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.

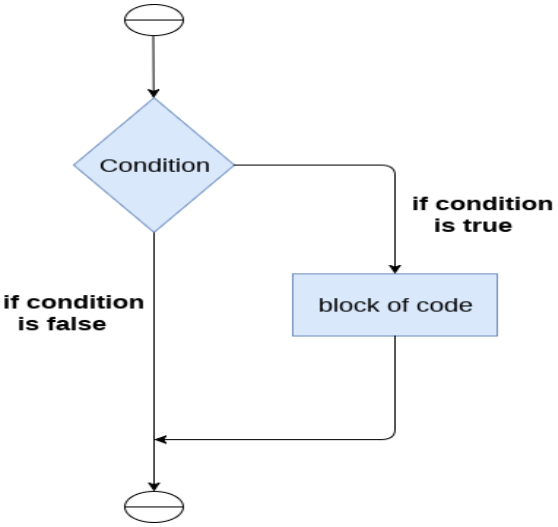

Figure: Flowchart of if statement

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.

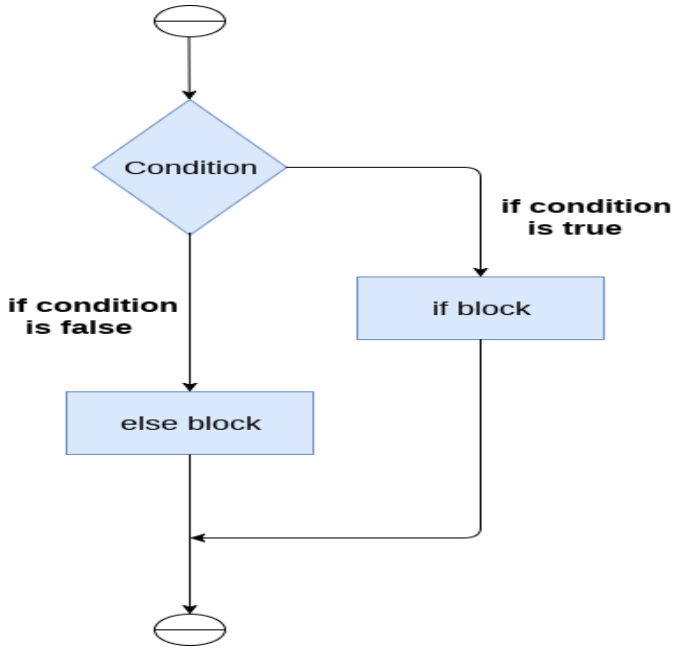

Figure: Flowchart of if-else statement

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
```

Flowchart of if...elif...else statement

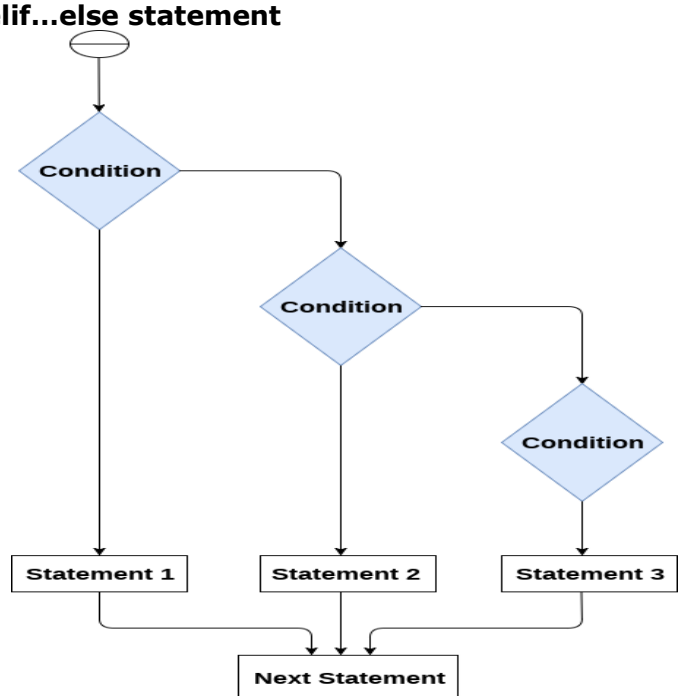

Figure: Flowchart of if...elif...else statement

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.

Example:

```
# if the number is positive or Negative or zero and display an appropriate message
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	for loop is used in the case where we need to execute some part of the code until the given condition is satisfied. 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 condition:
        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.

