:

- A group of statements which are part of another statement or a function are called block or code-block or suite in python.
- Indentation is used to show blocks in python. Four spaces together mark the next indent-level.



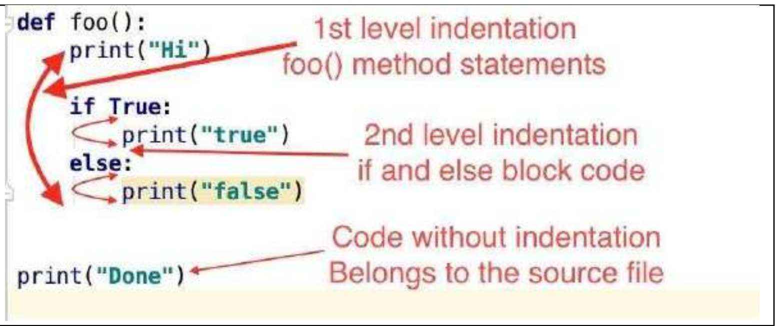
A Sample Python Program!

```
#This is a sample python program
#Definition of function goinggood() follows
def goinggood():
    print("It is easy to code in Python!“)
#Main Program-code
x = 10
y = x + 5
print(x)
If  Y < 15 :                                   #Colon means it’s a Block
     print("Initial value of 'x' is less than 10")
else:
     print("Initial value of 'x' is more than or equal to 10")
goinggood()                   #Function goinggood() is called
```

Python code Indention and execution

- Indentation refers to the spaces at the beginning of a code line.
- Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important.
- Python uses indentation to indicate a block of code.
- The leading whitespaces (space and tabs) at the start of a line is used to determine the indentation level of the line.
- Increase the indent level to group the statements for that code block. Similarly, reduce the indentation to close the grouping.
- Example:

```
def foo():
    print("Hi")

    if True:
        print("true")
    else:
        print("false")

print("Done")
```



```
def foo():
    print("Hi")

    if True:
        print("true")
    else:
        print("false")

print("Done")
```

Python Indentation Rules:

- We can’t split indentation into multiple lines using backslash.
- The first line of Python code can’t have indentation, it will throw IndentationError.
- You should avoid mixing tabs and whitespaces to create indentation.
- It is preferred to use whitespaces for indentation than the tab character.
- The best practice is to use 4 whitespaces for first indentation and then keep adding additional 4 whitespaces to increase the indentation.

2.2 Python Variables:

Variables:

- Variables are containers for storing data values.

Rules for Python Variable:

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

Valid Variable Name:	Invalid Variable Name:

`myvar = "John"`
`my_var = "John"`
`_my_var = "John"`
`myVar = "John"`
`MYVAR = "John"`
`myvar2 = "John"`
	`2myvar = "John"`
	`my-var = "John"`
	`my var = "John"`

Creating Variables

- Python has no command for declaring a variable.
- A variable is created the moment you first assign a value to it.
- Variables do not need to be declared with any particular type, and can even change type after they have been set.

Example:

```
x = 4      # x is of type int
z = "Sally" # x is now of type str
y='Alice'
print(x)
```

Note: String variables can be declared either by using single or double quotes.

Many Values to Multiple Variables

Python allows you to assign values to multiple variables in one line:

Example:

x, y, z = "Orange", "Banana", "Cherry"
print(x)
print(y)
print(z)
	Output:
	Orange
	Banana
	Cherry

One Value to Multiple Variables

We can assign the same value to multiple variables in one line:

Example:

x = y = z = "Orange"
print(x)
print(y)
print(z)
	Output:
	Orange
	Orange
	Orange

2.3 Python Datatypes:

- In programming, data type is an important concept
- Variables can store data of different types, and different types can do different things.
- Python has the following data types built-in by default, in these categories:

Data Types	Keywords
Text Types:	str
Numeric Types:	int, float, complex
Boolean Type:	bool

Getting the Data Type

- You can get the data type of any object by using the type() function:

Example:

x = 5
print(type(x))
	Output:
	<class 'int'>

Setting the Data Type

- In Python, the data type is set when you assign a value to a variable:

Example:	Data Types
x = "Hello World"	str
x = 20	int
x = 20.5	float
x = 1j	complex
x = True	bool

Type Conversions and Casting:

- If you want to specify the data type of a variable, this can be done with casting.

```
x = str(3)   # x will be '3'
y = int(3)  # y will be 3
z = float(3)  # z will be 3.0
```

Other Examples:

Example	Data Type
x = str("Hello World")	str
x = int(20)	int
x = float(20.5)	float
x = complex(1j)	Complex
x = bool(5)	bool

Python Casting

- There may be times when you want to specify a type on to a variable. This can be done with casting. Python is an object-orientated language, and as such it uses classes to define data types, including its primitive types.
- Casting in python is therefore done using functions as follows:
- int() – an integer number from an integer literal, a float literal (by removing all decimals), or a string literal (providing the string represents a whole number)
- float() - a float number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer)
- str() - a string from a wide variety of data types, including strings, integer literals and float literals

Example:

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

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

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

2.4 User defined function

- A function is a block of code which only runs when it is called.
- You can pass data, known as parameters, into a function.
- A function can return data as a result.

Creating a Function

- In Python a function is defined using the def keyword:

Syntax:

```
def function_name(Parameter List):
        function body
```

Example:

```
def my_function():
  print("Hello from a function")
```

Calling a Function

- To call a function, use the function name followed by parenthesis:

Example

```
def my_function():
  print("Hello from a function")

my_function()  #function call
```

Arguments

- Information can be passed into functions as arguments.

- Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.
- The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

Example:

```
def my_function(fname):
  print(" Welcome  " + fname )

my_function("Jay")
my_function("Parth")
my_function("Dhruvi")
```

```
Output:
Welcome Jay
Welcome Parth
Welcome Dhruvi
```

Number of Arguments

- By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you must call the function with 2 arguments, not more, and not less.

Example

(This function expects 2 arguments, and gets 2 arguments:)

```
def my_function(fname, lname):
  print(fname + " " + lname)

my_function("Alice", " Evan")
```

```
Output:
Alice Evan
```

Function with Default Parameter Value

- The following example shows how to use a default parameter value.
- If we call the function without argument, it uses the default value:

Example:

```
def my_function(country = "Norway"):
  print("I am from " + country)

my_function("Sweden")
my_function("India")
my_function() // this function will take default value
my_function("Brazil")
```

```
Output:

I am from Sweden
I am from India
I am from Norway
I am from Brazil
```

Function with Return Values

- To let a function return a value, use the return statement:

Example

```
def my_function(x):
  return 5 * x
```

```
Output:
15
25
```

```
print(my_function(3))
print(my_function(5))
print(my_function(9))
```

```
45
```

Global Variables

- Variables that are created outside of a function are known as global variables.
- Global variables can be used by everyone, both inside of functions and outside.

Example:

```
x = "awesome" #Global Variable

def myfunc():
  print("Python is " + x)# Printing global Variable value

myfunc() //function call
```

```
Output:
Python is awesome
```

Note: If you create a variable with the same name inside a function, this variable will be local, and can only be used inside the function. The global variable with the same name will remain as it was, global and with the original value.

Example:

```
x = "awesome" #global variable

def myfunc():
  x = "fantastic"
  print("Python is " + x)  #local variable

myfunc()  #function call

print("Python is " + x) #printing global variable
```

```
Output:
Python is fantastic
Python is awesome
```

The global Keyword

- Normally, when you create a variable inside a function, that variable is local, and can only be used inside that function.
- To create a global variable inside a function, you can use the global keyword.

Example

- If you use the global keyword, the variable belongs to the global scope:

```
def myfunc():
  global x
  x = "fantastic"

myfunc()  #function call

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

```
Output:
Python is fantastic
```

Example

- To change the value of a global variable inside a function, refer to the variable by using the global keyword:

x = "awesome"
 	
def myfunc():
  global x
  x = "fantastic"
 	
myfunc() #function call
 	
print("Python is " + x)
	Output:
	Python is fantastic

UNIT-3: Python Strings and Operators

3.1 Python Strings:

Python string is the collection of the characters surrounded by single quotes, double quotes, or triple quotes. The computer does not understand the characters; internally, it stores manipulated character as the combination of the 0's and 1's. Each character is encoded in the ASCII or Unicode character. So we can say that Python strings are also called the collection of Unicode characters.
In Python, strings can be created by enclosing the character or the sequence of characters in the quotes. Python allows us to use single quotes, double quotes, or triple quotes to create the string.

Consider the following example in Python to create a string.

