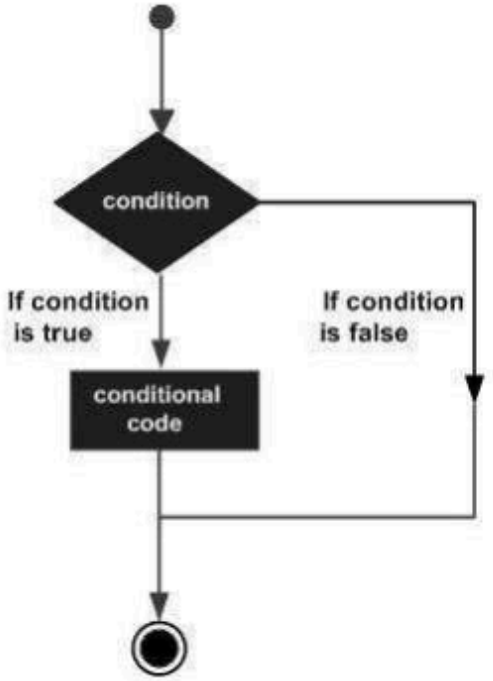
7. Python 3 – Decision Making

Decision-making is the anticipation of conditions occurring during the execution of a program and specified actions taken according to the conditions.

Decision structures evaluate multiple expressions, which produce TRUE or FALSE as the outcome. You need to determine which action to take and which statements to execute if the outcome is TRUE or FALSE otherwise.

Following is the general form of a typical decision making structure found in most of the programming languages-



Python programming language assumes any **non-zero** and **non-null** values as TRUE, and any **zero** or **null values** as FALSE value.

Python programming language provides the following types of decision-making statements.

Statement	Description
if statements	An if statement consists of a Boolean expression followed by one or more statements.
if...else statements	An if statement can be followed by an optional else statement, which executes when the boolean expression is FALSE.

nested if statements	You can use one if or else if statement inside another if or else if statement(s).

Let us go through each decision-making statement quickly.

IF Statement

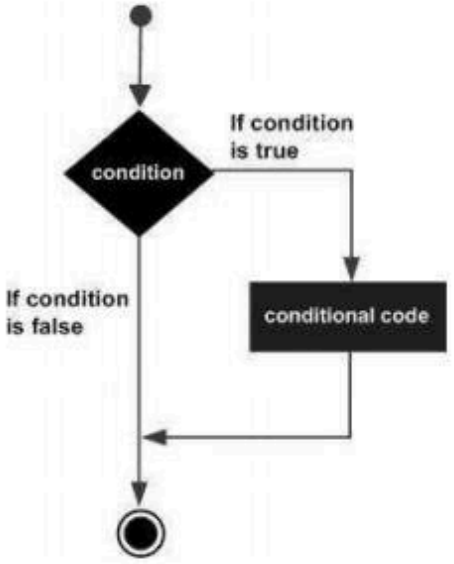
The IF statement is similar to that of other languages. The **if** statement contains a logical expression using which the data is compared and a decision is made based on the result of the comparison.

Syntax

```
if expression:
   statement(s)
```

If the boolean expression evaluates to TRUE, then the block of statement(s) inside the if statement is executed. In Python, statements in a block are uniformly indented after the : symbol. If boolean expression evaluates to FALSE, then the first set of code after the end of block is executed.

Flow Diagram



Example

```
#!/usr/bin/python3
var1 = 100
if var1:
   print ("1 - Got a true expression value")
   print (var1)
```

```
var2 = 0
if var2:
   print ("2 - Got a true expression value")
   print (var2)
print ("Good bye!")
```

When the above code is executed, it produces the following result –

```
1 - Got a true expression value
100
Good bye!
```

