EduTECH
WARJE
Trusted by student's and professional's

List, Tuple, Set and Dictionary in Python.

Get in touch with us

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

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

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

```
print(type(L1))
print(type(L2))
```

Output:

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

Characteristics of Lists

The list has the following characteristics:

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

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

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

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

Output:

True

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

```
emp = ["John", 102, "USA"]
Dep1 = ["CS",10]
Dep2 = ["IT",11]
HOD_CS = [10,"Mr. Holding"]
HOD_IT = [11, "Mr. Bewon"]
print("printing employee data...")
print("Name : %s, ID: %d, Country: %s"%(emp[0],emp[1],emp[2]))
print("printing departments...")
```

```
print("Department 1:\nName: %s, ID: %d\nDepartment 2:\nName: %s, ID:
 %s"%(Dep1[0],Dep2[1],Dep2[0],Dep2[1]))
print("HOD Details ....")
print("CS HOD Name: %s, Id: %d"%(HOD_CS[1],HOD_CS[0]))
print("IT HOD Name: %s, Id: %d"%(HOD_IT[1],HOD_IT[0]))
print(type(emp),type(Dep1),type(Dep2),type(HOD_CS),type(HOD_IT))
```

Output:

```
printing employee data...
Name : John, ID: 102, Country: USA
printing departments...
Department 1:
Name: CS, ID: 11
Department 2:
Name: IT, ID: 11
HOD Details ....
CS HOD Name: Mr. Holding, Id: 10
IT 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.

List = [0, 1, 2, 3, 4, 5]

0	1	2	3	4	5

List[0] = 0

List[1] = 1

List[2] = 2

List[3] = 3

List[4] = 4

List[5] = 5

List[0:] = [0,1,2,3,4,5]

List[:] = [0,1,2,3,4,5]

List[2:4] = [2, 3]

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

List[:4] = [0, 1, 2, 3]

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

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:

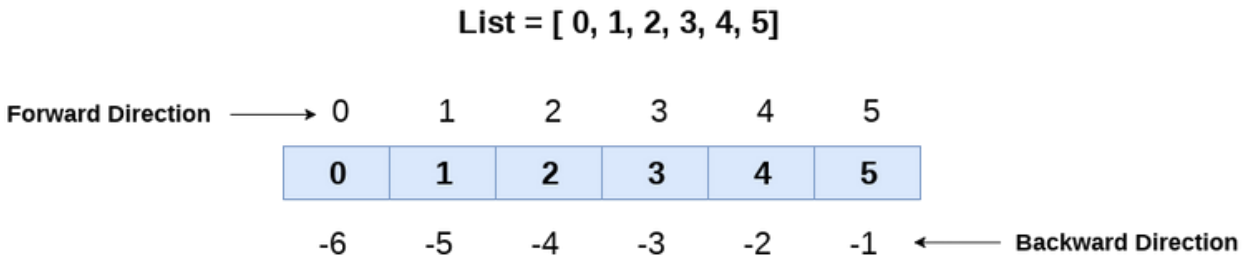
```
list = [1,2,3,4,5,6,7]
print(list[0])
print(list[1])
print(list[2])
print(list[3])
# Slicing the elements
print(list[0:6])
```

By default the index value is 0 so its starts from the 0th element and go fo
r index -1.
print(list[:])
print(list[2:5])
print(list[1:6:2])

Output:

1
2
3
4
[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.



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

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

print(list[-1])
print(list[-3:])

```
print(list[:-1])
print(list[-3:-1])
```

Output:

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

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.

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

Output:

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

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

Consider the following example to delete the list elements.

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

Output:

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

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.

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 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** 1 2 3 4
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.

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

Output:

```
John
David
James
Jonathan
```

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.

```
#Declaring the empty list
l =[]
#Number of elements will be entered by the user
n = int(input("Enter the number of elements in the list:"))
# for loop to take the input
for i in range(0,n):
# The input is taken from the user and added to the list as the item
l.append(input("Enter the item:"))
print("printing the list items..")
```

```
# traversal loop to print the list items
for i in l:
print(i, end = " ")
```

Output:

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

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

Output:

```
printing original list:
0 1 2 3 4
```

printing the list after the removal of first element...
1. 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.

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

Output:

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

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

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

Output:

The sum is: 67

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

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

Output:

The common element is: 2

Python Tuples

A collection of ordered and immutable objects is known as a tuple. Tuples and lists are similar as they both are sequences. Though, tuples and lists are different because we cannot modify tuples, although we can modify lists after creating them, and also because we use parentheses to create tuples while we use square brackets to create lists.

Placing different values separated by commas and enclosed in parentheses forms a tuple. For instance,

Example

```
tuple_1 = ("Python", "tuples", "immutable", "object")
tuple_2 = (23, 42, 12, 53, 64)
tuple_3 = ("Python", "Tuples", "Ordered", "Collection" )
```

We represent an empty tuple by two parentheses enclosing nothing.

```
Empty_tuple = ()
```

We need to add a comma after the element to create a tuple of a single element.

```
Tuple_1 = (50,)
```

Tuple indices begin at 0, and similar to strings, we can slice them, concatenate them, and perform other operations.

Creating a Tuple

All the objects (elements) must be enclosed in parenthesis (), each separated by a comma, to form a tuple. Although using parenthesis is not required, it is recommended to do so.

Whatever the number of objects, even of various data types, can be included in a tuple (dictionary, string, float, list, etc.).

Code

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

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

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

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

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

Output:

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

Parentheses are not mandated to build tuples. Tuple packing is the term for this.

```
# Python program to create a tuple without using parentheses

# Creating a tuple
tuple_ = 4, 5.7, "Tuples", ["Python", "Tuples"]
```

```
# displaying the tuple created
print(tuple_)
# Checking the data type of object tuple_
print( type(tuple_) )
# trying to modify tuple_
try:
tuple_[1] = 4.2
except:
print( TypeError )
```

Output:

```
(4, 5.7, 'Tuples', ['Python', 'Tuples'])
<class 'tuple'>
<class 'TypeError'>
```

It can be challenging to build a tuple with just one element.

Placing just the element in parentheses is not sufficient. It will require a comma after the element to be recognized as a tuple.

Code

```
# Python program to show how to create a tuple having a single element

single_tuple = ("Tuple")
print( type(single_tuple) )
# Creating a tuple that has only one element
single_tuple = ("Tuple",)
print( type(single_tuple) )

# Creating tuple without parentheses
single_tuple = "Tuple",
```

```
print( type(single_tuple) )
```

Output:

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

Accessing Tuple Elements

We can access the objects of a tuple in a variety of ways.

Indexing

To access an object of a tuple, we can use the index operator [], where indexing in the tuple starts from 0.

A tuple with 5 items will have indices ranging from 0 to 4. An IndexError will be raised if we try to access an index from the tuple that is outside the range of the tuple index. In this case, an index above 4 will be out of range.

We cannot give an index of a floating data type or other kinds because the index in Python must be an integer. TypeError will appear as a result if we give a floating index.

The example below illustrates how indexing is performed in nested tuples to access elements.

Code

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

# Creating a tuple
tuple_ = ("Python", "Tuple", "Ordered", "Collection")
```

```
print(tuple_[0])
print(tuple_[1])
# trying to access element index more than the length of a tuple
try:
print(tuple_[5])
except Exception as e:
print(e)
# trying to access elements through the index of floating data type
try:
print(tuple_[1.0])
except Exception as e:
print(e)
# Creating a nested tuple
nested_tuple = ("Tuple", [4, 6, 2, 6], (6, 2, 6, 7))
# Accessing the index of a nested tuple
print(nested_tuple[0][3])
print(nested_tuple[1][1])
```

Output:

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

Negative Indexing

Python's sequence objects support negative indexing.

The last item of the collection is represented by -1, the second last item by -2, and so on.

Code

```
# Python program to show how negative indexing works in Python tuples

# Creating a tuple
tuple_ = ("Python", "Tuple", "Ordered", "Collection")
# Printing elements using negative indices
print("Element at -1 index: ", tuple_[-1])
print("Elements between -4 and -1 are: ", tuple_[-4:-1])
```

Output:

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

Slicing

We can use a slicing operator, a colon (:), to access a range of tuple elements.

Code

```
# Python program to show how slicing works in Python tuples

# Creating a tuple
tuple_ = ("Python", "Tuple", "Ordered", "Immutable", "Collection", "Objects")


# Using slicing to access elements of the tuple
print("Elements between indices 1 and 3: ", tuple_[1:3])
# Using negative indexing in slicing
print("Elements between indices 0 and -4: ", tuple_[:-4])
```

# Printing the entire tuple by using the **default** start and end values.
print("Entire tuple: ", tuple_[:])

Output:

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

Deleting a Tuple

The elements of a tuple cannot be changed, as was already said. Therefore, we are unable to eliminate or remove elements of a tuple.

However, the keyword del makes it feasible to delete a tuple completely.

Code

Python program to show how to delete elements of a Python tuple

# Creating a tuple
tuple_ = ("Python", "Tuple", "Ordered", "Immutable", "Collection", "Objects")

Deleting a particular element of the tuple
try:
del tuple_[3]
print(tuple_)
except Exception as e:
print(e)

Deleting the variable from the global space of the program

del tuple_

Trying accessing the tuple after deleting it

try:
print(tuple_)
except Exception as e:
print(e)

Output:

'tuple' object doesn't support item deletion
name 'tuple_' is not defined

Repetition Tuples in Python

Code

Python program to show repetition in tuples

tuple_ = ('Python',"Tuples")
print("Original tuple is: ", tuple_)

Repeting the tuple elements

tuple_ = tuple_ * 3
print("New tuple is: ", tuple_)

Output:

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

Tuple Methods

Tuple does not provide methods to add or delete elements, and there are only the following two choices.
Examples of these methods are given below.

Code

```
# Python program to show how to tuple methods (.index() and .count()) wo
rk
# Creating a tuple
tuple_ = ("Python", "Tuple", "Ordered", "Immutable", "Collection", "Ordered"
)

# Counting the occurrence of an element of the tuple using the count() meth
od
print(tuple_.count('Ordered'))

# Getting the index of an element using the index() method
print(tuple_.index('Ordered')) # This method returns index of the first occu
rrence of the element
```

Output:

```
2
2
```

Tuple Membership Test

Using the in keyword, we can determine whether an item is present in the given tuple or not.

Code

```
# Python program to show how to perform membership test for tuples

# Creating a tuple
tuple_ = ("Python", "Tuple", "Ordered", "Immutable", "Collection", "Ordered"
)

# In operator
print('Tuple' in tuple_)
print('Items' in tuple_)

# Not in operator
print('Immutable' not in tuple_)
print('Items' not in tuple_)
```

Output:

```
True
False
False
True
```

Iterating Through a Tuple

We can use a for loop to iterate through each element of a tuple.

Code

```
# Python program to show how to iterate over tuple elements

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

Output:

```
Python
Tuple
Ordered
Immutable
```

Changing a Tuple

Tuples, as opposed to lists, are immutable objects.

This implies that after a tuple's elements have been specified, we cannot modify them. However, we can modify the nested elements of an element if the element itself is a mutable data type like a list.

A tuple can be assigned to many values (reassignment).

Code

```
# Python program to show that Python tuples are immutable objects

# Creating a tuple
tuple_ = ("Python", "Tuple", "Ordered", "Immutable", [1,2,3,4])
# Trying to change the element at index 2
try:
tuple_[2] = "Items"
print(tuple_)
except Exception as e:
print( e )
# But inside a tuple, we can change elements of a mutable object
tuple_[-1][2] = 10
print(tuple_)

# Changing the whole tuple
tuple_ = ("Python", "Items")
print(tuple_)
```

Output:

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

To merge multiple tuples, we can use the + operator. Concatenation is the term for this.

Using the * operator, we may also repeat a tuple's elements for a specified number of times. This is already shown above.

The results of the operations + and * are new tuples.

Code

```
# Python program to show how to concatenate tuples

# Creating a tuple
tuple_ = ("Python", "Tuple", "Ordered", "Immutable")
# Adding a tuple to the tuple_
print(tuple_ + (4, 5, 6))
```

Output:

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

Advantages of Tuple over List

Tuples and lists are employed in similar contexts because of how similar they are. A tuple implementation has several benefits over a list, though. The following are a few of the primary benefits:

- We generally employ lists for homogeneous data types and tuples for heterogeneous data types.
- Tuple iteration is quicker than list iteration because tuples are immutable. There is such a modest performance improvement.
- Tuples with immutable components can function as the key for a Python dictionary object. This feature is not feasible with lists.
- Collecting data in a tuple will ensure that it stays write-protected if it never changes.

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.

Example 1: Using curly braces

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

Output:

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

Monday
Saturday
Thursday
Sunday
Wednesday

Example 2: Using set() method

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

Output:

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

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

```
# Creating a set which have immutable elements
set1 = {1,2,3, "Gamaka AI", 20.5, 14}
```

```
print(type(set1))
#Creating a set which have mutable element
set2 = {1,2,3,["Gamaka AI",4]}
print(type(set2))
```

Output:

```
<class 'set'>

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

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

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

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

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

Output:

<class 'dict'>

<class 'set'>

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

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

Output:

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

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

Adding items to the set

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

Example: 1 - Using add() method

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

print(i)

Output:

printing the original set ...
{'February', 'May', 'April', 'March', 'June', 'January'}
Adding other months to the set...

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

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

Example - 2 Using update() function

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

Output:

printing the original set ...

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

updating the original set ...

printing the modified set ...

{'January', 'February', 'April', 'August', 'October', 'May', 'June', 'July', 'September', 'March'}

Removing items from the set

Python provides the **discard()** method and **remove()** method which can be used to remove the items from the set. The difference between these 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

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

Output:

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

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

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

Example-2 Using remove() function

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

Output:

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

Removing some months from the set...

Printing the modified set...

{'February', 'June', 'April', 'March'}

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

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

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

Output:

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

Removing some months from the set...

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

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

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

Consider the following example.

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

Output:

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

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

Difference between discard() and remove()

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

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

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

Consider the following example.

Example-

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

Output:

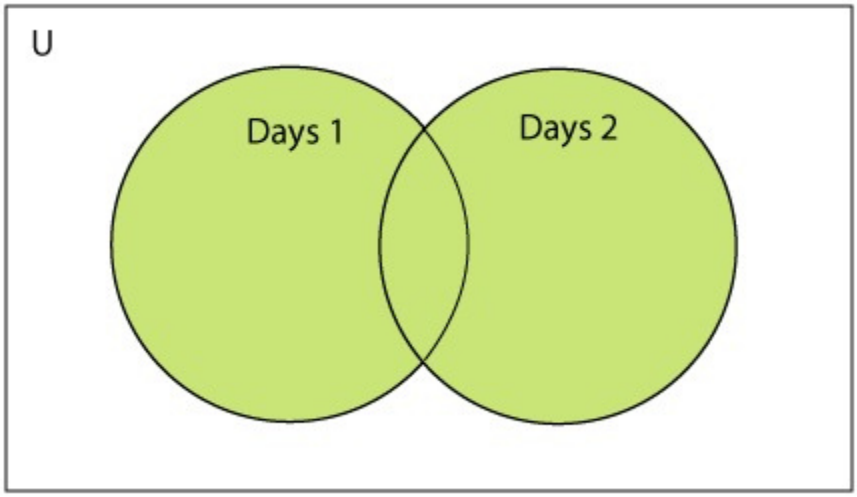
```
printing the original set ...
{'March', 'January', 'April', 'June', 'February', 'May'}
Removing items through discard() method...

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

Python Set Operations

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

Union of two Sets

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

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

Output:

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

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

Example 2: using union() method

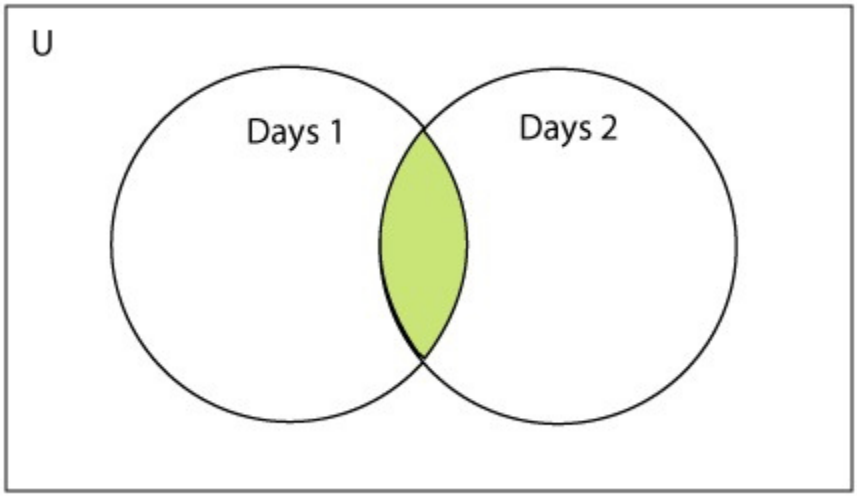
```
Days1 = {"Monday","Tuesday","Wednesday","Thursday"}
Days2 = {"Friday","Saturday","Sunday"}
print(Days1.union(Days2)) #printing the union of the sets
```

Output:

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

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

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

Output:

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

Example 2: Using intersection() method

```
set1 = {"PRABHANJAN","John", "David", "Martin"}
set2 = {"Steve", "Milan", "David", "Martin"}
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.

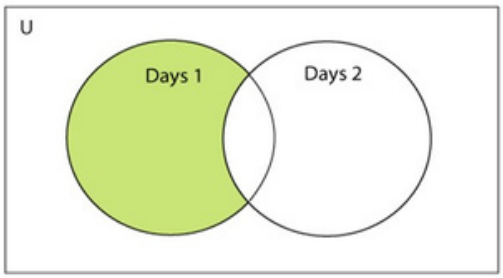
```
a = {"PRABHANJAN", "bob", "castle"}
b = {"castle", "dude", "emyway"}
c = {"fuson", "gaurav", "castle"}

a.intersection_update(b, c)

print(a)
```

Output:

```
{'castle'}
```

Difference between the two sets

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



Consider the following example.

Example 1 : Using subtraction (-) operator

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

Output:

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

Example 2 : Using difference() method

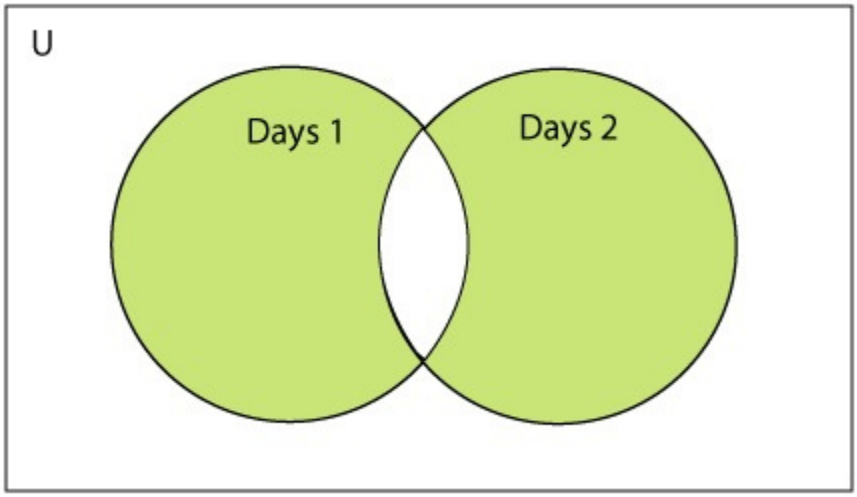
```
Days1 = {"Monday", "Tuesday", "Wednesday", "Thursday"}
Days2 = {"Monday", "Tuesday", "Sunday"}
print(Days1.difference(Days2)) # prints the difference of the two sets Days 1 and Days2
```

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

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

Output:

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

Example - 2: Using symmetric_difference() method

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

Output:

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

Set comparisons

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.

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

#Days1 is the superset of Days2 hence it will print true.
print (Days1>Days2)
#prints false since Days1 is not the subset of Days2
print (Days1<Days2)
#prints false since Days2 and Days3 are not equivalent
print (Days2 == Days3)
```

Output:

```
True
False
False
```

FrozenSets

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.

```
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 Fro
zenset after creation
```

Output:

```
<class 'frozenset'>

printing the content of frozen set...
1
2
3
4
5
Traceback (most recent call last):
File "set.py", line 6, in <module>
```

Frozenset.add(6) #gives an error since we can change the content of Frozenset after creation
AttributeError: 'frozenset' object has no attribute 'add'

Frozenset for the dictionary

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

Consider the following example.

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

Output:

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

Set Programming Example

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)
```

Output:

Enter the number you want to remove:12

After Removing: {1, 2, 3, 4, 5, 6, 24}

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

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

Output:

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

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

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

Output:

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

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

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

Output:

{56, 23}

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

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

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

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

Output:

```
False
False
True
True
```

Python Built-in set methods

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

SN	Method	Description
1	add(item)	It adds an item to the set. It has no effect if the item is already present in the set.
2	clear()	It deletes all the items from the set.
3	copy()	It returns a 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)	It removes the specified item from the set.
6	intersection()	It returns a new set that contains only the common elements of both the sets. (all the sets if more than two are specified).
7	intersection_update(....)	It removes the items from the original set that are not present in both the sets (all the sets if more than one are specified).
8	Isdisjoint(....)	Return True if two sets have a null intersection.
9	Issubset(....)	Report whether another set contains this set.
10	Issuperset(....)	Report whether this set contains

	another set.
11	pop() Remove and return an arbitrary set element that is the last element of the set. Raises KeyError if the set is empty.
12	remove(item) Remove an element from a set; it must be a member. If the element is not a member, raise a KeyError.
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 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.

o Keys must be a single element

o 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:

```
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.

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
print(type(Employee))
print("printing Employee data .... ")
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.

```
# Creating an empty Dictionary
Dict = {}
print("Empty Dictionary: ")
print(Dict)
# Creating a Dictionary
# with dict() method
Dict = dict({1: 'Gamaka', 2: 'T', 3:'AI'})
print("\nCreate Dictionary by using dict(): ")
print(Dict)

# Creating a Dictionary
# with each item as a Pair
Dict = dict([(1, 'PRABHANJAN'), (2, 'DHOBALE')])
print("\nDictionary with each item as a pair: ")
print(Dict)
```

Output:

Empty Dictionary:
{}
Create Dictionary by using dict():
{1: 'Gamaka', 2: 'T', 3: 'AI'}

Dictionary with each item as a pair:
{1: 'PRABHANJAN', 2: 'DHOBALE'}

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.

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGL
E"}
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"])
```

Output:

```
<class 'dict'>
printing Employee data ....
Name : John
Age : 29
Salary : 25000
Company : 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:

```
# 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] = 'Gamaka AI'
```

```
print("\nUpdated key value: ")
print(Dict)
```

Output:

Empty Dictionary:
{}
Dictionary after adding 3 elements:
{0: 'Peter', 2: 'Joseph', 3: 'