Chapater 1: Introduction to Python

Write down features of the Python

Python is a very popular general-purpose interpreted, interactive, object-oriented, and high-level programming language. Python is dynamically-typed (i.e. No need to specify data types of variables etc. like PHP) and garbage-collected programming language. It was created by Guido van Rossum during 1985- 1990. Like Perl, Python source code is also available under the GNU General Public License (GPL).

Python is Interpreted − Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.

Python is Interactive − You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.

Python is Object-Oriented − Python supports Object-Oriented style or technique of programming that encapsulates code within objects.

```
# This is my first Python program.

# This will print 'Hello, World!' as the output

print ("Hello, World!");
```

It supports functional and structured programming methods as well as OOP.

It can be used as a scripting language or can be compiled to byte-code for building large applications.

It provides very high-level dynamic data types and supports dynamic type checking.

It supports automatic garbage collection.

- **A broad standard library** − Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
- **Interactive Mode** − Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.
- **Portable** − Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- **Databases** − Python provides interfaces to all major commercial databases.
- **GUI Programming** − Python supports GUI applications that can be created and ported to many systems.
- **Scalable** − Python provides a better structure and support for large programs than shell scripting.

Latest version of Python: 3.12.0

What do we mean by “Interpreter Based” (slower) and “compiler based” (faster)?

Instructions in any programming languages must be translated into machine code for the processor to execute them. Programming languages are either compiler based or interpreter based. In case of a compiler, a machine language version of the entire source program is generated. The conversion fails even if there is a single erroneous statement. The C family languages (including C, C++, Java, C Sharp etc) are compiler based.

Python is an interpreter based language. The interpreter takes one instruction from the source code at a time, translates it into machine code and executes it. Instructions before the first occurrence of error are executed. With this feature, it is easier to debug the program.

Python's Standard Library

Even though it has a very few keywords (only Thirty Five), Python software is distributed with a standard library made of large number of modules and packages.

Python is Open Source and Cross Platform

Python's standard distribution can be downloaded from https://www.python.org/downloads/ without any restrictions.

Python is portabel

A Python program is first compiled to an intermediate platform independent byte code. The virtual machine inside the interpreter then executes the byte code. This behaviour makes Python a cross-platform language, and thus a Python program can be easily ported from one OS platform to other.

Python for GUI Applications

Python's standard distribution has an excellent graphics library called TKinter.

Python's Database Connectivity

Almost any type of database can be used as a backend with the Python application. DB-API is a set of specifications for database driver software to let Python communicate with a relational database. With many third party libraries, Python can also work with NoSQL databases such as MongoDB.

Python's Active Developer Community

As a result of Python's popularity and open-source nature, a large number of Python developers often interact with online forums and conferences.

Module Vs. Package Vs. Library in Python:

A module is a single file containing python code, whereas a package is a collection of modules that are organized in a directory hierarchy. A module is a set of code or functions with the.py extension. A library is a collection of related modules or packages.

Difference between Python and C++

Both Python and C++ are among the most popular programming languages. Both of them have their advantages and disadvantages. In this tutorial, we shall take a closure look at their characteristic features which differentiate one from another.

Compiled vs Interpreted

Like C, C++ is also a compiler-based language. A compiler translates the entire code in a machine language code specific to the operating system in use and processor architecture.

Python is interpreter-based language. The interpreter executes the source code line by line.

Cross platform

When a C++ source code such as hello.cpp is compiled on Linux, it can be only run on any other computer with Linux operating system. If required to run on other OS, it needs to be compiled.

Python interpreter doesn't produce compiled code. Source code is converted to byte code every time it is run on any operating system without any changes or additional steps.

Portability

Python code is easily portable from one OS to other. C++ code is not portable as it must be recompiled if the OS changes.

Speed of Development

C++ program is compiled to the machine code. Hence, its execution is faster than interpreter based language.

Python interpreter doesn't generate the machine code. Conversion of intermediate byte code to machine language is done on each execution of program.

If a program is to be used frequently, C++ is more efficient than Python.

Easy to Learn

Compared to C++, Python has a simpler syntax. Its code is more readable. Writing C++ code seems daunting in the beginning because of complicated syntax rule such as use of curly braces and semicolon for sentence termination.

Python doesn't use curly brackets for marking a block of statements. Instead, it uses indents. Statements of similar indent level mark a block. This makes a Python program more readable.

Static vs Dynamic Typing

C++ is a statically typed language. The type of variables for storing data need to be declared in the beginning. Undeclared variables can't be used. Once a variable is declared to be of a certain type, value of only that type can be stored in it.

Python is a dynamically typed language. It doesn't require a variable to be declared before assigning it a value. Since, a variable may store any type of data, it is called dynamically typed.

OOP Concepts

Both C++ and Python implement object oriented programming concepts. C++ is closer to the theory of OOP than Python. C++ supports the concept of data encapsulation as the visibility of the variables can be defined as public, private and protected.

Python doesn't have the provision of defining the visibility. Unlike C++, Python doesn't support method overloading. Because it is dynamically typed, all the methods are polymorphic in nature by default.

C++ is in fact an extension of C. One can say that additional keywords are added in C so that it supports OOP. Hence, we can write a C type procedure oriented program in C++.

Python is completely object oriented language. Python's data model is such that, even if you can adapt a procedure oriented approach, Python internally uses object-oriented methodology.

Garbage Collection

C++ uses the concept of pointers. Unused memory in a C++ program is not cleared automatically. In C++, the process of garbage collection is manual. Hence, a C++ program is likely to face memory related exceptional behavior.

Python has a mechanism of automatic garbage collection. Hence, Python program is more robust and less prone to memory related issues.

Application Areas

Because C++ program compiles directly to machine code, it is more suitable for systems programming, writing device drivers, embedded systems and operating system utilities.

Python program is suitable for application programming. Its main area of application today is data science, machine learning, API development etc.

Difference table Python vs C++

The following table summarizes the comparison between C++ and Python −

Criteria	**C++**	**Python**
Execution	Compiler based	Interpreter based
Typing	Static typing	Dynamic typing
Portability	Not portable	Highly portable
Garbage collection	Manual	Automatic
Syntax	Tedious	Simple
Performance	Faster execution	Slower execution
Application areas	Embedded systems, device drivers	Machine learning, web applications

LIST (like Array)

List is one of the built-in data types in Python. A Python list is a sequence of comma separated items, enclosed in square brackets [ ]. The items in a Python list need not be of the same data type.

Following are some examples of Python lists –

list1 = ["Rohan", "Physics", 21, 69.75]

list2 = [1, 2, 3, 4, 5]

list3 = ["a", "b", "c", "d"]

list4 = [25.50, True, -55, 1+2j]

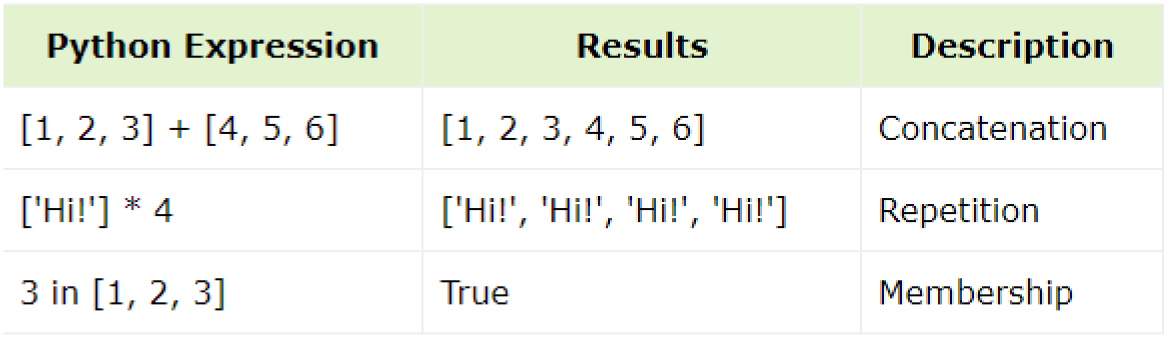
In Python, a list is a sequence data type. It is an ordered collection of items. Each item in a list has a unique position index, starting from 0.

A list in Python is similar to an array in C, C++ or Java. However, the major difference is that in C/C++/Java, the array elements must be of same type. On the other hand, Python lists may have objects of different data types.

A Python list is mutable. Any item from the list can be accessed using its index, and can be modified. One or more objects from the list can be removed or added. A list may have same item at more than one index positions.

Python List Operations

In Python, List is a sequence. Hence, we can concatenate two lists with "+" operator and concatenate multiple copies of a list with "*" operator. The membership operators "in" and "not in" work with list object.



Python Expression	Results	Description
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	Repetition
3 in [1, 2, 3]	True	Membership

How to access List Item?

In Python, a list is a sequence. Each object in the list is accessible with its index. The index starts from 0. Index or the last item in the list is "length-1". To access the values in a list, use the square brackets for slicing along with the index or indices to obtain value available at that index.

```
obj = list1[i]
```

Example 1

Take a look at the following example −

```
list1 = ["Rohan", "Physics", 21, 69.75]

list2 = [1, 2, 3, 4, 5]

print ("Item at 0th index in list1: ", list1[0])

print ("Item at index 2 in list2: ", list2[2])
```

It will produce the following output −

```
Item at 0th index in list1: Rohan

Item at index 2 in list2: 3
```

Python allows negative index to be used with any sequence type. The "-1" index refers to the last item in the list.

Example 2

Let's take another example −

```
list1 = ["a", "b", "c", "d"]

list2 = [25.50, True, -55, 1+2j]

print ("Item at 0th index in list1: ", list1[-1])

print ("Item at index 2 in list2: ", list2[-3])
```

It will produce the following output −

```
Item at 0th index in list1: d

Item at index 2 in list2: True
```

The slice operator extracts a sublist from the original list.

```
Sublist = list1[i:j]
```

Parameters

i − index of the first item in the sublist

j − index of the item next to the last in the sublist

This will return a slice from ith to (j-1)th items from the list1.

Example 3

While slicing, both operands "i" and "j" are optional. If not used, "i" is 0 and "j" is the last item in the list. Negative index can be used in slicing. Take a look at the following example −

```
list1 = ["a", "b", "c", "d"]

list2 = [25.50, True, -55, 1+2j]

print ("Items from index 1 to 2 in list1: ", list1[1:3])

print ("Items from index 0 to 1 in list2: ", list2[0:2])
```

It will produce the following output −

```
Items from index 1 to 2 in list1: ['b', 'c']

Items from index 0 to 1 in list2: [25.5, True]
```

Example 4

```
list1 = ["a", "b", "c", "d"]

list2 = [25.50, True, -55, 1+2j]

list4 = ["Rohan", "Physics", 21, 69.75]

list3 = [1, 2, 3, 4, 5]


print ("Items from index 1 to last in list1: ", list1[1:])

print ("Items from index 0 to 1 in list2: ", list2[:2])

print ("Items from index 2 to last in list3", list3[2:-1])

print ("Items from index 0 to index last in list4", list4[:])
```

It will produce the following output −

```
Items from index 1 to last in list1: ['b', 'c', 'd']

Items from index 0 to 1 in list2: [25.5, True]

Items from index 2 to last in list3 [3, 4]

Items from index 0 to index last in list4 ['Rohan', 'Physics', 21, 69.75]
```

How to change elements?

List is a mutable data type in Python. It means, the contents of list can be modified in place, after the object is stored in the memory. You can assign a new value at a given index position in the list

Syntax

```
list1[i] = newvalue
```

Example 1

In the following code, we change the value at index 2 of the given list.

```
list3 = [1, 2, 3, 4, 5]

print ("Original list ", list3)

list3[2] = 10

print ("List after changing value at index 2: ", list3)
```

It will produce the following output −

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

List after changing value at index 2: [1, 2, 10, 4, 5]
```

You can replace more consecutive items in a list with another sublist.

Example 2

In the following code, items at index 1 and 2 are replaced by items in another sublist.