IF...ELIF...ELSE Statements

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

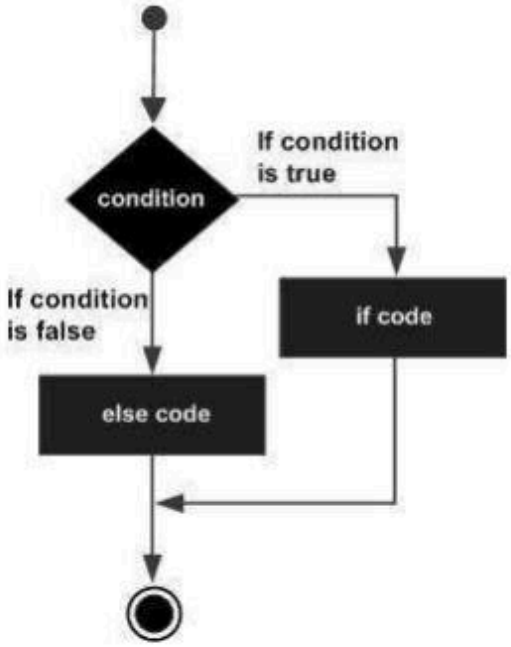
The else statement is an optional statement and there could be at the most only one **else** statement following **if**.

Syntax

The syntax of the **if...else** statement is-

```
if expression:
   statement(s)
else:
   statement(s)
```

Flow Diagram



Example

```
#!/usr/bin/python3
amount=int(input("Enter amount: "))
if amount<1000:
    discount=amount*0.05
    print ("Discount",discount)
else:
    discount=amount*0.10
    print ("Discount",discount)

print ("Net payable:",amount-discount)
```

In the above example, discount is calculated on the input amount. Rate of discount is 5%, if the amount is less than 1000, and 10% if it is above 10000. When the above code is executed, it produces the following result-

```
Enter amount: 600
Discount 30.0
Net payable: 570.0
Enter amount: 1200
Discount 120.0
```

```
Net payable: 1080.0
```

The elif Statement

The **elif** statement allows you to check multiple expressions for TRUE and execute a block of code as soon as one of the conditions evaluates to TRUE.

Similar to the **else**, the **elif** statement is optional. However, unlike **else**, for which there can be at the most one statement, there can be an arbitrary number of **elif** statements following an **if**.

Syntax

```
if expression1:
   statement(s)
elif expression2:
   statement(s)
elif expression3:
   statement(s)
else:
   statement(s)
```

Core Python does not provide switch or case statements as in other languages, but we can use if..elif...statements to simulate switch case as follows-

Example

```
#!/usr/bin/python3
amount=int(input("Enter amount: "))

if amount<1000:
    discount=amount*0.05
    print ("Discount",discount)
elif amount<5000:
    discount=amount*0.10
    print ("Discount",discount)
else:
    discount=amount*0.15
    print ("Discount",discount)
print ("Net payable:",amount-discount)
```

When the above code is executed, it produces the following result-

```
Enter amount: 600
Discount 30.0
Net payable: 570.0

Enter amount: 3000
Discount 300.0
Net payable: 2700.0

Enter amount: 6000
Discount 900.0
Net payable: 5100.0
```

Nested IF Statements

There may be a situation when you want to check for another condition after a condition resolves to true. In such a situation, you can use the nested **if** construct.

In a nested **if** construct, you can have an **if...elif...else** construct inside another **if...elif...else** construct.

Syntax

The syntax of the nested if...elif...else construct may be-

```
if expression1:
   statement(s)
   if expression2:
      statement(s)
   elif expression3:
      statement(s)
   else
      statement(s)
elif expression4:
   statement(s)
else:
   statement(s)
```

Example

```
# !/usr/bin/python3
num=int(input("enter number"))
```

```
if num%2==0:
    if num%3==0:
        print ("Divisible by 3 and 2")
    else:
        print ("divisible by 2 not divisible by 3")
else:
    if num%3==0:
        print ("divisible by 3 not divisible by 2")
    else:
        print  ("not Divisible by 2 not divisible by 3")
```

When the above code is executed, it produces the following result-

```
enter number8
divisible by 2 not divisible by 3

enter number15
divisible by 3 not divisible by 2

enter number12
Divisible by 3 and 2

enter number5
not Divisible by 2 not divisible by 3
```

Single Statement Suites

If the suite of an **if** clause consists only of a single line, it may go on the same line as the header statement.

