# Chapter 04

# Data Structures

In this chapter, you will learn about

- Integer to String Conversion
- The type Function
- List Indexes
- Slicing List
- Nested List
- Multiple Data Type List Elements
- List Operations
- Tuple
- Tuple Construction
- Tuple Operations
- Set Data Type
- Set Creation
- Set Operations
- Dictionary Data Type
- Dictionary Creation
- Dictionary Operations
- Looping Thru Sequences
- Sequence Comparison
- The range Data Type

## ❖ The type Function

The type function determines data type of the variable or constant.

```
1. print(type(20))
2. print(type(20.2))
3. print(type(c))
4. print(type('Wisen'))
```

**Line 1:** It prints <class 'int'> in the console.

**Line 2:** It prints <class 'float'> in the console.

**Line 3:** It prints <class 'bool'> in the console.

**Line 4:** It prints <class 'str'> in the console.

## ❖ String Conversion

Python allows you convert integer to string. The str function converts the boolean, integer or float data type to string data type.

```
1. x = 30
2. y = 31.33
3. print(x + y)
4. s1 = str(x)
5. s2 = str(y)
6. print(s1 + s2)
7.
8. b1 = False
9. bs1 = str(b1)
10.print(type(bs1))
```

**Line 3:** The expression x + y performs numeric addition. Therefore, it displays 61.33.

**Line 4:** The numeric x value 30 is converted to string data type.

**Line 5:** The numeric y value 31.33 is converted to string data type.

**Line 6:** The expression s1 + s2 performs string concatenation. Therefore, it displays 3031.33.

**Line 9:** The boolean b1 value False is converted to string data type.

**Line 10:** It displays "<class 'str'>" in the console.

## ❖ Python Data Structures

In order to work with information, any application often want to group data together and work on it in that group. In any programming languages, the most fundamental way of grouping data together is to use 'data-structures'. The data structure provides a powerful mechanism to organize and store data. All the data in the sequence data types are live in memory. Therefore, whenever the program is terminated, the data is lost.

Python provides a three powerful built in data structures

1. Sequence Types
    a. Basic Sequence Type
        i. list
        ii. tuple
        iii. range
    b. Text Sequence Type
        i. str
    c. Byte Sequence Type
        i. bytes
        ii. bytearray
        iii. memoryview
2. Set Types
    a. set
    b. frozenset
3. Mapping Types
    a. dict

These types are also called as Compound Data Types or Container Data Types.

The Compound Data Type acts a container to store collection of data grouped into a single variable. Each data that is inside of a list is called an element or item.

In Python, the same syntax and function names are used to perform operations on sequential data types.

## ❖ Homogenous & Heterogeneous Data Structure

### Homogenous Data Structure

A homogenous data structure is a container that only contains elements of the same data type.

data1 = [1,2,3,4,5]

It is homogenous list; since all elements in the data types are same data type integer.

data2 = [1,2,3,4,5, True]

It is homogenous list; since all elements in the data types are same data type integer and boolean. In Python boolean is also considered internally as integer.

data3 = [1,2,3,4,5, ‘6’]

It is NOT homogenous list; since one element 6 is the data type of string.

### Heterogeneous Data Structure

A heterogeneous data structure is a container that only contains elements of different data type.

data1 = [1,2,3,4,5, ‘6’]

It is heterogeneous list; since at least one element 6 is the data type of string.

data2 = [1,2,3,4,5, ‘A’]

It is heterogeneous list; since at least one element A is the data type of string.

data3 = [1,2,3,4,5]

It is NOT heterogeneous list; since all elements are in the same data type integer.

data4 = [1,2,3,4,5, True]

It is NOT heterogeneous list; since all elements are in the same data type integer. In Python boolean is also considered internally as integer.

## ❖ The list Type

List is used to store collection of data's in various data types one after the other. This means that it can contain arbitrary mixture of elements like numbers, strings, and other lists and so on. Lists are related to arrays of other programming languages like C#, Java etc; but Python lists are by far more flexible and powerful than "classical" arrays.

A list in Python is an ordered group of items or elements.

In Python, the important characteristics of lists are:

- Ordered collection of elements or items where the ordering is defined by the creator. In other words, it stores one or more data in specified order.
- Lists are intended to be homogeneous sequences.
- Mutable. This means that the elements in the can be modified.
- List elements can be accessed using index
- List elements can be any data type.
- Lists can contains other list also.
- List cannot be used as keys in Dictionary.

### List Construction

List can be constructed in three ways:

1. Using a pair of square brackets [ ].
2. Using the type constructor: list() or list(iterable)
3. Using a list comprehension. ([a for a in iterable])

### List Construction : Using [ ]

Python allows you to create list by using pair of square brackets. List elements are separated by comma.

Two types of lists you can create using square brackets.

1. Empty List
2. List with Data's

#### List Construction: Using [ ]: Empty List

You can create an Empty List just specifying a pair of square brackets [ ].

#### List Construction: Using [ ]: List with Data's

You can create a List using square brackets separating data's with commas.

```
1. colors = []
2. countries = ['Australia', 'India', "Ireland", "Finland"]
3. print(type(colors))
4. print(countries)
```

**Line 1:** Created an empty list object using [ ]. (**List Construction: Using [ ]: Empty List**)

**Line 2:** Created a list with data using [ ]. (**List Construction: Using [ ]: List with Data's**)

**Line 3:** It displays <class 'list'> in the console.

**Line 4:** It displays ['Australia', 'India', 'Ireland', 'Finland'] in the console.

## List Construction: Using Constructor

Python allows you to create list by calling the constructor. You can create list object in two ways using constructors.

1. Empty List
2. List with Data's

### List Construction: Using Constructor: Empty List

You can create an Empty List by calling constructor.

### List Construction: Using Constructor: List with Data's

You can create a List by calling constructor. You can pass below values to the constructor.

- **A String Value**
- **A Comma Separated values with in the parenthesis ()**

```
1. currencies = list()
2. mylist = list('Wisen')
3. countries = list(('Australia', 'India', "Ireland", "Finland"))
4. print(currencies)
5. print(mylist)
6. print(type(countries))
```

**Line 1:** Created an empty list object using list constructor. (**List Construction: Using Constructor: Empty List**)

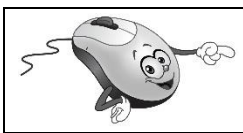
**Line 2:** Created a list with string data using list constructor. (**List Construction: Using [ ]: List with String Data's**)

**Line 3:** Created a list with list data using list constructor. (**List Construction: Using [ ]: List with List Data's**)

**Line 4:** It displays [ ] in the console.

**Line 5:** It displays ['W', 'i', 's', 'e', 'n'] in the console.

**Line 6:** It displays <class 'list'> in the console.

| | |
|---|---|
|  | Integers and Float values are non-iterable. Therefore, you can't pass to list constructor instead of string values. |

## List Construction: Comprehension

Comprehensions are constructs. In other words, Python supports computed list. It can be used to create list object in a very natural, well-known notation, like a mathematician is used to do. List comprehensions is an convenient way when we need to apply an arbitrary mathematic expression to a sequence

The following are common ways to describe lists (or sets, or tuples, or vectors) in mathematics.

$S = \{x^3 : x \in 1..100\}$ (In mathematics the cubes numbers of the natural numbers)

$V = (1, 2, 4, 8, ..., 2^{12})$

M = {x | x in S and x odd}

The general form is

**output-expression** for **variable** in **input-sequence** if **condition**

The **output-expression** indicates the valid python expression.

The **variable** indicates member of the input-sequence.

The **input-sequence** indicates the input list.

The **condition** is indicates the valid Python condition. The condition is optional.

**List Comprehension without Condition Example**

```
1. data = [1,2,3,4,5]
2. cubes = [  e * e * e for e in data ]
3. print(cubes)
```

**Line 2:** The right hand side of the assignment operator contains the list comprehension (e * e * e for e in data). This comprehension is used to find the cube for every element in lists called data and stores in the cubes list object.

**Line 3:** It displays [1, 8, 27, 64, 125] in console.

**List Comprehension with Condition Example**

```
1. data = [1,2,3,4,5]
2. cubes = [  e * e * e for e in data if e % 2 != 0]
3. print(cubes)
```

**Line 2:** The right hand side of the assignment operator contains the list comprehension (e * e * e for e in data if e % 2 != 0). This comprehension is used to find the cube for every odd element in lists called data and stores in the cubes list object.

**Line 3:** It displays [1, 27, 125] in console.

## ❖ List Indexes

Like Strings, List can be indexed. List element can be accessed using index value. The first element index starts with zero. Lists in python are zero indexed. This means, you can access individual elements in a list just as you would do in an array.

In Python, list are mutable.

## ❖ Nested List

Python supports Nested List. This means that you can define a list within another list, such lists are known as Nested List. In other words, a nested list is an element of its enclosing list.

| | |
|---|---|
|  | By default, Python limits 1000 stack frames stacked. So, Python throws exception when the list contains more than 1000 nested list in depth. |

```
1. data = [['A', 'B', 'C'], ['D', 'E'], ['F', 'G', 'H', 'I']]
2. print(data)
3. print(len(data))
4. print(len(data[0]))
```

Line 1: The list contains another list. Therefore, it is called as nested list.

The del Statement

The del statement removes an item from a list at the specified index.

The del statement can also be used to remove slices from a list or clear the entire list.

The general form is

del list[index | slice]

The del Statement

## ❖ List Operations

⇒ **append(item)**

| This method inserts the specified item after the last element of the list. The length of the list itself will increase by one. | | |
|---|---|---|
| **Parameters** | item | Indicates the element to append. |
| **Returns** | None | |
| **Example** | data = [10, 20, 30]<br>data.append(True) # it is equivalent to data[len(data):] = True<br>print(data) # It displays [10, 20, 30, True] in console. | |

⇒ **insert(index, item)**

| This method inserts the specified item into the list at the specified index position. The length of the list itself will increase by one. | | |
|---|---|---|
| **Parameters** | Index | Indicates the element to append. The specified index must be within the range; otherwise Python throws IndexError. |
| | item | Indicates the element to insert. |

| | | |
|---|---|---|
| **Returns** | None | |
| **Example** | data = [10, 20, 30]<br>data.insert(1, False)<br>print(data) # It displays [10, False, 20, 30] in console. | |

⇒ **extend(anotherList)**

| | | |
|---|---|---|
| It appends the specified list to the invoking list. | | |
| **Parameters** | anotherList | Indicates the list to append. |
| **Returns** | None | |
| **Example** | data = [10, 20, 30]<br>bools = [True, False]<br>result = data.extend(bools) # it is equivalent to data[len(data):] = bools<br>print(data) # It displays [10, 20, 30, True, False] in console. | |

⇒ **copy()**

| | | |
|---|---|---|
| It determines the new list object from the invoking list. | | |
| **Returns** | list | It contains the new list object. |
| **Example** | data = [10, 20, 30]<br>newList = data.copy()<br>print(newList) # It displays [10, 20, 30] in console.<br>print(id(data))<br>print(id(newList)) # This id value and previous statement id value will not be same; since both are different object. | |

⇒ **sorted(containerData, key=None, reverse=False)**

| | | |
|---|---|---|
| It is built-in function which can be used to sort the elements in the specified container object. | | |
| **Parameters** | **containerData** | It indicates the iterable object such as tuple, list set etc whose elements has to be sorted. |
| | key | The default value is None.<br>This parameter accepts function or None. If you any other data type value, Python throws TypeError. |
| | reverse | The default value is false.<br>The value False sorts the iterable object in ascending order.<br>The value True sorts the iterable object in descending order. |
| **Returns** | list | It contains the new sorted list. |

- **Sorting Numeric Values in Ascending**

```
1. data = [10, 8, 5, 3, 7, 2, 8]
2. sortedList = sorted(data)
3. print(sortedList)
```

Line 3: It displays “[2, 3, 5, 7, 8, 8, 10]” in the console.

- **Sorting Numeric Values in Descending**

```
1. data = [10, 8, 5, 3, 7, 2, 8, False, True]
2. sortedList = sorted(data, reverse=True)
3. print(sortedList)
```

Line 3: It displays “[10, 8, 8, 7, 5, 3, 2, True, False]” in the console.

- **Sorting String Values in Descending – Case Sensitive**

```
1. data = ["finland", "India", "France", "Germany", "Italy",
   "Australia", "ireland"]
2. countries = sorted(data, reverse=True)
3. print(countries)
```

Line 3: It displays “['ireland', 'finland', 'Italy', 'India', 'Germany', 'France', 'Australia']” in the console.

The ASCII value of small letter i is 105.

The ASCII value of small letter i is 102.

The ASCII value of capital letter I is 73.

Therefore, ireland has come first, then finland then Italy and so on,

- **Sorting String Values in Ascending – Case Sensitive**

```
1. data = ["finland", "India", "France", "Germany", "Italy",
   "Australia", "ireland"]
2. countries = sorted(data)
3. print(countries)
```

**Line 3:** It displays “['Australia', 'France', 'Germany', 'India', 'Italy', 'finland', 'ireland']” in the console.

- **Sorting String Values in Ascending – Case Insensitive**

  To sort in case insensitive way, you have to convert each element to lower case or upper case. To perform this operation, we use the key parameter.

```
1. data = ["finland", "India", "France", "Germany", "Italy",
   "Australia", "ireland"]
2. countries = sorted(data, key=str.lower)
3. print(countries)
```

**Line 2:** For each element in data, the lower method has been called and converted to lower case.

**Line 3:** It displays "['Australia', 'finland', 'France', 'Germany', 'India', 'ireland', 'Italy']" in the console.

### ➢ Sorting Tuple in Ascending - Default

By default, the tuples are sorted by first element of the tuple. If the first element in multiple tuples are same, then it is sorted by second element and so on

```
1. data = [["Mary", 29, 'F'], ["Jones", 26, 'M'], ["Mary", 24, 'F'],
   ["Kim", 32, 'F'],["John", 22, 'M']]
2. students = sorted(data)
3. print(students)
```

**Line 3:** It displays "[['John', 22, 'M'], ['Jones', 26, 'M'], ['Kim', 32, 'F'], ['Mary', 24, 'F'], ['Mary', 29, 'F']]" in the console.

Note: the first element value Mary is available in two tuple elements. Therefore, it is sorted by second element. Therefore, ['Mary', 24, 'F'] appears before ['Mary', 29, 'F']

### ➢ Sorting Tuple in Ascending – Other than First Element

By default, the tuples are sorted by first element of the tuple. If you want to sort by other than first element, then you have to use the key parameter.

```
1. def elementForSort(elem):
2.     print(elem)
3.     return elem[2]
4. data = [["Mary", 'F', 29], ["Jones", 'M', 26],["Kim", 'F',
   32],["John", 'M', 22]]
5. students = sorted(data, key=elementForSort)
6. print(students)
```

**Line 5:** Assign the function elementForSort to key parameter. Therefore,

**Iteration 1:** Python automatically invokes the elementForSort by passing the first element ["Mary", 'F', 29] to elem.

**Line 2:** It displays "["Mary", 'F', 29]" in the console.

**Line 3:** It returns the 29 to the caller.

**Iteration 2:** Python automatically invokes the elementForSort by passing the first element ["Jones", 'M', 26]to elem.

**Line 2:** It displays "["Jones", 'M', 26]" in the console.

**Line 3:** It returns the 26 to the caller.

In the same way, Iteration 3 & 4 performs.

