|  |  |
| --- | --- |
| Operators | Meaning |
| () | Parentheses |
| \*\* | Exponent |
| +x, -x, ~x | Unary plus, Unary minus, Bitwise NOT |
| \*, /, //, % | Multiplication, Division, Floor division, Modulus |
| +, - | Addition, Subtraction |
| <<, >> | Bitwise shift operators |
| & | Bitwise AND |
| ^ | Bitwise XOR |
| | | Bitwise OR |
| ==, !=, >, >=, <, <=, is, is not, in, not in | Comparisons, Identity, Membership operators |
| not | Logical NOT |
| and | Logical AND |
| or | Logical OR |

X=+5+4\*5 (PEMDA)

10- 0000(0X2power3+0X2power2+0X2power1+0X2power0)

1010(1X2power3+ 0X2power2+1X2power1+0X2power0)0100

**PYTHON LIST**

# Python List

A list in Python is used to store the sequence of various types of data. Python lists are mutable type its mean we can modify its element after it created. However, Python consists of six data-types that are capable to store the sequences, but the most common and reliable type is the list.

A list can be defined as a collection of values or items of different types. The items in the list are separated with the comma (,) and enclosed with the square brackets [].

A list can be define as below

1. L1 = ["John", 102, "USA"]
2. L2 = [1, 2, 3, 4, 5, 6]

IIf we try to print the type of L1, L2, and L3 using type() function then it will come out to be a list.

7.1M

104

SQL CREATE TABLE

1. **print**(type(L1))
2. **print**(type(L2))

**Output:**

<class 'list'><class 'list'>

### Characteristics of Lists

The list has the following characteristics:

* The lists are ordered.
* The element of the list can access by index.
* The lists are the mutable type.
* The lists are mutable types.
* A list can store the number of various elements.

Let's check the first statement that lists are the ordered.

1. a = [1,2,"Peter",4.50,"Ricky",5,6]
2. b = [1,2,5,"Peter",4.50,"Ricky",6]
3. a ==b

**Output:**

False

Both lists have consisted of the same elements, but the second list changed the index position of the 5th element that violates the order of lists. When compare both lists it returns the false.

Lists maintain the order of the element for the lifetime. That's why it is the ordered collection of objects.

1. a = [1, 2,"Peter", 4.50,"Ricky",5, 6]
2. b = [1, 2,"Peter", 4.50,"Ricky",5, 6]
3. a == b

**Output:**

True

Let's have a look at the list example in detail.

1. emp = ["John", 102, "USA"]
2. Dep1 = ["CS",10]
3. Dep2 = ["IT",11]
4. HOD\_CS = [10,"Mr. Holding"]
5. HOD\_IT = [11, "Mr. Bewon"]
6. **print**("printing employee data...")
7. **print**("Name : %s, ID: %d, Country: %s"%(emp[0],emp[1],emp[2]))
8. **print**("printing departments...")
9. **print**("Department 1:\nName: %s, ID: %d\nDepartment 2:\nName: %s, ID: %s"%(Dep1[0],Dep2[1],Dep2[0],Dep2[1]))
10. **print**("HOD Details ....")
11. **print**("CS HOD Name: %s, Id: %d"%(HOD\_CS[1],HOD\_CS[0]))
12. **print**("IT HOD Name: %s, Id: %d"%(HOD\_IT[1],HOD\_IT[0]))
13. **print**(type(emp),type(Dep1),type(Dep2),type(HOD\_CS),type(HOD\_IT))

**Output:**

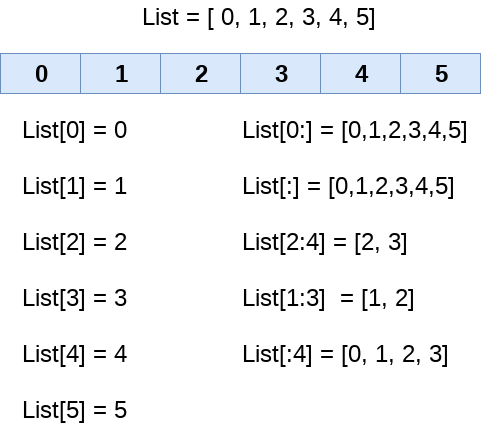
printing employee data...Name : John, ID: 102, Country: USAprinting departments...Department 1:Name: CS, ID: 11Department 2:Name: IT, ID: 11HOD Details ....CS HOD Name: Mr. Holding, Id: 10IT HOD Name: Mr. Bewon, Id: 11<class 'list'> <class 'list'> <class 'list'> <class 'list'> <class 'list'>

In the above example, we have created the lists which consist of the employee and department details and printed the corresponding details. Observe the above code to understand the concept of the list better.

## List indexing and splitting

The indexing is processed in the same way as it happens with the strings. The elements of the list can be accessed by using the slice operator [].

The index starts from 0 and goes to length - 1. The first element of the list is stored at the 0th index, the second element of the list is stored at the 1st index, and so on.



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

1. list\_varible(start:stop:step)

* The **start** denotes the starting index position of the list.
* The **stop** denotes the last index position of the list.
* The **step** is used to skip the nth element within a **start:stop**

Consider the following example:

1. list = [1,2,3,4,5,6,7]
2. **print**(list[0])
3. **print**(list[1])
4. **print**(list[2])
5. **print**(list[3])
6. # Slicing the elements
7. **print**(list[0:6])
8. # By default the index value is 0 so its starts from the 0th element and go for index -1.
9. **print**(list[:])
10. **print**(list[2:5])
11. **print**(list[1:6:2])

**Output:**

1234[1, 2, 3, 4, 5, 6][1, 2, 3, 4, 5, 6, 7][3, 4, 5][2, 4, 6]

Unlike other languages, Python provides the flexibility to use the negative indexing also. The negative indices are counted from the right. The last element (rightmost) of the list has the index -1; its adjacent left element is present at the index -2 and so on until the left-most elements are encountered.

Python Lists

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

1. list = [1,2,3,4,5]
2. **print**(list[-1])
3. **print**(list[-3:])
4. **print**(list[:-1])
5. **print**(list[-3:-1])

**Output:**

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

As we discussed above, we can get an element by using negative indexing. In the above code, the first print statement returned the rightmost element of the list. The second print statement returned the sub-list, and so on.

## Updating List values

Lists are the most versatile data structures in Python since they are mutable, and their values can be updated by using the slice and assignment operator.

Python also provides append() and insert() methods, which can be used to add values to the list.

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

1. list = [1, 2, 3, 4, 5, 6]
2. **print**(list)
3. # It will assign value to the value to the second index
4. list[2] = 10
5. **print**(list)
6. # Adding multiple-element
7. list[1:3] = [89, 78]
8. **print**(list)
9. # It will add value at the end of the list
10. list[-1] = 25
11. **print**(list)

**Output:**

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

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

Consider the following example to delete the list elements.

1. list = [1, 2, 3, 4, 5, 6]
2. **print**(list)
3. # It will assign value to the value to second index
4. list[2] = 10
5. **print**(list)
6. # Adding multiple element
7. list[1:3] = [89, 78]
8. **print**(list)
9. # It will add value at the end of the list
10. list[-1] = 25
11. **print**(list)

**Output:**

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

## Python List Operations

The concatenation (+) and repetition (\*) operators work in the same way as they were working with the strings.

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

1. Consider a Lists l1 = [1, 2, 3, 4], **and** l2 = [5, 6, 7, 8] to perform operation.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| Repetition | The repetition operator enables the list elements to be repeated multiple times. | L1\*2 = [1, 2, 3, 4, 1, 2, 3, 4] |
| Concatenation | It concatenates the list mentioned on either side of the operator. | l1+l2 = [1, 2, 3, 4, 5, 6, 7, 8] |
| Membership | It returns true if a particular item exists in a particular list otherwise false. | print(2 in l1) prints True. |
| Iteration | The for loop is used to iterate over the list elements. | for i in l1: print(i)**Output**1234 |
| Length | It is used to get the length of the list | len(l1) = 4 |

## Iterating a List

A list can be iterated by using a for - in loop. A simple list containing four strings, which can be iterated as follows.

1. list = ["John", "David", "James", "Jonathan"]
2. **for** i **in** list:
3. # The i variable will iterate over the elements of the List and contains each element in each iteration.
4. **print**(i)

**Output:**

JohnDavidJamesJonathan

## Adding elements to the list

Python provides append() function which is used to add an element to the list. However, the append() function can only add value to the end of the list.

Consider the following example in which, we are taking the elements of the list from the user and printing the list on the console.

1. #Declaring the empty list
2. l =[]
3. #Number of elements will be entered by the user
4. n = int(input("Enter the number of elements in the list:"))
5. # for loop to take the input
6. **for** i **in** range(0,n):
7. # The input is taken from the user and added to the list as the item
8. l.append(input("Enter the item:"))
9. **print**("printing the list items..")
10. # traversal loop to print the list items
11. **for** i **in** l:
12. **print**(i, end = " ")

**Output:**

Enter the number of elements in the list:5Enter the item:25Enter the item:46Enter the item:12Enter the item:75Enter the item:42printing the list items25 46 12 75 42

## Removing elements from the list

Python provides the **remove()** function which is used to remove the element from the list. Consider the following example to understand this concept.

