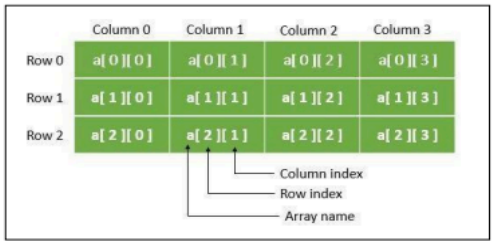
1.1.1 Declaring Two-Dimensional numeric array:
The two-dimensional array can be defined as an array of arrays. The 2D array is organized as matrices which can be represented as the collection of rows and columns. However, 2D arrays are created to implement a relational database look alike data structure. 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
```

Difference Between Structure and Union in C

Parameter	Structure	Union
Keyword	A user can deploy the keyword **struct** to define a Structure.	A user can deploy the keyword **union** to define a Union.
Internal Implementation	The implementation of Structure in C occurs internally- because it contains separate memory locations allotted to every input member.	In the case of a Union, the memory allocation occurs for only one member with the largest size among all the input variables. It shares the same location among all these members/objects.
Accessing Members	A user can access individual members at a given time.	A user can access only one member at a given time.
Syntax	The Syntax of declaring a Structure in C is: struct [structure name] { type element_1; type element_2; . . } variable_1, variable_2, ...;	The Syntax of declaring a Union in C is: union [union name] { type element_1; type element_2; . . } variable_1, variable_2, ...;
Size	A Structure does not have a shared location for all of its members. It makes the size of a Structure to be greater than or equal to the sum of the size of its data members.	A Union does not have a separate location for every member in it. It makes its size equal to the size of the largest member among all the data members.
Value Altering	Altering the values of a single member does not affect the other members of a Structure.	When you alter the values of a single member, it affects the values of other members.
Storage of Value	In the case of a Structure, there is a specific memory location for every input data member. Thus, it can store multiple values of the various members.	In the case of a Union, there is an allocation of only one shared memory for all the input data members. Thus, it stores one value at a time for all of its members.

Difference Between Structure and Array in C

Parameter	Structure in C	Array in C
Definition	It is a type of data structure in the form of a container that holds variables of different types.	It is a type of data structure that works as a container to hold variables of the very same type. Array does not support variables of multiple data types.
Allocation of Memory	In a structure, the memory allocation for the input data doesn't require being in consecutive memory locations.	The array stores the input data in a memory allocation of contiguous type. It means that the array stores its data in a type of memory model where the memory blocks hold consecutive addresses (it assigns memory blocks consecutively).
Accessibility	For a user to access the elements present in a structure, they require the name of that particular element (it is mandatory for retrieval).	On the other hand, any user can easily access the elements by index in an array's case.
Pointer	A structure holds no concept of internal Pointer.	An array, on the other hand, implements Pointer internally. It always points at the very first element present in the array.
Instantiation	One can create an object from the structure after a later declaration in its program.	An array does not allow the creation of an object after the declaration.
Types of Data Type Variables	A structure includes multiple forms of data-type variables in the form of input.	A user cannot have multiple forms of data-type variables in an array because it supports only the same form of data-type variables.
Performance	A structure becomes very slow in performance due to the presence of multiple data-types. The process of searching and accessing elements becomes very slow in these.	The process of searching and accessing elements is much faster in the case of an array due to the absence of multiple data-type variables. It is, thus, better and faster in performance.
Syntax	**struct** sructure_name{ element type 1; element type 2; . . } variable no.1, variable no.2, . .;	type name_of_array [size]

❖ **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();
   ... .. ...
}
```

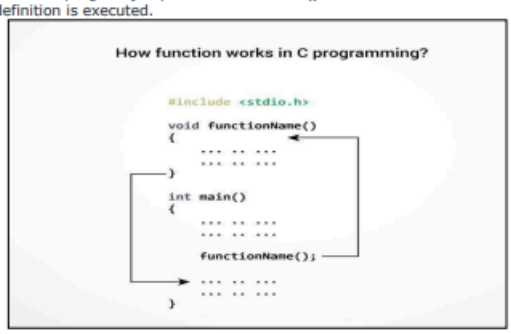
The execution of a C program begins from the main() function.

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.



❖ **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.

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

Chapter 2: Python Fundamentals

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

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
```

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://python37')      # 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
```

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.

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

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)

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.

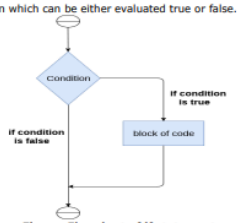


Figure: Flowchart of if statement

The syntax of the if-statement is given below.

```
if expression:
    statement
```


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.

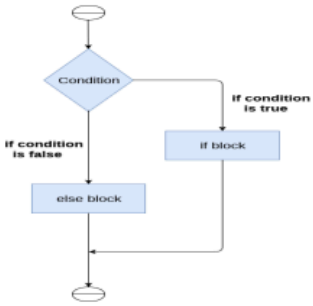


Figure: Flowchart of if-else statement

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 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")
```

A.Y-2020-2021
FYBCA-SEM2-204 - Programming Skills

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
```

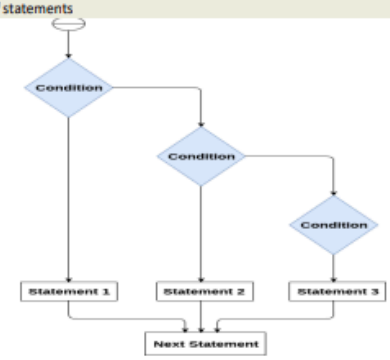


Figure: Flowchart of if...elif...else statement

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");
```

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")
```

4.2.1 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.

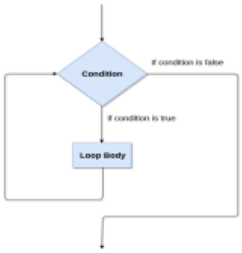


Figure: Flowchart of while loop

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)
```



```
                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)
```

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':
```

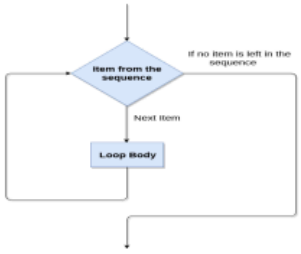


Figure: Flowchart of for loop

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:
```


❖ **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)
```

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())

FYBCA-SEM2-204 - Programming Skills

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=...)
```