**Line 6:** It displays "[['John', 'M', 22], ['Jones', 'M', 26], ['Mary', 'F', 29], ['Kim', 'F', 32]]" in the console.

- **The sorted function is guaranteed to be stable. This means that when two or more elements have the same key their original order is preserved.**

```
1. def elementForSort(elem):
2.     print(elem)
3.     return elem[0]
4. data = [["Mary", 29, 'F'], ["Jones", 26, 'M'], ["Mary", 24, 'F'],
   ["Kim", 32, 'F'],["John", 22, 'M']]
5. students = sorted(data, key=elementForSort)
6. print(students)
```

**Line 6:** It displays "[['John', 22, 'M'], ['Jones', 26, 'M'], ['Kim', 32, 'F'], ['Mary', 29, 'F'], ['Mary', 24, 'F']]" in the console.

It is sorted by 0th element. There are two list contains Mary as 0th element. Both are same. In the original list ['Mary', 29, 'F'] appeared before ['Mary', 24, 'F']. In the sorted list also list ['Mary', 29, 'F'] appeared before ['Mary', 24, 'F']. Therefore, the original order is preserved. The sorted function is guaranteed to be stable.

⇒ **sort(key=None, reverse=False)**

| It is built-in function which can be used to sort the elements in the invoking list object. | | |
|---|---|---|
| **Parameters** | key | The default value is None.<br>This parameter accepts function or None. If you specify any other data type value, Python throws TypeError. |
| | reverse | The default value is false.<br>The value False sorts the list object in ascending order.<br>The value True sorts the list object in descending order. |
| **Returns** | None | It returns None |

- **Sorting Numeric Values in Ascending**

```
1. data = [10, 8, 5, 3, 7, 2, 8]
2. data.sort()
3. print(data)
```

Line 3: It displays "[2, 3, 5, 7, 8, 8, 10]" in the console.

- **Sorting Numeric Values in Descending**

```
1. data = [10, 8, 5, 3, 7, 2, 8, False, True]
2. data.sort(reverse=True)
3. print(data)
```

Line 3: It displays "[10, 8, 8, 7, 5, 3, 2, True, False]" in the console.

- **Sorting String Values in Descending – Case Sensitive**

```
1. countries = ["finland", "India", "France", "Germany", "Italy",
   "Australia", "ireland"]
2. countries.sort(reverse=True)
3. print(countries)
```

Line 3: It displays “['ireland', 'finland', 'Italy', 'India', 'Germany', 'France', 'Australia']” in the console.

The ASCII value of small letter i is 105.

The ASCII value of small letter i is 102.

The ASCII value of capital letter I is 73.

Therefore, ireland has come first, then finland then Italy and so on,

- **Sorting String Values in Ascending – Case Sensitive**

```
1. countries = ["finland", "India", "France", "Germany", "Italy",
   "Australia", "ireland"]
2. countries.sort()
3. print(countries)
```

**Line 3:** It displays “['Australia', 'France', 'Germany', 'India', 'Italy', 'finland', 'ireland']” in the console.

- **Sorting String Values in Ascending – Case Insensitive**

  To sort in case insensitive way, you have to convert each element to lower case or upper case. To perform this operation, we use the key parameter.

```
1. countries = ["finland", "India", "France", "Germany", "Italy",
   "Australia", "ireland"]
2. countries.sort(key=str.lower)
3. print(countries)
```

**Line 2:** For each element in data, the lower method has been called and converted to lower case.

**Line 3:** It displays “['Australia', 'finland', 'France', 'Germany', 'India', 'ireland', 'Italy']” in the console.

- **Sorting Tuple in Ascending - Default**

  By default, the tuples are sorted by first element of the tuple. If the first element in multiple tuples are same, then it is sorted by second element and so on

```
1. students = [["Mary", 29, 'F'], ["Jones", 26, 'M'], ["Mary", 24,
   'F'], ["Kim", 32, 'F'],["John", 22, 'M']]
2. students.sort()
3. print(students)
```

**Line 3:** It displays "[['John', 22, 'M'], ['Jones', 26, 'M'], ['Kim', 32, 'F'], ['Mary', 24, 'F'], ['Mary', 29, 'F']]" in the console.

Note: the first element value Mary is available in two tuple elements. Therefore, it is sorted by second element. Therefore, ['Mary', 24, 'F'] appears before ['Mary', 29, 'F']

- **Sorting Tuple in Ascending – Other than First Element**

By default, the tuples are sorted by first element of the tuple. If you want to sort by other than first element, then you have to use the key parameter.

```
1. def elementForSort(elem):
2.     print(elem)
3.     return elem[2]
4. students = [["Mary", 'F', 29], ["Jones", 'M', 26],["Kim", 'F',
   32],["John", 'M', 22]]
5. students.sort(key=elementForSort)
6. print(students)
```

**Line 5:** Assign the function elementForSort to key parameter. Therefore,

**Iteration 1:** Python automatically invokes the elementForSort by passing the first element ["Mary", 'F', 29] to elem.

**Line 2:** It displays "["Mary", 'F', 29]" in the console.

**Line 3:** It returns the 29 to the caller.

**Iteration 2:** Python automatically invokes the elementForSort by passing the first element ["Jones", 'M', 26]to elem.

**Line 2:** It displays "["Jones", 'M', 26]" in the console.

**Line 3:** It returns the 26 to the caller.

In the same way, Iteration 3 & 4 performs.

**Line 6:** It displays "[['John', 'M', 22], ['Jones', 'M', 26], ['Mary', 'F', 29], ['Kim', 'F', 32]]" in the console.

- **The sorted function is guaranteed to be stable. This means that when two or more elements have the same key their original order is preserved.**

```
1. def elementForSort(elem):
2.     print(elem)
3.     return elem[0]
4. students = [["Mary", 29, 'F'], ["Jones", 26, 'M'], ["Mary", 24,
   'F'], ["Kim", 32, 'F'],["John", 22, 'M']]
5. students.sort(key=elementForSort)
6. print(students)
```

**Line 6:** It displays "[['John', 22, 'M'], ['Jones', 26, 'M'], ['Kim', 32, 'F'], ['Mary', 29, 'F'], ['Mary', 24, 'F']]" in the console.

It is sorted by $0^{th}$ element. There are two list contains Mary as $0^{th}$ element. Both are same. In the original list ['Mary', 29, 'F'] appeared before ['Mary', 24, 'F']. In the sorted list also list ['Mary', 29, 'F'] appeared before ['Mary', 24, 'F']. Therefore, the original order is preserved. The sorted function is guaranteed to be stable.

|  | The sort function can't sort the heterogeneous list except number & boolean |
|---|---|

⇒ **reverse()**

| It reverse the elements in the invoking list with respect to position. The last element will become first element, the last before element will become second element and so on. | |
|---|---|
| **Example** | currencies = ["USD", "EUR", "INR", "dsd", "AUD", "egp", "nok", "NONE"]<br>currencies.reverse()<br>print(currencies) # it prints ['NONE', 'nok', 'egp', 'AUD', 'dsd', 'INR', 'EUR', 'USD'] |

⇒ **pop([idx])**

| This method determines the element at the specified index position and removes the element from the invoking list. | | |
|---|---|---|
| **Parameters** | idx | Indicate the index position.<br>The parameter is optional. If the index postion is not specified, Python determines the last element and removes from the invoking list object.<br>If the specified index is out of range, Python throws "**IndexError**" with this message "**pop index out of range**". |
| **Returns** | Any | It returns any data type element. |

```
1. currencies = ["USD", "EUR", "INR", "dsd", "AUD"]
2. elem = currencies.pop(2)
3. print(elem)
4. print(currencies)
```

In the above program, Line 2, the pop method determines the element at index position 2 i-e INR and removes from the invoking list object. The INR is assigned to elem.

Line 3: It displays INR in the console.

Line 4: It displays "['USD', 'EUR', 'dsd', 'AUD']" in the console

```
1. currencies = ["USD", "EUR", "INR", "dsd", "AUD"]
2. elem = currencies.pop()
3. print(elem)
4. print(currencies)
```

In the above program, Line 2, the pop method determines the last element (since no index position is specified) i-e AUD and removes from the invoking list object. The AUD is assigned to elem.

Line 3: It displays AUD in the console.

Line 4: It displays “['USD', 'EUR', 'INR', 'dsd']” in the console

⇒ **remove(elem)**

| This method removes first occurrence of the specified element from the invoking list. | | | |
|---|---|---|---|
| **Parameters** | elem | Mandatory | It indicates the element to remove. |
| **Returns** | None | It returns None; if the specified element is available in the invoking object.<br>If the specified element is NOT available in the invoking list object, it throws<br>ValueError: list.remove(x): x not in list | |

```
1. currencies = ["USD", "EUR", "INR", "dsd", "AUD"]
2. elem = currencies.remove("dsd")
3. print(elem)
4. print(currencies)
```

Line2: The remove method deletes the specified element “dsd” in the invoking object. It returns None; since the specified element “dsd” available in the invoking object.

Line 3: It displays “None” in the console.

Line 4: It displays “['USD', 'EUR', 'INR', 'AUD']” in the console.

```
1. currencies = ["USD", "EUR", "INR", "dsd", "AUD"]
2. elem = currencies.remove("acs")
3. print(elem)
4. print(currencies)
```

Line2: The remove method throws the below error; since the specified element “acs” element is not available in the invoking object.

elem = currencies.remove("acs")

ValueError: list.remove(x): x not in list

⇒ **clear()**

| It removes all the elements from the invoking list object. It is equivalent to del a[:]. | | |
|---|---|---|
| **Returns** | None | It returns None. |
| **Example** | currencies = ["USD", "EUR", "INR", "dsd", "AUD"]<br>elem = currencies.clear()<br>print(elem) # It displays None in the console.<br>print(currencies) # It display [] in the console. | |

## ❖ Tuple

A tuple consists of a number of values separated by commas. A tuple is a data structure, which can be used to group and organize data to make it easier to use. A tuple can be used to group any number of elements into a single compound value. The parentheses for the input tuple is optional. Except in the empty tuple case, or when they are needed to avoid syntactic ambiguity.

An output of the tuples are always enclosed in parentheses.

In Python, the important characteristics of tuples are:

- Tuples are much faster than lists.
- Ordered collection of elements or items.
- Tuples are heterogeneous data structures. This means that Tuple elements can be any data type.
- Tuple is an immutable List.
- It can be considered as read-only lists.
- Tuple elements can be accessed using corresponding index number. Therefore you can access the element using index. The index number can be positive or negative.
- Tuple can contains other list, tuple also.
- List cannot be used as keys in Dictionary.
- Allows to create nested tuples.

## ❖ Tuple Construction

Tuples may be constructed in four ways:

1. Constructing an empty tuple: ()
2. One element tuple: a, or (a,)
3. Separating items with commas: a, b, c or (a, b, c)
4. Using the built-in function tuple(): tuple() or tuple(iterable)

### ➢ Constructing an empty tuple

In Python empty tuples (zero element tuple) are constructed using opening and closing parentheses ( ).

```
1. data = ()
2. print(data)
3. print(type(data))
```

**Line 1:** The opening and closing parentheses () creates an empty tuple and assigned to data.

**Line 2:** It displays () in the console.

**Line 3:** It displays <class 'tuple'> in the console.

| | |
|---|---|
|  | In this case, The opening and closing parentheses are mandatory. The comma is not essential. |

### ➢ Constructing one element tuple

In Python one element tuple can be constructed using trailing comma with or without opening and closing parentheses. When you construct tuple without a parentheses, the tuple is inferred.

```
1. data = 'A',
2. print(type(data))
3. data1 = ('A',)
4. print(type(data1))
```

**Line 1:** The trailing comma creates a tuple with one element A and assigned to data.

**Line 2:** It displays <class 'tuple'> in the console.

**Line 4:** The trailing comma within the parentheses creates a tuple with one element A and assigned to data1.

**Line 3:** It displays <class 'tuple'> in the console.

|  | In this case, The training comma are mandatory. The opening and closing parentheses are not essential. |
|---|---|

```
1. data = 'A'
2. print(type(data))
3. data1 = ('A')
4. print(type(data1))
```

In the above code snippet, Line 1 & 2 does not contain the trailing comma after A. Therefore, data & data1 are considered as string data type.

The Line 2 & 4 displays <class 'str'> in the console.

### ➢ Tuple Construction: Comma Separated Items

Tuples may be constructed by separating items with commas with or without parentheses a, b, c or (a, b, c). In this case, the trailing comma (comma after the last element) is optional.

```
1. data = 'A', 'B', 'C'
2. print(type(data))
3. data1 = ('A', 'B', 'C')
4. print(type(data1))
```

**Line 1:** Construct a tuple object, each element separated by comma without parentheses. The trailing comma (comma after the last element) is optional.

**Line 2:** It displays <class 'tuple'> in the console.

**Line 3:** Construct a tuple object, each element separated by comma with parentheses. The trailing comma (comma after the last element) is optional.

**Line 4:** It displays <class 'tuple'> in the console.

### ➢ Tuple Construction: Using Built-in Function tuple()

Tuples may be constructed by invoking the built-in function called tuple().

⇒ **tuple([iterable])**

| It constructs a new tuple object | | | |
|---|---|---|---|
| **Parameters** | iterable | Optional | Indicates an iterable object. The iterable object can be String, List, Set etc.<br>If the argument is omitted, it creates an empty tuple.<br>It can accept only one parameter, which should be in the data type of iterable. |

```
1. data = tuple()
2. print(data)
3. data1 = tuple('ABC')
4. print(data1)
5. data1 = tuple(['A', 'B', 'C', 'D', 'E'])
6. print(data1)
```

**Line 1:** Construct an empty tuple object.

**Line 2:** It displays () in the console.

**Line 3:** Construct a tuple object, from an iterable string object.

**Line 4:** It displays ('A', 'B', 'C') in the console.

**Line 5:** Construct a tuple object, from an iterable list object.

**Line 6:** It displays ('A', 'B', 'C', 'D', 'E') in the console.

## ❖ The range Data Type

In Python, the important characteristics of lists are:

- ✍ Immutable. This means that the elements in the can NOT be modified.
- ✍ It returns only integer values.
- ✍ It is an iterable Object.