**Example -**

1. list = [0,1,2,3,4]
2. **print**("printing original list: ");
3. **for** i **in** list:
4. **print**(i,end=" ")
5. list.remove(2)
6. **print**("\nprinting the list after the removal of first element...")
7. **for** i **in** list:
8. **print**(i,end=" ")

**Output:**

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

## Python List Built-in functions

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

|  |  |  |  |
| --- | --- | --- | --- |
| **SN** | **Function** | **Description** | **Example** |
| 1 | cmp(list1, list2) | It compares the elements of both the lists. | This method is not used in the Python 3 and the above versions. |
| 2 | len(list) | It is used to calculate the length of the list. | L1 = [1,2,3,4,5,6,7,8]print(len(L1)) 8 |
| 3 | max(list) | It returns the maximum element of the list. | L1 = [12,34,26,48,72]print(max(L1))72 |
| 4 | min(list) | It returns the minimum element of the list. | L1 = [12,34,26,48,72]print(min(L1))12 |
| 5 | list(seq) | It converts any sequence to the list. | str = "Johnson"s = list(str)print(type(s))<class list> |

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

**Example: 1-** Write the program to remove the duplicate element of the list.

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

**Output:**

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

**Example:2-** Write a program to find the sum of the element in the list.

1. list1 = [3,4,5,9,10,12,24]
2. sum = 0
3. **for** i **in** list1:
4. sum = sum+i
5. **print**("The sum is:",sum)

**Output:**

The sum is: 67

**Example: 3-** Write the program to find the lists consist of at least one common element.

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

**Output:**

The common element is: 2

# **Python Set**

1. A Python set is the collection of the unordered items.
2. Each element in the set must be **unique**, **immutable**, and the sets remove the **duplicate** elements.
3. Sets are mutable which means we can modify it after its creation.

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

## Creating a set

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

### Example 1: Using curly braces

1. Days = {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"}
2. **print**(Days)
3. **print**(type(Days))
4. **print**("looping through the set elements ... ")
5. **for** i **in** Days:
6. **print**(i)

**Output:**

861.1K

Darwin Carpet Pythons on the Balcony

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

### Example 2: Using set() method

1. Days = set(["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"])
2. **print**(Days)
3. **print**(type(Days))
4. **print**("looping through the set elements ... ")
5. **for** i **in** Days:
6. **print**(i)

**Output:**

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

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

1. # Creating a set which have immutable elements
2. set1 = {1,2,3, "JavaTpoint", 20.5, 14}
3. **print**(type(set1))
4. #Creating a set which have mutable element
5. set2 = {1,2,3,["Javatpoint",4]}
6. **print**(type(set2))

**Output:**

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

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

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

1. # Empty curly braces will create dictionary
2. set3 = {}
3. **print**(type(set3))
5. # Empty set using set() function
6. set4 = set()
7. **print**(type(set4))

**Output:**

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

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

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

**Output:**

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

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

## Adding items to the set

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

### Example: 1 - Using add() method

1. Months = set(["January","February", "March", "April", "May", "June"])
2. **print**("\nprinting the original set ... ")
3. **print**(months)
4. **print**("\nAdding other months to the set...");
5. Months.add("July");
6. Months.add ("August");
7. **print**("\nPrinting the modified set...");
8. **print**(Months)
9. **print**("\nlooping through the set elements ... ")
10. **for** i **in** Months:
11. **print**(i)

**Output:**

printing the original set ... {'February', 'May', 'April', 'March', 'June', 'January'}Adding other months to the set...Printing the modified set...{'February', 'July', 'May', 'April', 'March', 'August', 'June', 'January'}looping through the set elements ... FebruaryJulyMayAprilMarchAugustJuneJanuary

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

Consider the following example.

### Example - 2 Using update() function

1. Months = set(["January","February", "March", "April", "May", "June"])
2. **print**("\nprinting the original set ... ")
3. **print**(Months)
4. **print**("\nupdating the original set ... ")
5. Months.update(["July","August","September","October"]);
6. **print**("\nprinting the modified set ... ")
7. **print**(Months);

**Output:**

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

## Removing items from the set

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

Consider the following example.

### Example-1 Using discard() method

1. months = set(["January","February", "March", "April", "May", "June"])
2. **print**("\nprinting the original set ... ")
3. **print**(months)
4. **print**("\nRemoving some months from the set...");
5. months.discard("January");
6. months.discard("May");
7. **print**("\nPrinting the modified set...");
8. **print**(months)
9. **print**("\nlooping through the set elements ... ")
10. **for** i **in** months:
11. **print**(i)

**Output:**

printing the original set ... {'February', 'January', 'March', 'April', 'June', 'May'}Removing some months from the set...Printing the modified set...{'February', 'March', 'April', 'June'}looping through the set elements ... FebruaryMarchAprilJune

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

### Example-2 Using remove() function

1. months = set(["January","February", "March", "April", "May", "June"])
2. **print**("\nprinting the original set ... ")
3. **print**(months)
4. **print**("\nRemoving some months from the set...");
5. months.remove("January");
6. months.remove("May");
7. **print**("\nPrinting the modified set...");
8. **print**(months)

**Output:**

printing the original set ... {'February', 'June', 'April', 'May', 'January', 'March'}Removing some months from the set...Printing the modified set...{'February', 'June', 'April', 'March'}

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

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

1. Months = set(["January","February", "March", "April", "May", "June"])
2. **print**("\nprinting the original set ... ")
3. **print**(Months)
4. **print**("\nRemoving some months from the set...");
5. Months.pop();
6. Months.pop();
7. **print**("\nPrinting the modified set...");
8. **print**(Months)

**Output:**

printing the original set ... {'June', 'January', 'May', 'April', 'February', 'March'}Removing some months from the set...Printing the modified set...{'May', 'April', 'February', 'March'}

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

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

Consider the following example.

1. Months = set(["January","February", "March", "April", "May", "June"])
2. **print**("\nprinting the original set ... ")
3. **print**(Months)
4. **print**("\nRemoving all the items from the set...");
5. Months.clear()
6. **print**("\nPrinting the modified set...")
7. **print**(Months)

**Output:**

printing the original set ... {'January', 'May', 'June', 'April', 'March', 'February'}Removing all the items from the set...Printing the modified set...set()

## Difference between discard() and remove()

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

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

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

Consider the following example.

### Example-

1. Months = set(["January","February", "March", "April", "May", "June"])
2. **print**("\nprinting the original set ... ")
3. **print**(Months)
4. **print**("\nRemoving items through discard() method...");
5. Months.discard("Feb"); #will not give an error although the key feb is not available in the set
6. **print**("\nprinting the modified set...")
7. **print**(Months)
8. **print**("\nRemoving items through remove() method...");
9. Months.remove("Jan") #will give an error as the key jan is not available in the set.
10. **print**("\nPrinting the modified set...")
11. **print**(Months)

**Output:**

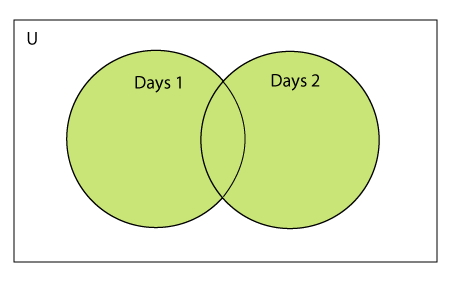
printing the original set ... {'March', 'January', 'April', 'June', 'February', 'May'}Removing items through discard() method...printing the modified set...{'March', 'January', 'April', 'June', 'February', 'May'}Removing items through remove() method...Traceback (most recent call last): File "set.py", line 9, in Months.remove("Jan")KeyError: 'Jan'

## Python Set Operations

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

### Union of two Sets

The union of two sets is calculated by using the pipe (|) operator. The union of the two sets contains all the items that are present in both the sets.



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

**Example 1: using union | operator**

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

**Output:**

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

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

**Example 2: using union() method**

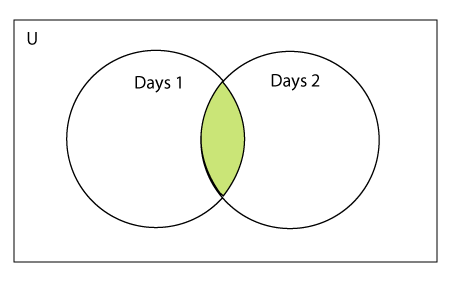
1. Days1 = {"Monday","Tuesday","Wednesday","Thursday"}
2. Days2 = {"Friday","Saturday","Sunday"}
3. **print**(Days1.union(Days2)) #printing the union of the sets

**Output:**

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

### Intersection of two sets

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



Consider the following example.

**Example 1: Using & operator**

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

**Output:**

{'Monday', 'Tuesday'}

**Example 2: Using intersection() method**

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

**Output:**

{'Martin', 'David'}

**Example 3:**

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

**Output:**

{1,2,5}

## The intersection\_update() method

The **intersection\_update()** method removes the items from the original set that are not present in both the sets (all the sets if more than one are specified).

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

Consider the following example.

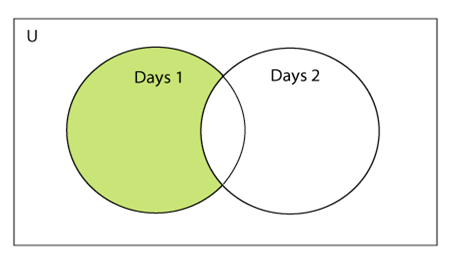
1. a = {"Devansh", "bob", "castle"}
2. b = {"castle", "dude", "emyway"}
3. c = {"fuson", "gaurav", "castle"}
5. a.intersection\_update(b, c)
7. **print**(a)

**Output:**

{'castle'}

## Difference between the two sets

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



Consider the following example.