```
str1 = "Hi Python !"
print(type(str1))
```

In Python, strings are treated as the sequence of characters, which means that Python doesn't support the character data-type; instead, a single character written as 'p' is treated as the string of length 1.

3.1.1 Multiline string, String as character array, triple quotes

Creating String in Python

We can create a string by enclosing the characters in single-quotes or double- quotes. Python also provides triple-quotes to represent the string, but it is generally used for multiline string or docstrings.

```
#Using single quotes
str1 = 'Hello Python'
print(str1)
#Using double quotes
str2 = "Hello Python"
print(str2)

#Using triple quotes
str3 = '''Triple quotes are generally used for
    represent the multiline or
    docstring'''
print(str3)
```

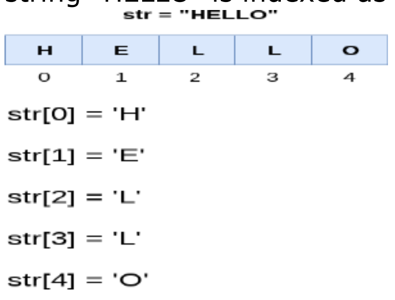
Output:

```
Hello Python
Hello Python
Triple quotes are generally used for
    represent the multiline or
    docstring
```

3.1.2 Slicing string, negative indexing, string length, concatenation

Strings indexing and Slicing (splitting)

Like other languages, the indexing of the Python strings starts from 0. For example, The string "HELLO" is indexed as given in the below figure.



Consider the following example:

```
str = "HELLO"
print(str[0])
print(str[1])
print(str[2])
print(str[3])
print(str[4])
print(str[6])   # It returns the IndexError because 6th index doesn't exist
```

Output:

```
H
E
L
L
O
IndexError: string index out of range
```

As shown in Python, the slice operator [] is used to access the individual characters of the string. However, we can use the **:** (colon) operator in Python to access the substring from the given string. Consider the following example.

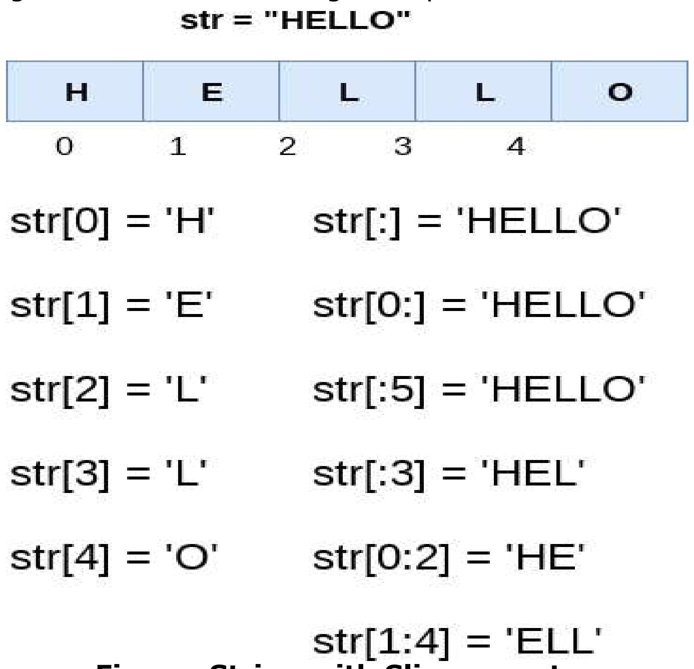


Figure: String with Slice operator

Here, we must notice that the upper range given in the slice operator is always exclusive i.e., if str = 'HELLO' is given, then str[1:3] will always include str[1] = 'E', str[2] = 'L' and nothing else.

Example:

```
str = "JAVATPOINT"
# Start Oth index to end
print(str[0:])
# Starts 1th index to 4th index
print(str[1:5])
# Starts 2nd index to 3rd index
print(str[2:4])
# Starts 0th to 2nd index
print(str[:3])
#Starts 4th to 6th index
```

```
print(str[4:7])
```
Output:
```
JAVATPOINT
AVAT
VA
JAV
TPO
```

We can do the negative slicing in the string; it starts from the rightmost character, which is indicated as -1. The second rightmost index indicates -2, and so on. Consider the following image.

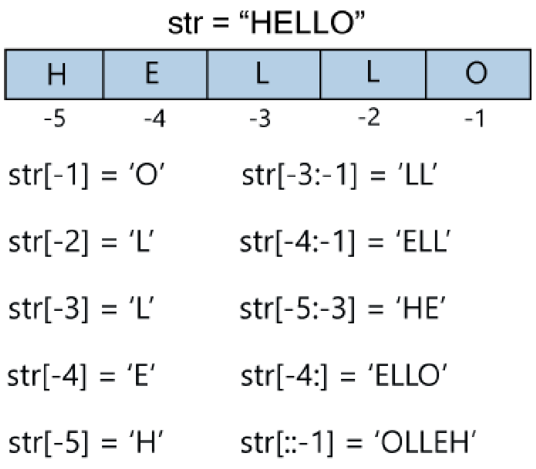


Figure: Negative slicing in the string

Consider the following example

```
str = 'JAVATPOINT'
print(str[-1])
print(str[-3])
print(str[-2:])
print(str[-4:-1])
print(str[-7:-2])
print(str[::-1])     # Reversing the given string
print(str[-12])
```
Output:
```
T
I
NT
OIN
ATPOI
TNIOPTAVAJ
IndexError: string index out of rang
```

Reassigning Strings: Updating the content of the strings is as easy as assigning it to a new string. The string object doesn't support item assignment i.e., A string can only be replaced with new string since its content cannot be partially replaced. Strings are immutable in Python.
Consider the following example.

```
str = "HELLO"
str[0] = "h"
print(str)
```
Output:

```
Traceback (most recent call last):
  File "12.py", line 2, in <module>
    str[0] = "h";
TypeError: 'str' object does not support item assignment
```

However, in example 1, the string str can be assigned completely to a new content as specified in the following example.

```
str = "HELLO"
print(str)
str = "hello"
print(str)
OUTPUT:
HELLO
hello
```

Deleting the String

As we know that strings are immutable. We cannot delete or remove the characters from the string. But we can delete the entire string using the del keyword.

```
str = "JAVATPOINT"
del str[1]
Output:
TypeError: 'str' object doesn't support item deletion
```

Now we are deleting entire string.

```
str1 = "JAVATPOINT"
del str1
print(str1)
Output:
NameError: name 'str1' is not defined
```

String Operators

Operator	Description
+	It is known as concatenation operator used to join the strings given either side of the operator.
*****	It is known as repetition operator. It concatenates the multiple copies of the same string.
[]	It is known as slice operator. It is used to access the sub-strings of a particular string.
[:]	It is known as range slice operator. It is used to access the characters from the specified range.
in	It is known as membership operator. It returns if a particular sub-string is present in the specified string.
not in	It is also a membership operator and does the exact reverse of in. It returns true if a particular substring is not present in the specified string.
r/R	It is used to specify the raw string. Raw strings are used in the cases where we need to print the actual meaning of escape characters such as "C://python". To define any string as a raw string, the character r or R is followed by the string.
%	It is used to perform string formatting. It makes use of the format specifiers used in C programming like %d or %f to map their values in python. We will discuss how formatting is done in python.

Consider the following example to understand the real use of Python operators.

```
str = "Hello"
str1 = " world"
print(str*3)                 # prints HelloHelloHello
print(str+str1)              # prints Hello world
print(str[4])                # prints o
print(str[2:4]);             # prints ll
print('w' in str)            # prints false as w is not present in str
print('wo' not in str1)      # prints false as wo is present in str1.
print(r'C://python39')       # prints C://python37 as it is written
print("The string str : %s"%(str))          # prints The string str : Hello
Output:
HelloHelloHello
Hello world
o
ll
False
False
C://python39
The string str : Hello
```

Escape Sequence

Let's suppose we need to write the text as - They said, "Hello what's going on?"- the given statement can be written in single quotes or double quotes but it will raise the SyntaxError as it contains both single and double-quotes.

Consider the following example to understand the real use of Python operators.