⇒ **range(start, stop[, step])**

| It generates a sequence of integer values. The data type of generated list is range. It does not return a list object. Rather it acts like a list.<br>All parameters must be integers. All parameters can be positive or negative. | | | |
|---|---|---|---|
| **Parameters** | start | Mandatory | Indicates starting number in the sequence. |
| | stop | Optional | Indicates where the list ends. It is exclusive. |
| | step | Optional | Indicates the difference between each number in the sequence. The default step is 1. |

| **Returns** | Range or Error | It returns the range object. If the step value is **zero ( 0 )** it **throws the ValueError**. |
|---|---|---|

## ➢ range(n)

It generates a sequence of numbers starts from 0 to n-1. If n is less than or equal to 0, it returns empty range object.

```
1. data = range(5)
2. print(data)
3. print(type(data))
4. for val in data:
5.     print(val)
6. data = range(0)
7. print(data)
8. print(len(data))
```

**Line 1:** It creates a range object which contains the integer value from 0 to 4.

**Line 2:** It displays “range(0, 5)” in the console.

**Line 3:** It displays “<class 'range'>” in the console.

**Line 4 to 5:** It loops thru the range object and prints “0 1 2 3 4” in the console line by line.

**Line 6:** It creates a range object which contains zero ( 0 ) elements.

**Line 7:** It prints “range(0, 0)” in the console.

**Line 8:** It prints “0” in the console.

## ➢ range(start, end)

It generates a sequence of numbers starts from start to end-1. If start is greater than or equal to end, it returns empty range object.

```
1. data = range(1, 5)
2. print(data)
3. for val in data:
4.     print(val)
5. data = range(1, -5)
6. print(data)
7. print(len(data))
8. data = range(-1, -5)
9. print(data)
10.print(len(data))
11.data = range(-1, 5)
12.print(data)
13.print(len(data))
14.data = range(-5, -1)
15.for val in data:
16.    print(val)
```

**Line 1:** It creates a range object which contains the integer value from 1 to 4.

**Line 2:** It displays “range(1, 5)” in the console.

**Line 3 to 4:** It loops thru the range object and prints “1 2 3 4” in the console line by line.

**Line 5:** It creates a range object which contains zero ( 0 ) elements; since start index is greater than end index.

**Line 6:** It prints “range(1, -5)” in the console.

**Line 7:** It prints “0” in the console.

**Line 8:** It creates a range object which contains zero ( 0 ) elements; since start index is greater than end index.

**Line 9:** It prints “range(-1, -5)” in the console.

**Line 10:** It prints “0” in the console.

**Line 11:** It creates a range object which contains the integer value from -1 to 4.

**Line 12:** It prints “range(-1, 5)” in the console.

**Line 13:** It prints “6” in the console.

**Line 14:** It creates a range object which contains the integer value from -5 to -2.

**Line 15 to 16:** It loops thru the range object and prints “-5 -4 -3 -2” in the console line by line.

## ➢ range(start, end, step)

It generates a sequence of numbers starts from start to end-1 with the specified increment. If step is 0, Python generates ValueError. If start is less than or equal to end and step is negative, it returns empty range object. If start is greater than or equal to end and step is positive, it returns empty range object.

If the step is greater than or equal to end – start, it returns only the start element.

```
1. data = range(1, 5, 2)
2. print(list(data))
3. data = range(1, 5, -2)
4. print(list(data))
5. data = range(10, 5, 2)
6. print(list(data))
7. data = range(1, 5, 5)
8. print(list(data))
9. data = range(10, 5, -2)
10.print(list(data))
11.data = range(-5, -10, -2)
12.print(list(data))
13.data = range(-5, -10, 0)
14.print(list(data))
```

**Line 1:** It creates a range object which contains the integer value from 1 to 4 with increment of 2.

**Line 2:** A list object is created by passing the range iterable object data. It prints “[1, 3]” in the console.

**Line 3:** It creates a range object which contains zero elements. Since, start less than end and step is negative.

**Line 4:** A list object is created by passing the range iterable object data. It prints “[]” in the console.

**Line 5:** It creates a range object which contains zero elements. Since, start greater than end and step is positive.

**Line 6:** A list object is created by passing the range iterable object data. It prints “[]” in the console.

**Line 7:** It creates a range object which contains the integer value 1. Since, the step is greater than end – start.

**Line 8:** A list object is created by passing the range iterable object data. It prints “[1]” in the console.

**Line 9:** It creates a range object which contains the integer value from 10 to 6 with increment of -2.

**Line 10:** A list object is created by passing the range iterable object data. It prints “[10, 8, 6]” in the console.

**Line 11:** It creates a range object which contains the integer value from -5 to -9 with increment of -2.

**Line 12:** A list object is created by passing the range iterable object data. It prints “[-5, -7, -9]” in the console.

**Line 13:** It throws ‘ValueError: range() arg 3 must not be zero”. Since step is 0.

## ❖ Set

Python supports a data type called sets. Python support two types of built-in set types:

1. set
2. frozenset

In Python, the important characteristics of sets and frozenset are:

- Unordered collection. This means that Python does not maintain element position or order of insertion.
- Set does not support duplicate elements or items
- Set elements must be hashable.
- Set supports mathematical set operations like Union, Intersection, Difference, Symmetric Difference etc

In Python, the important characteristics of sets are:

- Sets are mutable. This means that elements can be changed.
- Set can NOT be used as dictionary key.
- It can NOT be an element of another set. But, it can be an element in list and tuple.

| | |
|---|---|
|  | Set can NOT be nested. This means that Nested Set is not possible. |

In Python, the important characteristics of frozenset are:

- The frozenset are immutable. This means that elements can NOT be changed after it is created.
- The frozen set object can be used as dictionary key.
- The frozen set object can be an element of another set.
- The frozenset object element must be hashable type.

## ❖ Constructing Set Object

Set can be created in three ways:

1. Using Curly Braces { }
2. Using set() Function
3. Set Comprehensions

### ➢ Set Creation: Using { }

Python allows you to create a non-empty sets using { }.

```
1. data = {'A', 'B', 'C', 'A', 'E', 'B'}
2. print(data)
3. print(type(data))
```

**Line 1:** The set object is constructed by using { }.

**Line 2:** It displays {'B', 'C', 'E', 'A'} in the console. Duplicates elements are removed. The order of elements are not same. Therefore set objects are unordered elements.

**Line 3:** It displays <class 'set'> in the console.

| | |
|---|---|
|  | Python does NOT allow to create empty sets using { } |

## ➢ Set Creation : Using set Function

Python allows you create sets using built in set function.

⇒ **set([iterable])**

| It constructs a new empty or non-empty set object of an unordered collection with no duplicate elements. | | | |
|---|---|---|---|
| **Parameters** | iterable | Optional | Indicates an iterable object. The iterable object can be String, List, Set, Tuple etc.<br>If the argument is omitted, it creates an empty set.<br>It can accept only one parameter, which should be in the data type of iterable. |

```
1. data = set()
2. print(data)
3. data1 = set('ABC')
4. print(data1)
5. data1 = set(['A', 'B', 'A', 'D', 'A'])
6. print(data1)
```

**Line 1:** Construct an empty set object.

**Line 2:** It displays set() in the console.

**Line 3:** Construct a set object, from an iterable string object.

**Line 4:** It displays {'A', 'B', 'C'} in the console.

**Line 5:** Construct a set object, from an iterable list object.

**Line 6:** It displays {'A', 'D', 'B'} in the console.

## ➢ Set Creation : Using set Comprehensions

Like List, Python allows you create sets using set Comprehensions.

```
1. data = [1,2,3,4,5]
2. cubes = {  e * e * e for e in data if e % 2 != 0}
3. print(cubes)
4. print(type(cubes))
```

**Line 2:** The right hand side of the assignment operator contains the set comprehension (e * e * e for e in data if e % 2 != 0). This comprehension is used to find the cube for every odd element in lists called data and stores in the cubes set object.

**Line 3:** It displays {1, 27, 125} in console.

**Line 4:** It displays <class 'set'> in console.

# ❖ Constructing Frozen Set Object

Frozen Set object can be created in one way:

1. Using fronzenset() Function

## ➢ Frozen Set Creation : Using frozenset Function

Python allows you create sets using built in set function.

⇒ **frozenset([iterable])**

| It constructs a new empty or non-empty frozen set object of an unordered collection with no duplicate elements. It cannot be modifiable. | | | |
|---|---|---|---|
| **Parameters** | iterable | Optional | Indicates an iterable object. The iterable object can be String, List, Set, Tuple etc.<br>If the argument is omitted, it creates an empty frozen set.<br>It can accept only one parameter, which should be in the data type of iterable. |

```
1. data = frozenset()
2. print(data)
3. data1 = frozenset('ABC')
4. print(data1)
5. data1 = set(['A', 'B', 'A', 'Wisen', 'A'])
6. print(data1)
```

**Line 1:** Construct an empty frozen set object.

**Line 2:** It displays frozenset() in the console.

**Line 3:** Construct a frozenset object, from an iterable string object.

**Line 4:** It displays frozenset({'A', 'C', 'B'}) in the console.

**Line 5:** Construct a frozenset object, from an iterable list object.

**Line 6:** It displays frozenset({'A', 'Wisen', 'B'}) in the console.

## ❖ Mutable & Immutable Types

The id function is used to check if a type is mutable or not.

### Mutable Types

For mutable objects, its content can be altered without changing their identity.

### List of mutable types:

list, dict, set, bytearray, user-defined classes

```
1. data1 = [1, 2, 3, 4]
2. data1[2] = 99
3. print(data1)
4. data2 = {1, 2, 3, 4}
5. data2.add(99)
6. print(data2)
```

**Line 2:** The list object data1 is mutable. Therefore, you can modify the element in index position 2.

**Line 4:** The set object data2 is mutable. Therefore, you can add the new element in the set object.

### Immutable Types

For immutable objects, when its content is changed, Python creates a new object. Therefore, it identity will be changed.

**List of immutable types:**

int, float, decimal, complex, bool, string, tuple, range, frozenset, bytes

```
1. a = 10
2. print(id(a))
3. a += 5
4. print(id(a))
5. s1 = "Wis"
6. print(id(s1))
7. s1 += "en"
8. print(id(s1))
9. data1 = (1, 2, 3, 4)
10.data1[2] = 99
```

**Line 1:** Defined a variable called a and assign the value 10.

**Line 2:** It displays the identifier of the variable a. i-e 1623635296

**Line 3:** Modify the existing variable value. Therefore Python assign a new memory location for variable a.

**Line 4:** It displays the identifier of the variable a. i-e 1623635456. Note that Line 2 and Line 4 values are not same. We are not able to modify the value in the existing identity. Therefore numeric data types are immutable type.

**Line 5:** Defined a variable called s1 and assign the value Wis.

**Line 6:** It displays the identifier of the variable a. i-e 2249971902312

**Line 7:** Modify the existing variable value. Therefore Python assign a new memory location for variable s1.

**Line 8:** It displays the identifier of the variable a. i-e 2250002966600. Note that Line 6 and Line 8 values are not same. We are not able to modify the value in the existing identity. Therefore string data type is immutable type.

**Line 10:** It throws TypeError: 'tuple' object does not support item assignment; since tuple type is immutable type. Therefore you can't modify the element in the tuple.

Note: The identifier's value may be different in your computer.

## ❖ Hashable Type

An object is hashable if and only is

- It has a hash value which never changes during its lifetime; therefore it needs a hash() method.
- It can be compared to other objects; therefore it needs an eq() method. Hashable objects which compare equal must have the same hash value.

Only Hashable object can be used as a dictionary key and a set member, because these data structures use the hash value internally.

### Hashable Types

- The microscopic immutable types are all hashable, such as bool, int, float, str, bytes.
- A frozen set is always hashable(its elements must be hashable by definition)
- A tuple is hashable only if all its elements are hashable
- User-defined types are hashable by default because their hash value is their id()

### Non Hashable Types

- The list & set data's are non hashable type.
- If the tuple contains non hashable element, then that tuple is non hashable type.

For user-defined classes, it is not necessarily the case that immutable==hashable, because a class may have effectively immutable objects and just fail to implement a hash.

Hashability and immutability refer to object instances, not data type. For example, an object of type tuple can be hashable or not.

## ❖ Set & Frozenset Operations

Sets supports below mathematical operations:

- Union,
- Intersection,

- Difference,
- Symmetric Difference and more.

## ➢ Set Operations: Union (Applicable to set & frozenset instance)

Sets supports union operation. It implemented using Pipe | Symbol.

⇒ **union(*others)**

⇒ **set | other | ...**

| It determines a new set object using union operator from the specified set. This means that it determines a new set object with elements from the set object and all other set objects. | | |
|---|---|---|
| **Parameters** | *others | Indicates the variable length parameters of set object for the union operation. |
| | set | Indicates the first set object for the union operation. |
| | other \| … | Indicates the second set object for the union operation and so on |
| **Returns** | set | It returns the new set object union of two or more set object. |

```
1. data1 = {'A', 'B', 'C', 'D', 'A'}
2. data2 = {1, 2, 3, 4}
3. data3 = {True, False}
4. result = data1.union(data2, data3)
5. print(result)
6. result1 = data1 | data2 | data3
7. print(result1)
```

**Line 4:** It returns the new set object union of data1, data2 & data3 set object and assigns to result. This operation is performed by invoking the union method.

**Line 5:** It displays “{False, 1, 2, 3, 4, 'C', 'B', 'D', 'A'}” in the console. Set does not support duplicate; therefore, one A is removed from the return set object. In Python True is considered as 1. There, True is removed from the return set object.

Note: In your computer you may get the output in different order; since set is unordered collection of elements.

**Line 4:** It returns the new set object union of data1, data2 & data3 set object and assigns to result. This operation is performed using the pipe | symbol.

**Line 5:** It displays “{False, 1, 2, 3, 4, 'C', 'B', 'D', 'A'}” in the console.

Note: In your computer you may get the output in different order; since set is unordered collection of elements.

## ➢ Set Operations: Intersection (Applicable to set & frozenset instance)

Sets supports intersection operation. It implemented using Ampersand & Symbol.

⇒ **intersection(*others)**

⇒ **set & other & ...**

| It determines a new set object using intersection operator from the specified set. In other words, the returned set object contains all elements that are members of all set objects. | | |
|---|---|---|
| **Parameters** | *others | Indicates the variable length parameters of set object for the intersection operation. |
| | set | Indicates the first set object for the intersection operation. |
| | other \| … | Indicates the second set object for the intersection operation and so on |
| **Returns** | set | It returns the new set object intersects of two or more set object. |

```
1. data1 = {'A', 'B', 'C', 'D', False}
2. data2 = {1, 2, 3, 4, 'A', 'B', False}
3. data3 = {True, False, 1, 2, 'A', 'B', 'C'}
4. result = data1.intersection(data2, data3)
5. print(result)
6. result1 = data1 & data2 & data3
7. print(result1)
```

**Line 4:** It returns the new set object intersect of data1, data2 & data3 set object and assigns to result. This operation is performed by invoking the intersection method.

**Line 5:** It displays "{False, 'A', 'B'}" in the console.

Note: In your computer you may get the output in different order; since set is unordered collection of elements.

**Line 4:** It returns the new set object intersect of data1, data2 & data3 set object and assigns to result. This operation is performed using the Ampersand & symbol.

**Line 5:** It displays "{False, 'A', 'B'}" in the console.

Note: In your computer you may get the output in different order; since set is unordered collection of elements.

## ➢ Set Operations: Difference (Applicable to set & frozenset instance)

Sets supports difference operation. It implemented using minus - symbol.

⇒ **difference(*others)**

⇒ **set - other - ...**

| It determines a new set object using difference operator from the specified set. In other words, the returned set object contains all elements in the invoking set object that are NOT members in the specified set others. | | |
|---|---|---|
| **Parameters** | *others | Indicates the variable length parameters of set object for the difference operation. |
| | set | Indicates the first set object for the difference operation. |
| | other \| … | Indicates the second set object for the difference operation and so on |
| **Returns** | set | It returns the new set object difference of two or more set object. |

```
1. data1 = {'A', 'B', 'C', 'D', 'E', False}
2. data2 = {1, 2, 3, 4, 'A', 'B', False}
3. data3 = {True, False, 1, 2, 'A', 'B', 'C'}
4. diff = data1.difference(data2, data3)
5. print(diff)
6. diff = data1 - data2 - data3
7. print(diff)
```

**Line 4:** It returns the new set object difference of data1, data2 & data3 set object and assigns to result. This operation is performed by invoking the difference method. It contains all elements in the invoking set object that are NOT members in the specified set others. The element D and E is available in data1 but both are not available in data2 & data3.

**Line 5:** It displays "{'E', 'D'}" in the console.

**Line 6:** It returns the new set object intersect of data1, data2 & data3 set object and assigns to result. This operation is performed using the minus - symbol.

**Line 7:** It displays "{'E', 'D'}" in the console.

Note: In your computer you may get the output in different order; since set is unordered collection of elements.

## ➢ Set Operations: Symmetric Difference (Applicable to set & frozenset instance)

Sets supports symmetric difference operation. It implemented using Caret ^ operator.

⇒ **symmetric_difference (other)**

⇒ **set ^ other**

| It determines a new set object using symmetric difference operator from the specified set. In other words, the returned set object contains all those elements that are members either to set object or to other set object but not to both. | | |
|---|---|---|
| **Parameters** | set | Indicates the first set object for the symmetric difference operation. |
| | other | Indicates the second set object for the symmetric difference operation |
| **Returns** | set | It returns the new set object symmetric difference of two set object. |

```
1. data1 = {'A', 'B', 'C', 'D', 'E', False}
2. data2 = {1, 2, 3, 4, 'A', 'B', False}
3. sdiff = data1.symmetric_difference(data2)
4. print(sdiff)
5. sdiff = data1 ^ data2
6. print(sdiff)
```

**Line 4:** It returns the new set object symmetric difference of data1 & data2 set object and assigns to result. This operation is performed by invoking the symmetric_difference method.

**Line 5:** It displays "{1, 2, 3, 4, 'D', 'E', 'C'}" in the console.

Note: In your computer you may get the output in different order; since set is unordered collection of elements.

**Line 4:** It returns the new set object symmetric difference of data1 & data2 set object and assigns to result. This operation is performed using the caret ^ symbol.

**Line 5:** It displays "{1, 2, 3, 4, 'D', 'E', 'C'}" in the console.

Note: In your computer you may get the output in different order; since set is unordered collection of elements.

## ➢ Set Operations: Is Subset (Applicable to set & frozenset instance)

⇒ **issubset(other)**

⇒ **set <= other**

| It determines TRUE if every element in the set object is available in the other set object; otherwise false. | | |
|---|---|---|
| **Parameters** | set | Indicates the first set object. |
| | other | Indicates the second set object. |
| **Returns** | boolean | It returns true if set object is subset of other set object; otherwise false |

```
1. data1 = frozenset(['A', 'B', 'C', False])
2. data2 = frozenset(['A', 'B', False])
3. isSubSet = data1.issubset(data2)
4. print(data1, " <= ", data2, isSubSet)
5. data1 = {'A', 'B', 'C', False}
6. data2 = {'A', 'B', 'C', False}
7. isSubSet = data1 <= data2
8. print(data1, " <= ", data2, isSubSet)
9. data1 = {'A', 'B', 'C', False}
10.data2 = {'A', 'B', 'C', 'D', False}
11.isSubSet = data1.issubset(data2)
12.print(data1, " <= ", data2, isSubSet)
```

**Line 3:** The issubset method determines False; since element C is not available in data2 set object.

**Line 4:** It displays "frozenset({False, True, 'A', 'B'}) <= frozenset({False, 'A', 'C', 'B'}) False" in the console.

**Line 7:** The issubset symbol <= determines True; since all elements on data1 object is available in data2 set object.

**Line 8:** It displays "{'A', 'B', 'C', False} <= {'A', 'B', 'C', False} True" in the console.

**Line 11:** The issubset method determines True; since all elements on data1 object is available in data2 set object.

**Line 12:** It displays "{'A', 'B', 'C', False} <= {'A', False, 'B', 'D', 'C'} True" in the console.

## ➢ Set Operations: Is Proper Subset (Applicable to set & frozenset instance)

⇒ **set < other**

| It determines TRUE if every element in the set object is available in the other set object and set object should not equal to other set object; otherwise false. | | |
|---|---|---|
| **Parameters** | set | Indicates the first set object. |
| | other | Indicates the second set object. |
| **Returns** | boolean | It returns true if set object is proper subset of other set object; otherwise false |

```
1. data1 = frozenset(['A', 'B', 'C', False])
2. data2 = frozenset(['A', 'B', 'C', False])
3. isProperSubSet = data1 < data2
4. print(data1, " < ", data2, isProperSubSet)
5. data1 = {'A', 'B', 'C', False}
6. data2 = {'A', 'B', 'C', 'D', False}
7. isProperSubSet = data1 < data2
8. print(data1, " < ", data2, isProperSubSet)
```

**Line 3:** All elements in data1 is available in data2; but data1 elements == data2 elements. Therefore the proper sub set symbol < determines False.

**Line 4:** It displays "frozenset({False, 'A', 'C', 'B'}) < frozenset({False, 'A', 'C', 'B'}) False" in the console.

**Line 7:** All elements in data1 is available in data2; but data1 elements != data2 elements. Therefore the proper sub set symbol < determines True.

**Line 8:** It displays "{'A', False, 'C', 'B'} < {False, 'B', 'A', 'D', 'C'} True" in the console.

## ➢ Set Operations: Is Superset (Applicable to set & frozenset instance)

⇒ **issuperset(other)**

⇒ **set >= other**

| It determines TRUE if every element in the other set object is available in the set object; otherwise false. | | |
|---|---|---|
| **Parameters** | set | Indicates the first set object. |
| | other | Indicates the second set object. |
| **Returns** | boolean | It returns true if set object is superset of other set object; otherwise false |

```
1. data1 = frozenset(['A', 'B',  False])
2. data2 = frozenset(['A', 'B', 'C', False])
3. isSuperSet = data1.issuperset(data2)
4. print(data1, " >= ", data2, isSuperSet)
5. data1 = {'A', 'B', 'C', False}
6. data2 = {'A', 'B', 'C', False}
7. isSuperSet = data1 >= data2
8. print(data1, " >= ", data2, isSuperSet)
9. data1 = {'A', 'B', 'C', 'D', False}
10.data2 = {'A', 'B', 'C',  False}
11.isSuperSet = data1.issuperset(data2)
12.print(data1, " >= ", data2, isSuperSet)
```

**Line 3:** The issuperset method determines False; since element C is not available in data1 set object.

**Line 4:** It displays “frozenset({False, 'B', 'A'}) >= frozenset({False, 'B', 'A', 'C'}) False” in the console.

**Line 7:** The issuperset symbol >= determines True; since all elements on data2 object is available in data1 set object.

**Line 8:** It displays “{False, 'B', 'A', 'C'} >= {False, 'B', 'A', 'C'} True” in the console.

**Line 11:** The issuperset method determines True; since all elements on data2 object is available in data1 set object.

**Line 12:** It displays “{False, 'D', 'A', 'B', 'C'} >= {False, 'B', 'A', 'C'} True” in the console.

## ➢ Set Operations: Is Proper Superset (Applicable to set & frozenset instance)

⇒ **set > other**

| It determines TRUE if every element in the other set object is available in the set object and set object should not equal to other set object; otherwise false. | | |
|---|---|---|
| **Parameters** | set | Indicates the first set object. |
| | other | Indicates the second set object. |
| **Returns** | boolean | It returns true if set object is proper superset of other set object; otherwise false |

```
1. data1 = frozenset(['A', 'B', 'C', False])
2. data2 = frozenset(['A', 'B', 'C', False])
3. isProperSuperSet = data1 > data2
4. print(data1, " > ", data2, isProperSuperSet)
5. data1 = {'A', 'B', 'C', 'D', False}
6. data2 = {'A', 'B', 'C', False}
7. isProperSuperSet = data1 > data2
8. print(data1, " > ", data2, isProperSuperSet)
```

**Line 3:** All elements in data2 is available in data1; but data1 elements == data2 elements. Therefore the proper super set symbol > determines False.

**Line 4:** It displays “frozenset({False, 'B', 'A', 'C'}) > frozenset({False, 'B', 'A', 'C'}) False” in the console.

**Line 7:** All elements in data2 is available in data1; but data1 elements != data2 elements. Therefore the proper super set symbol > determines True.

**Line 8:** It displays “{False, 'D', 'A', 'B', 'C'} > {False, 'B', 'A', 'C'} True” in the console.

## ➢ Set Operations: Is Disjoint (Applicable to set & frozenset instance)

⇒ **isdisjoint(other)**

| It determines TRUE if the invoking set object has no elements in common with the specified set object; otherwise false. | | |
|---|---|---|
| **Parameters** | other | Indicates the set object. |
| **Returns** | boolean | It returns true if invoking set object is disjoint with other set object; otherwise false. |

```
1. data1 = frozenset(['A', 'B',  False])
2. data2 = frozenset([1, 2, 3, True])
3. result = data1.isdisjoint(data2)
4. print(data1, " disjoint ", data2, result)
5. data1 = {'A', 'B', 'C', False}
6. data2 = {1, 2, 'C', True}
7. result = data1.isdisjoint(data2)
8. print(data1, " disjoint ", data2, result)
```

**Line 3:** The data1 set has no common elements with data2. Therefore the isdisjoint determines True.

**Line 4:** It displays “frozenset({False, 'A', 'B'}) disjoint frozenset({1, 2, 3}) True” in the console.

**Line 7:** The data1 set has one common element ‘C’ with data2. Therefore the isdisjoint determines False.

**Line 8:** It displays “{False, 'A', 'B', 'C'} disjoint {1, 2, 'C'} False” in the console.

The below methods are available only to the Set object; since it is mutable object. They are not available to the frozenset object; since it is not immutable object.

## ➢ Set Operations: update (Applicable to ONLY set instance)

⇒ **update(*others)**

⇒ **set |= other1 | other2 | ….**

| It adds the elements in the invoking set object from the specified set object. | | |
|---|---|---|
| **Parameters** | others | Indicates the iterable object. |
| | other1, other2 … | Indicates **ONLY** the set object. **Not even** frozen set object. |

| | | |
|---|---|---|
| **Returns** | None | It returns None value. |

```
1. data1 = {1, 2, 3, 4, 'C'}
2. result = data1.update('A', 'B')
3. print(result)
4. print(data1)
5. data1 = {1, 2, 3, 4, 'C'}
6. data2 = {'A', 'B', 1, 2, 3}
7. data3 = set("Wisen")
8. data1 |= data2 | data3
9. print(data1)
```

**Line 2:** The update method appends the specified variable parameter values A & B are added to the invoking set object data1. This method returns None and assigns to the result.

**Line 3:** It displays None in console.

**Line 4:** It displays {1, 2, 3, 4, 'C', 'A', 'B'} in console.

**Line 7:** It creates a set object from the iterable string object Wisen and assign the created set object to data3.

**Line 8:** The update symbol | appends the elements in data2 and data3 to data1 set object.

**Line 9:** It displays "{1, 2, 3, 4, 'C', 'A', 's', 'W', 'e', 'n', 'B', 'i'}" in the console.

## ➢ Set Operations: intersection_update (Applicable to ONLY set instance)

⇒ **intersection_update(*others)**

⇒ **set &= other1 & other2 & ….**

| It modifies the invoking set object, by keeping only the elements found in the invoking set object and all other specified set object. | | |
|---|---|---|
| **Parameters** | others | Indicates the iterable object. |
| | other1, other2 … | Indicates **ONLY** the set object. **Not even** frozen set object. |
| **Returns** | None | It returns None value. |

```
1. data1 = {1, 2, '3', '4', 'A', 'B', 'C'}
2. result = data1.intersection_update('AB345E')
3. print(result)
4. print(data1)
5. data1 = {'W', 'I', 3, 4, 'C'}
6. data2 = {'A', 'B', 'W', 's', 3}
7. data3 = set("Wisen")
8. data1 &= data2 & data3
9. print(data1)
```

**Line 2:** The intersection_update method modifies the invoking set object data1 by keeping the elements found in data1 and the iterable object AB34. The elements which is found in both objects are A B 3 4. This method returns None and assigns to the result variable.

**Line 3:** It displays None in console.

**Line 4:** It displays {'3', '4', 'A', 'B'} in console.

**Line 8:** The intersection_update symbol & find the elements common in data1 and data2 and data3 and store it in data1. The elements common to three objects are W

**Line 9:** It displays "{'W'}" in the console.

## ➢ Set Operations: difference_update (Applicable to ONLY set instance)

⇒ **difference_update (*others)**

⇒ **set -= other1 | other2 | ….**

| It removes the elements in the invoking set object which are found in the specified set. | | |
|---|---|---|
| **Parameters** | others | Indicates the iterable object. |
| | other1, other2 … | Indicates **ONLY** the set object. **Not even** frozen set object. |
| **Returns** | None | It returns None value. |

```
1. data1 = {1, 2, 3, 4, 5}
2. result = data1.difference_update({1, 2, 3})
3. print(result)
4. print(data1)
5. data1 = {'1', '2', '3', '4', '5', '6', '7', '8', '9'}
6. data2 = {'1', '2', '3'}
7. data3 = set("67")
8. data4 = set("89")
9. data1 -= data2 | data3 | data4
10.print(data1)
```

**Line 2:** The difference_update removes all the elements in the data1 object which are found in {1, 2, 3} set object. Therefore, elements 1, 2, 3 are removed from data1 object. This method returns None and assigns to the result.

**Line 3:** It displays None in console.

**Line 4:** It displays {4, 5} in console.

**Line 8:** The difference_update symbol removes all the elements in the data1 object when are found in data2, data3, and data4. Therefore, elements '1', '2', '3', '6', '7', '8', '9' are removed from the data1 set object.

**Line 9:** It displays "{'5', '4'}" in the console.

## ➢ Set Operations: symmetric_difference_update (Applicable to ONLY set instance)

⇒ **symmetric_difference_update (other)**

⇒ **set ^= other1**

| It updates the invoking set object keeping the elements which are found in either invoking set or specified set; but not in both. | | |
|---|---|---|
| **Parameters** | other | Indicates the iterable object. |
| | other1 | Indicates **ONLY** the set object. **Not even** frozen set object. |
| **Returns** | None | It returns None value. |

```
1. data1 = {1, 2, 3, 4, 5}
2. result = data1.symmetric_difference_update({1, 2, 3, 8, 9})
3. print(result)
4. print(data1)
5. data1 = {'1', '2', '3', '4', '5'}
6. data2 = {'1', '2', '3', 'A', "Wisen"}
7. data1 ^= data2
8. print(data1)
```

**Line 2:** The symmetric_difference_update updates the data1 object keeping the elements which are found in data1 set object or {1, 2, 3, 8, 9} set object; but not in both. The elements which are not in both are 4, 5, 8, 9. This method returns None and assigns to the result.