Here is an example of a **one-line if** clause-

```
#!/usr/bin/python3
var = 100
if ( var  == 100 ) : print ("Value of expression is 100")
print ("Good bye!")
```

When the above code is executed, it produces the following result-

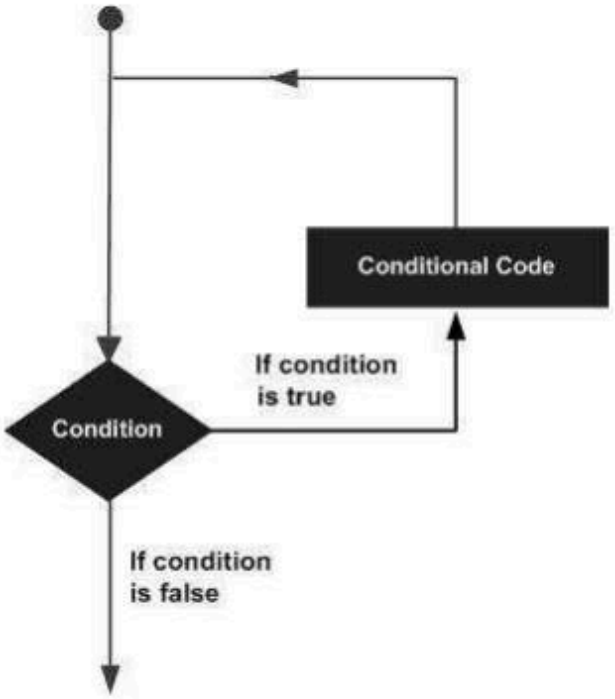
```
Value of expression is 100
Good bye!
```

8. Python 3 – Loops

In general, statements are executed sequentially- The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times.

Programming languages provide various control structures that allow more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times. The following diagram illustrates a loop statement.



Python programming language provides the following types of loops to handle looping requirements.

Loop Type	Description
while loop	Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.
for loop	Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

nested loops	You can use one or more loop inside any another while, or for loop.

while Loop Statements

A **while** loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.

Syntax

The syntax of a **while** loop in Python programming language is-

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

Here, **statement(s)** may be a single statement or a block of statements with uniform indent. The **condition** may be any expression, and true is any non-zero value. The loop iterates while the condition is true.

When the condition becomes false, program control passes to the line immediately following the loop.

In Python, all the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.

Flow Diagram

- for loop
- while loop

for loop – A for loop is used to iterate over a sequence that is either a list, tuple, dictionary, or a set. We can execute a set of statements once for each item in a list, tuple, or dictionary.

```
print("1st example")

lst = [1, 2, 3]
for i in range(len(lst)):
    print(lst[i], end = " \n")

print("2nd example")

for j in range(0,5):
    print(j, end = " \n")
```

Output
1.85s
1st example
1 2 3

2nd example
0 1 2 3

while loop:

In Python, while loops are used to execute a block of statements repeatedly until a given condition is satisfied. Then, the expression is checked again and, if it is still true, the body is executed again. This continues until the expression becomes false.

```
m = 5
i = 0
while i < m:
    print(i, end = " ")
    i = i + 1
print("End")
```

Output
0 1 2 3 4 end

Indentation

Python requires indentation to define statement blocks. A block of code must have a consistent number of spaces.

To indent in Python, whitespaces are preferred over tabs. Also, use either whitespace or tabs to indent; mixing tabs and whitespaces in indentation can result in incorrect indentation errors.

Rules of Indentation in Python

Here are the rules of indentation in python:

- Python's default indentation spaces are four spaces. The number of spaces, however, is entirely up to the user. However, a minimum of one space is required to indent a statement.
- Indentation is not permitted on the first line of Python code.
- Python requires indentation to define statement blocks.
- A block of code must have a consistent number of spaces.
- To indent in Python, whitespaces are preferred over tabs. Also, use either whitespace or tabs to indent; mixing tabs and whitespaces in indentation can result in incorrect indentation errors.
    - List
    - Lists are used to store multiple items in a single variable.
    - Lists are one of 4 built-in data types in Python used to store collections of data, the other 3 are Tuple, Set, and Dictionary, all with different qualities and usage.
    - Lists are created using square brackets:
    - Example
    - Create a List:
    - thislist = ["apple", "banana", "cherry"]
      print(thislist)