```
str = "They said, "Hello what's going on?""
print(str)
Output:
SyntaxError: invalid syntax
```

We can use the triple quotes to accomplish this problem but Python provides the escape sequence.

The backslash(/) symbol denotes the escape sequence. The backslash can be followed by a special character and it interpreted differently. The single quotes inside the string must be escaped. We can apply the same as in the double quotes.

Example

```
# using triple quotes
print('''They said, "What's there?"''')

# escaping single quotes
print('They said, "What\'s going on?"')

# escaping double quotes
print("They said, \"What's going on?\"")
OUTPUT:
They said, "What's there?"
They said, "What's going on?"
They said, "What's going on?"
```

The list of an escape sequence is given below:

Sr.	Escape Sequence	Description	Example
1.	\newline	It ignores the new line.	print("Python1 \ Python2 \ Python3")

			Output: Python1 Python2 Python3
2.	\\	Backslash	print("\\") Output: \
3.	\'	Single Quotes	print('\'') Output: '
4.	\\"	Double Quotes	print("\"") Output: "
5.	\a	ASCII Bell	print("\a")
6.	\b	ASCII Backspace(BS)	print("Hello \b World") Output: Hello World
7.	\f	ASCII Formfeed	print("Hello \f World!") Hello World!
8.	\n	ASCII Linefeed	print("Hello \n World!") Output: Hello World!
9.	\r	ASCII Carriege Return(CR)	print("Hello \r World!") Output: World!
10.	\t	ASCII Horizontal Tab	print("Hello \t World!") Output: Hello World!
11.	\v	ASCII Vertical Tab	print("Hello \v World!") Output: Hello World!
12.	\ooo	Character with octal value	print("\110\145\154\154\157") Output: Hello
13	\xHH	Character with hex value.	print("\x48\x65\x6c\x6c\x6f") Output: Hello

Here is the simple example of escape sequence.

```
print("C:\\Users\\Bhumika\\Python39\\Lib")
print("This is the \n multiline quotes")
print("This is \x48\x45\x58 representation")
Output:
C:\Users\Bhumika\Python39\Lib
This is the
 multiline quotes
This is HEX representation
```

Python String Formatting Using % Operator

Python allows us to use the format specifiers used in C's printf statement. The format specifiers in Python are treated in the same way as they are treated in C. However, Python provides an additional operator %, which is used as an interface between the format specifiers and their values. In other words, we can say that it binds the format specifiers to the values.**Consider the following example.**

```
Integer = 10;
Float = 1.290
String = "Mihir"
```

```
print("Hi I am Integer ... My value is %d\nHi I am float ... My value is %f\nHi I am string ... My value is %s"%(Integer,Float,String))
Output:
Hi I am Integer ... My value is 10
Hi I am float ... My value is 1.290000
Hi I am string ... My value is Mihir
```

3.1.3 String Methods: (center, count, join, len, max, min, replace, lower, upper, replace, split)

No	Method	Description

1	format(value)	It returns a formatted version of S, using the passed value.
		Eg: #named indexes:
		txt1 = "My name is {fname}, I'm {age}".format(fname = "John", age = 36)
		#numbered indexes:
		txt2 = "My name is {0}, I'm {1}".format("John",36)
		#empty placeholders:
		txt3 = "My name is {}, I'm {}".format("John",36)
		print(txt1)
		print(txt2)
		print(txt3)

2	center(width ,fillchar)	It returns a space padded string with the original string centred with equal number of left and right spaces.
		Eg: str = "Hello Javatpoint"
		# Calling function
		str2 = str.center(20,'#')
		# Displaying result
		print("Old value:", str)
		print("New value:", str2)

3	string.count(value, start, end)	It returns the number of times a specified value appears in the string. value: Required. A String. The string to value to search for start: Optional. An Integer. The position to start the search. Default is 0 end: Optional. An Integer. The position to end the search. Default is the end of the string Eg1: txt = "I love apples, apple are my favorite fruit" x = txt.count("apple") print(x) Eg2: txt = "I love apples, apple are my favorite fruit" x = txt.count("apple", 10, 24) print(x)

4	join(seq)	It merges the strings representation of the given sequence.
		Eg : str = "->" # string
		list = {'Java','C#','Python'} # iterable
		str2 = str.join(list)
		print(str2)

5	len(string)	It returns the length of a string. Eg: x = len("Hello") print(x)

6	max()	It returns the item with the highest value, or the item with the highest value in an iterable. If the values are strings, an alphabetically comparison is done. Eg: x = max("Mike", "John", "Vicky") print(x)
7	min()	It returns the item with the lowest value, or the item with the lowest value in an iterable. If the values are strings, an alphabetically comparison is done. Eg: x = min("Bhumi", "John", "Vicky") print(x)

8	replace(old,new[,count])	It replaces the old sequence of characters with the new sequence. The max characters are replaced if max is given.
		Eg1: txt = "one one was a race horse, two two was one too."
		x = txt.replace("one", "three")
		print(x)
		Eg2: txt = "one one was a race horse, two two was one too."
		x = txt.replace("one", "three", 2)
		print(x)

9	upper()	It converts all the characters of a string to Upper Case.
		Eg: txt = "Hello my friends"
		x = txt.upper()
		print(x)

10	lower()	It converts all the characters of a string to Lower case.
		Eg1: txt = "Hello my FRIENDS"
		x = txt.lower()
		print(x)

11	split(separator, maxsplit)	Splits the string according to the delimiter str. The string splits according to the space if the delimiter is not provided. It returns the list of substring concatenated with the delimiter.
		separator :Optional. Specifies the separator to use when splitting the string. By default any whitespace is a separator
		maxsplit :Optional. Specifies how many splits to do. Default value is -1, which is "all occurrences"
		eg1: txt = "apple#banana#cherry#orange"
		x = txt.split("#")
		print(x)
		Eg2: txt = "apple#banana#cherry#orange"
		# setting the maxsplit parameter to 1, will return a list with 2 elements!
		x = txt.split("#", 1)
		print(x)

3.2 Operators:
The operator can be defined as a symbol which is responsible for a particular operation between two operands. Operators are the pillars of a program on which the logic is built in a specific programming language. Python provides a variety of operators, which are described as follows.

3.2.1 Arithmetic Operators (+,-,*, /, %,,//)**
Arithmetic operators are used to perform arithmetic operations between two operands. It includes + (addition), - (subtraction), *(multiplication), /(divide), %(reminder), //(floor division), and exponent (**) operators.
Consider the following table for a detailed explanation of arithmetic operators.

Operator	**Description**
+ (Addition)	It is used to add two operands. For example, if a = 20, b = 10 => a+b = 30
- (Subtraction)	It is used to subtract the second operand from the first operand. If the first operand is less than the second operand, the value results negative. For example, if a = 20, b = 10 => a - b = 10
/ (divide)	It returns the quotient after dividing the first operand by the second operand. For example, if a = 20, b = 10 => a/b = 2.0
*** (Multiplication)**	It is used to multiply one operand with the other. For example, if a = 20, b = 10 => a * b = 200
% (reminder)	It returns the reminder after dividing the first operand by the second operand. For example, if a = 20, b = 10 => a%b = 0
**** (Exponent)**	It is an exponent operator represented as it calculates the first operand power to the second operand.
// (Floor division)	It gives the floor value of the quotient produced by dividing the two operands.

3.2.2 Assignment Operators (=,+=,-=,/=,*=,//=)
The assignment operators are used to assign the value of the right expression to the left operand. The assignment operators are described in the following table.

Operator	**Description**
=	It assigns the value of the right expression to the left operand.
+=	It increases the value of the left operand by the value of the right operand and assigns the modified value back to left operand. For example, if a = 10, b = 20 => a+ = b will be equal to a = a+ b and therefore, a = 30.
-=	It decreases the value of the left operand by the value of the right operand and assigns the modified value back to left operand. For example, if a = 20, b = 10 => a- = b will be equal to a = a- b and therefore, a = 10.
***=**	It multiplies the value of the left operand by the value of the right operand and assigns the modified value back to then the left operand. For example, if a = 10, b = 20 => a* = b will be equal to a = a* b and therefore, a = 200.
%=	It divides the value of the left operand by the value of the right operand and assigns the reminder back to the left operand. For example, if a = 20, b = 10 => a % = b will be equal to a = a % b and therefore, a = 0.
****=**	a**=b will be equal to a=a**b, for example, if a = 4, b =2, a**=b will assign 4**2 = 16 to a.
//=	A//=b will be equal to a = a// b, for example, if a = 4, b = 3, a//=b will assign 4//3 = 1 to a.

3.2.3 Comparison Operators (==, !=, >,<,>=,<=)

Comparison operators are used to comparing the value of the two operands and returns Boolean true or false accordingly. The comparison operators are described in the following table.

Operator	Description
==	If the value of two operands is equal, then the condition becomes true.
!=	If the value of two operands is not equal, then the condition becomes true.
<=	If the first operand is less than or equal to the second operand, then the condition becomes true.
>=	If the first operand is greater than or equal to the second operand, then the condition becomes true.
>	If the first operand is greater than the second operand, then the condition becomes true.
<	If the first operand is less than the second operand, then the condition becomes true.

3.2.4 Logical Operators (and, or, not)

The logical operators are used primarily in the expression evaluation to make a decision. Python supports the following logical operators.

Operator	Description
and	If both the expression are true, then the condition will be true. If a and b are the two expressions, a → true, b → true => a and b → true.
or	If one of the expressions is true, then the condition will be true. If a and b are the two expressions, a → true, b → false => a or b → true.
not	If an expression a is true, then not (a) will be false and vice versa.

3.2.5 Identity and member operators (is, is not, in, not in)

Membership Operators

Python membership operators are used to check the membership of value inside a Python data structure. If the value is present in the data structure, then the resulting value is true otherwise it returns false.

Operator	Description
in	It is evaluated to be true if the first operand is found in the second operand (list, tuple, or dictionary).
not in	It is evaluated to be true if the first operand is not found in the second operand (list, tuple, or dictionary).

Identity Operators

The identity operators are used to decide whether an element certain class or type.

Operator	Description
is	It is evaluated to be true if the reference present at both sides point to the same object.
is not	It is evaluated to be true if the reference present at both sides do not point to the same object.

Operator Precedence

The precedence of the operators is essential to find out since it enables us to know which operator should be evaluated first. The precedence table of the operators in Python is given below.

Operator	Description
**	exponent operator is given priority over all the others used in the expression.
~ + -	negation, unary plus, and minus.
* / % //	multiplication, divide, modules, reminder, and floor division.
+ -	Binary plus, and minus
>> <<	Left shift. and right shift
&	Binary and(Bitwise)
^ |	Binary xor, and or (Bitwise)
<= < > >=	Comparison operators (less than, less than equal to, greater than, greater then equal to).
<> == !=	Equality operators.
= %= /= //= -= += *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

A.Y-2020-2021

FYBCA-SEM2-204 - Programming Skills

UNIT-4: Python conditional and iterative statements

4.1 if statement, if..elif statement, if..elif…else statements, nested if

Decision making is the most important aspect of almost all the programming languages. As the name implies, decision making allows us to run a particular block of code for a particular decision. Here, the decisions are made on the validity of the particular conditions. Condition checking is the backbone of decision making.

In python, decision making is performed by the following statements.

Statement	Description
if statement	if evaluates the test expression and will execute statement(s) only if the test expression is True. If the test expression is False, the statement(s) is not executed.
If…else statement	The if..else statement evaluates test expression and will execute the body of if only when the test condition is True. If the condition is False, the body of else is executed. Indentation is used to separate the blocks.
if...elif...else statement	The elif is short for else if. It allows us to check for multiple expressions.
nested if statement	Nested if statements enable us to use if…else statement inside an outer if statement.

Indentation in Python

For the ease of programming and to achieve simplicity, python doesn't allow the use of parentheses for the block level code. In Python, indentation is used to declare a block. If two statements are at the same indentation level, then they are the part of the same block.

Generally, four spaces are given to indent the statements which are a typical amount of indentation in python.

Indentation is the most used part of the python language since it declares the block of code. All the statements of one block are intended at the same level indentation.

if statement: It is used to test a particular condition and if the condition is true, it executes a block of code known as if-block. The condition of if statement can be any valid logical expression which can be either evaluated true or false.

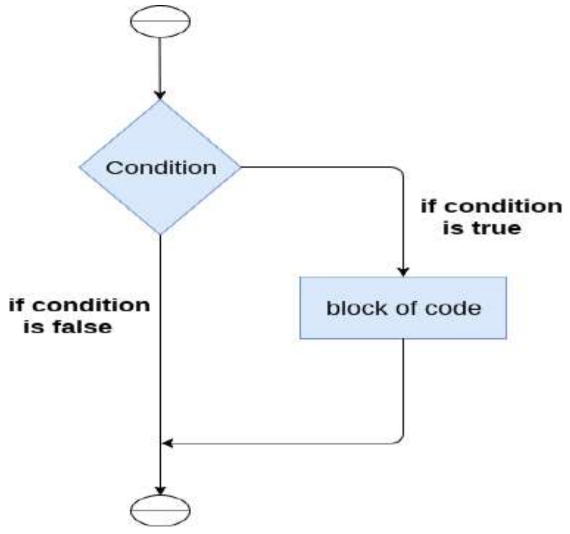


Figure: Flowchart of if statement

The syntax of the if-statement is given below.