**Line 4:** It displays {4, 5, 8, 9} in console.

**Line 8:** The symmetric_difference_update symbol updates the data1 object keeping the elements which are found in data1 set object or other set object; but not in both. The elements which are not in both are 4, 5, ‘A’, ‘Wisen’.

**Line 9:** It displays “{'Wisen', '4', '5', 'A'}” in the console.

## ➢ Set Operations: add (Applicable to ONLY set instance)

⇒ **add (elem)**

| It adds the specified element in the invoking set object. If the specified element already exists; nothing happens. | | |
|---|---|---|
| **Parameters** | elem | It indicates an element to add. |
| **Returns** | None | It returns None value. |

```
1. data1 = {1, 2, 3, 4}
2. result = data1.add(99)
3. print(result)
4. print(data1)
```

**Line 2:** It adds the specified element 99 to the invoking set object data1.

**Line 3:** It displays "{1, 2, 99, 3, 4}" in the console.

**Line 4:** It displays "None" in the console.

Note: In your computer you may get the output; since set is unordered collection of elements.

## ➢ Set Operations: remove (Applicable to ONLY set instance)

⇒ **remove (elem)**

| It deletes the specified element from the invoking set object. If the specified element does not exists in the invoking set object, Python throws the KeyError. | | |
|---|---|---|
| **Parameters** | elem | It indicates the element to remove. |
| **Returns** | None | It returns None value. |
| **Error** | KeyError | It throws when the specified element is not available in the set object. |

```
1. data1 = {1, 2, 3, 4}
2. result = data1.remove(3)
3. print(result)
4. print(data1)
5. data2 = {1, 2, 3, 4}
6. result = data2.remove(99)
```

**Line 2:** It deletes the specified element 3 from the invoking set object data1. It returns None and assign to result variable.

**Line 4:** It displays "{1, 2, 4}" in the console.

**Line 6:** In the remove method invocation, the specified element 99 is not available in the invoking set object; therefore, it throws KeyError: 99

Note: In your computer you may get the output; since set is unordered collection of elements.

## ➢ Set Operations: discard (Applicable to ONLY set instance)

⇒ **discard (elem)**

| It deletes the specified element from the invoking set object. If the specified element does not exists in the invoking set object, nothing happens. | | |
|---|---|---|
| **Parameters** | elem | It indicates an element to delete from the invoking set object. |
| **Returns** | None | It returns None value. |

```
1. data1 = {1, 2, 3, 4}
2. result = data1.discard(3)
3. print(result)
4. print(data1)
5. data2 = {1, 2, 3, 4}
6. result = data2.discard(99)
```

**Line 2:** It deletes the specified element 3 from the invoking set object data1. It returns None and assign to result variable.

**Line 4:** It displays "{1, 2, 4}" in the console.

**Line 6:** In the discard method invocation, the specified element 99 is not available in the invoking set object; the discard method does nothing.

Note: In your computer you may get the output; since set is unordered collection of elements.

## ➢ Set Operations: pop (Applicable to ONLY set instance)

⇒ **pop ()**

| It return and delete an arbitrary element from the invoking set object. | | |
|---|---|---|
| **Returns** | Any | It returns element. |
| **Error** | KeyError | It throws an KeyError, if the invoking set is empty |

```
1. data1 = {'U', 'V', 'W', 'X', 'Y', 'Z'}
2. result = data1.pop()
3. print(result)
4. print(data1)
```

**Line 2:** The pop method returns and deletes an arbitrary element from the invoking set object. The return element assigns to the result variable..

**Line 3:** It displays 'U' in console.

**Line 4:** It displays "{'V', 'Y', 'W', 'X', 'Z'}" in the console.

Note: In your computer you may get the output; since set is unordered collection of elements. If you execute this program multiple times, you may get different output each time you run.

## ➢ Set Operations: clear (Applicable to ONLY set instance)

⇒ **clear ()**

| It delete all the elements in the invoking set object. | | |
|---|---|---|
| **Returns** | None | It returns None value. |

```
1. data1 = {'U', 'V', 'W', 'X', 'Y', 'Z'}
2. result = data1.pop()
3. print(result)
4. print(data1)
```

**Line 2:** The clear method deletes all the element from the invoking object data1. This method returns None and assigns to the result.

**Line 3:** It displays None in console.

**Line 4:** It displays “set()” in the console.

## ❖ Dictionary

Python supports another important mapping data type called dictionary. In other languages it is called as “Associative Memories”, “Associative Array”, “Map”, “Hash Maps” or “Unordered Map”. It is also called as mapping, since it maps the set of keys onto the set of values (in the algebraic sense).

In Python, the important characteristics of dict are:

- Dictionary objects are mutable.
- Zero or more ordered pairs ($k_i$, $v_i$) of Key-Value pairs.
- Each $k_i$ indicates the key. Each $v_i$ indicates the value.
- It is Key-Value hash table structure.
- The key must be unique.
- Each key must be immutable type.
- The value can be any data type.
- By default, the order of elements are not user defined; the order of the pairs is arbitrary: it is essentially an unordered set of ordered pairs.
- Dictionary element cannot access by index; keys are used to retrieve value.

Example

Yellow Pages

### ➢ Dictionary Creation

Dictionary can be created in three ways:

1. Using Curly Braces { }
2. Using dict() Function
3. Set Comprehensions

## ➢ Dictionary Creation : Using { }

Python allows you create empty or non-empty dictionary using { }. Each key value pair ($k_i$, $v_i$) are separated by comma.

```
1. data = {}
2. print(data)
3. print(type(data))
4. counCurr = {"India": "INR", "Australia": "AUD", "Finland":
   "EUR", "Italy": "EUR"}
5. print(counCurr)
6. print(type(counCurr))
```

**Line 1:** The empty dict object is constructed by using { }.

**Line 2:** It displays "{}" in the console.

**Line 3:** It displays <class 'dict'> in the console.

**Line 4:** The non-empty dict object is constructed by using { }.

**Line 5:** It displays "{'India': 'INR', 'Australia': 'AUD', 'Finland': 'EUR', 'Italy': 'EUR'}" in the console.

**Line 6:** It displays <class 'dict'> in the console.

| | |
|---|---|
|  | Python does NOT allow to create empty sets using { }; but it allows to create empty dict obkject using { }. |

## ➢ Dictionary Creation : Using dict Constructors

Python allows you create dictionary object using built in dict constructor.

⇒ **dict([**kwargs])**

| It constructs a new empty or non-empty dict object of an unordered collection of key value pairs from the specified keyword arguments with no duplicate element as key. | | | |
|---|---|---|---|
| **Parameters** | kwargs | Optional | It indicates a one or more keyword arguments.<br>If the argument is omitted, it creates an empty dict object.<br>It can accept keyword arguments. |

```
1. data = dict()
2. print(data)
3. print(type(data))
4. counCurr = dict(India="INR", Australia="AUD")
5. print(counCurr)
6. print(type(counCurr))
```

**Line 1:** The dict() built function creates a new dict object and assign to data.

**Line 2:** It displays “{}” in the console.

**Line 3:** It displays <class 'dict'> in the console.

**Line 4:** A new dict object is created by invoking the dict built-in function by passing the keyword arguments.

**Line 5:** It displays “{'India': 'INR', 'Australia': 'AUD'}” in the console.

**Line 6:** It displays <class 'dict'> in the console.

⇒ **dict([mapping, **kwargs])**

| It constructs a new empty or non-empty dict object of an unordered collection of key value pairs from the specified mapping object with no duplicate element as key. | | | |
|---|---|---|---|
| **Parameters** | mapping | Optional | Indicates an mapping object. The mapping object can be dict etc.<br>If the argument is omitted, it creates an empty dict object. |
| | kwargs | Optional | It indicates a one or more keyword arguments. |

```
1. counCurr = dict({"India": "INR", "Australia": "AUD",
   "Finland": "EUR", "Italy": "EUR"})
2. print(counCurr)
3. print(type(counCurr))
```

**Line 1:** A new dict object is created by invoking the dict built-in function by passing the mapping object.

**Line 2:** It displays “{'India': 'INR', 'Australia': 'AUD', 'Finland': 'EUR', 'Italy': 'EUR'}” in the console.

**Line 3:** It displays <class 'dict'> in the console.

⇒ **dict([iterable, **kwargs])**

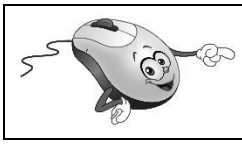
| It constructs a new empty or non-empty dict object of an unordered collection of key value pairs from the specified iterable object with no duplicate element as key. | | | |
|---|---|---|---|
| **Parameters** | iterable | Optional | Indicates an iterable object. The iterable object can be String, List, Set, Tuple etc.<br>If the argument is omitted, it creates an empty dict object. |
| | kwargs | Optional | It indicates a one or more keyword arguments. |

```
1. data = dict([("A", 1), ("B",2), ("C",3)], India="INR", Australia="AUD")
2. print(data)
3. print(type(data))
```

**Line 1:** A new dict object is created by invoking the dict built-in function by passing the iterable object and keyword arguments.

**Line 2:** It displays "{'A': 1, 'B': 2, 'C': 3, 'India': 'INR', 'Australia': 'AUD'}" in the console.

**Line 3:** It displays <class 'dict'> in the console.



Tuples can be used as keys in dict object if and only if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it CANNOT be used as a key.

## ➢ Dictionary Creation : Using dict Comprehensions

Like List, Python allows you create dictionary object using dict Comprehensions.

```
1. data = [1,2,3,4,5]
2. cubes = {  e: e * e * e for e in data if e % 2 != 0}
3. print(cubes)
4. print(type(cubes))
```

**Line 2:** The right hand side of the assignment operator contains the dict comprehension {e: e * e * e for e in data if e % 2 != 0}. This comprehension is used to find the cube for every odd element in lists called data and stores in the cubes dict object.

**Line 3:** It displays {1: 1, 3: 27, 5: 125} in console.

**Line 4:** It displays <class 'dict'> in console.

## ❖ Accessing Dictionary

Dictionary values can be accessed by specifying keys within square brackets. When you try to access the non-existent key, Python raises Key Error. The general form is

dictionaryVarible[key]

```
1. data1 = {'Australia': "AUD", 'Italy': "EUR", 'Germany': "EUR",
   'India': "INR"}
2. print(data1["Italy"])
3. print(data1["USA"])
```

**Line 1:** Created a new dict object called data1 using { } brackets with mapping data

**Line 2:** Access the value by specifying the key Italy within [ ] brackets. It displays “EUR” in the console.

**Line 3:** The Key “USA” is not available in the dict object data1. Therefore, Python throws KeyError: 'USA'.

### Modify Dictionary Value

Dictionary are mutable in Python. Therefore, it allows you modify the value for a particular key. If the specified key is already available, value gets modified, else a new key: value pair is added to the dictionary.

```
1. data1 = {'Australia': "AUD", 'Italy': "None", 'Germany': "EUR",
   'India': "INR"}
2. data1["Italy"] = "EUR"
3. data1["USA"] = "USD"
4. print(data1)
```

**Line 1:** Created a new dict object called data1 using { } brackets with mapping data

**Line 2:** The specified key “Italy” is already available in data1. Therefore, the Italy value “None” is updated with the new value “EUR”.

**Line 3:** The Key “USA” is not available in the dict object data1. Therefore, Python creates a new Key Value pair called “USA : USD” and added to the data1 dict object.

**Line 3:** It displays “{'Australia': 'AUD', 'Italy': 'EUR', 'Germany': 'EUR', 'India': 'INR', 'USA': 'USD'}” in the console.

## ❖ Dictionary Operations

⇒ **get(key[, defaultValue])**

<table>
<tr><td colspan="4">It determines the value of the specified key from the invoking dict object.</td></tr>
<tr><td rowspan="2">Parameters</td><td>key</td><td>Mandatory</td><td>It indicates the key whose value to be determined.</td></tr>
<tr><td>defaultValue</td><td>Optional</td><td>It indicates the default value which has to return when the specified key is not available in the dict object.</td></tr>
<tr><td>Returns</td><td>Any</td><td colspan="2">If the key is available in the dict object, Python returns the value of the specified key.<br><br>If the key is NOT available in the dict object and defaultValue is not specified (omitted), Python returns None.<br><br>If the key is NOT available in the dict object and defaultValue is specified, Python returns the specified defaultValue.</td></tr>
</table>

```
1. curr = {'Australia': "AUD", 'Italy': "EUR", 'Germany': "EUR",
   'India': "INR"}
2. print(curr.get("Italy"))
3. print(curr.get("USA"))
4. print(curr.get("USA", "Not Found"))
```

**Line 2:** The get method determines the value for the key Italy and the print function displays “EUR” in the console.

**Line 3:** The USA key is not available in curr dict object. Therefore, the get method returns None. The print function displays “None” in the console; since the default value is omitted.

**Line 3:** The USA key is not available in curr dict object. Therefore, the get method returns Not Found. The print function displays “Not Found” in the console; since the default value is specified.

⇒ **clear ()**

| It delete all the elements in the invoking dict object. | | |
|---|---|---|
| **Returns** | None | It returns None value. |

⇒ **items()**

| It determines a new view object of type dict_items which contains all key-value pairs from the invoking dict object. If the invoking list object is modified at any time, then the modification are automatically reflected in the view object. | | |
|---|---|---|
| **Returns** | dict_items | It returns a new dict_items object. |

⇒ **keys()**

| It determines a new view object of type dict_keys which contains all key from the invoking dict object. If the invoking list object is modified at any time, then the modification are automatically reflected in the view object. | | |
|---|---|---|
| **Returns** | dict_keys | It returns a new dict_keys object. |

⇒ **values()**

| It determines a new view object of type dict_values which contains all value from the invoking dict object. If the invoking list object is modified at any time, then the modification are automatically reflected in the view object. | | |
|---|---|---|
| **Returns** | dict_values | It returns a new dict_values object. |

```
1. curr = {'Australia': "AUD", 'Italy': "EUR", 'Germany': "EUR",
   'India': "INR"}
2. newCurr = curr.items()
3. print(newCurr)
4. print(type(newCurr))
5. keys = curr.keys()
6. print(keys)
7. print(type(keys))
8. values = curr.values()
9. print(values)
10.print(type(values))
11.curr["USA"] = "USD"
12.print(newCurr)
13.print(keys)
14.print(values)
```

**Line 2:** The items method determines new view object of type dict_items which contains all key-value pairs from the invoking dict object curr.

**Line 3:** It displays “dict_items([('Australia', 'AUD'), ('Italy', 'EUR'), ('Germany', 'EUR'), ('India', 'INR')])” in the console.

**Line 4:** It displays “<class 'dict_items'>” in the console.

**Line 5:** The items method determines new view object of type dict_keys which contains all key pairs from the invoking dict object curr.