**Example 1 : Using subtraction ( - ) operator**

1. Days1 = {"Monday", "Tuesday", "Wednesday", "Thursday"}
2. Days2 = {"Monday", "Tuesday", "Sunday"}
3. **print**(Days1-Days2) #{"Wednesday", "Thursday" will be printed}

**Output:**

{'Thursday', 'Wednesday'}

**Example 2 : Using difference() method**

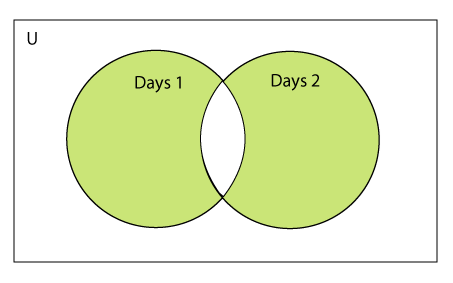
1. Days1 = {"Monday", "Tuesday", "Wednesday", "Thursday"}
2. Days2 = {"Monday", "Tuesday", "Sunday"}
3. **print**(Days1.difference(Days2)) # prints the difference of the two sets Days1 and Days2

**Output:**

{'Thursday', 'Wednesday'}

## Symmetric Difference of two sets

The symmetric difference of two sets is calculated by ^ operator or **symmetric\_difference()** method. Symmetric difference of sets, it removes that element which is present in both sets. Consider the following example:



**Example - 1: Using ^ operator**

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

**Output:**

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

**Example - 2: Using symmetric\_difference() method**

1. a = {1,2,3,4,5,6}
2. b = {1,2,9,8,10}
3. c = a.symmetric\_difference(b)
4. **print**(c)

**Output:**

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

## Set comparisons

Python allows us to use the comparison operators i.e., <, >, <=, >= , == with the sets by using which we can check whether a set is a subset, superset, or equivalent to other set. The boolean true or false is returned depending upon the items present inside the sets.

Consider the following example.

1. Days1 = {"Monday", "Tuesday", "Wednesday", "Thursday"}
2. Days2 = {"Monday", "Tuesday"}
3. Days3 = {"Monday", "Tuesday", "Friday"}
5. #Days1 is the superset of Days2 hence it will print true.
6. **print** (Days1>Days2)
8. #prints false since Days1 is not the subset of Days2
9. **print** (Days1<Days2)
11. #prints false since Days2 and Days3 are not equivalent
12. **print** (Days2 == Days3)

**Output:**

TrueFalseFalse

## FrozenSets

The frozen sets are the immutable form of the normal sets, i.e., the items of the frozen set cannot be changed and therefore it can be used as a key in the dictionary.

The elements of the frozen set cannot be changed after the creation. We cannot change or append the content of the frozen sets by using the methods like add() or remove().

The frozenset() method is used to create the frozenset object. The iterable sequence is passed into this method which is converted into the frozen set as a return type of the method.

Consider the following example to create the frozen set.

1. Frozenset = frozenset([1,2,3,4,5])
2. **print**(type(Frozenset))
3. **print**("\nprinting the content of frozen set...")
4. **for** i **in** Frozenset:
5. **print**(i);
6. Frozenset.add(6) #gives an error since we cannot change the content of Frozenset after creation

**Output:**

<class 'frozenset'>printing the content of frozen set...12345Traceback (most recent call last): File "set.py", line 6, in <module> Frozenset.add(6) #gives an error since we can change the content of Frozenset after creation AttributeError: 'frozenset' object has no attribute 'add'

## Frozenset for the dictionary

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

Consider the following example.

1. Dictionary = {"Name":"John", "Country":"USA", "ID":101}
2. **print**(type(Dictionary))
3. Frozenset = frozenset(Dictionary); #Frozenset will contain the keys of the dictionary
4. **print**(type(Frozenset))
5. **for** i **in** Frozenset:
6. **print**(i)

**Output:**

<class 'dict'><class 'frozenset'>NameCountryID

### Set Programming Example

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

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

**Output:**

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

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

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

**Output:**

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

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

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

**Output:**

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

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

1. set1 = {23,44,56,67,90,45,"Javatpoint"}
2. set2 = {13,23,56,76,"Sachin"}
3. set3 = set1.intersection(set2)
4. **print**(set3)

**Output:**

{56, 23}

**Example - 5:** Write the program to add element to the frozenset.

1. set1 = {23,44,56,67,90,45,"Javatpoint"}
2. set2 = {13,23,56,76,"Sachin"}
3. set3 = set1.intersection(set2)
4. **print**(set3)

**Output:**

TypeError: 'frozenset' object does not support item assignment

Above code raised an error because frozensets are immutable and can't be changed after creation.

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

1. set1 = set(["Peter","James","Camroon","Ricky","Donald"])
2. set2 = set(["Camroon","Washington","Peter"])
3. set3 = set(["Peter"])
5. issubset = set1 >= set2
6. **print**(issubset)
7. issuperset = set1 <= set2
8. **print**(issuperset)
9. issubset = set3 <= set2
10. **print**(issubset)
11. issuperset = set2 >= set3
12. **print**(issuperset)

**Output:**

FalseFalseTrueTrue

## Python Built-in set methods

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

|  |  |  |
| --- | --- | --- |
| **SN** | **Method** | **Description** |
| 1 | [add(item)](https://www.javatpoint.com/python-set-add-method) | It adds an item to the set. It has no effect if the item is already present in the set. |
| 2 | clear() | It deletes all the items from the set. |
| 3 | copy() | It returns a shallow copy of the set. |
| 4 | difference\_update(....) | It modifies this set by removing all the items that are also present in the specified sets. |
| 5 | [discard(item)](https://www.javatpoint.com/python-set-discard-method) | It removes the specified item from the set. |
| 6 | intersection() | It returns a new set that contains only the common elements of both the sets. (all the sets if more than two are specified). |
| 7 | intersection\_update(....) | It removes the items from the original set that are not present in both the sets (all the sets if more than one are specified). |
| 8 | Isdisjoint(....) | Return True if two sets have a null intersection. |
| 9 | Issubset(....) | Report whether another set contains this set. |
| 10 | Issuperset(....) | Report whether this set contains another set. |
| 11 | [pop()](https://www.javatpoint.com/python-set-pop-method) | Remove and return an arbitrary set element that is the last element of the set. Raises KeyError if the set is empty. |
| 12 | [remove(item)](https://www.javatpoint.com/python-set-remove-method) | Remove an element from a set; it must be a member. If the element is not a member, raise a KeyError. |
| 13 | symmetric\_difference(....) | Remove an element from a set; it must be a member. If the element is not a member, raise a KeyError. |
| 14 | symmetric\_difference\_update(....) | Update a set with the symmetric difference of itself and another. |
| 15 | union(....) | Return the union of sets as a new set. (i.e. all elements that are in either set.) |
| 16 | update() | Update a set with the union of itself and others. |

# Python Tuple

Python Tuple is used to store the sequence of **immutable Python objects**. The tuple is similar to lists since the value of the items stored in the list can be changed, whereas the tuple is immutable, and the value of the items stored in the tuple cannot be changed.

## Creating a tuple

A tuple can be written as the collection of comma-separated (,) values enclosed with the small () brackets. The parentheses are optional but it is good practice to use. A tuple can be defined as follows.

1. T1 = (101, "Peter", 22)
2. T2 = ("Apple", "Banana", "Orange")
3. T3 = 10,20,30,40,50
5. print(type(T1))
6. print(type(T2))
7. print(type(T3))

**Output:**

<class 'tuple'><class 'tuple'><class 'tuple'>

#### Note: The tuple which is created without using parentheses is also known as tuple packing.

An empty tuple can be created as follows.

T4 = ()

Creating a tuple with single element is slightly different. We will need to put comma after the element to declare the tuple.

1. tup1 = ("JavaTpoint")
2. print(type(tup1))
3. #Creating a tuple with single element
4. tup2 = ("JavaTpoint",)
5. print(type(tup2))

**Output:**

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

A tuple is indexed in the same way as the lists. The items in the tuple can be accessed by using their specific index value.

Consider the following example of tuple:

### Example - 1

1. tuple1 = (10, 20, 30, 40, 50, 60)
2. print(tuple1)
3. count = 0
4. **for** i in tuple1:
5. print("tuple1[%d] = %d"%(count, i))
6. count = count+1

**Output:**

(10, 20, 30, 40, 50, 60)tuple1[0] = 10tuple1[1] = 20tuple1[2] = 30tuple1[3] = 40tuple1[4] = 50tuple1[5] = 60

### Example - 2

1. tuple1 = tuple(input("Enter the tuple elements ..."))
2. print(tuple1)
3. count = 0
4. **for** i in tuple1:
5. print("tuple1[%d] = %s"%(count, i))
6. count = count+1

**Output:**

Enter the tuple elements ...123456('1', '2', '3', '4', '5', '6')tuple1[0] = 1tuple1[1] = 2tuple1[2] = 3tuple1[3] = 4tuple1[4] = 5tuple1[5] = 6

A tuple is indexed in the same way as the lists. The items in the tuple can be accessed by using their specific index value.