```
if expression:

        statement
```

Example 1: Program to print even number. (even.py)

```
num = int(input("enter the number?"))

if num%2 == 0:

    print("Number is even")
```

Example 2: Program to print the largest of the three numbers. (largenum.py)

```
a = int(input("Enter a : "))
b = int(input("Enter b : "))
c = int(input("Enter c : "))
if a>b and a>c:
    print("a is largest")
if b>a and b>c:
    print("b is largest")
if c>a and c>b:
    print("c is largest")
```

The if-else statement

The if-else statement provides an else block combined with the if statement which is executed in the false case of the condition. If the condition is true, then the if-block is executed. Otherwise, the else-block is executed.

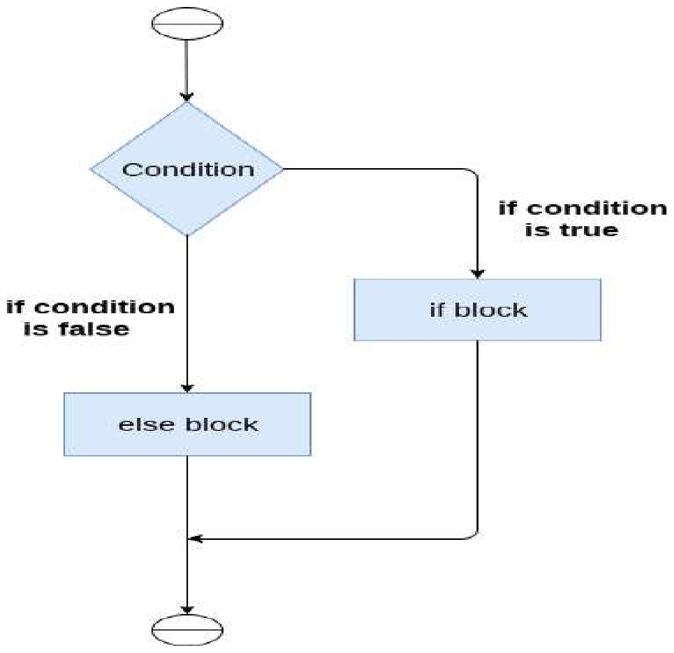


Figure: Flowchart of if-else statement

The syntax of the if-else statement is given below.

```
if condition:
        #block of statements
else:
        #another block of statements (else-block)
```

Example 1: Program to check whether a person is eligible to vote or not.

```
age = int (input("Enter your age? "))
if age >= 18:
   print("You are eligible to vote !!")
else:
   print("Sorry ! you have to wait !!")
```

Example 2: Program to check whether a number is even or not.

```
num = int(input("enter the number?"))
if num%2 == 0:
   print("Number is even...")
else:
   print("Number is odd...")
```

The elif statement

The elif statement enables us to check multiple conditions and execute the specific block of statements depending upon the true condition among them. We can have any number of elif statements in our program depending upon our need. However, using elif is optional. The elif statement works like an if-else-if ladder statement in C. It must be succeeded by an if statement.

The syntax of the if...elif...else statement is given below.