**Line 6:** It displays “dict_keys(['Australia', 'Italy', 'Germany', 'India'])” in the console.

**Line 7:** It displays “<class 'dict_keys>” in the console.

**Line 8:** The items method determines new view object of type dict_values which contains all key pairs from the invoking dict object curr.

**Line 9:** It displays “dict_values(['AUD', 'EUR', 'EUR', 'INR'])” in the console.

**Line 10:** It displays “<class 'dict_values>” in the console.

**Line 11:** The Key “USA” is not available in the dict object curr. Therefore, Python creates a new Key Value pair called “USA : USD” and added to the data1 curr object. It also reflects in all the view objects newCurr, keys and values.

**Line 12:** It displays “dict_items([('Australia', 'AUD'), ('Italy', 'EUR'), ('Germany', 'EUR'), ('India', 'INR'), ('USA', 'USD')])” in the console.

**Line 13:** It displays “dict_keys(['Australia', 'Italy', 'Germany', 'India', 'USA'])” in the console.

**Line 14:** It displays “dict_values(['AUD', 'EUR', 'EUR', 'INR', 'USD'])” in the console.

⇒ **fromkeys(seq[, defaultValue])**

It creates a new dictionary object from the specified sequence of elements as keys with the specified value or None.

If the **defaultValue** is **NOT specified**, None is assigned to each key in the new dictionary object.

If the **defaultValue** is **specified**, The specified value is assigned to each key in the new dictionary object. If the specified value is a mutable object like list, set, dict etc, whenever the specified mutable object is modified, each value in the new dict object is also modified.

It is a class method. Therefore, it has to be invoked by the class name, not using the instance.

| | | | |
|---|---|---|---|
| **Parameters** | seq | Mandatory | It indicates the sequence of elements used as keys. |
| | defaultValue | Optional | It indicates the default value for the new dictionary object. |
| **Returns** | dict | It returns the new dict object. | |

```
1. curr1 = dict.fromkeys(["Australia", "Italy"])
2. print(curr1)
3. curr2 = dict.fromkeys(["Australia", "Italy"], "No Value")
4. print(curr2)
5. cr = {"AUD", "EUR"}
6. curr3 = dict.fromkeys(["Australia", "Italy"], cr)
7. print(curr3)
8. cr.add("USD")
9. print(curr3)
```

**Line 1:** The fromKeys method is invoked using class name dict; since it is a class method. It determines a new dictionary object from the specified sequence "Australia", "Italy" as keys. The defaultValue is not specified; therefore each key value contains None.

**Line 2:** It prints “"No Value"” in the console.

**Line 3:** It determines a new dictionary object from the specified sequence "Australia", "Italy" as keys. The defaultValue is specified; therefore each key value contains "No Value".

**Line 4:** It prints “{'Australia': 'No Value', 'Italy': 'No Value'}” in the console.

**Line 6:** It determines a new dictionary object from the specified sequence "Australia", "Italy" as keys. The defaultValue is specified; therefore each key value contains "{"AUD", "EUR"}". It is mutable object.

**Line 7:** It prints “{'Australia': {'EUR', 'AUD'}, 'Italy': {'EUR', 'AUD'}}” in the console.

**Line 8:** It adds a new element in the cr set object. Therefore, each value in the curr3 dict object is automatically modified.

**Line 9:** It prints “{'Australia': {'EUR', 'AUD', 'USD'}, 'Italy': {'EUR', 'AUD', 'USD'}}” in the console.

⇒ **popitem()**

| It returns and remove an arbitrary element (key-value pair) from the invoking dict object. | | |
|---|---|---|
| **Returns** | tuple | It returns the tuple object which contains the removed element if the dict it not empty.<br>If the dict object is empty, Python throws the key error. |

```
1. curr = {'Australia': "AUD", 'Italy': "EUR", 'Germany': "EUR",
   'India': "INR"}
2. result = curr.popitem()
3. print(result)
4. print(type(result))
5. print(curr)
```

**Line 2:** It returns and deletes an arbitrary element from the invoking dict object curr. The removed element key-value pair is assigned as tuple to the variable result.

**Line 3:** It prints “('India', 'INR')” in the console.

**Line 4:** It prints “<class 'tuple'>” in the console.

**Line 5:** It prints “{'Australia': 'AUD', 'Italy': 'EUR', 'Germany': 'EUR'}” in the console.

Note: In your computer the output may differ; since it returns the arbitrary element.

⇒ **pop(key[, defaultValue])**

<table>
<tr><td colspan="4">It deletes an element which contains the specified key and returns the value of the deleted key-value pair.</td></tr>
<tr><td rowspan="2">Parameters</td><td>key</td><td>Mandatory</td><td>It indicates the key to be searched for deletion.</td></tr>
<tr><td>defaultValue</td><td>Optional</td><td>It indicated the value which will be returned when the specified key is not available in the invoking dict object.</td></tr>
<tr><td>Returns</td><td>Any</td><td colspan="2">If the specified key is available, Python returns the value of the specified key.<br><br>If the specified key is NOT available and defaultValue is specified, Python returns the specified default value.<br><br>If the specified key is NOT available and defaultValue is NOT specified, Python throws KeyError.</td></tr>
</table>

```
1. curr = {'Australia': "AUD", 'Italy': "EUR", 'Germany': "EUR",
   'India': "INR"}
2. result = curr.pop("Italy")
3. print(result)
4. print(curr)
5. result = curr.pop("Canada", "Not Found")
6. print(result)
7. result = curr.pop("USA")
8. print(result)
```

**Line 2:** The specified key “Italy” is available in the invoking dict object curr. Therefore, 'Italy': "EUR" key-value pair is removed and returns the value “EUR” and assign to result variable.

**Line 3:** It prints “EUR” in the console.

**Line 4:** It prints “” in the console.

**Line 5:** The specified key “Canada” is NOT available in the invoking dict object curr. Therefore, it returns the specified default value “Not Found”.

**Line 6:** It prints “Not Found” in the console.

**Line 7:** The specified key “USA” is NOT available in the invoking dict object curr. The defaultValue is NOT specified. Therefore, it throws “KeyError: 'USA'”.

⇒ **setdefault(key[, defaultValue])**

<table>
<tr><td colspan="4">If the specified key is available in the invoking dict object, Python returns the value associated with the key.<br><br>If the specified key is NOT available in the invoking dict object,<br>1. If the defaultValue is NOT specified, Python inserts key with value None in the invoking object.<br>2. If the defaultValue is specified, Python inserts key with the specified value in the invoking object.</td></tr>
<tr><td rowspan="2">Parameters</td><td>key</td><td>Mandatory</td><td>Indicates the key to be searched.</td></tr>
<tr><td>defaultValue</td><td>Optional</td><td>Indicates the value for the key when the key is not available in the invoking object.</td></tr>
</table>

| | | |
|---|---|---|
| **Returns** | Any | If the specified key is available in the invoking dict object, it returns the value associated with the specified key.<br>If the specified key is available in the invoking dict object, and defaultValue is not specified, it returns None.<br>If the specified key is available in the invoking dict object, and defaultValue is specified, it returns specified defaultValue. |

```
1. curr = {'Australia': "AUD", 'Italy': "EUR", 'Germany': "EUR",
   'India': "INR"}
2. result = curr.setdefault("Italy")
3. print(result)
4. print(curr)
5. result = curr.setdefault("Canada", "Not Found")
6. print(result)
7. print(curr)
8. result = curr.setdefault("USA")
9. print(result)
10.print(curr)
```

**Line 2:** Invoke the setdefault method by passing Italy as parameter. The Italy key is available in curr dict object. It returns EUR and assign to result variable.

**Line 3:** It prints “EUR” in the console.

**Line 4:** It prints “{'India': 91, 'Australia': 61, 'Denmark': 45, 'Egypt': 20}” in the console.

**Line 5:** Invoke the setdefault method by passing Canada & Not Found as parameter. The Canada key is NOT available in curr dict object. Therefore, Python inserts “Canada – Not Found” key value pair in the invoking dict object curr. It also returns Not Found and assign to result variable.

**Line 6:** It prints “Not Found” in the console.

**Line 7:** It prints “{'India': 91, 'Australia': 61, 'Denmark': 45, 'Egypt': 20, 'Canada': 'Not Found'}” in the console.

**Line 8:** Invoke the setdefault method by passing USA as parameter. The defaultValue is not specified. The USA key is NOT available in curr dict object. Therefore, Python inserts “Canada – None” key value pair in the invoking dict object curr. It also returns None and assign to result variable.

**Line 9:** It prints “None” in the console.

**Line 10:** It prints “{'India': 91, 'Australia': 61, 'Denmark': 45, 'Egypt': 20, 'Canada': 'Not Found', 'USA': None}” in the console.

⇒ **update([other])**

| |
|---|
| It updates the invoking dict object from the specified dictionary or iterable of key value pairs.<br>If the key in the specified dict object is NOT available in the invoking dict object, the specified key-value pair is inserted in the invoking object.<br>If the key in the specified dict object is available in the invoking dict object, it updates the specified value for the key in the invoking object.<br>If it is invoked without parameter, the invoking dict object remains unchanged. |

| | | | |
|---|---|---|---|
| **Parameters** | other | Optional | Indicates a dictionary object or iterable object of key value pairs. |
| **Returns** | None | It returns None. | |

```
1. curr = {'Australia': "AUD", 'Italy': "None", 'Germany': "None",
   'India': "INR"}
2. result = curr.update()
3. print(result)
4. print(curr)
5. result = curr.update({'Indonesia': 'IDR', 'Italy': 'EUR'})
6. print(result)
7. print(curr)
8. result = curr.update(Germany = 'EUR', USA = 'USD')
9. print(result)
10.print(curr)
```

**Line 2:** Invoke the update method with zero parameters. Therefore, the invoking dict object remains unchanged.

**Line 3:** It displays “None” in the console.

**Line 4:** It displays “{'Australia': 'AUD', 'Italy': 'None', 'Germany': 'None', 'India': 'INR'}” in the console.

**Line 5:** Invoke the update method with one dict parameter. The parameter dict object contains two key value pairs. The Indonesia key is not available in the invoking dict object curr. Therefore, the 'Indonesia': 'IDR' key value pair is added to the invoking object curr. The Italy key is available in the invoking dict object curr. Therefore, the ' Italy' key value ‘EUR' is updated to the invoking object curr.

**Line 7:** It displays “{'Australia': 'AUD', 'Italy': 'EUR', 'Germany': 'None', 'India': 'INR', 'Indonesia': 'IDR'}” in the console.

**Line 8:** Invoke the update method with one keyword parameters. The Germanykey is available in the invoking dict object curr. Therefore, the 'Germany’ key value ‘EUR' is updated to the invoking object curr. The USAkey is not available in the invoking dict object curr. Therefore, the ‘USA’: 'USD' key value pair is added to the invoking object curr.

**Line 10:** It displays “{'Australia': 'AUD', 'Italy': 'EUR', 'Germany': 'EUR', 'India': 'INR', 'Indonesia': 'IDR', 'USA': 'USD'}” in the console.

## ❖ Common Sequence Operations

The most of the sequence types both mutable and immutable types supports the below operations:

1. The in Operator
2. The not in Operator
3. The + (concatenation) Operator
4. The * (repetition) Operator
5. The slicing Operator
6. The len & count Function
7. The min & max Function

8. The index Function

## ➢ The in Operator

The general form is

element in sequence

This operation determines true if the specified element is available in the specified list; otherwise false.

```
1. data1 = set("Wisen")
2. result = 'i' in data1
3. print("'i' in data1 ", result)
4. result = 'a' in data1
5. print("'a' in data1 ",result)
```

**Line 2:** The element 'i' is available in "Wisen"; therefore it determines True.

**Line 4:** The element 'a' is NOT available in "Wisen"; therefore it determines False.

## ➢ The not in Operation

The general form is

element not in sequence

This operation determines true if the specified element is NOT available in the specified list; otherwise false.

```
1. data1 = set("Wisen")
2. result = 'i' not in data1
3. print("'i' not in data1 ", result)
4. result = 'a' not in data1
5. print("'a' not in data1 ",result)
```

**Line 2:** The element 'i' is available in "Wisen"; therefore it determines false.

**Line 4:** The element 'a' is NOT available in "Wisen"; therefore it determines true.

⇒ **len(s)**

| It determines the length ( the number of elements ) in the specified sequence object | | |
|---|---|---|
| **Parameters** | s | It indicates a sequence type object (string, bytes, tuple, list, or range) or a collection object (dictionary, set or frozen set). |
| **Returns** | Integer | It returns the length in integer data type.<br>If you pass invalid argument, Python throws TypeError. |
| **Example** | data = []<br>print(len(data)) # It displays 0 in the console.<br>data = (10, 20, 30) | |

| | |
|---|---|
| | print(len(data)) # It displays 3 in the console.<br>result = len(11) # TypeError: object of type 'int' has no len() |

⇒ **count(elem)**

| | | |
|---|---|---|
| It determines the no of times the specified element available in the invoking list. It should be the same data type. Note: In Python boolean is also considered as number data type. | | |
| **Parameters** | elem | Indicates the element to check whether it is appeared in the invoking sequence types. It is applicable to list, tuple & string data types. It is not applicable set, frozenset and dict data types. |
| **Returns** | Integer | It returns the count in integer data type.<br>If you invoke this function using set, frozenset or dict object, Python throws AttributeError. |
| **Example** | data = [1, 2, 3, 4, 1, '1', 2, True, 1.0]<br>cnt = data.count(1)<br>print(cnt) # It displays 4; since integer, boolean and floating point comes under data type. | |

⇒ **min(iterable, *iterables[, key, default])**

| | | | |
|---|---|---|---|
| It determines the smallest element in a specified iterable object or smallest of two or more specified iterable objects. | | | |
| **Parameters** | iterable | Mandatory | Indicates sequence object (string, list, tuple), collection object (set, dictionary) or an iterator object |
| | *iterables | Optional | Indicates two or more iterable object. |
| | key | Optional | Indicates the function where comparison logic is implemented. |
| | default | Optional | It indicates an default value |

⇒ **max(iterable, *iterables[, key, default])**

| |
|---|
| It determines the largest element. Remaining every concept same like above min signature. |

| | |
|---|---|
| **Returns** | Like below |

✍ **min(iterable) max(iterable)**

✍ It throws ValueError if the specified iterable object is empty.

```
1. data1 = []
2. minResult = min(data1);
3. print(minResult);
4. maxResult = max(data1);
5. print(maxResult);
```

**Line 2:** It throws ValueError: min() arg is an empty sequence

- **min(iterable [, default]) max(iterable[, default])**
- It determines default value when the specified iterable object is empty.

```
1. data1 = []
2. minResult = min(data1, default="No Minimum");
3. print(minResult);
4. maxResult = max(data1, default="No Maximum");
5. print(maxResult);
```

**Line 2:** It returns “No Minimum”; since data1 sequence object is empty.

**Line 4:** It returns “No Maximum”; since data1 sequence object is empty.

- **min(iterable [, default]) max(iterable[, default])**
- It determines smallest value when the specified iterable object is NOT empty. If it is not empty list, the default value will be ignored.