```
list1 = ["a", "b", "c", "d"]

print ("Original list: ", list1)

list2 = ['Y', 'Z']

list1[1:3] = list2


print ("List after changing with sublist: ", list1)
```

It will produce the following output −

```
Original list: ['a', 'b', 'c', 'd']

List after changing with sublist: ['a', 'Y', 'Z', 'd']
```

Example 3

If the source sublist has more items than the slice to be replaced, the extra items in the source will be inserted. Take a look at the following code −

```
list1 = ["a", "b", "c", "d"]

print ("Original list: ", list1)

list2 = ['X','Y', 'Z']

list1[1:3] = list2

print ("List after changing with sublist: ", list1)
```

It will produce the following output −

```
Original list: ['a', 'b', 'c', 'd']

List after changing with sublist: ['a', 'X', 'Y', 'Z', 'd']
```

Example 4

If the sublist with which a slice of original list is to be replaced, has lesser items, the items with match will be replaced and rest of the items in original list will be removed.

In the following code, we try to replace "b" and "c" with "Z" (one less item than items to be replaced). It results in Z replacing b and c removed.

```
list1 = ["a", "b", "c", "d"]

print ("Original list: ", list1)

list2 = ['Z']

list1[1:3] = list2

print ("List after changing with sublist: ", list1)
```

It will produce the following output −

```
Original list: ['a', 'b', 'c', 'd']

List after changing with sublist: ['a', 'Z', 'd']
```

Python List

In Python, the sequence of various data types is stored in a list. A list is a collection of different kinds of values or items. Since Python lists are mutable, we can change their elements after forming. The comma (,) and the square brackets [enclose the List's items] serve as separators.

A list, a type of sequence data, is used to store the collection of data. Tuples and Strings are two similar data formats for sequences.

Lists written in Python are identical to dynamically scaled arrays defined in other languages, such as Array List in Java and Vector in C++. A list is a collection of items separated by commas and denoted by the symbol [].

List Declaration

Code

```python
# a simple list
list1 = [1, 2, "Python", "Program", 15.9]
list2 = ["Amy", "Ryan", "Henry", "Emma"]

# printing the list
print(list1)
print(list2)

# printing the type of list
print(type(list1))
print(type(list2))
```

Output:

```
[1, 2, 'Python', 'Program', 15.9]
['Amy', 'Ryan', 'Henry', 'Emma']
< class ' list ' >
< class ' list ' >
```

Characteristics of Lists

The characteristics of the List are as follows:

- The lists are in order.
- The list element can be accessed via the index.
- The mutable type
- The number of various elements can be stored in a list.

Ordered List Checking

Code

```
# example
 a = [ 1, 2, "Ram", 3.50, "Rahul", 5, 6 ]
b = [ 1, 2, 5, "Ram", 3.50, "Rahul", 6 ]
if(a == b):
    print("a==b")
else:
    print("a!=b")
```

Code

```
# example
a = [ 1, 2, "Ram", 3.50, "Rahul", 5, 6 ]
b = [ 1, 2, "Ram", 3.50, "Rahul", 5, 6 ]

if(a == b):
    print("a==b")
else:
    print("a!=b")
```

Let's take a closer look at the list example.

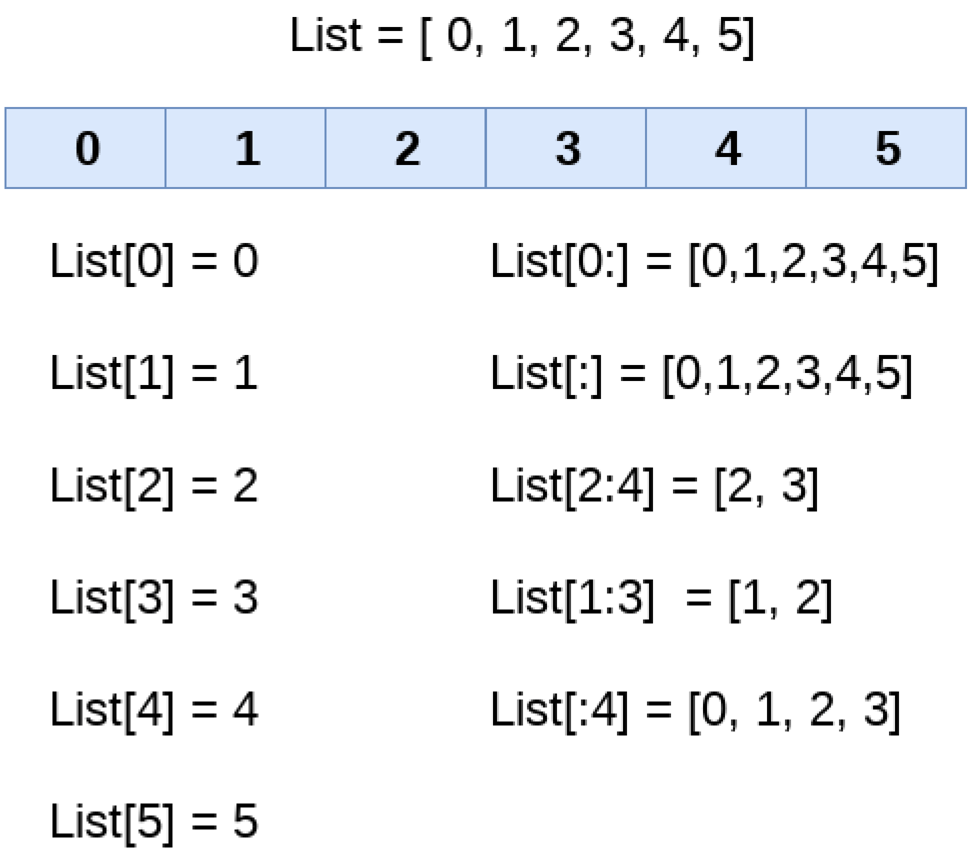
Code

```
# list example in detail
emp = [ "John", 102, "USA"]
print("printing employee data ...")
print(" Name : %s, ID: %d, Country: %s" %(emp[0], emp[1], emp[2]))
```

List Indexing and Splitting

The indexing procedure is carried out similarly to string processing. The slice operator can be used to get to the List's components.

The index ranges from 0 to length -1. The 0th index is where the List's first element is stored; the 1st index is where the second element is stored, and so on.



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

```
list_varible(start:stop:step)
```

Consider the following example:

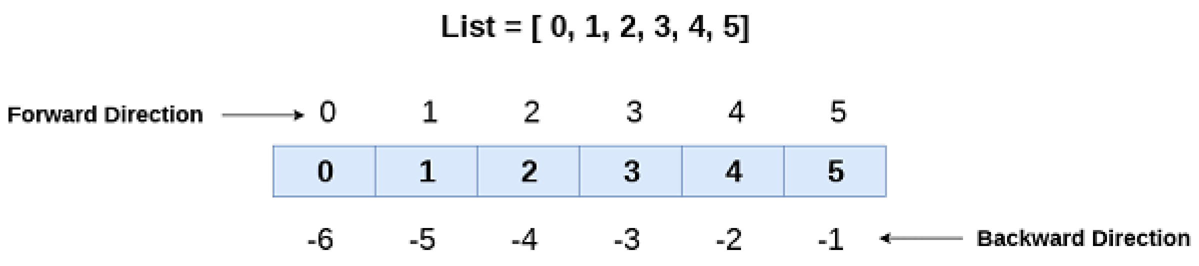
Code

```
list = [1,2,3,4,5,6,7]
print(list[0])
print(list[1])
print(list[2])
print(list[3])
# Slicing the elements
print(list[0:6])
print(list[:])
print(list[2:5])
print(list[1:6:2])
```

Output:

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

In contrast to other programming languages, Python lets you use negative indexing as well. The negative indices are counted from the right. The index -1 represents the final element on the List's right side, followed by the index -2 for the next member on the left, and so on, until the last element on the left is reached.



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

Code

```
# negative indexing example
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]
```

Updating List Values

Due to their mutability and the slice and assignment operator's ability to update their values, lists are Python's most adaptable data structure. Python's append() and insert() methods can also add values to a list.

Consider the following example to update the values inside the List.

Code

```python
# updating list values
list = [1, 2, 3, 4, 5, 6]
print(list)
# It will assign value to the value to the second index
list[2] = 10
print(list)
# Adding multiple-element
list[1:3] = [89, 78]
print(list)
# It will add value at the end of the list
list[-1] = 25
print(list)
```

Output:

```
[1, 2, 3, 4, 5, 6]
[1, 2, 10, 4, 5, 6]
[1, 89, 78, 4, 5, 6]
[1, 89, 78, 4, 5, 25]
```

The list elements can also be deleted by using the **del** keyword. Python also provides us the **remove()** method if we do not know which element is to be deleted from the list.

Let us understand what is Del and Remove() in a Python List before moving towards the difference.

Del Keyword in Python List

The del keyword in Python is used to remove an element or multiple elements from a List. We can also remove all the elements i.e. delete the entire list.

Example

Delete an element from a Python List using the del keyword

#Create a List

myList = ["Toyota", "Benz", "Audi", "Bentley"]

print("List = ",myList)

```
# Delete a single element
del myList[2]


print("Updated List = \n",myList)
```

Output

```
List =  ['Toyota', 'Benz', 'Audi', 'Bentley']
Updated List =
 ['Toyota', 'Benz', 'Bentley']
```

Example

Delete multiple elements from a Python List using the del keyword

```
# Create a List
myList = ["Toyota", "Benz", "Audi", "Bentley", "Hyundai", "Honda", "Tata"]


print("List = ",myList)


# Delete multiple element
del myList[2:5]
print("Updated List = \n",myList)
```

Output

```
List =  ['Toyota', 'Benz', 'Audi', 'Bentley', 'Hyundai', 'Honda', 'Tata']
Updated List = ['Toyota', 'Benz', 'Honda', 'Tata']
```

Example

Delete all the elements from a Python List using the del keyword

```
# Create a List

myList = ["Toyota", "Benz", "Audi", "Bentley"]

print("List = ",myList)


# Deletes the entire List

del myList


# The above deleted the List completely and all its elements
```

Output

```
List =  ['Toyota', 'Benz', 'Audi', 'Bentley']
```

Remove() method in Python List

The remove() built-in method in Python is used to remove elements from a List.

Example

Remove an element from a Python using the remove() method

```
# Create a List

myList = ["Toyota", "Benz", "Audi", "Bentley"]

print("List = ",myList)


# Remove a single element

myList.remove("Benz")
```

Display the updated List

print("Updated List = \n",myList)

Output

List = ['Toyota', 'Benz', 'Audi', 'Bentley']

Updated List =

 ['Toyota', 'Audi', 'Bentley']

Del vs Remove()

Let us now see the difference between del and remove() in Python –

del in Python	**remove()**
The del is a keyword in Python.	The remove(0) is a built-in method in Python.
An indexError is thrown if the index doesn’t exist in the Python list.	The valueError is thrown if the value doesn’t exist in the Python list.
The del works on index.	The remove() works on value.
The del deletes an element at a specified index number.	It removes the first matching value from the Python List.
The del is simple deletion.	The remove() searches the list to find the item.

Python List Operations

The concatenation (+) and repetition (*) operators work fine. The different operations of list are

1. Repetition
2. Concatenation
3. Length
4. Iteration

5. Membership

Let's see how the list responds to various operators.

1. Repetition

Code

```
# repetition of list
# declaring the list
list1 = [12, 14, 16, 18, 20]
# repetition operator *
l = list1 * 2
print(l)
```

Output:

```
[12, 14, 16, 18, 20, 12, 14, 16, 18, 20]
```

2. Concatenation

It concatenates the list mentioned on either side of the operator.

Code

```
# concatenation of two lists
# declaring the lists
list1 = [12, 14, 16, 18, 20]
list2 = [9, 10, 32, 54, 86]
# concatenation operator +
l = list1 + list2
print(l)
```

Output:

```
[12, 14, 16, 18, 20, 9, 10, 32, 54, 86]
```

3. Length

It is used to get the length of the list

Code

list= [12, 14, 16, 18, 20]

print(len(list))

Output:

```
5
```

4. Iteration

The for loop is used to iterate over the list elements.

Code

```
# iteration of the list
# declaring the list
list1 = [12, 14, 16, 39, 40]
# iterating
for i in list1:
    print(i)
```

Output:

```
12
14
16
39
40
```

While loop and a list:

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

i=0

while (i<len(list)):

    print(list[i])

    i=i+1
```

Output:

1

2

3

5. Membership

It returns true if a particular item exists in a particular list otherwise false.