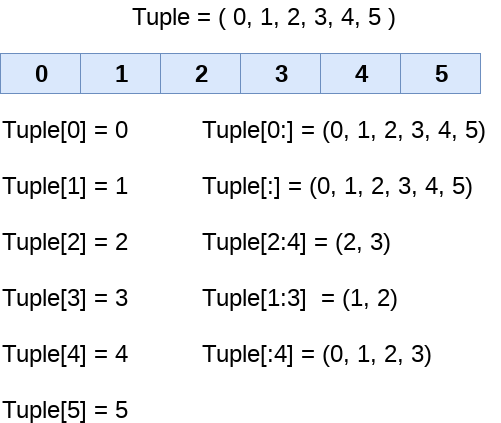
We will see all these aspects of tuple in this section of the tutorial.

## Tuple indexing and slicing

The indexing and slicing in the tuple are similar to lists. The indexing in the tuple starts from 0 and goes to length(tuple) - 1.

The items in the tuple can be accessed by using the index [] operator. Python also allows us to use the colon operator to access multiple items in the tuple.

Consider the following image to understand the indexing and slicing in detail.



Consider the following example:

1. tup = (1,2,3,4,5,6,7)
2. print(tup[0])
3. print(tup[1])
4. print(tup[2])
5. # It will give the IndexError
6. print(tup[8])

**Output:**

123tuple index out of range

In the above code, the tuple has 7 elements which denote 0 to 6. We tried to access an element outside of tuple that raised an **IndexError**.

1. tuple = (1,2,3,4,5,6,7)
2. #element 1 to end
3. print(tuple[1:])
4. #element 0 to 3 element
5. print(tuple[:4])
6. #element 1 to 4 element
7. print(tuple[1:5])
8. # element 0 to 6 and take step of 2
9. print(tuple[0:6:2])

**Output:**

(2, 3, 4, 5, 6, 7)(1, 2, 3, 4)(1, 2, 3, 4)(1, 3, 5)

## Negative Indexing

The tuple element can also access by using negative indexing. The index of -1 denotes the rightmost element and -2 to the second last item and so on.

The elements from left to right are traversed using the negative indexing. Consider the following example:

1. tuple1 = (1, 2, 3, 4, 5)
2. print(tuple1[-1])
3. print(tuple1[-4])
4. print(tuple1[-3:-1])
5. print(tuple1[:-1])
6. print(tuple1[-2:])

**Output:**

52(3, 4)(1, 2, 3, 4)(4, 5)

## Deleting Tuple

Unlike lists, the tuple items cannot be deleted by using the **del** keyword as tuples are immutable. To delete an entire tuple, we can use the **del** keyword with the tuple name.

Consider the following example.

1. tuple1 = (1, 2, 3, 4, 5, 6)
2. print(tuple1)
3. del tuple1[0]
4. print(tuple1)
5. del tuple1
6. print(tuple1)

**Output:**

(1, 2, 3, 4, 5, 6)Traceback (most recent call last): File "tuple.py", line 4, in <module> print(tuple1)NameError: name 'tuple1' is not defined

## Basic Tuple operations

The operators like concatenation (+), repetition (\*), Membership (in) works in the same way as they work with the list. Consider the following table for more detail.

Let's say Tuple t = (1, 2, 3, 4, 5) and Tuple t1 = (6, 7, 8, 9) are declared.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| Repetition | The repetition operator enables the tuple elements to be repeated multiple times. | T1\*2 = (1, 2, 3, 4, 5, 1, 2, 3, 4, 5) |
| Concatenation | It concatenates the tuple mentioned on either side of the operator. | T1+T2 = (1, 2, 3, 4, 5, 6, 7, 8, 9) |
| Membership | It returns true if a particular item exists in the tuple otherwise false | print (2 in T1) prints True. |
| Iteration | The for loop is used to iterate over the tuple elements. | for i in T1: print(i)**Output**12345 |
| Length | It is used to get the length of the tuple. | len(T1) = 5 |

## Gurpreet

## Python Tuple inbuilt functions

|  |  |  |
| --- | --- | --- |
| **SN** | **Function** | **Description** |
| 1 | cmp(tuple1, tuple2) | It compares two tuples and returns true if tuple1 is greater than tuple2 otherwise false. |
| 2 | len(tuple) | It calculates the length of the tuple. |
| 3 | max(tuple) | It returns the maximum element of the tuple |
| 4 | min(tuple) | It returns the minimum element of the tuple. |
|  |  |  |
| 5 | tuple(seq) | It converts the specified sequence to the tuple. |

## Where use tuple?

Using tuple instead of list is used in the following scenario.

1. Using tuple instead of list gives us a clear idea that tuple data is constant and must not be changed.

2. Tuple can simulate a dictionary without keys. Consider the following nested structure, which can be used as a dictionary.

1. [(101, "John", 22), (102, "Mike", 28), (103, "Dustin", 30)]

## List vs. Tuple

|  |  |  |
| --- | --- | --- |
| **SN** | **List** | **Tuple** |
| 1 | The literal syntax of list is shown by the []. | The literal syntax of the tuple is shown by the (). |
| 2 | The List is mutable. | The tuple is immutable. |
| 3 | The List has the a variable length. | The tuple has the fixed length. |
| 4 | The list provides more functionality than a tuple. | The tuple provides less functionality than the list. |
| 5 | The list is used in the scenario in which we need to store the simple collections with no constraints where the value of the items can be changed. | The tuple is used in the cases where we need to store the read-only collections i.e., the value of the items cannot be changed. It can be used as the key inside the dictionary. |
| 6 | The lists are less memory efficient than a tuple. | The tuples are more memory efficient because of its immutability. |

# Python Function

Functions are the most important aspect of an application. A function can be defined as the organized block of **reusable code**, which can be called whenever required.

Python allows us to divide a large program into the basic building blocks known as a function. The function contains the set of programming statements enclosed by {}. A function can be called multiple times to provide reusability and **modularity** to the Python program.

The Function helps to programmer to break the program into the smaller part. It organizes the code very effectively and avoids the repetition of the code. As the program grows, function makes the program more organized.

Python provide us various inbuilt functions like **range()** or **print()**. Although, the user can create its functions, which can be called user-defined functions.

10 Sec

Java Try Catch

There are mainly two types of functions.

* **User-define functions** - The user-defined functions are those define by the **user** to perform the specific task.
* **Built-in functions** - The built-in functions are those functions that are **pre-defined** in Python.

In this tutorial, we will discuss the user define functions.

## Advantage of Functions in Python

There are the following advantages of Python functions.

* Using functions, we can avoid rewriting the same logic/code again and again in a program.
* We can call Python functions multiple times in a program and anywhere in a program.
* We can track a large Python program easily when it is divided into multiple functions.
* **Reusability** is the main achievement of Python functions.
* However, Function calling is always overhead in a Python program.

### Creating a Function

Python provides the **def** keyword to define the function. The syntax of the define function is given below.

**Syntax:**

1. **def** my\_function(parameters):
2. function\_block
3. **return** expression

Let's understand the syntax of functions definition.

* The **def** keyword, along with the function name is used to define the function.
* The identifier rule must follow the function name.
* A function accepts the parameter (argument), and they can be optional.
* The function block is started with the colon (:), and block statements must be at the same indentation.
* The **return** statement is used to return the value. A function can have only one **return**

### Function Calling

In Python, after the function is created, we can call it from another function. A function must be defined before the function call; otherwise, the Python interpreter gives an error. To call the function, use the function name followed by the parentheses.

Consider the following example of a simple example that prints the message "Hello World".

1. #function definition
2. **def** hello\_world():
3. **print**("hello world")
4. # function calling
5. hello\_world()

**Output:**

hello world

## The return statement

The return statement is used at the end of the function and returns the result of the function. It terminates the function execution and transfers the result where the function is called. The return statement cannot be used outside of the function.

**Syntax**

1. **return** [expression\_list]

It can contain the expression which gets evaluated and value is returned to the caller function. If the return statement has no expression or does not exist itself in the function then it returns the **None** object.

Consider the following example:

### Example 1

1. # Defining function
2. **def** sum():
3. a = 10
4. b = 20
5. c = a+b
6. **return** c
7. # calling sum() function in print statement
8. **print**("The sum is:",sum())

**Output:**

The sum is: 30

In the above code, we have defined the function named **sum,** and it has a statement **c = a+b,** which computes the given values, and the result is returned by the return statement to the caller function.

### Example 2 Creating function without return statement

1. # Defining function
2. **def** sum():
3. a = 10
4. b = 20
5. c = a+b
6. # calling sum() function in print statement
7. **print**(sum())

**Output:**

None

In the above code, we have defined the same function without the return statement as we can see that the **sum()** function returned the **None** object to the caller function.

## Arguments in function

The arguments are types of information which can be passed into the function. The arguments are specified in the parentheses. We can pass any number of arguments, but they must be separate them with a comma.

Consider the following example, which contains a function that accepts a string as the argument.

### Example 1

1. #defining the function
2. **def** func (name):
3. **print**("Hi ",name)
4. #calling the function
5. func("Devansh")

**Output:**

Hi Devansh

### Example 2

1. #Python function to calculate the sum of two variables
2. #defining the function
3. **def** sum (a,b):
4. **return** a+b;
6. #taking values from the user
7. a = int(input("Enter a: "))
8. b = int(input("Enter b: "))
10. #printing the sum of a and b
11. **print**("Sum = ",sum(a,b))

**Output:**

Enter a: 10Enter b: 20Sum = 30

## Call by reference in Python

In Python, call by reference means passing the actual value as an argument in the function. All the functions are called by reference, i.e., all the changes made to the reference inside the function revert back to the original value referred by the reference.

