By: Dr. Pankaj Malik

Python Dictionary

A dictionary in Python is a collection of key values, used to store data values like a map, which, unlike other data types holds **Key-value** only a single value as an element.

Dictionaries are a useful data structure for storing data in Python because they are capable of imitating real-world data arrangements where a certain value exists for a given key.

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

- This data structure is mutable
- The components of dictionary were made using keys and values.
- Keys must only have one component.
- Values can be of any type, including integer, list, and tuple.

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

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

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

Example:

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

Example:

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

Example:

```
# with dict() method
Dict = dict({1: 'Hcl', 2: 'WIPRO', 3:'Facebook'})
```

```
print("\nCreate Dictionary by using  dict(): ")
print(Dict)
```

Accessing the dictionary values

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

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

Adding Dictionary Values

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

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

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

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

```
print(Dict)

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

Example

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

Deleting Elements using del Keyword

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

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

Deleting Elements using pop() Method

A dictionary is a group of key-value pairs in Python. You can retrieve, insert, and remove items using this unordered, mutable data type by using their keys. The pop() method is one of the ways to get rid of elements from a dictionary. In this post, we'll talk about how to remove items from a Python dictionary using the pop() method. The value connected to a specific key in a dictionary is removed using the pop() method, which then returns the value. The key of the element to be removed is the only argument needed. The pop() method can be used in the following ways:

```
# Creating a Dictionary
Dict1 = {1: 'JavaTpoint', 2: 'Educational', 3: 'Website'}
# Deleting a key
# using pop() method
pop_key = Dict1.pop(2)
print(Dict1)
```

Iterating Dictionary

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

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"WIPRO"}
for x in Employee:
  print(x)
```

Example 2

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"WIPRO"}
for x in Employee.values():
  print(x)
```

Properties of Dictionary Keys

. 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.

Example:

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

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

Example:

```
Employee = {"Name": "John", "Age": 29, "salary":26000,"Company":"WIPRO",[100,201,301]:"Department I
D"}
for x,y in Employee.items():
   print(x,y)
```

Built-in Dictionary Functions

A function is a method that can be used on a construct to yield a value. Additionally, the construct is unaltered. A few of the Python methods can be combined with a Python dictionary.

- **len()**

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

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

- **any()**

  Like how it does with lists and tuples, the any() method returns True indeed if one dictionary key does have a Boolean expression that evaluates to True.

```
dict = {1: "Ayan", 2: "Bunny", 3: "Ram", 4: "Bheem"}
any({'':'','':'','3':''})
```

- **all()**

  Unlike in any() method, all() only returns True if each of the dictionary's keys contain a True Boolean value.

```
dict = {1: "Ayan", 2: "Bunny", 3: "Ram", 4: "Bheem"}
all({1:'',2:'','':''})
```

- **sorted()**

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

```
dict = {7: "Ayan", 5: "Bunny", 8: "Ram", 1: "Bheem"}
sorted(dict)
```

- **clear()**

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

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

- **copy()**

It returns a shallow copy of the dictionary which is created.

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

- **pop()**

It mainly eliminates the element using the defined key.

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

popitem()

removes the most recent key-value pair entered

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

- **keys()**

It returns all the keys of the dictionary.

```
# dictionary methods
dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
```

```
# keys() method
print(dict_demo.keys())
```

- **items()**

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

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

- **get()**

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

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

- **update()**

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

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

- **values()**

It returns all the values of the dictionary with respect to given input.

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

Python Set

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

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

Example 2: Using set() method

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

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

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

# Empty set using set() function
set4 = set()
print(type(set4))
```

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.

Example: 1 - Using add() method

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

Example - 2 Using update() function

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

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.

Example-1 Using discard() method

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

```
months.discard("January");
months.discard("May");
print("\nPrinting the modified set...");
print(months)
print("\nlooping through the set elements ... ")
for i in months:
   print(i)
```

Example-2 Using remove() function

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

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.