Code

```
# membership of the list
# declaring the list
list1 = [100, 200, 300, 400, 500]
# true will be printed if value exists
# and false if not

print(600 in list1)
print(700 in list1)
print(1040 in list1)

print(300 in list1)
print(100 in list1)
print(500 in list1)
```

Output:

```
False
False
False
True
True
True
```

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.

Code

```python
# iterating a list
list = ["John", "David", "James", "Jonathan"]
for i in list:
    # The i variable will iterate over the elements of the List and contains each element in each iteration.
    print(i)
```

Output:

```
John
David
James
Jonathan
```

Adding Elements to the List

The append() function in Python can add a new item to the List.

Code

```python
#Declaring the empty list
l =[]
#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
    l.append(input("Enter the item:"))
print("printing the list items..")
# traversal loop to print the list items
for i in l:
    print(i)
```

Output:

```
Enter the number of elements in the list:10
Enter the item:32
Enter the item:56
Enter the item:81
Enter the item:2
Enter the item:34
Enter the item:65
Enter the item:09
Enter the item:66
Enter the item:12
Enter the item:18
printing the list items..
32  56  81  2  34  65  09  66  12  18
```

Insert Method and Extend Method

```
myList = ['one', 'two', 'three']

myList.insert(0, 'zero')

print(myList)

# ['zero', 'one', 'two', 'three']
```

Extend Method:

```
myList1 = ['one', 'two', 'three']
myList2 = ['four', 'five', 'six']

myList1.extend(myList2)

print(myList1)
# ['one', 'two', 'three', 'four', 'five', 'six']
```

Removing Elements from the List

The remove() function in Python can remove an element from the List. To comprehend this idea, look at the example that follows.

Example -

Code

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

Output:

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

Python List Built-in Functions

Python provides the following built-in functions, which can be used with the lists.

1. len()
2. max()
3. min()

len( )

It is used to calculate the length of the list.

Code

```
# size of the list
# declaring the list
list1 = [12, 16, 18, 20, 39, 40]
# finding length of the list
Print(len(list1))
```

Output:

```
6
```

Max( )

It returns the maximum element of the list

Code

```
# maximum of the list
list1 = [103, 675, 321, 782, 200]
# large element in the list
print(max(list1))
```

Output:

```
782
```

Min( )

It returns the minimum element of the list

Code

```python
# minimum of the list
list1 = [103, 675, 321, 782, 200]
# smallest element in the list
print(min(list1))
```

Output:

```
103
```

Let's have a look at the few list examples.

Example: 1- Create a program to eliminate the List's duplicate items.

Code

```python
list1 = [1,2,2,3,55,98,65,65,13,29]
# Declare an empty list that will store unique values
list2 = []
for i in list1:
    if i not in list2:
        list2.append(i)
print(list2)
```

Output:

```
[1, 2, 3, 55, 98, 65, 13, 29]
```

Example Code

```python
list1 = [3,4,5,9,10,12,24]
sum = 0
for i in list1:
    sum = sum+i
```

```
print("The sum is:",sum)
```

Output:

```
The sum is: 67
```

Example:

Code

```
list1 = [1,2,3,4,5,6]
list2 = [7,8,9,2,10]
for x in list1:
    for y in list2:
        if x == y:
            print("The common element is:",x)
```

Output:

```
The common element is: 2
```

Example:

```
list=[[1,2,3],[4,5,6]]

print(list)


for i in list:

   print(i)


for i in list:

   for j in i:

      print(j)
```

Output:

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

[1, 2, 3]

[4, 5, 6]
```

1

2

3

4

5

6

Python Set

A Python set is the collection of the unordered items. Each element in the set must be unique, immutable and the sets remove the duplicate elements. Sets are mutable which means we can add/remove items from it.

Set items are unchangeable, meaning that we cannot change the immutable items after the set has been created. Once a set is created, you cannot change its items, but you can remove items and add new items.

Unlike other collections in Python, there is no index attached to the elements of the set, i.e., we cannot directly access any element of the set by the index. However, we can print them all together, or we can get the list of elements by looping through the set.

Python Collections

There are four collection data types in the Python programming language:

- **List** is a collection which is ordered and changeable. Allows duplicate members.
- **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
- **Set** is a collection which is unordered, unchangeable*, and unindexed. No duplicate members.
- **Dictionary** is a collection which is ordered and changeable. No duplicate members.

*Set *items* are unchangeable, but you can remove items and add new items.

Creating a set

The set can be created by enclosing the comma-separated immutable items with the curly braces {}. Python also provides the set() method, which can be used to create the set by the passed sequence.

Example 1: Using curly braces

```
Days = {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"}
print(Days)
print(type(Days))
print("looping through the set elements ... ")
for i in Days:
    print(i)
```

Output:

```
{'Friday',   'Tuesday',   'Monday',   'Saturday',   'Thursday',   'Sunday',
'Wednesday'}
<class 'set'>
looping through the set elements ...
Friday
Tuesday
Monday
Saturday
Thursday
Sunday
Wednesday
```

Example:

```
set1={"sachin","dhoni","mahi"}

set2=set(("srk","salman","ranveer"))

set3=set({"one","two","three"})

print(set1,set2)



for i in set2:

    print(i)
```

Output:

```
{'sachin', 'dhoni', 'mahi'} {'srk', 'salman', 'ranveer'}

srk

salman

ranveer
```

Example:

```
thisset = {"apple", "banana", "cherry", False, True, 0}

print(thisset)

output:
{False, True, 'cherry', 'apple', 'banana'}
```

Unordered means that the items in a set do not have a defined order.

Set items can appear in a different order every time you use them, and cannot be referred to by index or key.

Example:

```
set1={"sachin","dhoni","mahi","dhoni"}

set2=set(("srk","salman","ranveer"))

set3=set({"one","two","three"})

print(set1,set2)

set4={}

set4=set1

print(set4)

thisset = {"apple", "banana", "cherry"}
```

print("banana" in thisset)

print("bana" in thisset)

Output:

{'sachin', 'mahi', 'dhoni'} {'srk', 'salman', 'ranveer'}

{'sachin', 'mahi', 'dhoni'}

True

False

Example 2: Using set() method

```python
Days = set(["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"])
print(Days)
print(type(Days))
print("looping through the set elements ... ")
for i in Days:
    print(i)
```

Output:

```
{'Friday', 'Wednesday', 'Thursday', 'Saturday', 'Monday', 'Tuesday',
'Sunday'}
<class 'set'>
looping through the set elements ...
Friday
Wednesday
Thursday
Saturday
Monday
Tuesday
Sunday
```

It can contain any type of element such as integer, float, tuple etc. But mutable elements (list, dictionary, set) can't be a member of set. Consider the following example.

Example:

```
set={{1,2,3},{4,5,6}}

ERROR!

Traceback (most recent call last):

  File "<string>", line 15, in <module>

TypeError: unhashable type: 'set'
```

```
# Creating a set which have immutable elements
set1 = {1,2,3, "amit", 20.5, 14}
print(type(set1))
#Creating a set which have mutable element
set2 = {1,2,3,["amit",4]}
print(type(set2))
```

Output:

```
<class 'set'>

Traceback (most recent call last)
<ipython-input-5-9605bb6fbc68> in <module>
      4
      5 #Creating a set which holds mutable elements
----> 6 set2 = {1,2,3,["amit",4]}
      7 print(type(set2))

TypeError: unhashable type: 'list'
```

In the above code, we have created two sets, the set **set1** have immutable elements and set2 have one mutable element as a list. While checking the type of set2, it raised an error, which means set can contain only immutable elements.

Creating an empty set is a bit different because empty curly {} braces are also used to create a dictionary as well. So Python provides the set() method used without an argument to create an empty set.

```
# Empty curly braces will create dictionary
set3 = {}
print(type(set3))

# Empty set using set() function
```

```
set4 = set()
print(type(set4))
```

Output:

```
<class 'dict'>
<class 'set'>
```

Let's see what happened if we provide the duplicate element to the set.

```
set5 = {1,2,4,4,5,8,9,9,10}
print("Return set with unique elements:",set5)
```

Output:

```
Return set with unique elements: {1, 2, 4, 5, 8, 9, 10}
```

In the above code, we can see that **set5** consisted of multiple duplicate elements when we printed it remove the duplicity from the set.

Adding items to the set

Python provides the **add()** method and **update()** method which can be used to add some particular item to the set. The add() method is used to add a single element whereas the update() method is used to add multiple elements to the set. Consider the following example.

Example: 1 - Using add() method

```
Months = set(["January","February", "March", "April", "May", "June"])
print("\nprinting the original set ... ")
print(months)
print("\nAdding other months to the set...");
Months.add("July");
Months.add ("August");
print("\nPrinting the modified set...");
print(Months)
print("\nlooping through the set elements ... ")
for i in Months:
    print(i)
```

Output:

```
printing the original set ...
{'February', 'May', 'April', 'March', 'June', 'January'}

Adding other months to the set...

Printing the modified set...
{'February', 'July', 'May', 'April', 'March', 'August', 'June', 'January'}

looping through the set elements ...
February
July
May
April
March
August
June
January
```

To add more than one item in the set, Python provides the **update()** method. It accepts iterable as an argument.

Consider the following example.

Example:

set9={"one","two"}

set9.update("amit")

print(set9)

Output:

{'one', 'm', 't', 'i', 'a', 'two'}

Example - 2 Using update() function

```
Months = set(["January","February", "March", "April", "May", "June"])
print("\nprinting the original set ... ")
print(Months)
print("\nupdating the original set ... ")
Months.update(["July","August","September","October"]);
print("\nprinting the modified set ... ")
print(Months);
```

Output:

```
printing the original set ...
{'January', 'February', 'April', 'May', 'June', 'March'}

updating the original set ...
printing the modified set ...
{'January', 'February', 'April', 'August', 'October', 'May', 'June', 'July',
'September', 'March'}
```

Removing items from the set

Python provides the **discard()** method and **remove()** method which can be used to remove the items from the set. The difference between these functions, using discard() function if the item does not exist in the set then the set remain unchanged whereas remove() method will through an error.

Consider the following example.

Example-1 Using discard() method

```
months = set(["January","February", "March", "April", "May", "June"])
print("\nprinting the original set ... ")
print(months)
print("\nRemoving some months from the set...");
months.discard("January");
months.discard("May");
print("\nPrinting the modified set...");
print(months)
print("\nlooping through the set elements ... ")
for i in months:
    print(i)
```

Output:

```
printing the original set ...
{'February', 'January', 'March', 'April', 'June', 'May'}

Removing some months from the set...

Printing the modified set...
{'February', 'March', 'April', 'June'}

looping through the set elements ...
February
March
April
June
```

Python provides also the **remove()** method to remove the item from the set. Consider the following example to remove the items using **remove()** method.

Example-2 Using remove() function

```
months = set(["January","February", "March", "April", "May", "June"])
print("\nprinting the original set ... ")
print(months)
print("\nRemoving some months from the set...");
months.remove("January");
months.remove("May");
print("\nPrinting the modified set...");
print(months)
```

Output:

```
printing the original set ...
{'February', 'June', 'April', 'May', 'January', 'March'}

Removing some months from the set...

Printing the modified set...
{'February', 'June', 'April', 'March'}
```

Pop()

We can also use the pop() method to remove the item. Generally, the pop() method will always remove the last item but the set is unordered, we can't determine which element will be popped from set.

Consider the following example to remove the item from the set using pop() method.

```
Months = set(["January","February", "March", "April", "May", "June"])
print("\nprinting the original set ... ")
print(Months)
print("\nRemoving some months from the set...");
Months.pop();
Months.pop();
print("\nPrinting the modified set...");
print(Months)
```

Output:

```
printing the original set ...
```

```
{'June', 'January', 'May', 'April', 'February', 'March'}

Removing some months from the set...

Printing the modified set...
{'May', 'April', 'February', 'March'}
```

In the above code, the last element of the **Month** set is **March** but the pop() method removed the **June and January** because the set is unordered and the pop() method could not determine the last element of the set.

Python provides the clear() method to remove all the items from the set.

Consider the following example.

```
Months = set(["January","February", "March", "April", "May", "June"])
print("\nprinting the original set ... ")
print(Months)
print("\nRemoving all the items from the set...");
Months.clear()
print("\nPrinting the modified set...")
print(Months)
```

Output:

```
printing the original set ...
{'January', 'May', 'June', 'April', 'March', 'February'}

Removing all the items from the set...

Printing the modified set...
set()
```

Difference between discard() and remove()

Despite the fact that **discard()** and **remove()** method both perform the same task, There is one main difference between discard() and remove().

If the key to be deleted from the set using discard() doesn't exist in the set, the Python will not give the error. The program maintains its control flow.

On the other hand, if the item to be deleted from the set using remove() doesn't exist in the set, the Python will raise an error.

Consider the following example.

Example-

```
Months = set(["January","February", "March", "April", "May", "June"])
print("\nprinting the original set ... ")
print(Months)
print("\nRemoving items through discard() method...");
Months.discard("Feb"); #will not give an error although the key feb is not available in the set
print("\nprinting the modified set...")
print(Months)
print("\nRemoving items through remove() method...");
Months.remove("Jan") #will give an error as the key jan is not available in the set.
print("\nPrinting the modified set...")
print(Months)
```

Output:

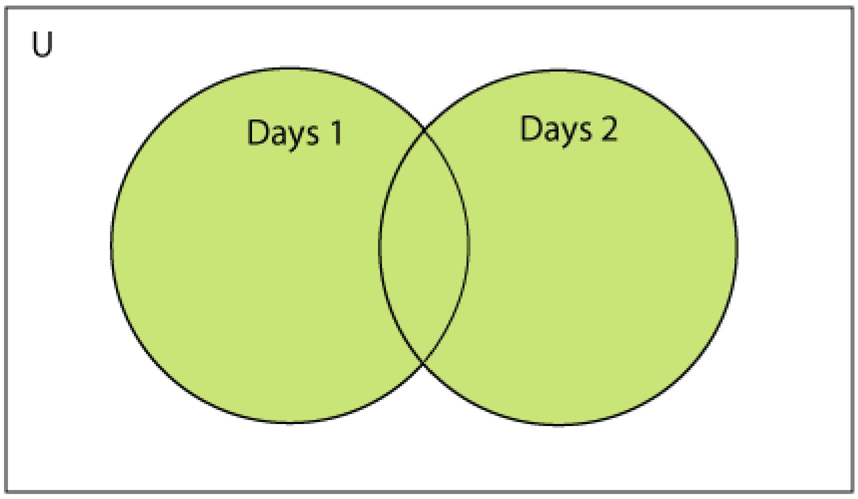
```
printing the original set ...
{'March', 'January', 'April', 'June', 'February', 'May'}

Removing items through discard() method...

printing the modified set...
{'March', 'January', 'April', 'June', 'February', 'May'}

Removing items through remove() method...
Traceback (most recent call last):
  File "set.py", line 9, in
    Months.remove("Jan")
KeyError: 'Jan'
```

Python Set Operations

Set can be performed mathematical operation such as union, intersection, difference, and symmetric difference. Python provides the facility to carry out these operations with operators or methods. We describe these operations as follows.

Union of two Sets

To combine two or more sets into one set in Python, use the union() function. All of the distinctive characteristics from each combined set are present in the final set. As parameters, one or more sets may be passed to the union() function. The function returns a copy of the set supplied as the lone parameter if there is just one set. The method returns a new set containing all the different items from all the arguments if more than one set is supplied as an argument.



Consider the following example to calculate the union of two sets.

Example 1: using union | operator

```
Days1 = {"Monday","Tuesday","Wednesday","Thursday", "Sunday"}
Days2 = {"Friday","Saturday","Sunday"}
print(Days1|Days2) #printing the union of the sets
```

Output:

```
{'Friday', 'Sunday', 'Saturday', 'Tuesday', 'Wednesday', 'Monday', 
'Thursday'}
```

Python also provides the **union()** method which can also be used to calculate the union of two sets. Consider the following example.

Example 2: using union() method

```
Days1 = {"Monday","Tuesday","Wednesday","Thursday"}
Days2 = {"Friday","Saturday","Sunday"}
print(Days1.union(Days2)) #printing the union of the sets
```

Output:

```
{'Friday', 'Monday', 'Tuesday', 'Thursday', 'Wednesday', 'Sunday', 
'Saturday'}
```

Now, we can also make the union of more than two sets using the union() function, for example:

Program:

```
# Create three sets
set1 = {1, 2, 3}
set2 = {2, 3, 4}
set3 = {3, 4, 5}

elements = set1.union(set2, set3)

# Print the common elements
print(elements)
```

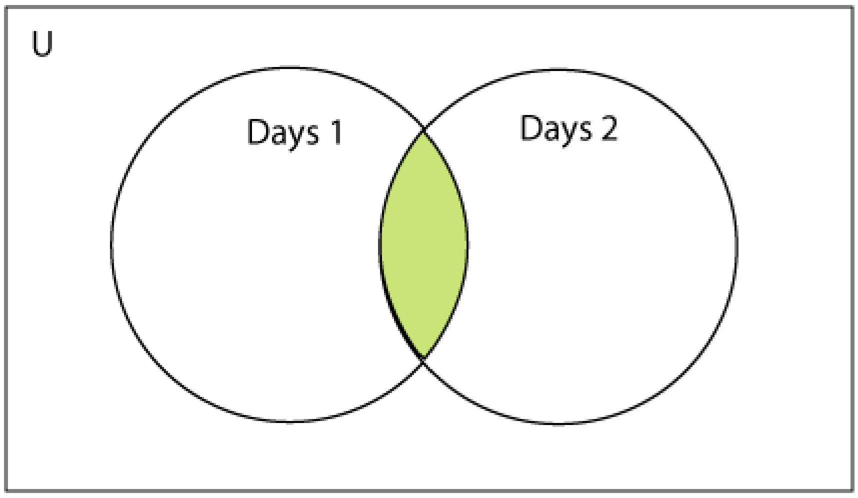
Output:

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

The intersection of two sets

To discover what is common between two or more sets in Python, apply the intersection() function. Only the items in all sets being compared are included in the final set. One or more sets can also be used as the intersection() function parameters. The function returns a copy of the set supplied as the lone parameter if there is just one set. The method returns a new set that only contains the elements in all the compared sets if multiple sets are supplied as arguments.

The intersection of two sets can be performed by the **&** operator or the **intersection() function**. The intersection of the two sets is given as the set of the elements that common in both sets.



Consider the following example.

Example 1: Using & operator

```
Days1 = {"Monday","Tuesday", "Wednesday", "Thursday"}
Days2 = {"Monday","Tuesday","Sunday", "Friday"}
print(Days1&Days2) #prints the intersection of the two sets
```

Output:

```
{'Monday', 'Tuesday'}
```

Example 2: Using intersection() method

```
set1 = {"Devansh","John", "David", "Martin"}
set2 = {"Steve", "Milan", "David", "Martin"}
print(set1.intersection(set2)) #prints the intersection of the two sets
```

Output:

```
{'Martin', 'David'}
```

Example 3:

```
set1 = {1,2,3,4,5,6,7}
set2 = {1,2,20,32,5,9}
set3 = set1.intersection(set2)
print(set3)
```

Output:

```
{1,2,5}
```

Similarly, as the same as union function, we can perform the intersection of more than two sets at a time,

For Example:

Program

```
# Create three sets
set1 = {1, 2, 3}
set2 = {2, 3, 4}
set3 = {3, 4, 5}
```

```

# Find the common elements between the three sets
common_elements = set1.intersection(set2, set3)

# Print the common elements
print(common_elements)
```

Output:

```
{3}
```

The intersection_update() method

The **intersection_update()** method removes the items from the original set that are not present in both the sets (all the sets if more than one are specified).So, this method will modify the original set to contain only the intersection (common) elements across all the sets.

The **intersection_update()** method is different from the intersection() method since it modifies the original set by removing the unwanted items, on the other hand, the intersection() method returns a new set.

If there are no common elements, then original set becomes empty set losing all our data in the original set.

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

set2={4,5,6}

print(set.intersection_update(set2))

print(set)

print(set2)
```

Output:

```
None

set()

{4, 5, 6}
```

Consider the following example.

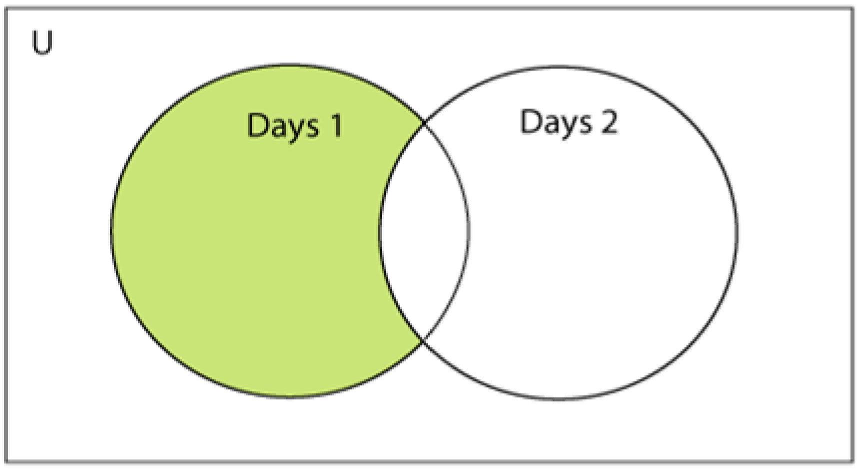
```
a = {"Devansh", "bob", "castle"}
b = {"castle", "dude", "emyway"}
c = {"fuson", "gaurav", "castle"}

a.intersection_update(b, c)

print(a)
```

Output:

```
{'castle'}
```

Difference between the two sets

The difference of two sets can be calculated by using the subtraction (-) operator or **intersection()** method. Suppose there are two sets A and B, and the difference is A-B that denotes the resulting set will be obtained that element of A, which is not present in the set B.



Consider the following example.

Example 1 : Using subtraction (-) operator

```
Days1 = {"Monday",  "Tuesday", "Wednesday", "Thursday"}
Days2 = {"Monday", "Tuesday", "Sunday"}
```

```
3. print(Days1-Days2) #{"Wednesday", "Thursday" will be printed}
```

Output:

```
{'Thursday', 'Wednesday'}
```

Example 2 : Using difference() method

```
1. Days1 = {"Monday",  "Tuesday", "Wednesday", "Thursday"}
2. Days2 = {"Monday", "Tuesday", "Sunday"}
3. print(Days1.difference(Days2)) # prints the difference of the two sets Days1 and Days 2
```

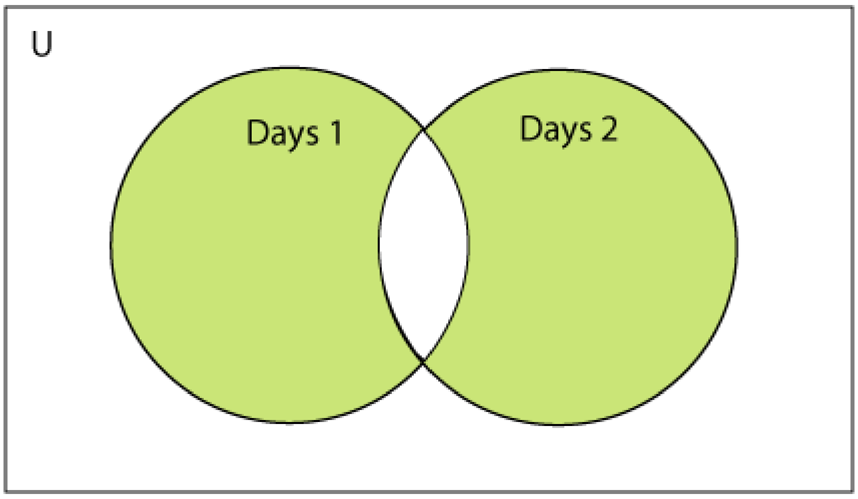
Output:

```
{'Thursday', 'Wednesday'}
```

Symmetric Difference of two sets

In Python, the symmetric Difference between set1 and set2 is the set of elements present in one set or the other but not in both sets. In other words, the set of elements is in set1 or set2 but not in their intersection. It will return a new set with all the elements from both the sets after dropping common elements.