### Example 1 Passing Immutable Object (List)

1. #defining the function
2. **def** change\_list(list1):
3. list1.append(20)
4. list1.append(30)
5. **print**("list inside function = ",list1)
7. #defining the list
8. list1 = [10,30,40,50]
10. #calling the function
11. change\_list(list1)
12. **print**("list outside function = ",list1)

**Output:**

list inside function = [10, 30, 40, 50, 20, 30]list outside function = [10, 30, 40, 50, 20, 30]

## Example 2 Passing Mutable Object (String)

1. #defining the function
2. **def** change\_string (str):
3. str = str + " How are you "
4. **print**("printing the string inside function :",str)
6. string1 = "Hi I am there"
8. #calling the function
9. change\_string(string1)
11. **print**("printing the string outside function :",string1)

**Output:**

printing the string inside function : Hi I am there How are you printing the string outside function : Hi I am there

## Types of arguments

There may be several types of arguments which can be passed at the time of function call.

1. Required arguments
2. Keyword arguments
3. Default arguments
4. Variable-length arguments

### Required Arguments

Till now, we have learned about function calling in Python. However, we can provide the arguments at the time of the function call. As far as the required arguments are concerned, these are the arguments which are required to be passed at the time of function calling with the exact match of their positions in the function call and function definition. If either of the arguments is not provided in the function call, or the position of the arguments is changed, the Python interpreter will show the error.

Consider the following example.

**Example 1**

1. **def** func(name,str):
2. message = "Hi "+name
3. **return** message
4. name = input("Enter the name:")
5. **print**(func(name))

**Output:**

Enter the name: JohnHi John

**Example 2**

1. #the function simple\_interest accepts three arguments and returns the simple interest accordingly
2. **def** simple\_interest(p,t,r):
3. **return** (p\*t\*r)/100
4. p = float(input("Enter the principle amount? "))
5. r = float(input("Enter the rate of interest? "))
6. t = float(input("Enter the time in years? "))
7. **print**("Simple Interest: ",simple\_interest(p,r,t))

**Output:**

Enter the principle amount: 5000Enter the rate of interest: 5Enter the time in years: 3Simple Interest: 750.0

**Example 3**

1. #the function calculate returns the sum of two arguments a and b
2. **def** calculate(a,b):
3. **return** a+b
4. calculate(10) # this causes an error as we are missing a required arguments b.

**Output:**

TypeError: calculate() missing 1 required positional argument: 'b'

## Default Arguments

Python allows us to initialize the arguments at the function definition. If the value of any of the arguments is not provided at the time of function call, then that argument can be initialized with the value given in the definition even if the argument is not specified at the function call.

**Example 1**

1. **def** printme(name,age=22):
2. **print**("My name is",name,"and age is",age)
3. printme(name = "john")

**Output:**

My name is John and age is 22

**Example 2**

1. **def** printme(name,age=22):
2. **print**("My name is",name,"and age is",age)
3. printme(name = "john") #the variable age is not passed into the function however the default value of age is considered in the function
4. printme(age = 10,name="David") #the value of age is overwritten here, 10 will be printed as age

**Output:**

My name is john and age is 22My name is David and age is 10

### Variable-length Arguments (\*args)

In large projects, sometimes we may not know the number of arguments to be passed in advance. In such cases, Python provides us the flexibility to offer the comma-separated values which are internally treated as tuples at the function call. By using the variable-length arguments, we can pass any number of arguments.

However, at the function definition, we define the variable-length argument using the **\*args** (star) as \*<variable - name >.

Consider the following example.

**Example**

1. **def** printme(\*names):
2. **print**("type of passed argument is ",type(names))
3. **print**("printing the passed arguments...")
4. **for** name **in** names:
5. **print**(name)
6. printme("john","David","smith","nick")

**Output:**

type of passed argument is <class 'tuple'>printing the passed arguments...johnDavidsmithnick

In the above code, we passed **\*names** as variable-length argument. We called the function and passed values which are treated as tuple internally. The tuple is an iterable sequence the same as the list. To print the given values, we iterated **\*arg names** using for loop.

### Keyword arguments(\*\*kwargs)

Python allows us to call the function with the keyword arguments. This kind of function call will enable us to pass the arguments in the random order.

The **name of the arguments is treated as the keywords and matched in the function calling** and definition. If the same match is found, the values of the arguments are copied in the function definition.

Consider the following example.

**Example 1**

1. #function func is called with the name and message as the keyword arguments
2. **def** func(name,message):
3. **print**("printing the message with",name,"and ",message)
5. #name and message is copied with the values John and hello respectively
6. func(name = "John",message="hello")

**Output:**

printing the message with John and hello

**Example 2 providing the values in different order at the calling**

1. #The function simple\_interest(p, t, r) is called with the keyword arguments the order of arguments doesn't matter in this case
2. **def** simple\_interest(p,t,r):
3. **return** (p\*t\*r)/100
4. **print**("Simple Interest: ",simple\_interest(t=10,r=10,p=1900))

**Output:**

Simple Interest: 1900.0

If we provide the different name of arguments at the time of function call, an error will be thrown.

Consider the following example.

**Example 3**

1. #The function simple\_interest(p, t, r) is called with the keyword arguments.
2. **def** simple\_interest(p,t,r):
3. **return** (p\*t\*r)/100
5. # doesn't find the exact match of the name of the arguments (keywords)
6. **print**("Simple Interest: ",simple\_interest(time=10,rate=10,principle=1900))

**Output:**

TypeError: simple\_interest() got an unexpected keyword argument 'time'

The Python allows us to provide the mix of the required arguments and keyword arguments at the time of function call. However, the required argument must not be given after the keyword argument, i.e., once the keyword argument is encountered in the function call, the following arguments must also be the keyword arguments.

Consider the following example.

**Example 4**

1. **def** func(name1,message,name2):
2. **print**("printing the message with",name1,",",message,",and",name2)
3. #the first argument is not the keyword argument
4. func("John",message="hello",name2="David")

**Output:**

printing the message with John , hello ,and David

The following example will cause an error due to an in-proper mix of keyword and required arguments being passed in the function call.

**Example 5**

1. **def** func(name1,message,name2):
2. **print**("printing the message with",name1,",",message,",and",name2)
3. func("John",message="hello","David")

**Output:**

SyntaxError: positional argument follows keyword argument

Python provides the facility to pass the multiple keyword arguments which can be represented as **\*\*kwargs**. It is similar as the **\*args** but it stores the argument in the dictionary format.

This type of arguments is useful when we do not know the number of arguments in advance.

Consider the following example:

**Example 6: Many arguments using Keyword argument**

1. **def** food(\*\*kwargs):
2. **print**(kwargs)
3. food(a="Apple")
4. food(fruits="Orange", Vagitables="Carrot")

**Output:**

{'a': 'Apple'}{'fruits': 'Orange', 'Vagitables': 'Carrot'}

## Scope of variables

The scopes of the variables depend upon the location where the variable is being declared. The variable declared in one part of the program may not be accessible to the other parts.

In python, the variables are defined with the two types of scopes.

1. Global variables
2. Local variables

The variable defined outside any function is known to have a global scope, whereas the variable defined inside a function is known to have a local scope.

Consider the following example.

### Example 1 Local Variable

1. **def** print\_message():
2. message = "hello !! I am going to print a message." # the variable message is local to the function itself
3. **print**(message)
4. print\_message()
5. **print**(message) # this will cause an error since a local variable cannot be accessible here.

**Output:**

hello !! I am going to print a message. File "/root/PycharmProjects/PythonTest/Test1.py", line 5, in print(message)NameError: name 'message' is not defined

### Example 2 Global Variable

1. **def** calculate(\*args):
2. sum=0
3. **for** arg **in** args:
4. sum = sum +arg
5. **print**("The sum is",sum)
6. sum=0
7. calculate(10,20,30) #60 will be printed as the sum
8. **print**("Value of sum outside the function:",sum) # 0 will be printed Output:

**Output:**

The sum is 60Value of sum outside the function: 0

# Python Built-in Functions

The Python built-in functions are defined as the functions whose functionality is pre-defined in Python. The python interpreter has several functions that are always present for use. These functions are known as Built-in Functions. There are several built-in functions in Python which are listed below:

## Python abs() Function

The python **abs()** function is used to return the absolute value of a number. It takes only one argument, a number whose absolute value is to be returned. The argument can be an integer and floating-point number. If the argument is a complex number, then, abs() returns its magnitude.

**Python abs() Function Example**

1. # integer number
2. integer = -20
3. **print**('Absolute value of -40 is:', abs(integer))
5. # floating number
6. floating = -20.83
7. **print**('Absolute value of -40.83 is:', abs(floating))

**Output:**

11M

225

History of Java

Absolute value of -20 is: 20Absolute value of -20.83 is: 20.83

## Python all() Function

The python **all()** function accepts an iterable object (such as list, dictionary, etc.). It returns true if all items in passed iterable are true. Otherwise, it returns False. If the iterable object is empty, the all() function returns True.