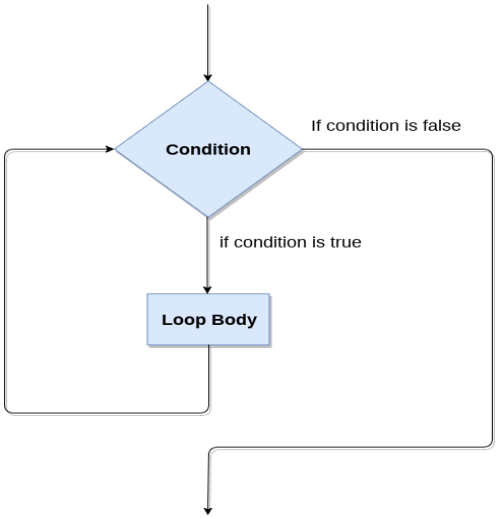

Figure: Flowchart of while loop

Example for while loop

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

nested while loop

Declaration

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

Syntax

```
#statement(s)
while condition_1 :
    #statement(s)
    while condition_2 :
        #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: Nested While Loop

```
i=1
while i<=3:
    print(i,"Outer loop is executed ....")
    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 = 'SDJ International College'

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 = "SDJ International college"
print("Current Letter :")
while i < len(str1):
        if str1[i] == "t" :
                i += 1
                break
        print(str1[i],end=' ')
        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)
```

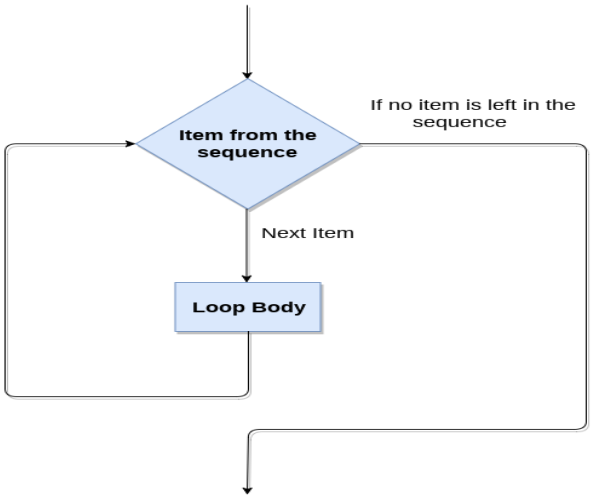

Figure: Flowchart of for loop

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_var in sequence:
          for iterating_var in sequence:
                    statements(s)
           statements(s)
```

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

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

List = [0, 1, 2, 3, 4, 5]

0	1	2	3	4	5

List[0] = 0

List[1] = 1

List[2] = 2

List[3] = 3

List[4] = 4

List[5] = 5

List[0:] = [0,1,2,3,4,5]

List[:] = [0,1,2,3,4,5]

List[2:4] = [2, 3]

List[1:3] = [1, 2]

List[:4] = [0, 1, 2, 3]

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])
print(list[0:6])    # Slicing the elements
# 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.

List = [0, 1, 2, 3, 4, 5]

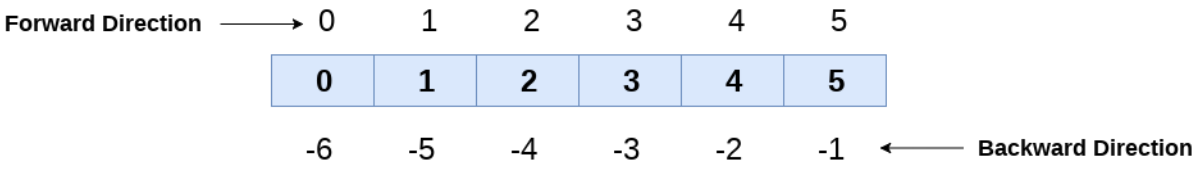

Figure: List negative indexing

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. The index must be an integer. 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']
print(my_list[0])
```

```
print(my_list[2])
print(my_list[4])
n_list = ["Happy", [2, 0, 1, 5]]     # Nested List
print(n_list[0][1])                  # Nested indexing
print(n_list[1][3])
print(my_list[4.0])                   # Error! Only integer can be used for indexing
```

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:

```
fruitlist = ["apple", "banana", "cherry"]
if "apple" in fruitlist:
        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.

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

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

n = int(input("Enter the number of elements in the list:"))

for i in range(0,n):
        ls.append(input("Enter the item:"))

print("printing the list items..")
for i in ls:                    # traversal loop to print the list items
        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

```
list = [{1, 2}, ('a'), ['1.1', '2.2']]        # Defining a list
# 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.

syntax:

```
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 thefirst 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: ");
fori in list:
print(i,end=" ")
list.remove(2)
print("\nprinting the list after the removal of first element...")
fori 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)]
```