Union of two Sets

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

Example 1: using union | operator

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

Example 2: using union() method

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

The intersection of two sets

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

The intersection of two sets can be performed by the **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.

Example 1: Using & operator

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

Example 2: Using intersection() method

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

Example 3:

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

The intersection_update() method

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.

```
a = {"Devansh", "bob", "castle"}
b = {"castle", "dude", "emyway"}
c = {"fuson", "gaurav", "castle"}
   a.intersection_update(b, c)
   print(a)
```

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.

Example 1 : Using subtraction (-) operator

```
Days1 = {"Monday",  "Tuesday", "Wednesday", "Thursday"}
Days2 = {"Monday", "Tuesday", "Sunday"}
print(Days1-Days2) #{"Wednesday", "Thursday" will be printed}
```

Example 2 : Using difference() method

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

Symmetric Difference of two sets

In Python, the symmetric Difference between set1 and set2 is the set of elements present in one set or the other but not in both sets. In other words, the set of elements is in set1 or set2 but not in their intersection.

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

Example - 1: Using ^ operator

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

```
print(c)
```

Example - 2: Using symmetric_difference() method

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

Set comparisons

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

```
Days1 = {"Monday",  "Tuesday", "Wednesday", "Thursday"}
Days2 = {"Monday", "Tuesday"}
Days3 = {"Monday", "Tuesday", "Friday"}

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

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

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

FrozenSets

In Python, a frozen set is an immutable version of the built-in set data type. It is similar to a set, but its contents cannot be changed once a frozen set is created.

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

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

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

example to create the frozen set.

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

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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Dictionary Comprehension in Python

What is Dictionary Comprehension?

Dictionary comprehension is the method used for transferring one dictionary into another dictionary. Throughout the process of transferring the dictionary into another, the user can also include the data of the original dictionary into the new dictionary, and each data can be transferred as per need.

Just like list comprehension, Python also allows the user to perform dictionary comprehensions. The user can create the dictionary by using the simple expression of the built-in method.

Example 1: (Method 1)

```
# let's assume the user have two lists named Key and values
key = ['p', 'q', 'r', 's', 't']
value = [56, 67, 43, 12, 6]
# the following method is used for comprehensiong the dictionary
user_Dict = { X:Y for (X,Y) in zip(key, value)}
print ("user_Dict: ", user_Dict)
```

Example 2: (Method 2)

```
# The user can also user the below method
key_1 = ['j', 'k', 'l', 'm', 'n', 'o']
value_1 = [34, 54, 13, 76, 98, 74]
user_Dict_1 = dict (zip (key_1, value_1))
print ("user_Dict_1: ", user_Dict_1)
```

Example 3: (Method 3)

```
user_Dict2 = {q: q**2 for q in [56, 67, 43, 12, 6]}
print ("user_Dict2: ", user_Dict2)
```

Example 6: (Method 4)

```
user_dict5 = dict()
for numm in range(3, 20):
   user_dict5 [numm] = numm*numm
print("user_dict5: ", user_dict5)
```

Example:

```
# dictionary comprehension example
```

```
square_dict = {num: num*num for num in range(1, 11)}
print(square_dict)
```

Example : If Conditional Dictionary Comprehension

```
original_dict = {'jack': 38, 'michael': 48, 'guido': 57, 'john': 33}
even_dict = {k: v for (k, v) in original_dict.items() if v % 2 == 0}
print(even_dict)
```

Example : Multiple if Conditional Dictionary Comprehension

```
original_dict = {'jack': 38, 'michael': 48, 'guido': 57, 'john': 33}
new_dict = {k: v for (k, v) in original_dict.items() if v % 2 != 0 if v < 40}
print(new_dict)
```

Example : if-else Conditional Dictionary Comprehension

```
original_dict = {'jack': 38, 'michael': 48, 'guido': 57, 'john': 33}
new_dict_1 = {k: ('old' if v > 40 else 'young')
    for (k, v) in original_dict.items()}
print(new_dict_1)
```

Example : Nested Dictionary with Two Dictionary Comprehensions