**Python all() Function Example**

1. # all values true
2. k = [1, 3, 4, 6]
3. **print**(all(k))
5. # all values false
6. k = [0, False]
7. **print**(all(k))
9. # one false value
10. k = [1, 3, 7, 0]
11. **print**(all(k))
13. # one true value
14. k = [0, False, 5]
15. **print**(all(k))
17. # empty iterable
18. k = []
19. **print**(all(k))

**Output:**

TrueFalseFalseFalseTrue

## Python bin() Function

The python **bin()** function is used to return the binary representation of a specified integer. A result always starts with the prefix 0b.

**Python bin() Function Example**

1. x = 10
2. y = bin(x)
3. **print** (y)

**Output:**

0b1010

## Python bool()

The python **bool()** converts a value to boolean(True or False) using the standard truth testing procedure.

**Python bool() Example**

1. test1 = []
2. **print**(test1,'is',bool(test1))
3. test1 = [0]
4. **print**(test1,'is',bool(test1))
5. test1 = 0.0
6. **print**(test1,'is',bool(test1))
7. test1 = None
8. **print**(test1,'is',bool(test1))
9. test1 = True
10. **print**(test1,'is',bool(test1))
11. test1 = 'Easy string'
12. **print**(test1,'is',bool(test1))

**Output:**

[] is False[0] is True0.0 is FalseNone is FalseTrue is TrueEasy string is True

## Python bytes()

The python **bytes()** in Python is used for returning a **bytes** object. It is an immutable version of the bytearray() function.

It can create empty bytes object of the specified size.

**Python bytes() Example**

1. string = "Hello World."
2. array = bytes(string, 'utf-8')
3. **print**(array)

**Output:**

b ' Hello World.'

## Python callable() Function

A python **callable()** function in Python is something that can be called. This built-in function checks and returns true if the object passed appears to be callable, otherwise false.

**Python callable() Function Example**

1. x = 8
2. **print**(callable(x))

**Output:**

## Python compile() Function

The python **compile()** function takes source code as input and returns a code object which can later be executed by exec() function.

**Python compile() Function Example**

1. # compile string source to code
2. code\_str = 'x=5\ny=10\nprint("sum =",x+y)'
3. code = compile(code\_str, 'sum.py', 'exec')
4. **print**(type(code))
5. **exec**(code)
6. **exec**(x)

**Output:**

<class 'code'>sum = 15

## Python exec() Function

The python **exec()** function is used for the dynamic execution of Python program which can either be a string or object code and it accepts large blocks of code, unlike the eval() function which only accepts a single expression.

**Python exec() Function Example**

1. x = 8
2. **exec**('print(x==8)')
3. **exec**('print(x+4)')

**Output:**

True12

## Python sum() Function

As the name says, python **sum()** function is used to get the sum of numbers of an iterable, i.e., list.

**Python sum() Function Example**

1. s = sum([1, 2,4 ])
2. **print**(s)
4. s = sum([1, 2, 4], 10)
5. **print**(s)

**Output:**

717

## Python any() Function

The python **any()** function returns true if any item in an iterable is true. Otherwise, it returns False.

**Python any() Function Example**

1. l = [4, 3, 2, 0]
2. **print**(any(l))
4. l = [0, False]
5. **print**(any(l))
7. l = [0, False, 5]
8. **print**(any(l))
10. l = []
11. **print**(any(l))

**Output:**

TrueFalseTrueFalse

## Python ascii() Function

The python **ascii()** function returns a string containing a printable representation of an object and escapes the non-ASCII characters in the string using \x, \u or \U escapes.

**Python ascii() Function Example**

1. normalText = 'Python is interesting'
2. **print**(ascii(normalText))
4. otherText = 'Pythön is interesting'
5. **print**(ascii(otherText))
7. **print**('Pyth\xf6n is interesting')

**Output:**

'Python is interesting''Pyth\xf6n is interesting'Pythön is interesting

## Python bytearray()

The python **bytearray()** returns a bytearray object and can convert objects into bytearray objects, or create an empty bytearray object of the specified size.

**Python bytearray() Example**

1. string = "Python is a programming language."
3. # string with encoding 'utf-8'
4. arr = bytearray(string, 'utf-8')
5. **print**(arr)

**Output:**

bytearray(b'Python is a programming language.')

## Python eval() Function

The python **eval()** function parses the expression passed to it and runs python expression(code) within the program.

**Python eval() Function Example**

1. x = 8
2. **print**(eval('x + 1'))

**Output:**

9

## Python float()

The python **float()** function returns a floating-point number from a number or string.

**Python float() Example**

1. # for integers
2. **print**(float(9))
4. # for floats
5. **print**(float(8.19))
7. # for string floats
8. **print**(float("-24.27"))
10. # for string floats with whitespaces
11. **print**(float(" -17.19\n"))
13. # string float error
14. **print**(float("xyz"))

**Output:**

9.08.19-24.27-17.19ValueError: could not convert string to float: 'xyz'

## Python format() Function

The python **format()** function returns a formatted representation of the given value.

**Python format() Function Example**

1. # d, f and b are a type
3. # integer
4. **print**(format(123, "d"))
6. # float arguments
7. **print**(format(123.4567898, "f"))
9. # binary format
10. **print**(format(12, "b"))

**Output:**

123123.4567901100

## Python frozenset()

The python **frozenset()** function returns an immutable frozenset object initialized with elements from the given iterable.

**Python frozenset() Example**

1. # tuple of letters
2. letters = ('m', 'r', 'o', 't', 's')
4. fSet = frozenset(letters)
5. **print**('Frozen set is:', fSet)
6. **print**('Empty frozen set is:', frozenset())

**Output:**

Frozen set is: frozenset({'o', 'm', 's', 'r', 't'})Empty frozen set is: frozenset()

## Python getattr() Function

The python **getattr()** function returns the value of a named attribute of an object. If it is not found, it returns the default value.

**Python getattr() Function Example**

1. **class** Details:
2. age = 22
3. name = "Phill"
5. details = Details()
6. **print**('The age is:', getattr(details, "age"))
7. **print**('The age is:', details.age)

**Output:**

The age is: 22The age is: 22

Practice Problems For Functions

Q1. Write a Python function to sum all the numbers in a list

Q2. Write a Python program to reverse a string.

Q3. Write a Python function to calculate the factorial of a number (a non-negative integer). The function accepts the number as an argument

Q4 Write a Python function that takes a list and returns a new list with unique elements of the first list.

Q5. Write a Python program to print the even numbers from a given list

# Python Dictionary

Python Dictionary is used to store the data in a key-value pair format. The dictionary is the data type in Python, which can simulate the real-life data arrangement where some specific value exists for some particular key. It is the mutable data-structure. The dictionary is defined into element Keys and values.

* Keys must be a single element
* Value can be any type such as list, tuple, integer, etc.

In other words, we can say that a dictionary is the collection of key-value pairs where the value can be any Python object. In contrast, the keys are the immutable Python object, i.e., Numbers, string, or tuple.

## Creating the dictionary

The dictionary can be created by using multiple key-value pairs enclosed with the curly brackets {}, and each key is separated from its value by the colon (:).The syntax to define the dictionary is given below.

**Syntax:**

1. Dict = {"Name": "Tom", "Age": 22}

In the above dictionary **Dict**, The keys **Name** and **Age** are the string that is an immutable object.

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

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. **print**(type(Employee))
3. **print**("printing Employee data .... ")
4. **print**(Employee)

**Output**

<class 'dict'>Printing Employee data .... {'Name': 'John', 'Age': 29, 'salary': 25000, 'Company': 'GOOGLE'}

Python provides the built-in function **dict()** method which is also used to create dictionary. The empty curly braces {} is used to create empty dictionary.

1. # Creating an empty Dictionary
2. Dict = {}
3. **print**("Empty Dictionary: ")
4. **print**(Dict)
6. # Creating a Dictionary
7. # with dict() method
8. Dict = dict({1: 'Java', 2: 'T', 3:'Point'})
9. **print**("\nCreate Dictionary by using dict(): ")
10. **print**(Dict)
12. # Creating a Dictionary
13. # with each item as a Pair
14. Dict = dict([(1, 'Devansh'), (2, 'Sharma')])
15. **print**("\nDictionary with each item as a pair: ")
16. **print**(Dict)

**Output:**

Empty Dictionary: {}Create Dictionary by using dict(): {1: 'Java', 2: 'T', 3: 'Point'}Dictionary with each item as a pair: {1: 'Devansh', 2: 'Sharma'}

## Accessing the dictionary values

We have discussed how the data can be accessed in the list and tuple by using the indexing.

However, the values can be accessed in the dictionary by using the keys as keys are unique in the dictionary.

The dictionary values can be accessed in the following way.

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. **print**(type(Employee))
3. **print**("printing Employee data .... ")
4. **print**("Name : %s" %Employee["Name"])
5. **print**("Age : %d" %Employee["Age"])
6. **print**("Salary : %d" %Employee["salary"])
7. **print**("Company : %s" %Employee["Company"])

**Output:**

<class 'dict'>printing Employee data .... Name : JohnAge : 29Salary : 25000Company : GOOGLE

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

## Adding dictionary values

The dictionary is a mutable data type, and its values can be updated by using the specific keys. The value can be updated along with key **Dict[key] = value**. The update() method is also used to update an existing value.