```
if expression 1:
        # block of statements
elif expression 2:
        # block of statements
elif expression 3:
        # block of statements
else:
        # block of statements
```

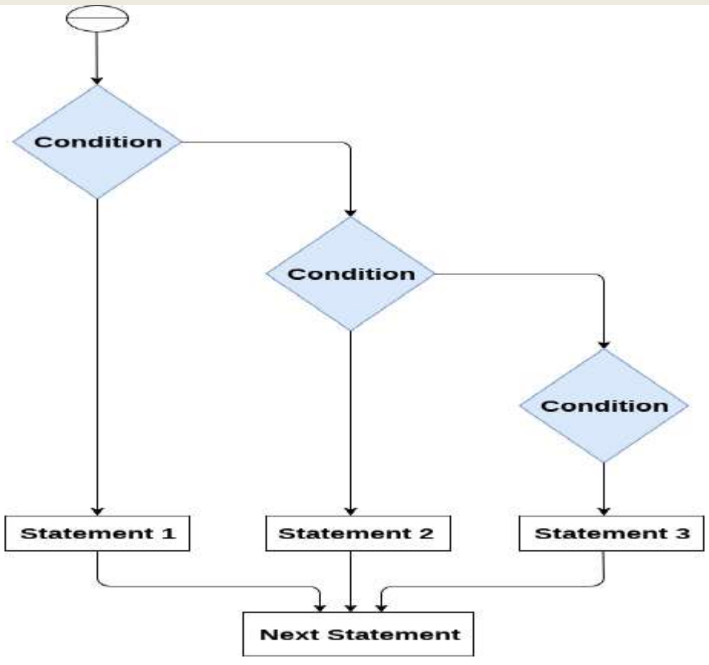


Figure: Flowchart of if...elif...else statement

Example 1 Example of

```
number = int(input("Enter the number : "))
if number==10:
   print("number is equals to 10")
elif number==50:
   print("number is equal to 50");
elif number==100:
   print("number is equal to 100");
else:
   print("number is not equal to 10, 50 or 100");
```

Example2: Python Program to take in the marks of 5 subjects and display the grade

```
rno=input("Enter Student Roll Number :")
sname=input("Enter Student Name :")
sub1=int(input("Enter marks of HTML subject: "))
sub2=int(input("Enter marks of ET & IT subject: "))
sub3=int(input("Enter marks of os subject: "))
sub4=int(input("Enter marks of PS subject: "))
sub5=int(input("Enter marks of RDBMS subject: "))

tot=sub1 + sub2 + sub3 + sub4 + sub5
avg=tot / 5

print("Student Roll No is : ",rno)
print("Student Name is :",sname)
print("Marks of HTML subject :",sub1)
print("Marks of ET & IT subject :",sub2)
print("Marks of os subject :",sub3)
print("Marks of PS subject :",sub4)
print("Marks of RDBMS subject :",sub5)
print("Total Marks : ",tot)
print("Percentage Marks : ",avg)
if(avg>=80):
   print("Grade: A+")
elif(avg>=70):
   print("Grade: A")
elif(avg>=60):
   print("Grade: B")
elif(avg>=50):
   print("Grade: C")
elif(avg>=35):
   print("Grade: D")
else:
   print("Grade: Fail")
```

Python Nested if statements

We can have if...elif...else statement inside another if...elif...else statement. This is called nesting in computer programming. Any number of these statements can be nested inside one another. Indentation is the only way to figure out the level of nesting. They can get confusing, so they must be avoided unless necessary.

Python Nested if Example

```
'''In this program, we input a number check if the number is positive or
   Negative or zero and display an appropriate message
   This time we use nested if statement    '''

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

4.2 Iterative statements:

The flow of the programs written in any programming language is sequential by default. Sometimes we may need to alter the flow of the program. The execution of a specific code may need to be repeated several numbers of times.

For this purpose, programming languages provide various types of loops which are capable of repeating some specific code several numbers of times. Consider the following diagram to understand the working of a loop statement.

Why we use loops in python?

The looping simplifies the complex problems into the easy ones. It enables us to alter the flow of the program so that instead of writing the same code again and again, we can repeat the same code for a finite number of times. For example, if we need to print the first 10 natural numbers then, instead of using the print statement 10 times, we can print inside a loop which runs up to 10 iterations.

Advantages of loops

There are the following advantages of loops in Python.

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

There are the following loop statements in Python.

Loop Statement	**Description**
while loop	The while loop is to be used in the scenario where we don't know the number of iterations in advance. The block of statements is executed in the while loop until the condition specified in the while loop is satisfied. It is also called a pre-tested loop.
for loop	The for loop is used in the case where we need to execute some part of the code until the given condition is satisfied. The for loop is also called as a pre-tested loop. It is better to use for loop if the number of iteration is known in advance.

4.2.1 while loop, nested while loop, break , continue statements.

while loop : The Python while loop allows a part of the code to be executed until the given condition returns false. It is also known as a pre-tested loop.
It can be viewed as a repeating if statement. When we don't know the number of iterations then the while loop is most effective to use.

Syntax:

```
while expression:
    statements
```

Here, the statements can be a single statement or a group of statements. The expression should be any valid Python expression resulting in true or false. The true is any non-zero value and false is 0.

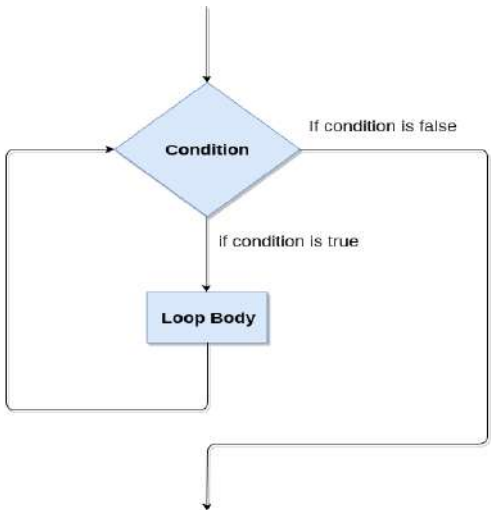


Figure: Flowchart of while loop

Example

```
i=1
number = int(input("Enter the number:"))
while i <= 10:
   print("%d X %d = %d \n"%(number, i, number * i))
   i = i+1
```

nested while loop

Declaration
The syntax of the nested- while loop in Python as follows:

Syntax

```
while expression:
        while expression:
                statement(s)
statement(s)
```

How works nested while loop

In the nested-while loop in Python, Two type of while statements are available:

1. Outer while loop
2. Inner while loop

Initially, Outer loop test expression is evaluated only once.

When it return true, the flow of control jumps to the inner while loop. The inner while loop executes to completion. However, when the test expression is false, the flow of control comes out of inner while loop and executes again from the outer while loop only once. This flow of control persists until test expression of the outer loop is false.

Thereafter, if test expression of the outer loop is false, the flow of control skips the execution and goes to rest.

Example1:

```
i=1
while i<=3 :
    print(i ,"Outer loop is executed only once")
    j=1
    while j<=3:
        print(j ,"Inner loop is executed until to completion")
        j+=1
    i+=1;
```

Example2: nested while loop to find the prime numbers from 2 to 100

```
i = 2
while(i < 100):
   j = 2
   while(j <= (i/j)):
      if not(i%j): break
      j = j + 1
   if (j > i/j) : print i, " is prime"
   i = i + 1

print "Good bye!"
```

Loop Control Statements

We can change the normal sequence of while loop's execution using the loop control statement. When the while loop's execution is completed, all automatic objects defined in that scope are demolished. Python offers the following control statement to use within the while loop.

1. Continue Statement - When the continue statement is encountered, the control transfer to the beginning of the loop. Let's understand the following example.

Example:

```
# prints all letters except 'a' and 't'
i = 0
str1 = 'javatpoint'

while i < len(str1):
        if str1[i] == 'a' or str1[i] == 't':
                i += 1
                continue
        print('Current Letter :', str1[i])
        i += 1
```

2. Break Statement - When the break statement is encountered, it brings control out of the loop.

Example:

```
# The control transfer is transferred
# when break statement soon it sees t
i = 0
str1 = 'javatpoint'
while i < len(str1):
        if str1[i] == 't':
                i += 1
                break
        print('Current Letter :', str1[i])
        i += 1
```

3. Pass Statement - The pass statement is used to declare the empty loop. It is also used to define empty class, function, and control statement. In Python programming, the pass statement is a null statement. The difference between a comment and a pass statement in Python is that while the interpreter ignores a comment entirely, pass is not ignored. However, nothing happens when the pass is executed. It results in no operation (NOP).