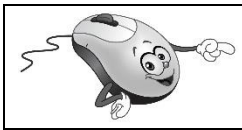
```
1. data1 = [1, 2, 3, 4]
2. data2 = 'abcABC';
3. minResult1 = min(data1, default="No Minimum");
4. print(minResult1);
5. minResult2 = min(data2);
6. print(minResult2);
7. maxResult1 = max(data1, default="No Maximum");
8. print(maxResult1);
9. maxResult2 = max(data2);
10.print(maxResult2);
```

**Line 3:** It returns 1 which is the smallest element in the data1 iterable object.

**Line 5:** It returns A which is the smallest element in the data2 iterable object. The ASCII value of a is 97, b is 98, c is 99, A is 65 and B is 66. The smallest of the ASCII values is A.

**Line 7:** It returns 4 which is the largest element in the data1 iterable object.

**Line 9:** It returns c which is the largest element in the data2 iterable object. The largest value of ASCII values is c.



Only the same data type can be compared. Different data types cannot be compared. When you try to compare a different data type, Python throws TypeError.

```
1. data1 = [1, 2, 3, 'A']
2. print(min(data1))
```

**Line 2:** It throws TypeError: '<' not supported between instances of 'str' and 'int'

- **min(iterable [, key]) max(iterable[, key])**
- It determines smallest value in the specified object according to the custom logic implemented in the specified function. For each element the custom function will be invoked and the element is automatically passed to the parameter of the custom function.

```
1. def characterCount(elem):
2.     return len(elem)
3. data1 = ('abc', 'bc', 'cdfer', 'B')
4. minResult1 = min(data1, key=characterCount);
5. print(minResult1);
6. maxResult1 = max(data1, key=characterCount);
7. print(maxResult1);
```

**Line 4:** Iteration 1:

The characterCount function is invoked and abc is passed to the parameter elem. Line2 returns 3.

Iteration 2:

The characterCount function is invoked and bc is passed to the parameter elem. Line2 returns 2.

Iteration 3:

The characterCount function is invoked and cdfer is passed to the parameter elem. Line2 returns 5.

Iteration 4:

The characterCount function is invoked and B is passed to the parameter elem. Line2 returns 1.

The smallest of the custom function return value is 1. Therefore, the min function determines B.

**Line 6:** The characterCount function is invoked for each element by passing element to the parameter. The largest of the custom function return value is 5. Therefore, the max function determines cdfer.

| | |
|---|---|
|  | If the key function is specified, then elements can be in different data type. |

| | |
|---|---|
|  | The function return values data type must be same. When you try to return a different data type, Python throws TypeError. |

- **min(iterables[, *iterables]) max(iterable[, *iterables])**
- It determines smallest among the specified iterable object.

```
1. data1 = [1, 2, 3, 4]
2. data2 = [0, 2, 3, 6]
3. minResult = min(data1, data2)
4. print(minResult)
5. maxResult = max(data1, data2)
6. print(maxResult)
7. sdata1 = "abCAB";
8. sdata2 = "aBCXB";
9. minResult = min(sdata1, sdata2)
10.print(minResult)
11.maxResult = max(sdata1, sdata2)
12.print(maxResult)
```

**Line 3:** There are two iterable objects data1 and data2 are passed to min function. Now, the min function determines the smallest between data1 and data2. The data1 object first element 1 is compared with data2 object first element 0, the 0 is smallest; therefore, it returns data2 as smallest object and stores in minResult.

**Line 5:** There are two iterable objects data1 and data2 are passed to max function. Now, the max function determines the largest between data1 and data2. The data1 object first element 1 is compared with data2 object first element 0, the 1 is largest; therefore, it returns data1 as largest object and stores in maxResult.

**Line 9:** There are two iterable objects sdata1 and sdata2 are passed to min function. Now, the min function determines the smallest between sdata1 and sdata2.

The sdata1 object first character a ASCII value 97 is compared with data2 object first element a ASCII value 97, both the ASCII values are same;

Next, The sdata1 object second character b ASCII value 98 is compared with data2 object second element B ASCII value 66, the ASCII value is 66 is smallest;

Therefore, it returns data2 as smallest object and stores in minResult.

**Line 9:** There are two iterable objects sdata1 and sdata2 are passed to min function. Now, the max function determines the smallest between sdata1 and sdata2.

The sdata1 object first character a ASCII value 97 is compared with data2 object first element a ASCII value 97, both the ASCII values are same;

Next, The sdata1 object second character b ASCII value 98 is compared with data2 object second element B ASCII value 66, the ASCII value is 98 is largest;

Therefore, it returns data1 as largest object and stores in maxResult.

The iterables objects must be same compatible data type; Otherwise, Python throws TypeError.

|  | The iterables objects cannot be dict objects; id they are dict objects, Python throws TypeError. |
|---|---|

- **min(iterables[, *iterables, key]) max(iterable[, *iterables, key])**
- It determines smallest among the specified iterable objects according to the custom logic implemented in the specified function.

```
1. def elemCount(mylist):
2.     return mylist.count(1)
3. data1 = [1, 2, 3, 1, 2]
4. data2 = [2, 4, 1, 5, 4, 7]
5. minResult = min(data1, data2, key=elemCount)
6. print(minResult)
7. maxResult = max(data1, data2, key=elemCount)
8. print(maxResult)
```

**Line 5:** For each iterable object the elemCount will be invoking by passing the iterable object to the parameter.

**Iteration 1:**

The data1 is passed to the elemCount function parameter mylist. Line 2 returns no of occurrence of 1 i-e 2.

**Iteration 2:**

The data2 is passed to the elemCount function parameter mylist. Line 2 returns no of occurrence of 1 i-e 1.

The smallest of the return values is 1. Therefore, the min function returns data2 and store in minResult.

Line7: The largest of the return values is 2. Therefore, the max function returns data1 and store in maxResult.

⇒ **min(arg1, arg2, *args[, key])**

| It determines the smallest among the specified arguments | | | |
|---|---|---|---|
| **Parameters** | arg1, arg2 | Mandatory | Indicates a value |
| | args | Optional | Indicates a value |
| | key | Optional | Indicates the function where comparison logic is implemented. |
| **Returns** | Any | It returns smallest value. | |

```
1. data1 = 1
2. data2 = 2
3. print(min(data1, data2))
4. print(max(data1, data2))
5. data1 = 'a'
6. data2 = 'A'
7. data3 = 'b'
8. print(min(data1, data2, data3))
9. print(max(data1, data2, data3))
10.def compareNoCaseSensitive(elem):
11.    return elem.upper()
12.print(min(data1, data2, data3, key=compareNoCaseSensitive))
13.print(max(data1, data2, data3, key=compareNoCaseSensitive))
```

**Line 3:** The min function determines the minimum between 1 and 2; it returns 1.

**Line 4:** The max function determines the maximum between 1 and 2; it returns 2.

**Line 8:** The min function determines the minimum between ASCII value of a, A, b. It returns A.

**Line 9:** The max function determines the minimum between ASCII value of a, A, b. It returns b.

**Line 10:** For each element, the compareNoCaseSensitive function is invokes. The returns the uppercase of every element. Therefore it a converts to A. The ASCII value is 65. Therefore it returns a.

⇒ **index(elem[, start[, end]])**

<table>
<tr><td colspan="4">It determines the index number in the invoking list of the first occurrence of the specified element, from starting at the specified start index and ending at the specified end index.</td></tr>
<tr><td rowspan="3">Parameters</td><td>elem</td><td>Mandatory</td><td>Indicate the element to search.</td></tr>
<tr><td>start</td><td>Optional</td><td>Indicate the index the start the search from. It is optional. In the search the start index is INCLUSIVE.<br>If the start index is positive, then the index starts from left to right and the first element index number is 0.<br>If the start index is negative, then the index numbering starts from right to left and the last element index number is -1.</td></tr>
<tr><td>end</td><td>Optional</td><td>Indicate the index the end the search to. It is optional. In the search the end index is EXCLUSIVE.<br>If the end index is positive, then the first element index number is 0.<br>If the end index is negative, then the last element index number is -1.</td></tr>
<tr><td>Returns</td><td>Integer or Error</td><td colspan="2">If the specified element is found, it returns the index position of the first occurrence. If the element is NOT found it throws the ValueError.</td></tr>
</table>

- **Simple List – start & end are positive values**

```
1. data = [0, 7, 2, '0', 0, 6, False, 6, 0.0]
2. print(data.index(0))
```

In the above program, Line 2 prints 0, It start search from left to right and find the first occurrence i-e in index position 0 the value 0 is available.

```
1. data = [0, 7, 9, '0', 0, 6, False, 6, 0.0]
2. print(data.index(9))
```

In the above program, Line 2 prints 2, It start search from left to right and find the first occurrence i-e in index position 2 the value 2 is available.

```
1. data = [0, 7, 9, '0', 0, 6, False, 6, 0.0]
2. print(data.index(3))
```

In the above program, Line 2 throws ValueError: 3 is not in list.

```
1. data = [0, 7, 9, '0', 0, 6, False, 6, 0.0]
2. print(data.index(0,2))
```

In the above program, Line 2, in index method the start index is specified as 2. Therefore, Python starts search from index position 2. In Index position 3, the value is 0. But the data type is string. It does not match.

Now, it checks the next element i-e index position 4. The value is 0 and data type is integer. It matches. It returns 4, the print function prints 4 in the console.

```
1. data = [0, 7, 9, '0', 0, 6, False, 6, 0.0]
2. print(data.index(0, 5, 8))
```

In the above program, Line 2, in index method the start index is specified as 5 and end index is specified as 8. Therefore, Python starts search from index position 5. In Index position 5, the value is False. It is a boolean data type. Its integer value is 0. It matches. It returns 6, the print function prints 6 in the console.

```
1. data = [0, 7, 9, '0', 0, 6, False, 6, 0.0]
2. print(data.index(0, 1, 4))
```

In the above program, Line 2, in index method the start index is specified as 1 and end index is specified as 4. Therefore, Python starts search from index position 1. In Index position 1, the value is 7. It does not match.

Python checks the next index position 2. In index position 2, the value is 9. It does not match.

Python checks the next index position 3. In index position 2, the value is ‘0’. But the data type is string. It does not match.

The end index is specified as 4. It is exclusive. Therefore, Python stops the search.

The specified value 0 is not found between the index position 1 (INCLUSIVE) and 4 (EXCLUSIVE). Python throws the below error.

print(data.index(0, 1, 4))

ValueError: 0 is not in list

- **Simple List – start index negative value & end index positive value**

```
1. data = [0, 7, 2, '0', 0, 6, False, 6, 0.0]
2. print(data.index(0,-2))
```

In the above program, Line 2 index method contains negative value for start index. Therefore, the index numbering starts from right to left and the last element index number is -1, the last element before element index position is -2 and so on. Python starts search from index position -2. In index position -2, the value is 6. It does not match.

Now, Python checks the next element i-e in index position -1. The value 0.0. In Python 0.0 is equals to 0. It matches. Therefore index method returns the original index number i-e 8.

```
3. data = [0, 7, 9, '0', 0, 6, False, 6, 0.0]
4. print(data.index(0, -7))
```

In the above program, Line 2 index method contains negative value for start index. Therefore, the index numbering starts from right to left and the last element index number is -1, the last element before element index position is -2 and so on.

In index number -7, the element value is 9. It does not match. Python checks the next element in index number -6. The element value is '0'. The element value is 0 but the data type is string. It does not match.

Python checks the next element in index number -5. The element value is 0. The element value is 0 and the data type is integer. It matches. Therefore index method returns the original index number i-e 4.

```
1. data = [10,20,30,40,50]
2. print(data.index(40, -4, 4))
```

In the above program, in start index number -4, the element value is 20. The value 40 does not match with 20.

Next, the index number -3, the element value is 30. The value 40 does not match with 30.

Next, the index number -2, the element value is 40. The value 40 matches with 40. Therefore index method returns the original index number i-e 4.

```
1. data = [10,20,30,40,50]
2. print(data.index(40, -4, 3))
```

In the above program, in start index number -4, the element value is 20. The value 40 does not match with 20.

Next, the index number -3, the element value is 30. The value 40 does not match with 30.

Next, the index number -2 and end index number is 3. The end index is exclusive. Therefore, Python will not check. It generates "ValueError: 40 is not in list".

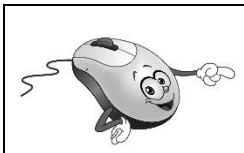
### ➢ Simple List – start index negative value & end index negative value

```
1. data = [10,20,30,40,50]
2. print(data.index(40, -4, -1))
```

In the above program, in start index number -4, the element value is 20. The value 40 does not match with 20.

Next, the index number -3, the element value is 30. The value 40 does not match with 30.

Next, the index number -2, the element value is 40. The value 40 matches with 40. Therefore index method returns the original index number i-e 4.

|  | The index function is applicable only for string, list, tuple data types.<br>It is not applicable for set & dict data type.<br>If try to invoke index function using set or dict object, Python throws AttributeError. |
|---|---|

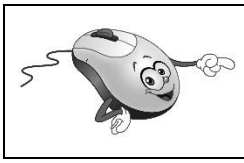
⇒ **The + Operator**

| It concatenates two or more sequences and returns the concatenated sequence. | | |
|---|---|---|
| **Parameters** | seq1, seq2. | Indicate the sequences. |
| **Returns** | list, tuple | It returns the new sequence object |

```
1. data1 = ['A', 'B', 'C', 'D']
2. data2 = [10, 20, 30, 40]
3. newData = data1 + data2;
4. print(newData)
```

**Line 3:** The + operator concatenates data1 and data2 elements and returns a new sequence object.

**Line 4:** It prints "['A', 'B', 'C', 'D', 10, 20, 30, 40]" in the console.

|  | For + operator, Sequence data type must be same. A list can be concatenated with another list. But a list object cannot concatenated with tuple object.<br>If you try, Python throws TypeError |
|---|---|

⇒ **The * Operator**

| It adds the elements by specified number of times. | | |
|---|---|---|
| **Parameters** | n | Indicate number of times. |
| **Returns** | list, tuple | It returns the new sequence object |

```
1. data1 = ['A', 'B', 'C', 'D']
2. newData = data1 * 2;
3. print(newData)
```

**Line 3:** The * operator adds data1 elements by 2 times.

**Line 4:** It prints "['A', 'B', 'C', 'D', 'A', 'B', 'C', 'D']" in the console.

The * operator is applicable only to list & tuple data type. It is not applicable to set & dict data type.

If you try, Python throws TypeError

## ❖ Accessing & Assigning Element using Square Bracket [ ]

You can retrieve the $i^{th}$ element from the sequence by specifying the index position within square bracket. You can also modify the element by specifying the index position within square bracket in the assignment operator.

| Type | Access | Assign |
|---|---|---|
| list | Element can be accessed using index position. | Element can be assigned using index position. |
| tuple | Element can be accessed using index position. | Item can NOT be assigned by index position. |
| range | Element can be accessed using index position. | Item can NOT be assigned by index position. |
| string | Element can be accessed using index position. | Value can NOT be assigned by index position. |
| set | Element can NOT be accessed by index position. | Item can NOT be assigned by index position. |
| frozenset | Element can NOT be accessed by index position. | Item can NOT be assigned by index position. |
| dict | Element can be accessed using key. | Element can be assigned using key. |

```
1. listData = ['A', 'B', 'C', 'D', 'E']
2. listData[2] = 999
3. print(listData[3])
4. print(listData)
5. rangeData = range(1, 5, 2)
6. print(rangeData[1])
7. rangeData[1] = 999
```

**Line 2:** Assign the value 999 in the index position of 2 in listData list object.

**Line 3:** Access the value in the index position 3 in listData list object and print the access data D in console.

**Line 4:** It prints "['A', 'B', 999, 'D', 'E']" in the console.

**Line 6:** Access the value in in the index position 1 in rangeData range object and print the access data 3 in console.