Note: If the key-value already present in the dictionary, the value gets updated. Otherwise, the new keys added in the dictionary.

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

**Example - 1:**

1. # Creating an empty Dictionary
2. Dict = {}
3. **print**("Empty Dictionary: ")
4. **print**(Dict)
6. # Adding elements to dictionary one at a time
7. Dict[0] = 'Peter'
8. Dict[2] = 'Joseph'
9. Dict[3] = 'Ricky'
10. **print**("\nDictionary after adding 3 elements: ")
11. **print**(Dict)
13. # Adding set of values
14. # with a single Key
15. # The Emp\_ages doesn't exist to dictionary
16. Dict['Emp\_ages'] = 20, 33, 24
17. **print**("\nDictionary after adding 3 elements: ")
18. **print**(Dict)
20. # Updating existing Key's Value
21. Dict[3] = 'JavaTpoint'
22. **print**("\nUpdated key value: ")
23. **print**(Dict)

**Output:**

Empty Dictionary: {}Dictionary after adding 3 elements: {0: 'Peter', 2: 'Joseph', 3: 'Ricky'}Dictionary after adding 3 elements: {0: 'Peter', 2: 'Joseph', 3: 'Ricky', 'Emp\_ages': (20, 33, 24)}Updated key value: {0: 'Peter', 2: 'Joseph', 3: 'JavaTpoint', 'Emp\_ages': (20, 33, 24)}

**Example - 2:**

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. **print**(type(Employee))
3. **print**("printing Employee data .... ")
4. **print**(Employee)
5. **print**("Enter the details of the new employee....");
6. Employee["Name"] = input("Name: ");
7. Employee["Age"] = int(input("Age: "));
8. Employee["salary"] = int(input("Salary: "));
9. Employee["Company"] = input("Company:");
10. **print**("printing the new data");
11. **print**(Employee)

**Output:**

Empty Dictionary: {}Dictionary after adding 3 elements: {0: 'Peter', 2: 'Joseph', 3: 'Ricky'}Dictionary after adding 3 elements: {0: 'Peter', 2: 'Joseph', 3: 'Ricky', 'Emp\_ages': (20, 33, 24)}Updated key value: {0: 'Peter', 2: 'Joseph', 3: 'JavaTpoint', 'Emp\_ages': (20, 33, 24)}

## Deleting elements using del keyword

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

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. **print**(type(Employee))
3. **print**("printing Employee data .... ")
4. **print**(Employee)
5. **print**("Deleting some of the employee data")
6. **del** Employee["Name"]
7. **del** Employee["Company"]
8. **print**("printing the modified information ")
9. **print**(Employee)
10. **print**("Deleting the dictionary: Employee");
11. **del** Employee
12. **print**("Lets try to print it again ");
13. **print**(Employee)

**Output:**

<class 'dict'>printing Employee data .... {'Name': 'John', 'Age': 29, 'salary': 25000, 'Company': 'GOOGLE'}Deleting some of the employee dataprinting the modified information {'Age': 29, 'salary': 25000}Deleting the dictionary: EmployeeLets try to print it again NameError: name 'Employee' is not defined

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

* **Using pop() method**

The **pop()** method accepts the key as an argument and remove the associated value. Consider the following example.

1. # Creating a Dictionary
2. Dict = {1: 'JavaTpoint', 2: 'Peter', 3: 'Thomas'}
3. # Deleting a key
4. # using pop() method
5. pop\_ele = Dict.pop(3)
6. **print**(Dict)

**Output:**

{1: 'JavaTpoint', 2: 'Peter'}

Python also provides a built-in methods popitem() and clear() method for remove elements from the dictionary. The popitem() removes the arbitrary element from a dictionary, whereas the clear() method removes all elements to the whole dictionary.

## Iterating Dictionary

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

### Example 1

**# for loop to print all the keys of a dictionary**

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. **for** x **in** Employee:
3. **print**(x)

**Output:**

NameAgesalaryCompany

### Example 2

**#for loop to print all the values of the dictionary**

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. **for** x **in** Employee:
3. **print**(Employee[x])

**Output:**

John2925000GOOGLE

### Example - 3

**#for loop to print the values of the dictionary by using values() method.**

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. **for** x **in** Employee.values():
3. **print**(x)

**Output:**

John2925000GOOGLE

### Example 4

**#for loop to print the items of the dictionary by using items() method.**

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. **for** x **in** Employee.items():
3. **print**(x)

**Output:**

('Name', 'John')('Age', 29)('salary', 25000)('Company', 'GOOGLE')

## Properties of Dictionary keys

1. In the dictionary, we cannot store multiple values for the same keys. If we pass more than one value for a single key, then the value which is last assigned is considered as the value of the key.

Consider the following example.

1. Employee={"Name":"John","Age":29,"Salary":25000,"Company":"GOOGLE","Name":"John"}
2. **for** x,y **in** Employee.items():
3. **print**(x,y)

**Output:**

Name JohnAge 29Salary 25000Company GOOGLE

2. In python, the key cannot be any mutable object. We can use numbers, strings, or tuples as the key, but we cannot use any mutable object like the list as the key in the dictionary.

Consider the following example.

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE",[100,201,301]:"Department ID"}
2. **for** x,y **in** Employee.items():
3. **print**(x,y)

**Output:**

Traceback (most recent call last): File "dictionary.py", line 1, in Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE",[100,201,301]:"Department ID"}TypeError: unhashable type: 'list'

## Built-in Dictionary functions

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

|  |  |  |
| --- | --- | --- |
| **SN** | **Function** | **Description** |
| 1 | cmp(dict1, dict2) | It compares the items of both the dictionary and returns true if the first dictionary values are greater than the second dictionary, otherwise it returns false. |
| 2 | len(dict) | It is used to calculate the length of the dictionary. |
| 3 | str(dict) | It converts the dictionary into the printable string representation. |
| 4 | type(variable) | It is used to print the type of the passed variable. |

## Built-in Dictionary methods

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

|  |  |  |
| --- | --- | --- |
| **SN** | **Method** | **Description** |
| 1 | [dic.clear()](https://www.javatpoint.com/python-dictionary-clear-method) | It is used to delete all the items of the dictionary. |
| 2 | [dict.copy()](https://www.javatpoint.com/python-dictionary-copy-method) | It returns a shallow copy of the dictionary. |
| 3 | [dict.fromkeys(iterable, value = None, /)](https://www.javatpoint.com/python-dictionary-fromkeys-method) | Create a new dictionary from the iterable with the values equal to value. |
| 4 | [dict.get(key, default = "None")](https://www.javatpoint.com/python-dictionary-get-method) | It is used to get the value specified for the passed key. |
| 5 | dict.has\_key(key) | It returns true if the dictionary contains the specified key. |
| 6 | [dict.items()](https://www.javatpoint.com/python-dictionary-items-method) | It returns all the key-value pairs as a tuple. |
| 7 | [dict.keys()](https://www.javatpoint.com/python-dictionary-keys-method) | It returns all the keys of the dictionary. |
| 8 | [dict.setdefault(key,default= "None")](https://www.javatpoint.com/python-dictionary-setdefault-method) | It is used to set the key to the default value if the key is not specified in the dictionary |
| 9 | [dict.update(dict2)](https://www.javatpoint.com/python-dictionary-update-method) | It updates the dictionary by adding the key-value pair of dict2 to this dictionary. |
| 10 | [dict.values()](https://www.javatpoint.com/python-dictionary-values-method) | It returns all the values of the dictionary. |
| 11 | [len()](https://www.javatpoint.com/python-dictionary-len-method) |  |
| 12 | [popItem()](https://www.javatpoint.com/python-dictionary-popitem-method) |  |
| 13 | [pop()](https://www.javatpoint.com/python-dictionary-pop-method) |  |
| 14 | [count()](https://www.javatpoint.com/python-dictionary-count-method) |  |
| 15 | [index()](https://www.javatpoint.com/python-dictionary-index-method) |  |

# Python Dictionary

Python Dictionary is used to store the data in a key-value pair format. The dictionary is the data type in Python, which can simulate the real-life data arrangement where some specific value exists for some particular key. It is the mutable data-structure. The dictionary is defined into element Keys and values.

* Keys must be a single element
* Value can be any type such as list, tuple, integer, etc.

In other words, we can say that a dictionary is the collection of key-value pairs where the value can be any Python object. In contrast, the keys are the immutable Python object, i.e., Numbers, string, or tuple.

## Creating the dictionary

The dictionary can be created by using multiple key-value pairs enclosed with the curly brackets {}, and each key is separated from its value by the colon (:).The syntax to define the dictionary is given below.

**Syntax:**

17.6M

288

Hello Java Program for Beginners

1. Dict = {"Name": "Tom", "Age": 22}

In the above dictionary **Dict**, The keys **Name** and **Age** are the string that is an immutable object.

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

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. **print**(type(Employee))
3. **print**("printing Employee data .... ")
4. **print**(Employee)

**Output**

<class 'dict'>Printing Employee data .... {'Name': 'John', 'Age': 29, 'salary': 25000, 'Company': 'GOOGLE'}

Python provides the built-in function **dict()** method which is also used to create dictionary. The empty curly braces {} is used to create empty dictionary.