Creating a List in Python

Lists in Python can be created by just placing the sequence inside the square brackets[]. Unlike Sets, a list doesn't need a built-in function for its creation of a list.

```
Python program to demonstrate

# Creation of List

 # Creating a List

List = []

print("Blank List: ")

print(List)

 # Creating a List of numbers

List = [10, 20, 14]

print("\nList of numbers: ")

print(List)

 # Creating a List of strings and accessing

# using index

List = ["Geeks", "For", "Geeks"]

print("\nList Items: ")

print(List[0])

print(List[2])
```

Output

Blank List:

[]

List of numbers:

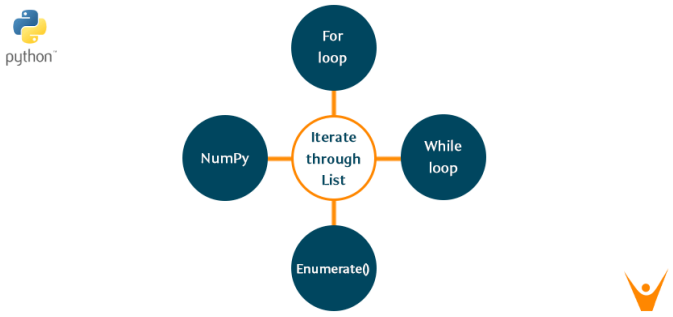
[10, 20, 14]

List Items:

Geeks

Geeks

Iterate a List in Python



Using for loop

The easiest method to iterate the list in python programming is by using them for a loop. The method of the iterating list using for loop is as given below

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

# Iterating list using for loop
for i in list:
    print(i)
```

output

1

2

3

4

5

Using loop and range() function

Another method to iterate the list while programming in python is by using the loop and range() function together. Iterating the list using this method is not recommended if iteration using for loop(method 1) is possible. The method to do so is as given below:

```
list = [1, 2, 3, 4, 5]
# getting length of list using len() function
length = len(list)

# using for loop along with range() function
for i in range(length):
    print(list[i])
```

output

1

2

3

4

5

Using While loop

We can also iterate the list in python language using a while loop. The method to use while loop for the iterating list is as given below:

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

# Getting length of list using len() function
length = len(list)
i = 0

while i < length:
   print(list[i])
   i += 1
```

output:

1

2

3

4

5

Using list comprehension

This is the most concrete way to iterate the list while programming in the python language. The method to iterate list using list comprehension is as given below:

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

# Iterating list using list comprehension
[print(i) for i in list]
```

Output:

1

2

3

4

5

Using enumerate() function

There are few times when you may need the display the index of the element along with the element in the list itself. In such cases, we use the enumerate() function for the iteration of the list. The method to use the enumerate() function to iterate the list is as given below:

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

# Using enumerate() function to iterate the list
for i, val in enumerate(list):
   print (i, ",",val)
```

output:

0,1

1,3

2,5

3,7

Using Numpy function

All the methods that we discussed till now are generally preferable for small or say single-dimensional lists. But when it comes to large n-dimensional lists it is advisable to use an external library such as NumPy for iterating lists. The method to use the Numpy function for iterating lists is as given below:

```
# Importing external library
import numpy as np

a = np.arange(5)

for x in np.nditer(a):
    print(x)
```

output:

0

1

2

3

More on Lists
List / Array Methods in Python
Let's look at some different methods for lists in Python:

S.no	Method	Description
1	append()	Used for adding elements to the end of the List.
2	copy()	It returns a shallow copy of a list
3	clear()	This method is used for removing all items from the list.
4	count()	These methods count the elements.
5	extend()	Adds each element of an

S.no	Method	Description
		iterable to the end of the List
6	index()	Returns the lowest index where the element appears.
7	insert()	Inserts a given element at a given index in a list.
8	pop()	Removes and returns the last value from the List or the given index value.
9	remove()	Removes a given object from the List.
10	reverse()	Reverses objects of the List in place.
11	sort()	Sort a List in ascending, descending, or user-defined order
12	min()	Calculates the minimum of all the elements of the List
13	max()	Calculates the maximum of all the elements of the List

Adding Element in List
1. Python append() method
Adds element to the end of a list.