**Line 7:** The range object does not support assignment operation. Therefore, Python throws "TypeError: 'range' object does not support item assignment" in the console.

```
1. setData = frozenset('Wisen')
2. print(setData[1])
```

**Line 2:** The set and frozenset object does not support indexing option. Therefore, Python throws “TypeError: 'frozenset' object does not support indexing”

## ❖ Slicing

The colon : inside the square bracket (subscriptable notation) makes the slicing syntax.

⇒ **seq[start:end]**

⇒ **seq[start:end:step]**

| In Python, the slice copies a range of elements from the invoking sequence types. | | | |
|---|---|---|---|
| **Parameters** | start | Optional | Indicate the beginning index of the slice. It is inclusive. |
| | end | Optional | Indicate the ending index of the slice. It is exclusive. |
| | step | Optional | Indicates the value at which the index increases. The default value is 1. |
| **Returns** | sequence | It returns a sequence of data. Whenever a slice is created from a sequence, the determined slice is also the same data type as the data type from which it was created.<br>For example, when you create a slice from tuple is always a tuple. | |

```
1. mylist = ['A1', 'A2', 'A3', 'A4', 'A5', 'A6', 'A7']
2. newData = mylist[2:6]
3. print("mylist[2:6] ",newData)
4. print(type(newData))
5. rlist = range(1, 10)
6. newRData = rlist[1:4]
7. print(list(newRData))
8. print(type(newRData))
9. myset = set("Wisen")
10.newSet = myset[1:3]
11.print("myset[1:4] ",newSet)
```

**Line 2:** It copies from index position 2 (inclusive) to index position 6 (exclusive) from mylist object. Therefore, A2 to A6 is copied from mylist as list data type and stored in newData.

**Line 3:** It prints “mylist[2:6]  ['A3', 'A4', 'A5', 'A6']” in the console.

**Line 4:** It prints “<class 'list'>” in the console.

**Line 6:** It copies from index position 1 (inclusive) to index position 4 (exclusive) from rlist object. Therefore, 2 to 4 is copied from rlist as range data type and stored in newRData.

**Line 7:** It prints “[2, 3, 4]” in the console.

**Line 8:** It prints “<class ‘range’>” in the console.

**Line 10:** The slicing is not supported for set & frozenset object; since both does not support subscriptable concept. Therefore, Python throws “TypeError: 'set' object is not subscriptable”

## ➢ The start, stop &step values are omitted or contains None value

When the start, stop & step values are omitted, the slicing syntax copies all the elements; i-e elements from the beginning to end. The general form is

seq[:]

It copies seq[0], seq[2] …….. seq[n-1]

```
1. mylist = ['A1', 'A2', 'A3', 'A4', 'A5', 'A6', 'A7']
2. newList = mylist[:]
3. print("mylist[:] ",newList)
```

## ➢ Only start Index is specified

**Case 1: The start index value is positive and within the range**

If the specified value for start index is positive, it copies from the specified start index to end of the sequence (until the last element). The index position starts counting from left to right. The general form is

seq[start:]

It copies seq[start], seq[start+1], seq[start+2] …… seq[n-1]

**Case 2: The start index value is negative and within the range**

If the specified value for start index is negative, it copies from the specified start index to end of the sequence (until the last element). The index position starts counting from right to left. The general form is

seq[-start:]

It copies seq[-start], seq[-start+1], seq[-start+2] …… seq[-1]

**Case 3: The start index value out of range (exceeds the length of the sequence)**

If the specified value for start index out of range, Python handles gracefully. This means that Python does not throw any error.

**Case A:** If out of range is positive, it returns empty sequence.

**Case B:** If out of range is negative, it returns seq[-n], seq[-n+1], seq[-n+2] …… seq[-1]

```
1. mylist = ['A1', 'A2', 'A3', 'A4', 'A5', 'A6', 'A7']
2. newList = mylist[4:]
3. print("mylist[4:] ",newList)
4. newList = mylist[-4:]
5. print("mylist[-4:] ",newList)
6. newList = mylist[12:]
7. print("mylist[12:] ",newList)
8. newList = mylist[-12:]
9. print("mylist[-12:] ",newList)
10.newList = mylist[-1:]
11.print("mylist[-1:] ",newList)
```

**Line 2:** It slices all the elements between the edges start and number of elements i-e 4 to 7. In other words, mylist[4], mylist[5], and mylist[7].

**Line 3:** It prints “mylist[4:] ['A5', 'A6', 'A7']” in the console.

**Line 4:** It slices all the elements between the edges start and end of the list i-e -4 to end of the list. In other words, mylist[-4], mylist[-3], mylist[-2] and mylist[-1].

**Line 5:** It prints “mylist[-4:] ['A4', 'A5', 'A6', 'A7']” in the console.

**Line 6:** The specified start value is out of range i-e 12 >= number of elements 7. It is positive out of range. Therefore, Python returns empty list object.

**Line 7:** It prints “[]” in the console.

**Line 8:** The specified start index is out of range; but negative. Therefore, it slices all the elements between the edges negative of no of elements i-e -7 to end of the list. In the other words, mylist[-7], mylist[-6], mylist[-5] , mylist[-4], mylist[-3], mylist[-2] and mylist[-1].

**Line 9:** It prints “mylist[-12:] ['A1', 'A2', 'A3', 'A4', 'A5', 'A6', 'A7']” in the console.

**Line 10:** It slices all the elements between the edges start and end of the list i-e -1 to end of the list. In other words, mylist[-1].

**Line 5:** It prints “mylist[-1:] ['A7']” in the console.

## ➢ Only end Index is specified

**Case 1: The end index value is positive and within the range**

If the specified value for end index is positive, it copies from 0 index position element (first element) to end of the specified end index of the sequence. The end index element is exclusive. The index position starts counting from left to right. The general form is

seq[:end]

It copies seq[0], seq[1], seq[2] …… seq[end-1]

**Case 2: The start index value is negative and within the range**

If the specified value for end index is negative, it copies from 0 index position element (first element) to end of the specified index of the sequence. The index position starts counting from right to left. The general form is

seq[: -end]

It copies seq[-n], seq[-n+1], seq[-n+2] …… seq[-end]

**Case 3: The start index value out of range (exceeds the length of the sequence)**

If the specified value for end index out of range, Python handles gracefully. This means that Python does not throw any error.

**Case A:** If out of range is positive, it returns seq[0], seq[1], seq[2] …… seq[n-1].

**Case B:** If out of range is negative, it returns empty sequence.

```
1. mylist = ['A1', 'A2', 'A3', 'A4', 'A5', 'A6', 'A7']
2. newList = mylist[None:4]
3. print("mylist[None:4] ",newList)
4. newList = mylist[:-4]
5. print("mylist[:-4] ",newList)
6. newList = mylist[:10]
7. print("mylist[:10] ",newList)
8. newList = mylist[:-10]
9. print("mylist[:-10] ",newList)
10.newList = mylist[:-7]
11.print("mylist[:-7] ",newList)
```

**Line 2:** The None is considered as 0. It slices all the elements between the edges 0 and specified end edge i-e 0 to 4. In other words, mylist[0], mylist[1], mylist[2] and mylist[3].

**Line 3:** It prints “mylist[None:4] ['A1', 'A2', 'A3', 'A4']” in the console.

**Line 4:** It slices all the elements between the edges 0 and specified end edge of the list i-e 0 to -4. In other words, mylist[-7], mylist[-6] and mylist[-5].

**Line 5:** It prints “mylist[:-4] ['A1', 'A2', 'A3']” in the console.

**Line 6:** The specified end value is out of range i-e 10 >= number of elements 7. It is positive out of range. Therefore, Python returns mylist[0], mylist[1], mylist[2] …. mylist[n-1].

**Line 7:** It prints “mylist[:10] ['A1', 'A2', 'A3', 'A4', 'A5', 'A6', 'A7']” in the console.

**Line 8:** The specified end index is out of range; but negative. Therefore, it empty list object.

**Line 9:** It prints “mylist[:-10] []” in the console.

**Line 10:** The specified end index is out of range; even specified end index == number of elements in the list object; but negative. Therefore, it empty list object.

**Line 5:** It prints “mylist[:-7] []” in the console.

## ➢ Both start & end Index are specified

**Case 1: The start & end index value are positive**

The general form is

seq[start : end]

If start index is less than to end index (both are within range), then It copies seq[start], seq[start+1] … seq[end-1] and returns.

If start index is less than to end index (start index is within range & end index is out of range), then It copies seq[start], seq[start+1] … seq[n-1] and returns. The end index converts to number of elements.

If start index is out of range, then it returns empty sequence object. The start index converts to 0.

If start index is greater than or equal to end index, then it returns empty sequence object.

```
1. mylist = ['A1', 'A2', 'A3', 'A4', 'A5', 'A6', 'A7']
2. newList = mylist[2:6]
3. print("mylist[2:6] ",newList)
4. newList = mylist[2:10]
5. print("mylist[2:10] ",newList)
6. newList = mylist[12:20]
7. print("mylist[12:20] ",newList)
8. newList = mylist[6:2]
9. print("mylist[2:6] ",newList)
```

**Line 2:** It slices all the elements between the specified edges start and end i-e 2 to 10. The end is out of boundary. Therefore, it consider as number of elements i-e 7. In other words, mylist[0], mylist[1], mylist[2], mylist[3], mylist[4], mylist[5] and mylist[6].

**Line 3:** It prints "mylist[2:10] ['A3', 'A4', 'A5', 'A6', 'A7']" in the console.

**Line 4:** The start index is out of boundary; there it returns empty list object.

**Line 5:** It prints "mylist[12:20] []" in the console.

**Line 6:** The start index is greater than end index; there it returns empty list object.

**Line 7:** It prints "mylist[2:6] []" in the console.

**Case 2: The start index value is negative and the end index is positive**

The general form is

seq[-start : end]

If start index & end index are within the range and abs(start) < end, it returns seq[-start], seq[-start+1] … seq[end-1].

If start index & end index are within the range and abs(start) >= end, it returns empty sequence object.

If start index is within range & end index is out of range, it returns seq[-start], seq[-start+1] … seq[n-1]. The end index converts to number of elements.

If start index is out of range & end index is within range, it returns seq[-n], seq[-n+1] … seq[end-1]. The start index converts to –n.

If start index is out of range & end index is out of range, it returns seq[-n], seq[-n+1] … seq[n-1]. The end index converts to number of elements.

```
1. mylist = ['A1', 'A2', 'A3', 'A4', 'A5', 'A6', 'A7']
2. newList = mylist[-3:6]
3. print("mylist[-3:6] ",newList)
4. newList = mylist[-3:2]
5. print("mylist[-3:2] ",newList)
6. newList = mylist[-3:16]
7. print("mylist[-3:16] ",newList)
8. newList = mylist[-12:2]
9. print("mylist[-12:2] ",newList)
10.newList = mylist[-12:12]
11.print("mylist[-12:12] ",newList)
```

**Line 2:** It slices all the elements between the specified edges start and end i-e -3 to 6 (abs(start) < end i-e 3 < 6 True). In other words, it returns mylist[-3] and mylist[6] elements.

**Line 3:** It prints “mylist[-3:6] ['A5', 'A6']” in the console.

**Line 4:** It slices all the elements between the specified edges start and end i-e -3 to 2 (abs(start) < end i-e 3 < 2 False). It returns empty object.

**Line 5:** It prints “mylist[-3:2] []” in the console.

**Line 6:** It slices all the elements between the specified edges start and end i-e -3 to 16 (abs(start) < end i-e 3 < 16 True). The end index is out of range. Therefore, it converts to 7. In other words, it returns mylist[-3], mylist[-2] and mylist[6] elements.

**Line 7:** It prints “mylist[-3:16] ['A5', 'A6', 'A7']” in the console.

**Line 8:** It slices all the elements between the specified edges start and end i-e -12 to 2. The start index is out of range. Therefore, it converts to -7. In other words, it returns mylist[-7] and mylist[1] elements.

**Line 9:** It prints “mylist[-12:2] ['A1', 'A2']” in the console.

**Line 4:** It slices all the elements between the specified edges start and end i-e -12 to 12. The start index is out of range. Therefore, it converts to -7. The end index is out of range. Therefore, it converts to 7. In other words, it returns mylist[-7], mylist[-6] , mylist[-5], mylist[-4] , mylist[-3], mylist[-2] and mylist[6] elements.

**Line 5:** It prints “mylist[-12:12] ['A1', 'A2', 'A3', 'A4', 'A5', 'A6', 'A7']” in the console.

**Case 3: The start index value out of range (exceeds the length of the sequence)**

If the specified value for end index out of range, Python handles gracefully. This means that Python does not throw any error.

**Case A:** If out of range is positive, it returns seq[0], seq[1], seq[2] …… seq[n-1].

**Case B:** If out of range is negative, it returns empty sequence.

```
12.mylist = ['A1', 'A2', 'A3', 'A4', 'A5', 'A6', 'A7']
13.newList = mylist[None:4]
14.print("mylist[None:4] ",newList)
15.newList = mylist[:-4]
16.print("mylist[:-4] ",newList)
17.newList = mylist[:10]
18.print("mylist[:10] ",newList)
19.newList = mylist[:-10]
20.print("mylist[:-10] ",newList)
21.newList = mylist[:-7]
22.print("mylist[:-7] ",newList)
```

**Line 2:** The None is considered as 0. It slices all the elements between the edges 0 and specified end edge i-e 0 to 4. In other words, mylist[0], mylist[1], mylist[2] and mylist[3].

**Line 3:** It prints “mylist[None:4] ['A1', 'A2', 'A3', 'A4']” in the console.

**Line 4:** It slices all the elements between the edges 0 and specified end edge of the list i-e 0 to -4. In other words, mylist[-7], mylist[-6] and mylist[-5].

**Line 5:** It prints “mylist[:-4] ['A1', 'A2', 'A3']” in the console.

**Line 6:** The specified end value is out of range i-e 10 >= number of elements 7. It is positive out of range. Therefore, Python returns mylist[0], mylist[1], mylist[2] …. mylist[n-1].

**Line 7:** It prints “mylist[:10] ['A1', 'A2', 'A3', 'A4', 'A5', 'A6', 'A7']” in the console.

**Line 8:** The specified end index is out of range; but negative. Therefore, it empty list object.

**Line 9:** It prints “mylist[:-10] []” in the console.

**Line 10:** The specified end index is out of range; even specified end index == number of elements in the list object; but negative. Therefore, it empty list object.

**Line 5:** It prints “mylist[:-7] []” in the console.