The Symmetric Difference of two sets can be computed using Python's symmetric_difference() method. This method returns a new set containing all the elements in either but not in both. Consider the following example:



Example - 1: Using ^ operator

```
a = {1,2,3,4,5,6}
b = {1,2,9,8,10}
c = a^b
print(c)
```

Output:

```
{3, 4, 5, 6, 8, 9, 10}
```

Example - 2: Using symmetric_difference() method

```
a = {1,2,3,4,5,6}
b = {1,2,9,8,10}
c = a.symmetric_difference(b)
print(c)
```

Output:

```
{3, 4, 5, 6, 8, 9, 10}
```

Set comparisons

In Python, you can compare sets to check if they are equal, if one set is a subset or superset of another, or if two sets have elements in common.

Here are the set comparison operators available in Python:

- ==: checks if two sets have the same elements, regardless of their order.
- !=: checks if two sets are not equal.
- <: checks if the left set is a proper subset of the right set (i.e., all elements in the left set are also in the right set, but the right set has additional elements).
- <=: checks if the left set is a subset of the right set (i.e., all elements in the left set are also in the right set).
- >: checks if the left set is a proper superset of the right set (i.e., all elements in the right set are also in the left set, but the left set has additional elements).
- >=: checks if the left set is a superset of the right set (i.e., all elements in the right set are also in the left).

Consider the following example.

```
Days1 = {"Monday",  "Tuesday", "Wednesday", "Thursday"}
```

```
Days2 = {"Monday", "Tuesday"}
Days3 = {"Monday", "Tuesday", "Friday"}

#Days1 is the superset of Days2 hence it will print true.
print (Days1>Days2)

#prints false since Days1 is not the subset of Days2
print (Days1<Days2)

#prints false since Days2 and Days3 are not equivalent
print (Days2 == Days3)
```

Output:

```
True
False
False
```

FrozenSets

In Python, a frozen set is an immutable version of the built-in set data type. It is similar to a set, **but its contents cannot be changed once a frozen set is created. It means we cannot add/remove elements from frozen set but we can do so with standard set**. However, elements of frozen sets are immutable just like set, and hence both they cannot contain, for example, other sets or lists as elements.

list=[1,2,3]

set={list} #error

Below will also give you the error:

fs2=frozenset()

fs2={list} #this will give error

Example:

list=[1,2,3]

fs=frozenset([1,2,3])

```
fs2=frozenset()

fs2={4,5,fs} #NO error


print(fs)

print(fs2)
```

Output:

```
frozenset({1, 2, 3})

{frozenset({1, 2, 3}), 4, 5}
```

Below will give error:

```
set1={1,2,3}

s=set(set1)

s2=set()

s2={4,5,s}   #this will give error
```

Tuples and frozen sets are immutable, means we cannot add / remove elements from them. Hence, Tuples and frozen sets can be elements of frozen sets, sets, dictionary keys etc.

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

list2=[1,2,3,list] #LIST CAN CONTAIN OTHER LISTS, NO ERROR, SEE THE OUTPUT

print(list)

print(list2)
```

```
set={1,2}

set={4,5,set} #THIS WILL GIVE ERROR AS SET CANNOT CONTAIN OTHER SET/LIST ETC.
```

```
OUTPUT:

[1, 2, 3]

[1, 2, 3, [1, 2, 3]]

Traceback (most recent call last):

File "<string>", line 8, in <module>

ERROR!

TypeError: unhashable type: 'set'
```

Frozen set objects are unordered collections of unique elements, just like sets. They can be used the same way as sets. Because they are immutable, frozen set objects can be used as elements of other sets or dictionary keys, while standard sets cannot be.

SET: Mutable, elements immutable so cannot contain other sets/lists as elements

FS: Immutable, elements also immutable

List: Mutable, elements also mutable so it can contain other lists/sets etc.

One of the main advantages of using frozen set objects is that they are hashable, meaning they can be used as keys in dictionaries or as elements of other sets. Their contents cannot change, so their hash values remain constant. Standard sets are not hashable because they can be modified (can add / remove elements), so their hash values can change.

Frozen set objects support many of the assets of the same operation, such as union, intersection, Difference, and symmetric Difference. They also support operations that do not modify the frozen set, such as len(), min(), max(), and in.

Consider the following example to create the frozen set.

1. Frozenset = frozenset([1,2,3,4,5])

```
print(type(Frozenset))
print("\nprinting the content of frozen set...")
for i in Frozenset:
    print(i);
Frozenset.add(6) #gives an error since we cannot change the content of Frozenset after creation
```

Output:

```
<class 'frozenset'>

printing the content of frozen set...
1
2
3
4
5
Traceback (most recent call last):
  File "set.py", line 6, in <module>
    Frozenset.add(6)  #gives an error since we can change the content of
Frozenset after creation
AttributeError: 'frozenset' object has no attribute 'add'
```

Frozenset for the dictionary

If we pass the dictionary as the sequence inside the frozenset() method, it will take only the keys from the dictionary and returns a frozenset that contains the key of the dictionary as its elements.

Consider the following example.

```
Dictionary = {"Name":"John", "Country":"USA", "ID":101}
print(type(Dictionary))
Frozenset = frozenset(Dictionary); #Frozenset will contain the keys of the dictionary
print(type(Frozenset))
for i in Frozenset:
    print(i)
```

Output:

```
<class 'dict'>
<class 'frozenset'>
Name
Country
ID
```

Set Example

Example - 1: Write a program to remove the given number from the set.

```python
my_set = {1,2,3,4,5,6,12,24}
n = int(input("Enter the number you want to remove"))
my_set.discard(n)
print("After Removing:",my_set)
```

Output:

```
Enter the number you want to remove:12
After Removing: {1, 2, 3, 4, 5, 6, 24}
```

Example - 2: Write a program to add multiple elements to the set.

```python
set1 = set([1,2,4,"John","CS"])
set1.update(["Apple","Mango","Grapes"])
print(set1)
```

Output:

```
{1, 2, 4, 'Apple', 'John', 'CS', 'Mango', 'Grapes'}
```

Example - 3: Write a program to find the union between two set.

```python
set1 = set(["Peter","Joseph", 65,59,96])
set2  = set(["Peter",1,2,"Joseph"])
set3 = set1.union(set2)
print(set3)
```

Output:

```
{96, 65, 2, 'Joseph', 1, 'Peter', 59}
```

Example- 4: Write a program to find the intersection between two sets.

```python
set1 = {23,44,56,67,90,45,"amit"}
set2 = {13,23,56,76,"Sachin"}
set3 = set1.intersection(set2)
print(set3)
```

Output:

```
{56, 23}
```

Example - 6: Write the program to find the issuperset, issubset and superset.

```
set1 = set(["Peter","James","Camroon","Ricky","Donald"])
set2 = set(["Camroon","Washington","Peter"])
set3 = set(["Peter"])

issubset = set1 >= set2
print(issubset)
issuperset = set1 <= set2
print(issuperset)
issubset = set3 <= set2
print(issubset)
issuperset = set2 >= set3
print(issuperset)
```

Output:

```
False
False
True
True
```

Python Built-in set methods

Python contains the following methods to be used with the sets.

SN	Method	Description
1	add(item)	It adds an item to the set. It has no effect if the item is already present in the set.
2	clear()	It deletes all the items from the set.
3	copy()	It returns a copy of the set.
4	difference_update(....)	It modifies this set by removing all the items that are also present in the specified sets.
5	discard(item)	It removes the specified item from the set.
6	intersection()	It returns a new set that contains only the common elements of both the sets. (all the sets if more than two are specified).
7	intersection_update(....)	It removes the items from the original set that are not present in both the sets (all the sets if more than one are specified).
8	Isdisjoint(....)	Return True if two sets have a null intersection.
9	Issubset(....)	Report whether another set contains this set.
10	Issuperset(....)	Report whether this set contains another set.
11	pop()	Remove and return an arbitrary set element that is the last element of the set. Raises KeyError if the set is empty.
12	remove(item)	Remove an element from a set; it must be a member. If the element is not a member, raise a KeyError.
14	symmetric_difference_update(....)	Update a set with the symmetric difference of itself and another. **(Keep all elements from both the sets except common elements)**
15	union(....)	Return the union of sets as a new set, repetition of an element is not allowed (i.e. all elements that are in either set.)

16	update()	Update a set with the union of itself and others.
		s={1,2,3}
		s2={10,20}

		s.update(s2)
		print(s)
		print(s2)
		
		Output:
		{1, 2, 3, 20, 10}
		{10, 20}

“”” or ‘’’(triple quotes) are for multiline comments

Python Dictionary

Dictionaries are a useful data structure for storing data in Python because they are given key.

The data is stored as key-value pairs using a Python dictionary.

- This data structure is mutable
- The components (elements) of dictionary were made using keys and values.
- Values can be of any type, including integer, list, and tuple.

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



dict={"name":"amit","set":s}

dict2={"mydict":dict}

print(dict)

print(dict2)
```

Output:

{'name': 'amit', 'set': {1, 2, 3}}

{'mydict': {'name': 'amit', 'set': {1, 2, 3}}}

A dictionary is, in other words, a group of key-value pairs**, where the values can be any Python object. The keys, in contrast, are immutable Python objects, such as strings, tuples, or numbers.** Dictionary entries are ordered as of Python version 3.7. In Python 3.6 and before, dictionaries are generally unordered.

Creating the Dictionary

Curly brackets are the simplest way to generate a Python dictionary, although there are other approaches as well. With many key-value pairs surrounded in curly brackets and a colon separating each key from its value, the dictionary can be built. (:). The following provides the syntax for defining the dictionary.

Syntax:

```
Dict = {"Name": "Gayle", "Age": 25}
```

In the above dictionary **Dict**, The keys **Name** and **Age** are the strings which comes under the category of an immutable object.

Let's see an example to create a dictionary and print its content.

Code

```
Employee = {"Name": "Johnny", "Age": 32, "salary":26000,"Company":"TCS"}
print(type(Employee))
print("printing Employee data .... ")
print(Employee)
```

Output

```
<class 'dict'>
printing Employee data ....
{'Name': 'Johnny', 'Age': 32, 'salary': 26000, 'Company': TCS}
```

Python provides the built-in function **dict()** method which is also used to create the dictionary.

The empty curly braces {} is used to create empty dictionary.

Code

```
# Creating an empty Dictionary
Dict = {}       #this is dictionary and not set
print("Empty Dictionary: ")
print(Dict)

# Creating a Dictionary
# with dict() method
Dict = dict({1: 'Hcl', 2: 'WIPRO', 3:'Facebook'})
print("\nCreate Dictionary by using  dict(): ")
print(Dict)

# Creating a Dictionary
# with each item as a Pair
Dict = dict([(4, 'Rinku'), (2, ‘Singh’)])
print("\nDictionary with each item as a pair: ")
print(Dict)
```

Output

```
Empty Dictionary:
{}

Create Dictionary by using  dict():
{1: 'Hcl', 2: 'WIPRO', 3: 'Facebook'}

Dictionary with each item as a pair:
{4: 'Rinku', 2: 'Singh'}
```

Accessing the dictionary values

To access data contained in lists and tuples, indexing has been studied. The keys of the dictionary can be used to obtain the values because they are unique from one another. The following method can be used to access dictionary values.

Code

```
Employee = {"Name": "Dev", "Age": 20, "salary":45000,"Company":"WIPRO"}
print(type(Employee))
print("printing Employee data .... ")
print("Name : %s" %Employee["Name"])
```

5. **print**("Age : %d" %Employee["Age"])
6. **print**("Salary : %d" %Employee["salary"])
7. **print**("Company : %s" %Employee["Company"])

Output

```
ee["Company"])
Output
<class 'dict'>
printing Employee data ....
Name : Dev
Age : 20
Salary : 45000
Company : WIPRO
```

Python provides us with an alternative to use the get() method to access the dictionary values. It would give the same result as given by the indexing.

Dict = dict({1: 'Hcl', 2: 'WIPRO', 3:'Facebook',"salary":1000})

print(type(Dict['salary']))

print("salary is ",Dict['salary'])

print(F"salary is {Dict['salary']}")

Output:

<class 'int'>

salary is 1000

salary is 1000

Adding Dictionary Values

The dictionary is a mutable data type, and utilising the right keys allows you to change its values. An existing value can also be updated using the update() method.

Note: The value is updated if the key-value pair is already present in the dictionary. Otherwise, the dictionary adds a new key.

Let's see an example to update the dictionary values.