Syntax of pass

```
pass
```

We generally use it as a placeholder.
Suppose we have a loop or a function that is not implemented yet, but we want to implement it in the future. They cannot have an empty body. The interpreter would give an error. So, we use the pass statement to construct a body that does nothing.
Suppose we have a loop, and we do not want to execute right this moment, but we will execute in the future. Here we can use the pass. Consider the following example.

Example of pass

```
# pass is just a placeholder for we will add functionality later.
values = {'P', 'y', 't', 'h','o','n'}
for val in values:
    pass
```

4.2.2 for loop, range, break, continue, pass and else with for loop, nested for loop.

for loop: The for loop in Python is used to iterate the statements or a part of the program several times. It is frequently used to traverse the data structures like list, tuple, or dictionary.The **syntax** of for loop in python is given below.

```
for iterating_var in sequence:
    statement(s)
```

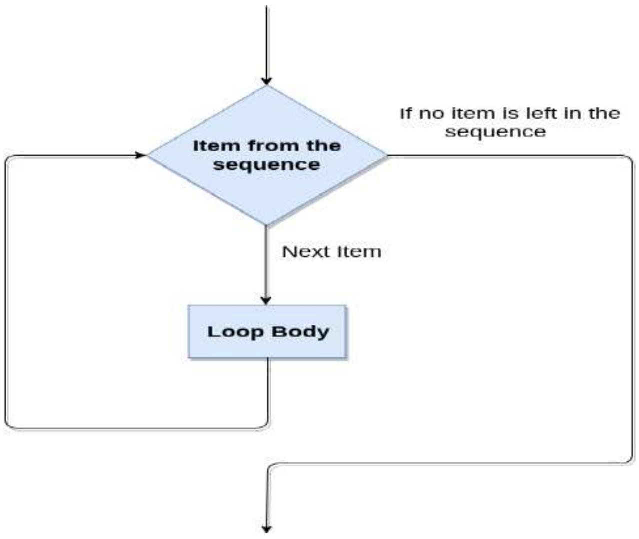


Figure: Flowchart of for loop

Example-1: Iterating string using for loop

```
str = "Python"
for i in str:
    print(i)
```

Example- 2: Program to print the table of the given number.

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

For loop Using range() function

The range() function: The range() function is used to generate the sequence of the numbers. If we pass the range(10), it will generate the numbers from 0 to 9.

Syntax:

```
range(start, stop, step size)
```

1. The start represents the beginning of the iteration.
2. The stop represents that the loop will iterate till stop-1. The range(1,5) will generate numbers 1 to 4 iterations. It is optional.
3. The step size is used to skip the specific numbers from the iteration. It is optional to use. By default, the step size is 1. It is optional.

Example-1: Program to print numbers in sequence.

```
for i in range(10):
    print(i,end = ' ')
```

Example-2: Program to print table of given number.

```
n = int(input("Enter the number "))
for i in range(1,11):
   c = n*i
   print(n,"*",i,"=",c)
```

Example-3: Program to print even number using step size in range().

```
n = int(input("Enter the number "))
for i in range(2,n,2):
   print(i)
```

Nested for loop in python

Python allows us to nest any number of for loops inside another for loop. The inner loop is executed n number of times for every iteration of the outer loop. The syntax is given below.

Syntax

```
for iterating_var1 in sequence:      #outer loop
    for iterating_var2 in sequence:        #inner loop
            #block of statements
#Other statements
```

Example- 1: Nested for loop to print Triangle Pattern.

```
rows = int(input("Enter the rows:"))      # User input for number of rows
# Outer loop will print number of rows
for i in range(0,rows+1):
# Inner loop will print number of Astrisk
    for j in range(i):
        print("*",end = '')
    print()
```

Example-2: Program to number pyramid.

```
rows = int(input("Enter the rows"))
for i in range(0,rows+1):
    for j in range(i):
        print(i,end = '')
    print()
```

Using else statement with for loop

Unlike other languages like C, C++, or Java, Python allows us to use the else statement with the for loop which can be executed only when all the iterations are exhausted. Here, we must notice that if the loop contains any of the break statement then the else statement will not be executed.

Example 1

```
for i in range(0,5):
    print(i)
else:
    print("for loop completely exhausted, since there is no break.")
```

Example 2

```
for i in range(0,5):
    print(i)
    break;
else:
    print("for loop is exhausted");
print("The loop is broken due to break statement...came out of the loop")
```

In the above example, the loop is broken due to the break statement; therefore, the else statement will not be executed. The statement present immediate next to else block will be executed.

4.3 List: creating list, indexing, accessing list members, range in list, List methods (append, clear, copy, count, index, insert, pop, remove, reverse, sort).

A **list** in Python is used to store the sequence of various types of data. Python lists are mutable type its mean we can modify its element after it created. However, Python consists of six data-types that are capable to store the sequences, but the most common and reliable type is the list.

A list can be defined as a collection of values or items of different types. The items in the list are separated with the comma (,) and enclosed with the square brackets [].

A list can be define as below:

```
L1 = ["John", 102, "USA"]
L2 = [1, 2, 3, 4, 5, 6]
print(type(L1))
print(type(L2))
#If we try to print the type of L1, L2 using type() function then it will come out to be a list.
OUTPUT
<class 'list'>
<class 'list'>
```

Characteristics of Lists

The list has the following characteristics:

1. The lists are ordered.
2. The element of the list can access by index.
3. The lists are the mutable type.
4. The lists are mutable types.
5. A list can store the number of various elements.

The following are the properties of a list.

1. **Mutable:** The elements of the list can be modified. We can add or remove items to the list after it has been created.
2. **Ordered**: The items in the lists are ordered. Each item has a unique index value. The new items will be added to the end of the list.
3. **Heterogeneous:** The list can contain different kinds of elements i.e; they can contain elements of string, integer, boolean or any type.
4. **Duplicates:** The list can contain duplicates i.e., lists can have two items with the same values.

Let's check the first statement that lists are the ordered.

```
a = [1,2,"Peter",4.50,"Ricky",5,6]
b = [1,2,5,"Peter",4.50,"Ricky",6]
print(a==b)
output:
False
```

Both lists have consisted of the same elements, but the second list changed the index position of the 5th element that violates the order of lists. When compare both lists it returns the false.

The list example in detail

```
emp = ["John", 102, "USA"]
Dep1 = ["CS",10]
Dep2 = ["IT",11]
HOD_CS = [10,"Mr. Holding"]
HOD_IT = [11, "Mr. Bewon"]
print("printing employee data...")
print("Name : %s, ID: %d, Country: %s" %(emp[0],emp[1],emp[2]))
print("printing departments...")
print("Department 1:\nName: %s, ID: %d\nDepartment 2:\nName: %s, ID: %s" %
(Dep1[0],Dep1[1],Dep2[0],Dep2[1]))
print("HOD Details.....")
print("CS HOD Name: %s, Id: %d"%(HOD_CS[1],HOD_CS[0]))
print("IT HOD Name: %s, Id: %d"%(HOD_IT[1],HOD_IT[0]))
print("\n \n",type(emp),type(Dep1),type(Dep2),type(HOD_CS),type(HOD_IT))
```

List indexing and slicing

The indexing is processed in the same way as it happens with the strings. The elements of the list can be accessed by using the **slice operator [].**

The index starts from 0 and goes to length - 1. The first element of the list is stored at the 0th index, the second element of the list is stored at the 1st index, and so on.