Syntax: *list.append (element)*
Example:
Python3

```
# Adds List Element as value of List.

List = ['Mathematics', 'chemistry', 1997, 2000]

List.append(20544)

print(List)
```

Output
['Mathematics', 'chemistry', 1997, 2000, 20544]

2. Python insert() method
Inserts an element at the specified position.

Syntax:
list.insert(<position, element)

Example:

```
List = ['Mathematics', 'chemistry', 1997, 2000]

# Insert at index 2 value 10087

List.insert(2, 10087)

print(List)
```

Output
['Mathematics', 'chemistry', 10087, 1997, 2000]

3. Python extend() method
Adds items of an iterable(list, array, string , etc.) to the end of a list

Syntax: *List1.extend(List2)*
Example:

```
List1 = [1, 2, 3]

List2 = [2, 3, 4, 5]

 # Add List2 to List1

List1.extend(List2)

print(List1)

 # Add List1 to List2 now

List2.extend(List1)

print(List2)
```

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

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

Important functions of the Python List
We have mentioned some essential Python list functions along with their syntax and example:

1. Python sum() method
Calculates the sum of all the elements of the List.

Syntax: *sum(List)*
Example:

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

print(sum(List))
```

Output
15

See example:

```
List = ['gfg', 'abc', 3]

print(sum(List))
```

Output:
Traceback (most recent call last):
  File "", line 1, in
    sum(List)
TypeError: unsupported operand type(s) for +: 'int' and 'str'

`2. Python count() method`

Calculates the total occurrence of a given element of the List.

Syntax: *List.count(element)*
Example:

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

print(List.count(1))
```

Output
4

3. Python len() method
Calculates the total length of the List.

Syntax: *len(list_name)*

Example:

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

print(len(List))
```

Output
10

4. Python index() method
Returns the index of the first occurrence. The start and end indexes are not necessary parameters.

Syntax: *List.index(element[,start[,end]])*
Example:

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

print(List.index(2))
```

Output
1

Another example:

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

print(List.index(2, 2))
```

Output
4

5. Python min() method
Calculates minimum of all the elements of List.

Syntax: *min(iterable, *iterables[, key])*
Example:

```
numbers = [5, 2, 8, 1, 9]

print(min(numbers))
```

Output
1

6. Python max() method
Calculates the maximum of all the elements of the List.

Syntax: *max(iterable, *iterables[, key])*

```
numbers = [5, 2, 8, 1, 9]

print(max(numbers))
```

Output
9

7. Python sort() method
Sort the given data structure (both tuple and list) in ascending order.

Key and **reverse_flag** are not necessary parameter and reverse_flag is set to False if nothing is passed through sorted().
Syntax: *list.sort([key,[Reverse_flag]])*
Example:

```
List = [2.3, 4.445, 3, 5.33, 1.054, 2.5]

 #Reverse flag is set True

List.sort(reverse=True)

  #List.sort().reverse(), reverses the sorted list

print(List)
```

Output
[5.33, 4.445, 3, 2.5, 2.3, 1.054]

8. Python reverse() method
reverse() function reverses the order of list.

Syntax: list. reverse()
Example:

```
# creating a list

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

```
#reversing the list

list.reverse()

#printing the list

print(list)
```

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

Deletion of List Elements
To Delete one or more elements, i.e. remove an element, many built-in Python functions can be used, such as **pop()** & **remove()** and keywords such as **del**.
1. Python pop() method
Removes an item from a specific index in a list.

Syntax: *list.pop([index])*
The index is not a necessary parameter, if not mentioned takes the last index.

Note: The index must be in the range of the List, elsewise IndexErrors occur.
Example 1:

```
List = [2.3, 4.445, 3, 5.33, 1.054, 2.5]

print(List.pop())
```

Output
2.5

Example 2:

```
List = [2.3, 4.445, 3, 5.33, 1.054, 2.5]

print(List.pop(0))
```

Output
2.3

2. Python del() method
Deletes an element from the list using it's index.

Syntax: *del list.[index]*
Example:

```
List = [2.3, 4.445, 3, 5.33, 1.054, 2.5]
```

```
del List[0]

print(List)
```

Output
[4.445, 3, 5.33, 1.054, 2.5]

3. Python remove() method
Removes a specific element using it's value/name.

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

```
List = [2.3, 4.445, 3, 5.33, 1.054, 2.5]

List.remove(3)

print(List)
```

Output
[2.3, 4.445, 5.33, 1.054, 2.5]

Python Tuples

A comma-separated group of items is called a Python triple.