1. # Creating an empty Dictionary
2. Dict = {}
3. **print**("Empty Dictionary: ")
4. **print**(Dict)
6. # Creating a Dictionary
7. # with dict() method
8. Dict = dict({1: 'Java', 2: 'T', 3:'Point'})
9. **print**("\nCreate Dictionary by using dict(): ")
10. **print**(Dict)
12. # Creating a Dictionary
13. # with each item as a Pair
14. Dict = dict([(1, 'Devansh'), (2, 'Sharma')])
15. **print**("\nDictionary with each item as a pair: ")
16. **print**(Dict)

**Output:**

Empty Dictionary: {}Create Dictionary by using dict(): {1: 'Java', 2: 'T', 3: 'Point'}Dictionary with each item as a pair: {1: 'Devansh', 2: 'Sharma'}

## Accessing the dictionary values

We have discussed how the data can be accessed in the list and tuple by using the indexing.

However, the values can be accessed in the dictionary by using the keys as keys are unique in the dictionary.

The dictionary values can be accessed in the following way.

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. **print**(type(Employee))
3. **print**("printing Employee data .... ")
4. **print**("Name : %s" %Employee["Name"])
5. **print**("Age : %d" %Employee["Age"])
6. **print**("Salary : %d" %Employee["salary"])
7. **print**("Company : %s" %Employee["Company"])

**Output:**

<class 'dict'>printing Employee data .... Name : JohnAge : 29Salary : 25000Company : GOOGLE

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

## Adding dictionary values

The dictionary is a mutable data type, and its values can be updated by using the specific keys. The value can be updated along with key **Dict[key] = value**. The update() method is also used to update an existing value.

Note: If the key-value already present in the dictionary, the value gets updated. Otherwise, the new keys added in the dictionary.

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

**Example - 1:**

1. # Creating an empty Dictionary
2. Dict = {}
3. **print**("Empty Dictionary: ")
4. **print**(Dict)
6. # Adding elements to dictionary one at a time
7. Dict[0] = 'Peter'
8. Dict[2] = 'Joseph'
9. Dict[3] = 'Ricky'
10. **print**("\nDictionary after adding 3 elements: ")
11. **print**(Dict)
13. # Adding set of values
14. # with a single Key
15. # The Emp\_ages doesn't exist to dictionary
16. Dict['Emp\_ages'] = 20, 33, 24
17. **print**("\nDictionary after adding 3 elements: ")
18. **print**(Dict)
20. # Updating existing Key's Value
21. Dict[3] = 'JavaTpoint'
22. **print**("\nUpdated key value: ")
23. **print**(Dict)

**Output:**

Empty Dictionary: {}Dictionary after adding 3 elements: {0: 'Peter', 2: 'Joseph', 3: 'Ricky'}Dictionary after adding 3 elements: {0: 'Peter', 2: 'Joseph', 3: 'Ricky', 'Emp\_ages': (20, 33, 24)}Updated key value: {0: 'Peter', 2: 'Joseph', 3: 'JavaTpoint', 'Emp\_ages': (20, 33, 24)}

**Example - 2:**

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. **print**(type(Employee))
3. **print**("printing Employee data .... ")
4. **print**(Employee)
5. **print**("Enter the details of the new employee....");
6. Employee["Name"] = input("Name: ");
7. Employee["Age"] = int(input("Age: "));
8. Employee["salary"] = int(input("Salary: "));
9. Employee["Company"] = input("Company:");
10. **print**("printing the new data");
11. **print**(Employee)

**Output:**

Empty Dictionary: {}Dictionary after adding 3 elements: {0: 'Peter', 2: 'Joseph', 3: 'Ricky'}Dictionary after adding 3 elements: {0: 'Peter', 2: 'Joseph', 3: 'Ricky', 'Emp\_ages': (20, 33, 24)}Updated key value: {0: 'Peter', 2: 'Joseph', 3: 'JavaTpoint', 'Emp\_ages': (20, 33, 24)}

## Deleting elements using del keyword

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

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. **print**(type(Employee))
3. **print**("printing Employee data .... ")
4. **print**(Employee)
5. **print**("Deleting some of the employee data")
6. **del** Employee["Name"]
7. **del** Employee["Company"]
8. **print**("printing the modified information ")
9. **print**(Employee)
10. **print**("Deleting the dictionary: Employee");
11. **del** Employee
12. **print**("Lets try to print it again ");
13. **print**(Employee)

**Output:**

<class 'dict'>printing Employee data .... {'Name': 'John', 'Age': 29, 'salary': 25000, 'Company': 'GOOGLE'}Deleting some of the employee dataprinting the modified information {'Age': 29, 'salary': 25000}Deleting the dictionary: EmployeeLets try to print it again NameError: name 'Employee' is not defined

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

* **Using pop() method**

The **pop()** method accepts the key as an argument and remove the associated value. Consider the following example.

1. # Creating a Dictionary
2. Dict = {1: 'JavaTpoint', 2: 'Peter', 3: 'Thomas'}
3. # Deleting a key
4. # using pop() method
5. pop\_ele = Dict.pop(3)
6. **print**(Dict)

**Output:**

{1: 'JavaTpoint', 2: 'Peter'}

Python also provides a built-in methods popitem() and clear() method for remove elements from the dictionary. The popitem() removes the arbitrary element from a dictionary, whereas the clear() method removes all elements to the whole dictionary.

## Iterating Dictionary

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

### Example 1

**# for loop to print all the keys of a dictionary**

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. **for** x **in** Employee:
3. **print**(x)

**Output:**

NameAgesalaryCompany

### Example 2

**#for loop to print all the values of the dictionary**

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. **for** x **in** Employee:
3. **print**(Employee[x])

**Output:**

John2925000GOOGLE

### Example - 3

**#for loop to print the values of the dictionary by using values() method.**

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. **for** x **in** Employee.values():
3. **print**(x)

**Output:**

John2925000GOOGLE

### Example 4

**#for loop to print the items of the dictionary by using items() method.**

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. **for** x **in** Employee.items():
3. **print**(x)

**Output:**

('Name', 'John')('Age', 29)('salary', 25000)('Company', 'GOOGLE')

## Properties of Dictionary keys

1. In the dictionary, we cannot store multiple values for the same keys. If we pass more than one value for a single key, then the value which is last assigned is considered as the value of the key.

Consider the following example.

1. Employee={"Name":"John","Age":29,"Salary":25000,"Company":"GOOGLE","Name":"John"}
2. **for** x,y **in** Employee.items():
3. **print**(x,y)

**Output:**

Name JohnAge 29Salary 25000Company GOOGLE

2. In python, the key cannot be any mutable object. We can use numbers, strings, or tuples as the key, but we cannot use any mutable object like the list as the key in the dictionary.

Consider the following example.

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE",[100,201,301]:"Department ID"}
2. **for** x,y **in** Employee.items():
3. **print**(x,y)

**Output:**

Traceback (most recent call last): File "dictionary.py", line 1, in Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE",[100,201,301]:"Department ID"}TypeError: unhashable type: 'list'

## Built-in Dictionary functions

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

|  |  |  |
| --- | --- | --- |
| **SN** | **Function** | **Description** |
| 1 | cmp(dict1, dict2) | It compares the items of both the dictionary and returns true if the first dictionary values are greater than the second dictionary, otherwise it returns false. |
| 2 | len(dict) | It is used to calculate the length of the dictionary. |
| 3 | str(dict) | It converts the dictionary into the printable string representation. |
| 4 | type(variable) | It is used to print the type of the passed variable. |

## Built-in Dictionary methods

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

|  |  |  |
| --- | --- | --- |
| **SN** | **Method** | **Description** |
| 1 | [dic.clear()](https://www.javatpoint.com/python-dictionary-clear-method) | It is used to delete all the items of the dictionary. |
| 2 | [dict.copy()](https://www.javatpoint.com/python-dictionary-copy-method) | It returns a shallow copy of the dictionary. |
| 3 | [dict.fromkeys(iterable, value = None, /)](https://www.javatpoint.com/python-dictionary-fromkeys-method) | Create a new dictionary from the iterable with the values equal to value. |
| 4 | [dict.get(key, default = "None")](https://www.javatpoint.com/python-dictionary-get-method) | It is used to get the value specified for the passed key. |
| 5 | dict.has\_key(key) | It returns true if the dictionary contains the specified key. |
| 6 | [dict.items()](https://www.javatpoint.com/python-dictionary-items-method) | It returns all the key-value pairs as a tuple. |
| 7 | [dict.keys()](https://www.javatpoint.com/python-dictionary-keys-method) | It returns all the keys of the dictionary. |
| 8 | [dict.setdefault(key,default= "None")](https://www.javatpoint.com/python-dictionary-setdefault-method) | It is used to set the key to the default value if the key is not specified in the dictionary |
| 9 | [dict.update(dict2)](https://www.javatpoint.com/python-dictionary-update-method) | It updates the dictionary by adding the key-value pair of dict2 to this dictionary. |
| 10 | [dict.values()](https://www.javatpoint.com/python-dictionary-values-method) | It returns all the values of the dictionary. |
| 11 | [len()](https://www.javatpoint.com/python-dictionary-len-method) |  |
| 12 | [popItem()](https://www.javatpoint.com/python-dictionary-popitem-method) |  |
| 13 | [pop()](https://www.javatpoint.com/python-dictionary-pop-method) |  |
| 14 | [count()](https://www.javatpoint.com/python-dictionary-count-method) |  |
| 15 | [index()](https://www.javatpoint.com/python-dictionary-index-method) |  |