```
dictionary = {
    k1: {k2: k1 * k2 for k2 in range(1, 6)} for k1 in range(2, 5)
}
print(dictionary)
```

Advantages of Using Dictionary Comprehension

As we can see, dictionary comprehension shortens the process of dictionary initialization by a lot. It makes the code more pythonic.

Using dictionary comprehension in our code can shorten the lines of code while keeping the logic intact.

Python Math Module

Python has a math module that can handle these complex calculations. The functions in the math module can perform simple mathematical calculations like addition (+) and subtraction (-) and advanced mathematical calculations like trigonometric operations and logarithmic operations.

It is a standard module, so we do not need to install it separately. We only must import it into the program we want to use. We can import the module, like any other module of Python, using import math to implement the functions to perform mathematical operations.

Since the source code of this module is in the C language, it provides access to the functionalities of the underlying C library.

Example

Square root calculation

import math

math.sqrt(4)

example

```
import math
n=int(input())
print(math.factorial(n))
```

Example

```
import math
num=int(input())
exp=int(input())
res=math.pow(num, exp)
print(res)
```

Example

```
import math
n1=int(input())
n2=int(input())
res=math.gcd(n1,n2)
print(res)
```

Example

```
import math
n = eval(input())
res = math.ceil(n)
print(res)
```

Example

```
import math
n = eval(input())
res = math.floor(n)
print(res)
```

Example

```
import math
n = eval(input())
res = math.fabs(n)
print(res)
```

Example

```
# Importing the required library
import math

# Printing the value of pi using the math module
print( "The value of pi is ", math.pi )
```

division concept in Python math module

```
a = int(input())
b = int(input())
res = a/b
print("Float Division" , res)
res = a//b
print("Integer Division" , res)
res = a % b
print("Reminder", res)
```

Python Random module

The Python Random module is a built-in module for generating random integers in Python. These numbers occur randomly and does not follow any rules or instructuctions. We can therefore use this module to generate random numbers, display a random item for a list or string, and so on.

The random() Function

The random.random() function gives a float number that ranges from 0.0 to 1.0. There are no parameters required for this function. This method returns the second random floating-point value within [0.0 and 1] is returned.

```
# Python program for generating random float number
import random
num=random.random()
print(num)
```

The randint() Function

The random.randint() function generates a random integer from the range of numbers supplied.

```
# Python program for generating a random integer
import random
num = random.randint(1, 500)
print( num )
```

The randrange() Function

The random.randrange() function selects an item randomly from the given range defined by the start, the stop, and the step parameters. By default, the start is set to 0. Likewise, the step is set to 1 by default.

```
# To generate value between a specific range
import random
num = random.randrange(1, 10)
print( num )
num = random.randrange(1, 10, 2)
print( num )
```

The choice() Function

The random.choice() function selects an item from a non-empty series at random. In the given below program, we have defined a string, list and a set. And using the above choice() method, random element is selected.

```
# To select a random element
import random
random_s = random.choice('Random Module') #a string
print( random_s )
random_l = random.choice([23, 54, 765, 23, 45, 45]) #a list
print( random_l )
random_s = random.choice((12, 64, 23, 54, 34)) #a set
print( random_s )
```

The shuffle() Function

The random.shuffle() function shuffles the given list randomly.

```
# To shuffle elements in the list
list1 = [34, 23, 65, 86, 23, 43]
random.shuffle( list1 )
print( list1 )
random.shuffle( list1 )
print( list1 )
```

random.seed( ) in Python

random() function is used to generate random numbers in Python. Not actually random, rather this is used to generate pseudo-random numbers. That implies that these randomly generated numbers can be determined. random() function generates numbers for some values. This value is also called *seed* value.

```
# random module is imported
import random
for i in range(5):

    # Any number can be used in place of '0'.
    random.seed(0)

    # Generated random number will be between 1 to 1000.
```

```
print(random.randint(1, 1000))
```