The ordering, settled items, and reiterations of a tuple are to some degree like those of a rundown, but in contrast to a rundown, a tuple is unchanging.

Example

("Suzuki", "Audi", "BMW"," Skoda ") is a tuple.

Creating a Tuple

A tuple is created by placing all the items (elements) inside parentheses (), separated by commas. The parentheses are optional, however, it is a good practice to use them.
A tuple can have any number of items and they may be of different types (integer, float, list, string, etc.).

```
# Different types of tuples

# Empty tuple
my_tuple = ()
print(my_tuple)

# Tuple having integers
my_tuple = (1, 2, 3)
print(my_tuple)

# tuple with mixed datatypes
my_tuple = (1, "Hello", 3.4)
print(my_tuple)

# nested tuple
```

```
my_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
print(my_tuple)
```
Run Code

Output

```
()
(1, 2, 3)
(1, 'Hello', 3.4)
('mouse', [8, 4, 6], (1, 2, 3))
```

Access Python Tuple Elements

Like a list, each element of a tuple is represented by index numbers **(0, 1, ...)** where the first element is at index **0**.
We use the index number to access tuple elements. For example,

1. Indexing

We can use the index operator [] to access an item in a tuple, where the index starts from 0.

```
# accessing tuple elements using indexing
letters = ("p", "r", "o", "g", "r", "a", "m", "i", "z")

print(letters[0])   # prints "p"
print(letters[5])   # prints "a"
```

Negative Indexing

Python allows negative indexing for its sequences.

The index of **-1** refers to the last item, **-2** to the second last item and so on. For example,

```
# accessing tuple elements using negative indexing
letters = ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')

print(letters[-1])   # prints 'z'
print(letters[-3])   # prints 'm'
```

3. Slicing

We can access a range of items in a tuple by using the slicing operator colon :.

```
# accessing tuple elements using slicing
my_tuple = ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')

# elements 2nd to 4th index
print(my_tuple[1:4])  #  prints ('r', 'o', 'g')

# elements beginning to 2nd
print(my_tuple[:-7]) # prints ('p', 'r')

# elements 8th to end
print(my_tuple[7:]) # prints ('i', 'z')

# elements beginning to end
print(my_tuple[:]) # Prints ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
```

Output

```
('r', 'o', 'g')
('p', 'r')
('i', 'z')
('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
```

Python Tuple Methods
Count() Method
The count() method of Tuple returns the number of times the given element appears in the tuple.

Syntax:
tuple.count(element)

```
# Creating tuples

Tuple1 = (0, 1, 2, 3, 2, 3, 1, 3, 2)

Tuple2 = ('python', 'geek', 'python',

        'for', 'java', 'python')

 # count the appearance of 3

res = Tuple1.count(3)

print('Count of 3 in Tuple1 is:', res)

  # count the appearance of python

res = Tuple2.count('python')

print('Count of Python in Tuple2 is:', res)
```

Output:
Count of 3 in Tuple1 is: 3

Count of Python in Tuple2 is: 3

Index() Method
The Index() method returns the first occurrence of the given element from the tuple.

Syntax:
tuple.index(element, start, end)

Parameters:
- **element:** The element to be searched.
- **start (Optional):** The starting index from where the searching is started
- **end (Optional):** The ending index till where the searching is done

```
Creating tuples
```

```
Tuple = (0, 1, 2, 3, 2, 3, 1, 3, 2)

  # getting the index of 3

res = Tuple.index(3)

print('First occurrence of 3 is', res)

 # getting the index of 3 after 4th

# index

res = Tuple.index(3, 4)

print('First occurrence of 3 after 4th index is:', res)
```

Output:

First occurrence of 3 is 3

First occurrence of 3 after 4th index is: 5