Example - 1:

Code

```
# Creating an empty Dictionary
Dict = {}
print("Empty Dictionary: ")
print(Dict)

# Adding elements to dictionary one at a time
Dict[0] = 'Peter'
Dict[2] = 'Joseph'
Dict[3] = 'Ricky'
print("\nDictionary after adding 3 elements: ")
print(Dict)

# Adding set of values
# with a single Key
# The Emp_ages doesn't exist to dictionary
Dict['Emp_ages'] = 20, 33, 24      (Yes, multiple values can be assigned to a single key)
print("\nDictionary after adding 3 elements: ")
print(Dict)

# Updating existing Key's Value
Dict[3] = 'amit'
print("\nUpdated key value: ")
print(Dict)
```

Output

```
Empty Dictionary:
{}

Dictionary after adding 3 elements:
{0: 'Peter', 2: 'Joseph', 3: 'Ricky'}

Dictionary after adding 3 elements:
{0: 'Peter', 2: 'Joseph', 3: 'Ricky', 'Emp_ages': (20, 33, 24)}

Updated key value:
{0: 'Peter', 2: 'Joseph', 3: 'amit', 'Emp_ages': (20, 33, 24)}
```

Example - 2:

Code

```
Employee = {"Name": "Dev", "Age": 20, "salary":45000,"Company":"WIPRO"}
print(type(Employee))
print("printing Employee data .... ")
print(Employee)
print("Enter the details of the new employee....");
Employee["Name"] = input("Name: ");
Employee["Age"] = int(input("Age: "));
Employee["salary"] = int(input("Salary: "));
Employee["Company"] = input("Company:");
print("printing the new data");
print(Employee)
```

Output

```
<class 'dict'>
printing Employee data ....
Employee  =  {"Name":  "Dev",  "Age":  20,  "salary":45000,"Company":"WIPRO"}
Enter the details of the new employee....
Name: Sunny
Age: 38
Salary: 39000
Company:Hcl
printing the new data
{'Name': 'Sunny', 'Age': 38, 'salary': 39000, 'Company': 'Hcl'}
```

Deleting Elements using del Keyword

The items of the dictionary can be deleted by using the **del** keyword as given below.

Code

```
Employee = {"Name": "David", "Age": 30, "salary":55000,"Company":"WIPRO"}
print(type(Employee))
print("printing Employee data .... ")
print(Employee)
print("Deleting some of the employee data")
del Employee["Name"]
del Employee["Company"]
print("printing the modified information ")
print(Employee)
print("Deleting the dictionary: Employee");
```

```
del Employee
print("Lets try to print it again ");
print(Employee)
```

Output

```
<class 'dict'>
printing Employee data ....
{'Name': 'David', 'Age': 30, 'salary': 55000, 'Company': 'WIPRO'}
Deleting some of the employee data
printing the modified information
{'Age': 30, 'salary': 55000}
Deleting the dictionary: Employee
Lets try to print it again
NameError: name 'Employee' is not defined.
```

The last print statement in the above code, it raised an error because we tried to print the Employee dictionary that already deleted.

Deleting Elements using pop() Method

A dictionary is a group of key-value pairs in Python. You can retrieve, insert, and remove items using this unordered, mutable data type by using their keys (and not indexes). The pop() method is one of the ways to get rid of elements from a dictionary. we'll talk about how to remove items from a Python dictionary using the pop() method.

The value connected to a specific key in a dictionary is removed using the pop() method, which then returns the value. The key of the element to be removed is the only argument needed. The pop() method can be used in the following ways:

Code

```
# Creating a Dictionary
Dict1 = {1: 'amit', 2: 'Educational', 3: ‘College'}
# Deleting a key
# using pop() method
pop_keyValue_returned = Dict1.pop(2)     #returns value at 2, i.e.Educational
print(Dict1)
```

Output

```
{1: 'amit', 3: 'Website'}
```

Additionally, Python offers built-in functions popitem() and clear() for removing dictionary items. In contrast to the clear() method, which removes all of the elements from the entire dictionary, popitem() removes any element from a dictionary.

```
di={"name":"amit"}

value=di.popitem()

print(di)

print(value)
```

Output:

```
{}

('name', 'amit')
```

Iterating Dictionary

A dictionary can be iterated using for loop as given below.

Example 1

Code

```
# for loop to print all the keys of a dictionary
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"WIPRO"}
for x in Employee:
    print(x)
```

Output

```
Name
Age
salary
Company
```

Example 2

Code

```
#for loop to print all the values of the dictionary
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"WIPRO"}
for x in Employee:
 print(Employee[x])
```

Output

```
John
29
25000
WIPRO
```

Example - 3

Code

```python
#for loop to print the values of the dictionary by using values() method.
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"WIPRO"}
for x in Employee.values():
    print(x)
```

Output

```
John
29
25000
WIPRO
```

Example 4

Code

Item=Key+Value

```python
#for loop to print the items of the dictionary by using items() method
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"WIPRO"}
for x in Employee.items():
    print(x)
```

Output

```
('Name', 'John')
('Age', 29)
('salary', 25000)
('Company', 'WIPRO')
```

Properties of Dictionary Keys

1. In the dictionary, we cannot store multiple (duplicate) keys with the same name. So, key-names are unique in a dictionary. If we use multiple keys

with the same name, then only the last duplicate key/value pair will be stored.

Consider the following example.

Code

```
Employee={"Name":"John","Age":29,"Salary":25000,"Company":"WIPRO","Name":
"amit"}
for x,y in Employee.items():
        print(x,y)
```

Output

```
Name amit
Age 29
Salary 25000
Company WIPRO
```

```
di={"name":"amit","surname":"patel","University":"PU"}

for x,y in di.items():

    print(x,"=",y)
```

Output:

name = amit

surname = patel

University = PU

2. The key cannot belong to any mutable object in Python. Numbers, strings, or tuples can be used as the key, however mutable objects like lists cannot be used as the key in a dictionary.

Consider the following example.

Code

1. Employee = {"Name": "John", "Age": 29, "salary":26000,"Company":"WIPRO",[100,201,301]:"Department ID"}
2. **for** x,y **in** Employee.items():
3.   **print**(x,y)

Output

```
Traceback (most recent call last):
  File "dictionary.py", line 1, in
    Employee        =        {"Name":        "John",        "Age":        29,
"salary":26000,"Company":"WIPRO",[100,201,301]:"Department ID"}
TypeError: unhashable type: 'list'
```

Built-in Dictionary Functions

A few of the Python methods can be combined with a Python dictionary.

The built-in Python dictionary methods are listed below, along with a brief description.

- **len()**

The dictionary's length is returned via the len() function in Python. The string is lengthened by one for each key-value pair.

Code

1. dict = {1: "Ayan", 2: "Bunny", 3: "Ramesh", 4: "Bheem"}
2. len(dict)

Output

```
4
```

- **sorted()**

Like it does with lists and tuples, the sorted() method returns an ordered series of the dictionary's keys. The ascending sorting has no effect on the original Python dictionary.

Code

1. dict = {7: "Ayan", 5: "Bunny", 8: "Ram", 1: "Bheem"}
2. sorted(dict)   #will sort according to keys, Capitals will be ordered first

Output

```
[ 1, 5, 7, 8]
```

Built-in Dictionary methods

The built-in python dictionary methods along with the description and Code are given below.

- **clear()**

It is mainly used to delete all the items of the dictionary.

Code

```
# dictionary methods
dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
# clear() method
dict.clear()
print(dict)
```

Output

```
{ }
```

- **copy()**

It returns a copy of the dictionary which is created.

Code

```
# dictionary methods
dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
# copy() method
dict_demo = dict.copy()
print(dict_demo)
```

Output

```
{1: 'Hcl', 2: 'WIPRO', 3: 'Facebook', 4: 'Amazon', 5: 'Flipkart'}
```

- **pop()**

It mainly eliminates the element using the defined key.

Code

```python
# dictionary methods
dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
# pop() method
dict_demo = dict.copy()
x = dict_demo.pop(1)
print(x)
```

Output

```
{2: 'WIPRO', 3: 'Facebook', 4: 'Amazon', 5: 'Flipkart'}
```

popitem()

removes the most recent (Last) key-value pair entered

Code

```python
# dictionary methods
dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
# popitem() method
dict.popitem()
dict.popitem()
print(dict)
```

Output

```
{1: 'Hcl', 2: 'WIPRO', 3: 'Facebook'}
```

- **keys()**

It returns all the keys of the dictionary.

Code

```python
# dictionary methods
dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
# keys() method
print(dict.keys())
```

Output

```
dict_keys([1, 2, 3, 4, 5])
```

- **items()**

It returns all the key-value pairs as a tuple.

Code

```python
# dictionary methods
dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
# items() method
print(dict.items())
```

Output

```
dict_items([(1, 'Hcl'), (2, 'WIPRO'), (3, 'Facebook'), (4, 'Amazon'), (5, 
'Flipkart')])
```

- **get()**

It is used to get the value specified for the passed key.

Code

```python
# dictionary methods
dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
# get() method
print(dict.get(3))
```

Output

```
Facebook
```

- **update()**

It mainly updates the dictionary1 by adding the key-value pair of dict2 to this dictionary1.

Code

```python
# dictionary methods
dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
# update() method
dict.update({3: "TCS"})    #will add 3,TCS as new key-value pair
dict2={}
dict2.update(dict) # dict2 gets all pairs from dict
print(dict)
print(dict2)
```

values()

It returns all the values of the dictionary

Code

```
# dictionary methods
dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
# values() method
print(dict.values())
```

#it will print values , HCL Wipro etc.

Python Tuples

We cannot alter the components of a tuple once they have been assigned. On the other hand, we can edit the contents of a list.

Example

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

Features of Python Tuple

- Tuples are an immutable data type, meaning their elements cannot be changed after they are generated.
- Each element in a tuple has a specific order that will never change because tuples are ordered sequences.

Forming a Tuple:

All the objects-also known as "elements"-must be separated by a comma, enclosed in parenthesis (). Although parentheses are not required, they are recommended.

Any number of items, including those with various data types (dictionary, string, float, list, etc.), can be contained in a tuple.

Code

```
# Python program to show how to create a tuple
```

```python
# Creating an empty tuple
empty_tuple = ()
print("Empty tuple: ", empty_tuple)

# Creating tuple having integers
int_tuple = (4, 6, 8, 10, 12, 14)
print("Tuple with integers: ", int_tuple)

# Creating a tuple having objects of different data types
mixed_tuple = (4, "Python", 9.3)
print("Tuple with different data types: ", mixed_tuple)

# Creating a nested tuple
nested_tuple = ("Python", {4: 5, 6: 2, 8:2}, (5, 3, 5, 6))
print("A nested tuple: ", nested_tuple)
```

Output:

```
Empty tuple:  ()
Tuple with integers:  (4, 6, 8, 10, 12, 14)
Tuple with different data types:  (4, 'Python', 9.3)
A nested tuple:  ('Python', {4: 5, 6: 2, 8: 2}, (5, 3, 5, 6))
```

Code

```python
# Python program to create a tuple without using parentheses
# Creating a tuple
tuple_ = 4, 5.7, "Tuples", ["Python", "Tuples"]
# Displaying the tuple created
print(tuple_)
# Checking the data type of object tuple_
print(type(tuple_) )
# Trying to modify tuple_
try:
    tuple_[1] = 4.2
except:
    print(TypeError )
```

Output:

```
(4, 5.7, 'Tuples', ['Python', 'Tuples'])
```

```
<class 'tuple'>
<class 'TypeError'>
```

Code

```
# Python program to show how to create a tuple having a single element
single_tuple = ("Tuple")   #( ) not must, it is a string “Tuple” without ( )
print( type(single_tuple) )
# Creating a tuple that has only one element
single_tuple = ("Tuple",)    # ( ) not must but , is must for tuples
print( type(single_tuple) )
# Creating tuple without parentheses
single_tuple = "Tuple",    # ( ) not must but , is must
print( type(single_tuple) )
```

Output:

```
<class 'str'>
<class 'tuple'>
<class 'tuple'>
```

Accessing Tuple Elements

A tuple's objects can be accessed in a variety of ways.

Indexing

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

The indices of a tuple with five items will range from 0 to 4. An Index Error will be raised assuming we attempt to get to a list from the Tuple that is outside the scope of the tuple record. An index above four will be out of range in this scenario.