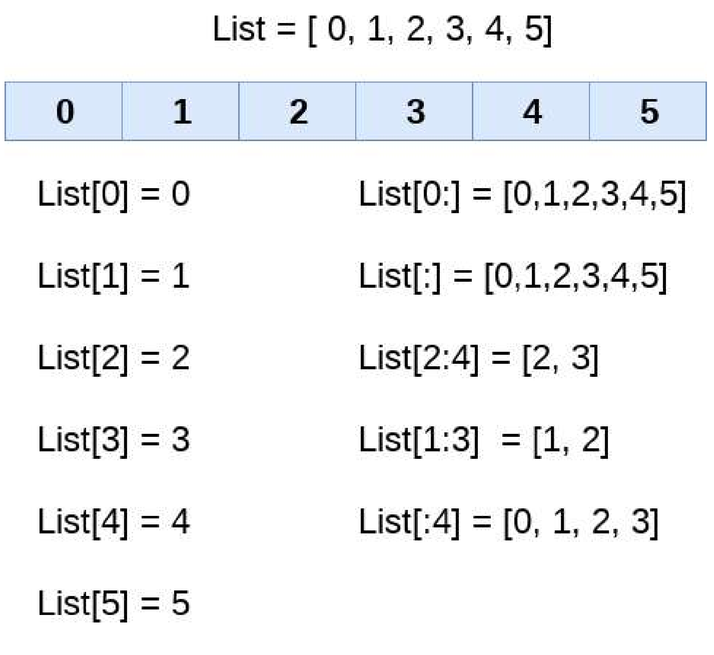


Figure: List indexing and slicing

We can get the sub-list of the list using the following syntax.

```
list_varible[start:stop:step]
```

1. The start denotes the starting index position of the list.
2. The stop denotes the last index position of the list.
3. The step is used to skip the nth element within a start:stop

Consider the following example:

```
list = [1,2,3,4,5,6,7]
print(list[0])
print(list[1])
print(list[2])
print(list[3])
# Slicing the elements
print(list[0:6])
# By default the index value is 0 so its starts from the 0th element and go for index -1.
print(list[:])
print(list[2:5])
print(list[1:6:2])
```

OUTPUT:

```
1
2
3
4
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 4, 5, 6, 7]
[3, 4, 5]
[2, 4, 6]
```

Unlike other languages, Python provides the flexibility to use the negative indexing also. The negative indices are counted from the right. The last element (rightmost) of the list has the index -1; its adjacent left element is present at the index -2 and so on until the left-most elements are encountered.

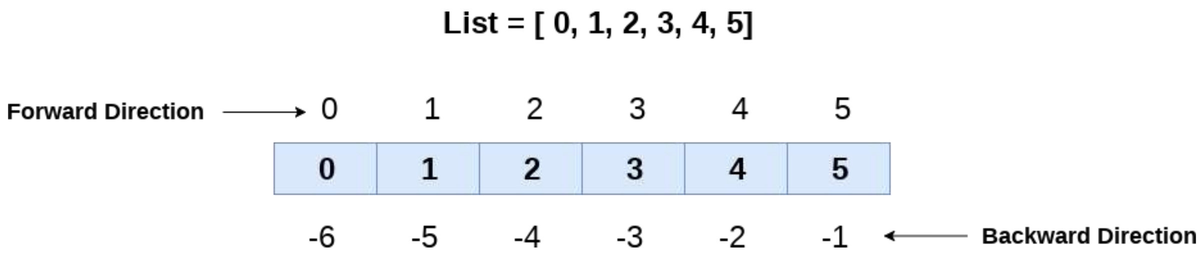


Figure: List negative indexing

Let's have a look at the following example where we will use negative indexing to access the elements of the list.

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

OUTPUT:

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

We can use the index operator [] to access an item in a list. In Python, indices start at 0. So, a list having 5 elements will have an index from 0 to 4.
Trying to access indexes other than these will raise an IndexError. <u>The index must be an integer.</u> We can't use float or other types, this will result in TypeError.
Nested lists are accessed using nested indexing.

```
# List indexing
my_list = ['p', 'r', 'o', 'b', 'e']

# Output: p
print(my_list[0])

# Output: o
print(my_list[2])

# Output: e
print(my_list[4])

# Nested List
n_list = ["Happy", [2, 0, 1, 5]]
# Nested indexing
print(n_list[0][1])

print(n_list[1][3])

# Error! Only integer can be used for indexing
print(my_list[4.0])
```

Output:

```
p
o
e
a
5
Traceback (most recent call last):
  File "<string>", line 21, in <module>
TypeError: list indices must be integers or slices, not float
```

Iterating a List
A list can be iterated by using a for - in loop. A simple list containing four strings, which can be iterated as follows.

```
list = ["John", "David", "James", "Jonathan"]
for i in list:
    print(i)
```

OUTPUT:

```
John
David
James
Jonathan
```

Check if Item Exists
To determine if a specified item is present in a list use the in keyword:
Check if "apple" is present in the list:

```
thislist = ["apple", "banana", "cherry"]
if "apple" in thislist:
  print("Yes, 'apple' is in the fruits list")
```

Add/Change List Elements
Lists are mutable, meaning their elements can be changed unlike string or tuple.
We can use the assignment operator = to change an item or a range of items.

```
# Correcting mistake values in a list
odd = [2, 4, 6, 8]
# change the 1st item
odd[0] = 1

print(odd)

# change 2nd to 4th items
odd[1:4] = [3, 5, 7]
print(odd)
Output:
[1, 4, 6, 8]
[1, 3, 5, 7]
```

range() to a list in Python
Often times we want to create a list containing a continuous value like, in a range of 100-200. Let's discuss how to create a list using the range() function.

```
# Create a list in a range of 10-20
My_list = [range(10, 20, 1)]

# Print the list
print(My_list)

output:
[range(10, 20)]
```

As we can see in the output, the result is not exactly what we were expecting because Python does not unpack the result of the range() function.
Code #1: We can use argument-unpacking operator i.e. *.

```
# Create a list in a range of 10-20
My_list = [*range(10, 21, 1)]

# Print the list
print(My_list)

output:
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

As we can see in the output, the argument-unpacking operator has successfully unpacked the result of the range function.

❖ List methods (append, clear, copy, count, index, insert, pop, remove, reverse, sort).

1. Adding elements to the list

Python provides append() function which is used to add an element to the list. However, the append() function can only add value to the end of the list.

The syntax of the append() method is:

list.append(item)

append() Parameters
The method takes a single argument
item - an item to be added at the end of the list
The item can be numbers, strings, dictionaries, another list, and so on.

Return Value from append()

The method doesn't return any value (returns None).
Consider the following example in which, we are taking the elements of the list from the user and printing the list on the console.

```
ls =[]    #Declaring the empty list
#Number of elements will be entered by the user
n = int(input("Enter the number of elements in the list:"))
# for loop to take the input
for i in range(0,n):
    # The input is taken from the user and added to the list as the item
    ls.append(input("Enter the item:"))
print("printing the list items..")
# traversal loop to print the list items
for i in ls:
    print(i, end = " ")
Output:
Enter the number of elements in the list:5
Enter the item:25
Enter the item:46
Enter the item:12
Enter the item:75
Enter the item:42
printing the list items
25  46  12 75  42
```

2. Python List clear()

The clear() method removes all items from the list.

The syntax of clear() method is:

list.clear()

clear() Parameters

The clear() method doesn't take any parameters.

Return Value from clear()

The clear() method only empties the given list. It doesn't return any value.

Example 1: Working of clear() method

```
# Defining a list
list = [{1, 2}, ('a'), ['1.1', '2.2']]
# clearing the list
list.clear()
print('List:', list)
Output:
List: []
```

3. Python List copy()

The copy() method returns a shallow copy of the list.
A list can be copied using the = operator. For example,
old_list = [1, 2, 3]
new_list = old_list
The problem with copying lists in this way is that if you modify new_list, old_list is also modified. It is because the new list is referencing or pointing to the same old_list object.

```
old_list = [1, 2, 3]
new_list = old_list

new_list.append('a')      # add an element to list
print('New List:', new_list)
print('Old List:', old_list)
```
Output:
```
Old List: [1, 2, 3, 'a']
New List: [1, 2, 3, 'a']
```
However, if you need the original list unchanged when the new list is modified, you can use the copy() method.

The syntax of the copy() method is:
```
new_list = list.copy()
```

copy() parameters
The copy() method doesn't take any parameters.

Return Value from copy()
The copy() method returns a new list. It doesn't modify the original list.

Example 1: Copying a List
```
# mixed list
my_list = ['cat', 0, 6.7]

# copying a list
new_list = my_list.copy()

print('Copied List:', new_list)
```
Output:
```
Copied List: ['cat', 0, 6.7]
```
If you modify the new_list in the above example, my_list will not be modified.

4. Python List count()

The count() method returns the number of times the specified element appears in the list.

The syntax of the count() method is:
```
list.count(element)
```

count() Parameters
The count() method takes a single argument:
element - the element to be counted

Return value from count()
The count() method returns the number of times element appears in the list.

Example 1: Use of count()
```
vowels = ['a', 'e', 'i', 'o', 'i', 'u']
# count element 'i'
count = vowels.count('i')
```

```
# print count
print('The count of i is:', count)
# count element 'p'
count = vowels.count('p')
# print count
print('The count of p is:', count)
```
Output:
```
The count of i is: 2
The count of p is: 0
```

5. Python List index()

The index() method returns the index of the specified element in the list.

The syntax of the list index() method is:

list.index(element, start, end)

list index() parameters

The list index() method can take a maximum of three arguments:

1. element - the element to be searched
2. start (optional) - start searching from this index
3. end (optional) - search the element up to this index

Return Value from List index()

The index() method returns the index of the given element in the list.
If the element is not found, a ValueError exception is raised.
Note: The index() method only returns the first occurrence of the matching element.

Example 1: Find the index of the element

```
vowels = ['a', 'e', 'i', 'o', 'i', 'u'] # vowels list
# index of 'e' in vowels
index = vowels.index('e')
print('The index of e:', index)
index = vowels.index('I')
print('The index of i:', index)
```
Output:
```
The index of e: 1
Traceback (most recent call last):
  File "C:/Users/AMD/Desktop/mypython/LIST/index.py", line 5, in <module>
    index = vowels.index('I') # element 'i' is searched & index of the first 'I' is returned
ValueError: 'I' is not in list
```

Example 2: Index of the Element not Present in the List

```
vowels = ['a', 'e', 'i', 'o', 'u']
index = vowels.index('p')
print('The index of p:', index)
```
Output:
```
ValueError: 'p' is not in list
```

6. Python List insert()

The list insert() method inserts an element to the list at the specified index.

The syntax of the insert() method is

list.insert(i, elem)

Here, elem is inserted to the list at the ith index. All the elements after elem are shifted to the right.

insert() Parameters
The insert() method takes two parameters:
index - the index where the element needs to be inserted
element - this is the element to be inserted in the list
Notes:
If index is 0, the element is inserted at the beginning of the list.
If index is 3, the element is inserted after the 3rd element. Its position will be 4th.

Return Value from insert()
The insert() method doesn't return anything; returns None. It only updates the current list.

Example 1: Inserting an Element to the List

```
vowel = ['a', 'e', 'i', 'u']      # vowel list
# 'o' is inserted at index 3
# the position of 'o' will be 4th
vowel.insert(3, 'o')

print('Updated List:', vowel)
```

Output:

```
Updated List: ['a', 'e', 'i', 'o', 'u']
```

7. Python List pop()

The pop() method removes the item at the given index from the list and returns the removed item.

The syntax of the pop() method is:

```
list.pop(index)
```

pop() parameters
The pop() method takes a single argument (index).
The argument passed to the method is optional. If not passed, the default index -1 is passed as an argument (index of the last item).
If the index passed to the method is not in range, it throws IndexError: pop index out of range exception.

Return Value from pop()
The pop() method returns the item present at the given index. This item is also removed from the list.

Example 1: Pop item at the given index from the list

```
languages = ['Python', 'Java', 'C++', 'French', 'C']   # programming languages list
return_value = languages.pop(3)      # remove and return the 4th item
print('Return Value:', return_value)
print('Updated List:', languages)     # Updated List
```

Output:

```
Return Value: French
Updated List: ['Python', 'Java', 'C++', 'C']
```

Note: Index in Python starts from 0, not 1.
If you need to pop the 4th element, you need to pass 3 to the pop() method.

8. Removing elements from the list

Python provides the remove() function which is used to remove the first matching element from the list.

The syntax of the remove() method is:

```
list.remove(element)
```

remove() Parameters

The remove() method takes a single element as an argument and removes it from the list.

If the element doesn't exist, it throws ValueError: list.remove(x): x not in list exception.

Return Value from remove()

The remove() doesn't return any value (returns None).

Example 1: Remove element from the list

```
animals = ['cat', 'dog', 'rabbit', 'guinea pig']          # animals list
animals.remove('rabbit')     # 'rabbit' is removed
print('Updated animals list: ', animals)            # Updated animals List
```

Output:

```
Updated animals list:  ['cat', 'dog', 'guinea pig']
```

Example 2: Deleting element that doesn't exist

```
animals = ['cat', 'dog', 'rabbit', 'guinea pig']      # animals list
animals.remove('fish')  # Deleting 'fish' element
print('Updated animals list: ', animals)    # Updated animals List
```

Output:

```
Traceback (most recent call last):
  File "....... ", line 5, in <module>
    animal.remove('fish')
ValueError: list.remove(x): x not in list
```

Here, we are getting an error because the animals list doesn't contain 'fish'.

NOTE:

- If you need to delete elements based on the index (like the fourth element), you can use the pop() method.
- Also, you can use the Python del statement to remove items from the list.

Example 3: Consider the following example to understand this concept.

```
list = [0,1,2,3,4]
print("printing original list: ");
for i in list:
     print(i,end=" ")
list.remove(2)
print("\nprinting the list after the removal of first element...")
for i in list:
     print(i,end=" ")
```

OUTPUT:

```
printing original list:
0 1 2 3 4
printing the list after the removal of first element...
0 1 3 4
```

9. Python List sort()

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

The syntax of the sort() method is:

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

Alternatively, you can also use Python's built-in sorted() function for the same purpose.

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

Note: The simplest difference between sort() and sorted() is: sort() changes the list directly and doesn't return any value, while sorted() doesn't change the list and returns the sorted list.

sort() Parameters
By default, sort() doesn't require any extra parameters. However, it has two optional parameters:
1. reverse - If True, the sorted list is reversed (or sorted in Descending order)
2. key - function that serves as a key for the sort comparison

Return value from sort()
The sort() method doesn't return any value. Rather, it changes the original list.
If you want a function to return the sorted list rather than change the original list, use sorted().
Example 1: Sort a given list

```
vowels = ['e', 'a', 'u', 'o', 'i']   # vowels list
vowels.sort()                         # sort the vowels
print('Sorted list:', vowels)   # print vowels
```

Output:

```
Sorted list: ['a', 'e', 'i', 'o', 'u']
```

Sort in Descending order
The sort() method accepts a reverse parameter as an optional argument.
Setting reverse = True sorts the list in the descending order.

```
list.sort(reverse=True)
```

Alternately for sorted(), you can use the following code.

```
sorted(list, reverse=True)
```

Example 2: Sort the list in Descending order

```
vowels = ['e', 'a', 'u', 'o', 'i']
vowels.sort(reverse=True)
print('Sorted list (in Descending):', vowels)
```

Output:

```
Sorted list (in Descending): ['u', 'o', 'i', 'e', 'a']
```

10. Python List reverse()
The reverse() method reverses the elements of the list.
The syntax of the reverse() method is:

```
list.reverse()
```

reverse() parameter
The reverse() method doesn't take any arguments.
Return Value from reverse()
The reverse() method doesn't return any value. It updates the existing list.
Example 1: Reverse a List

```
systems = ['Windows', 'macOS', 'Linux']
print('Original List:', systems)
systems.reverse()           # List Reverse
print('Updated List:', systems) # updated list
```

Output:

```
Original List: ['Windows', 'macOS', 'Linux']
Updated List: ['Linux', 'macOS', 'Windows']
```

NOTE: There are other several ways to reverse a list. (Using Slicing Operator & reversed())

11. Python List extend()

The extend() method adds all the elements of an iterable (list, tuple, string etc.) to the end of the list.

The syntax of the extend() method is:

```
list1.extend(iterable)
```

Here, all the elements of iterable are added to the end of list1.
extend() Parameters
As mentioned, the extend() method takes an iterable such as list, tuple, string etc.

Return Value from extend()

The extend() method modifies the original list. It doesn't return any value.

Example 1: Using extend() Method

```
languages = ['French', 'English']
languages1 = ['Spanish', 'Portuguese']
languages.extend(languages1)   # appending language1 elements to language
print('Languages List:', languages)
```

Output:

```
Languages List: ['French', 'English', 'Spanish', 'Portuguese']
```

Python extend() Vs append()

If you need to add an element to the end of a list, you can use the append() method.

```
a1 = [1, 2]
a2 = [1, 2]
b = (3, 4)
a1.extend(b)      # a1 = [1, 2, 3, 4]
print(a1)
a2.append(b)      # a2 = [1, 2, (3, 4)]
print(a2)
```

Output:

```
[1, 2, 3, 4]
[1, 2, (3, 4)]
```