Because the index in Python must be an integer, we cannot provide an index of a floating data type or any other type. If we provide a floating index, the result will be TypeError.

The method by which elements can be accessed through nested tuples can be seen in the example below.

Code

```
# Python program to show how to access tuple elements
```

```python
# Creating a tuple
tuple_ = ("Python", "Tuple", "Ordered", "Collection")
print(tuple_[0])
print(tuple_[1])
# trying to access element index more than the length of a tuple
try:
    print(tuple_[5])
except Exception as e:
    print(e)
# trying to access elements through the index of floating data type
try:
    print(tuple_[1.0])
except Exception as e:
    print(e)
# Creating a nested tuple
nested_tuple = ("Tuple", [4, 6, 2, 6], (6, 2, 6, 7))

# Accessing the index of a nested tuple
print(nested_tuple[0][3])
print(nested_tuple[1][1])
```

Output:

```
Python
Tuple
tuple index out of range
tuple indices must be integers or slices, not float
l
6
```

- **Negative Indexing**

Python's sequence objects support negative indexing.

The last element is addressed by - 1, the second last element by - 2, etc.

Code

```python
# Python program to show how negative indexing works in Python tuples
# Creating a tuple
tuple_ = ("Python", "Tuple", "Ordered", "Collection")
# Printing elements using negative indices
```

5. print("Element at -1 index: ", tuple_[-1])
6. print("Elements between -4 and -1 are: ", tuple_[-4:-1])   # gives from -4 till -2

Output:

```
Element at -1 index:  Collection
Elements between -4 and -1 are:  ('Python', 'Tuple', 'Ordered')
```

Slicing

Tuple slicing is a common practice in Python and the most common way for programmers to deal with practical issues. Look at a tuple in Python. Slice a tuple to access a variety of its elements. Using the colon as a straightforward slicing operator (:) is one strategy.

t=1,2,3,4,5,6

print(t[-2:0])

Output:

()

t=1,2,3,4,5,6

print(t[1:-4])

Output:

(2,)

t=1,2,3,4,5,6

print(t[1:-1])

Output:

(2, 3, 4, 5)

t=1,2,3,4,5,6

print(t[-2:-5])

Output:

()

t=1,2,3,4,5,6

print(t[2:2])

Output:

()

t=1,2,3,4,5,6

print(t[2:3])

(3,)

t=1,2,3,4,5,6

print(t[-5:-6])

Output:

()

t=1,2,3,4,5,6

print(t[-5:-6])

Output:

(2,)

To gain access to various tuple elements, we can use the slicing operator colon (:).

Code

```
# Python program to show how slicing works in Python tuples
# Creating a tuple
tuple_ = ("Python", "Tuple", "Ordered", "Immutable", "Collection", "Objects")
# Using slicing to access elements of the tuple
print("Elements between indices 1 and 3: ", tuple_[1:3])
# Using negative indexing in slicing
```

```
print("Elements between indices 0 and -4: ", tuple_[:-4])
# Printing the entire tuple by using the default start and end values.
print("Entire tuple: ", tuple_[:])
```

Output:

```
Elements between indices 1 and 3:  ('Tuple', 'Ordered')
Elements between indices 0 and -4:  ('Python', 'Tuple')
Entire tuple:   ('Python', 'Tuple', 'Ordered', 'Immutable', 'Collection',
'Objects')
```

Deleting a Tuple

A tuple's parts can't be modified. We are unable to eliminate or remove tuple components.

However, the keyword del can completely delete a tuple.

Code

```
# Python program to show how to delete elements of a Python tuple
# Creating a tuple
tuple_ = ("Python", "Tuple", "Ordered", "Immutable", "Collection", "Objects")
# Deleting a particular element of the tuple
try:
    del tuple_[3]
    print(tuple_)
except Exception as e:
    print(e)
# Deleting the variable from the global space of the program
del tuple_
# Trying accessing the tuple after deleting it
try:
    print(tuple_)
except Exception as e:
    print(e)
```

Output:

```
'tuple' object does not support item deletion
name 'tuple_' is not defined
```

Repetition Tuples in Python

Code

```
# Python program to show repetition in tuples
tuple_ = ('Python',"Tuples")
print("Original tuple is: ", tuple_)
# Repeting the tuple elements
tuple_ = tuple_ * 3
print("New tuple is: ", tuple_)
```

Output:

```
Original tuple is:  ('Python', 'Tuples')
New tuple is:  ('Python', 'Tuples', 'Python', 'Tuples', 'Python', 'Tuples')
```

Tuple Methods

Tuples is a collection of immutable objects. The following are some examples of these methods.

- **Count () Method**

The times the predetermined component happens in the Tuple is returned by the count () capability of the Tuple.

Code

" this is a string to be ignored"

'this string is also ignored'

'''this is

Multiline comment'''

"""again this is

A multiline comment"""

```
t=[6,6,6]
print(t.count(6))
print(t)
```

Output:

```
3
[6, 6, 6]
```

```
# Creating tuples
T1 = (0, 1, 5, 6, 7, 2, 2, 4, 2, 3, 2, 3, 1, 3, 2)
T2 = ('python', 'java', 'python', 'php', 'python', 'java')
# counting the appearance of 3
res = T1.count(2)
print('Count of 2 in T1 is:', res)
# counting the appearance of java
res = T2.count('java')
print('Count of Java in T2 is:', res)
```

Output:

```
Count of 2 in T1 is: 5
Count of java in T2 is: 2
```

Index() Method:

The Index() function returns the first instance of the requested element from the Tuple.

Parameters:

- The value/element that must be looked for.
- Start search from the index ...(Optional)

Code (Below works for List as well)

```
# Creating tuples
Tuple_data = (0, 1, 2, 3, 2, 3, 1, 3, 2)
# getting the index of 3
res = Tuple_data.index(3)
print('First occurrence of 3 is', res)
# getting the index of 3 after 4th
# index
res = Tuple_data.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
```

Tuple Membership Test

Code (Below works for List as well)

```python
# Python program to show how to perform membership test for tuples
# Creating a tuple
tuple_ = ("Python", "Tuple", "Ordered", "Immutable", "Collection", "Ordered")
# In operator
print('Tuple' in tuple_)
print('Items' in tuple_)
# Not in operator
print('Immutable' not in tuple_)
print('Items' not in tuple_)
```

Output:

```
True
False
False
True
```

Iterating Through a Tuple

A for loop can be used to iterate through each tuple element.

Code

```python
# Python program to show how to iterate over tuple elements
# Creating a tuple
tuple_ = ("Python", "Tuple", "Ordered", "Immutable")
# Iterating over tuple elements using a for loop
for item in tuple_:
    print(item)
```

Output:

```
Python
Tuple
Ordered
Immutable
```

Changing a Tuple

Once the elements of a tuple have been defined, we cannot change them. However, the nested elements can be altered if the element itself is a changeable data type like a list.

New values can be assigned to a tuple through reassignment.

Code

```
# Python program to show that Python tuples are immutable objects
# Creating a tuple
tuple_ = ("Python", "Tuple", "Ordered", "Immutable", [1,2,3,4])
# Trying to change the element at index 2
try:
    tuple_[2] = "Items"
    print(tuple_)
except Exception as e:
    print( e )
# But inside a tuple, we can change elements of a mutable object
tuple_[-1][2] = 10
print(tuple_)
# Changing the whole tuple
tuple_ = ("Python", "Items") # new tuple_ with new values
print(tuple_)
```

Output:

```
'tuple' object does not support item assignment
('Python', 'Tuple', 'Ordered', 'Immutable', [1, 2, 10, 4])
('Python', 'Items')
```

The + operator can be used to combine multiple tuples into one. This phenomenon is known as concatenation.

We can also repeat the elements of a tuple a predetermined number of times by using the * operator. This is already demonstrated above.

The aftereffects of the tasks + and * are new tuples.

Code

```python
# Python program to show how to concatenate tuples
# Creating a tuple
tuple_ = ("Python", "Tuple", "Ordered", "Immutable")
# Adding a tuple to the tuple_
print(tuple_ + (4, 5, 6))
```

Output:

```
('Python', 'Tuple', 'Ordered', 'Immutable', 4, 5, 6)
```

Tuples have the following advantages over lists:

- Due to tuples, the code is protected from accidental modifications. It is desirable to store non-changing information in "tuples" instead of "lists”.
- A tuple can be used as a dictionary key if it contains immutable values like strings, numbers, or another tuple. "Lists" cannot be utilized as dictionary keys because they are mutable.

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

for i in range(4):

    l.append(5)

print(l)
```

output:

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

```
l=list()

for i in range(4):

    l.append(5)

print(l)
```

Output:

```
[5, 5, 5, 5]
```

```
num=int(input('enter number:'))

l=[]

for i in range(num):

    l.append(input('enter name {}:'.format(i+1)))

print(l)
```

Output:

```
enter number:3

enter name 1:amit

enter name 2:patel

enter name 3:PU

['amit', 'patel', 'PU']
```

Use of format string (method of String class)

```
txt1 = "My name is {fname}, I'm {age}".format(fname = "John", age
= 36)
txt2 = "My name is {0}, I'm {1}".format("John",36)
txt3 = "My name is {}, I'm {}".format("John",36)
```

Tuples or Lists as Dictionary Keys:

We can't employ a list as a key of a dictionary because it is mutable. This is because a key of a Python dictionary is an immutable object. As a result, tuples can be used as keys to a dictionary if required.

Let's consider the example highlighting the difference between lists and tuples in immutability and mutability.

Example Code

```
# Updating the element of list and tuple at a particular index

# creating a list and a tuple
list_ = ["Python", "Lists", "Tuples", "Differences"]
tuple_ = ("Python", "Lists", "Tuples", "Differences")

# modifying the last string in both data structures
list_[3] = "Mutable"
print( list_ )
try:
    tuple_[3] = "Immutable"
    print( tuple_ )
except TypeError:
    print( "Tuples cannot be modified because they are immutable" )
```

Output:

```
['Python', 'Lists', 'Tuples', 'Mutable']
Tuples cannot be modified because they are immutable
```

Python String

Python string is the collection of the characters surrounded by single quotes, double quotes, or **triple quotes.**

Consider the following example in Python to create a string.

Syntax:

```
str = "Hi Python !"
str2='a' #this is also a string, not a character
```

Here, if we check the type of the variable **str** using a Python script

```
print(type(str)), then it will print a string (str).
```

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

Creating String in Python

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

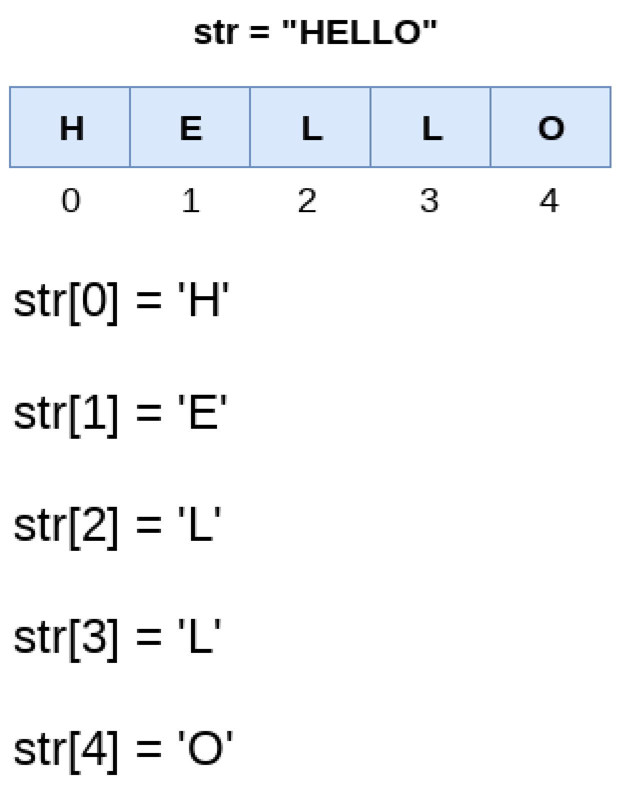
```
#Using single quotes
str1 = 'Hello Python'
print(str1)
#Using double quotes
str2 = "Hello Python"
print(str2)

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

Output:

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

Strings indexing and splitting

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



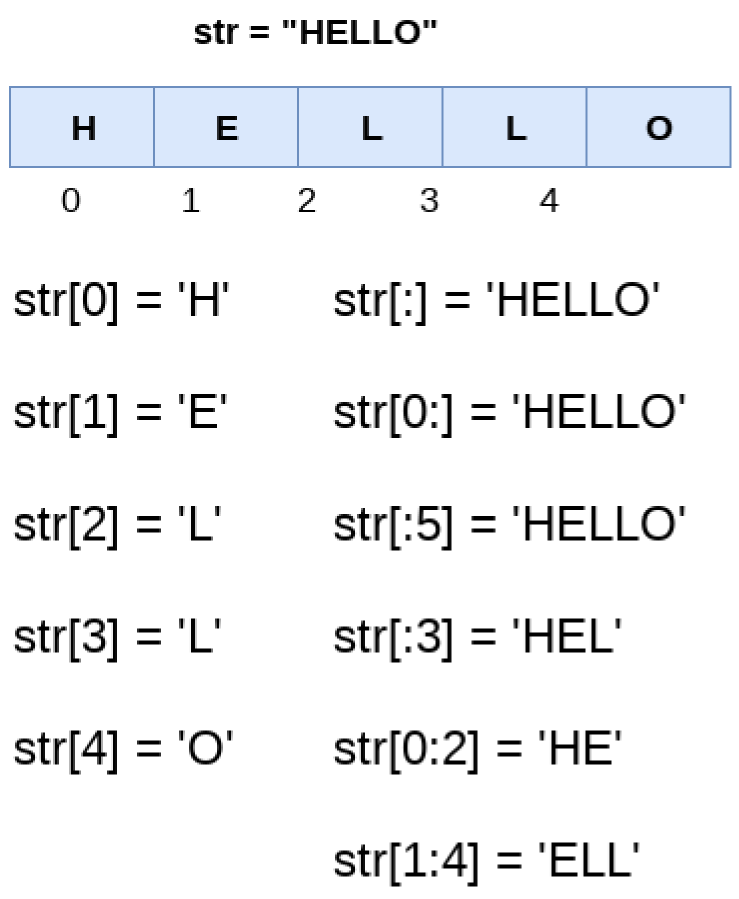
Consider the following example:

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

Output:

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

we can use the [:] operator in Python to access the substring from the given string. Consider the following example.



```
str[0] = 'H'      str[:] = 'HELLO'
str[1] = 'E'      str[0:] = 'HELLO'
str[2] = 'L'      str[:5] = 'HELLO'
str[3] = 'L'      str[:3] = 'HEL'
str[4] = 'O'      str[0:2] = 'HE'
                  str[1:4] = 'ELL'
```

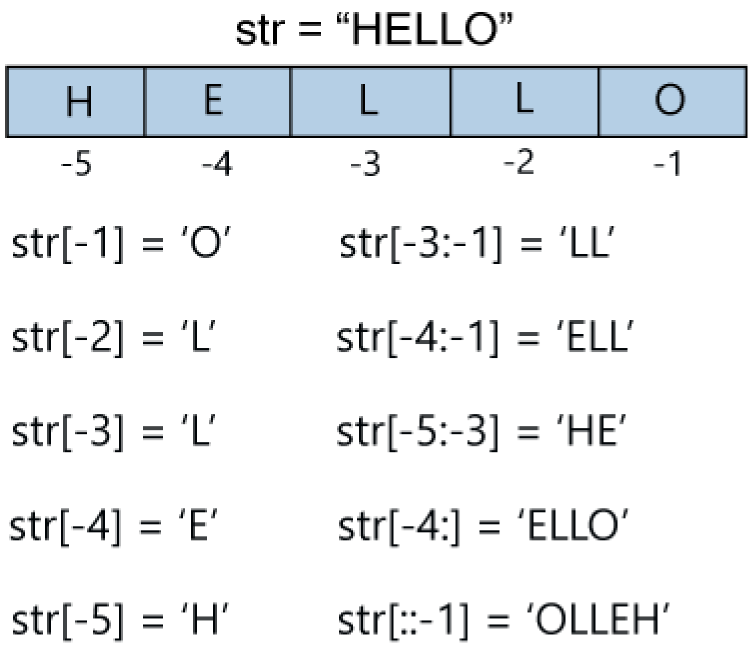
Here, we must notice that the upper range given in the slice operator is always exclusive i.e., if str = 'HELLO' is given, then str[1:3] will always include str[1] = 'E', str[2] = 'L' and nothing else.

Consider the following example:

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

```
print(str[4:7])
```

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



```
str[-1] = 'O'     str[-3:-1] = 'LL'
str[-2] = 'L'     str[-4:-1] = 'ELL'
str[-3] = 'L'     str[-5:-3] = 'HE'
str[-4] = 'E'     str[-4:] = 'ELLO'
str[-5] = 'H'     str[::-1] = 'OLLEH'
```

Consider the following example

```
str = 'amitNavneetPatel'
print(str[-1])
print(str[-3])
print(str[-2:])
print(str[-4:-1])
print(str[-7:-2])
# Reversing the given string
print(str[::-1])
print(str[-12])  #Error Index out of range
```

Reversing Strings Through Slicing

Slicing is a useful technique that allows you to extract items from a given sequence using different combinations of **integer indices** known as offsets. When it comes to slicing strings, these offsets define the index of the first character in the slicing, the index of the character that stops the slicing, and a value that defines how many characters you want to jump through in each iteration.

To slice a string, you can use the following syntax:

Python
```
a_string[start:stop:step]
```

All the offsets are optional, and they have the following default values:

Offset	Default Value
`start`	`0`
`stop`	`len(a_string) or Last Index`
`step`	`1`

Here, `start` represents the index of the first character in the slice, while `stop` holds the index that stops the slicing operation. The third offset, `step`, allows you to decide how many characters the slicing will jump through on each iteration.

Note: A slicing operation finishes when it reaches the index equal to or greater than `stop`. This means that it never includes the item at that index, if any, in the final slice.

The `step` offset allows you to fine-tune how you extract desired characters from a string while skipping others:

Python
```
>>> letters = "AaBbCcDd"

>>> # Get all characters relying on default offsets
>>> letters[::]
'AaBbCcDd'
>>> letters[:]
'AaBbCcDd'

>>> # Get every other character from 0 to the end
>>> letters[::2]
```

```
'ABCD'

>>> # Get every other character from 1 to the end
>>> letters[1::2]
'abcd'
```

Why are slicing and this third offset relevant to reversing strings in Python? The answer lies in how `step` works with **negative values**. If you provide a negative value to `step`, then the slicing runs backward, meaning from right to left.

For example, if you set `step` equal to `-1`, then you can build a slice that retrieves all the characters in reverse order:

Python
```
>>> letters = "ABCDEF"

>>> letters[::-1]
'FEDCBA'
```

This slicing returns all the characters from the right end of the string, where the index is equal to `len(letters) - 1`, back to the left end of the string, where the index is `0`. When you use this trick, you get a copy of the original string in reverse order without affecting the original content of `letters`.

Reassigning Strings

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

s="amit"

s="patel"

print(s)

print(s[0])

s2=""

s2=input("enter University:")

print(s2)

#s[0]='A'

#error item assignment not allowed

Output:

patel

p

enter University:parul

parul

Consider the following example.

Example 1

1. str = "HELLO"
2. str[0] = "h"
3. **print**(str)

Output:

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

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

Example 2

1. str = "HELLO"
2. **print**(str)

3. str = "hello"
4. print(str)

Output:

```
HELLO
hello
```

Deleting the String

As we know that strings are immutable. We cannot delete or remove the character(s) from the string. But we can delete the entire string using the **del** keyword.

1. str = "amit"
2. del str[1]

Output:

```
TypeError: 'str' object doesn't support item deletion
```

Now we are deleting entire string.

1. str1 = "amit"
2. del str1
3. print(str1)

Output:

```
NameError: name 'str1' is not defined
```

String Operators

Operator	Description
+	It is known as concatenation operator used to join the strings given either side of the operator.
*	It is known as repetition operator. It concatenates the multiple copies of the same string.
[]	It is known as slice operator. It is used to access the sub-strings of a particular string.
[:]	It is known as range slice operator. It is used to access the characters from the specified range.
in	It is known as membership operator. It returns if a particular sub-string is present in the specified string.
not in	It is also a membership operator and does the exact reverse of in. It returns true if a particular substring is not present in the specified string.
%	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.

Example

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("The string str : %s"%(str)) # prints The string str : Hello
```

Output:

```
HelloHelloHello
Hello world
```

```
o
ll
False
False
The string str : Hello
```

Python String Formatting

Escape Sequence

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

Example

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

```
str = "They said, "Hello what's going on?""
print(str)
```

Output:

```
SyntaxError: invalid syntax
```

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

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

Example -

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

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

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

Output:

```
They said, "What's there?"
They said, "What's going on?"
They said, "What's going on?"
```

The format() method

The **format()** method is the most flexible and useful method in formatting strings. The curly braces {} are used as the placeholder in the string and replaced by the **format()** method argument. Let's have a look at the given an example:

```
# Using Curly braces
print("{} and {} both are the best friend".format("Devansh","Abhishek"))

#Positional Argument
print("{1} and {0} best players ".format("Virat","Rohit"))

#Keyword Argument
print("{a},{b},{c}".format(a = "James", b = "Peter", c = "Ricky"))
```

Output:

```
Devansh and Abhishek both are the best friend
Rohit and Virat best players
James,Peter,Ricky

print("{a},{b},{c}".format(b = "James", c = "Peter", a = "Ricky"))

Output:
Ricky,James,Peter
```

Python String Formatting Using % Operator

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

Consider the following example.

```
Integer = 10;
Float = 1.290
```

3. String = "Devansh"
4. print("Hi I am Integer ... My value is %d\nHi I am float ... My value is %f\nHi I am string ... My value is %s"%(Integer,Float,String))

Output:

```
Hi I am Integer ... My value is 10
Hi I am float ... My value is 1.290000
Hi I am string ... My value is Devansh
```

Python String functions

Method	Description
capitalize()	It capitalizes the first character of the String. This function is deprecated in python3
casefold()	It returns a version of s suitable for case-less comparisons.
count(string,begin,end)	It counts the number of occurrences of a substring in a String between begin and end index.
endswith(suffix ,begin=0,end =len(string))	It returns a Boolean value if the string terminates with given suffix between begin and end.
find(substring ,beginIndex, endIndex)	It returns the index value of the string where substring is found between begin index and end index.
index(subsring, beginIndex, endIndex)	It throws an exception if string is not found. It works same as find() method.
isalnum()	It returns true if the characters in the string are alphanumeric i.e., alphabets or numbers
isalpha()	It returns true if all the characters are alphabets
isdecimal()	It returns true if all the characters of the string are decimals.
isdigit()	It returns true if all the characters are digits
isidentifier()	It returns true if the string is the valid identifier.
islower()	It returns true if the characters of a string are in lower case, otherwise false.
isnumeric()	It returns true if the string contains only numeric characters.
isprintable()	It returns true if all the characters of s are printable or s is empty, false otherwise.

isupper()	It returns false if characters of a string are in Upper case, otherwise False.
isspace()	It returns true if the characters of a string are white-space, otherwise false.
istitle()	It returns true if the string is titled properly and false otherwise. A title string is the one in which the first character is upper-case whereas the other characters are lower-case. Example: Amit Patel is Title string amit patel is not Title string
isupper()	It returns true if all the characters of the string(if exists) is true otherwise it returns false.
len(string)	It returns the length of a string.
lower()	It converts all the characters of a string to Lower case.
replace(old,new[,count])	It replaces the old sequence of characters with the new sequence. The max characters are replaced if max is given.
rfind(str,beg=0,end=len(str))	It is similar to find but it traverses the string in backward direction.
rindex(str,beg=0,end=len(str))	It is same as index but it traverses the string in backward direction.
split(str,num=string.count(str))	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.
startswith(str,beg=0,end=len(str))	It returns a Boolean value if the string starts with given str between begin and end.
swapcase()	It inverts case of all characters in a string.
title()	It is used to convert the string into the title-case i.e., The string **meEruT** will be converted to Meerut.

upper()	It converts all the characters of a string to Upper Case.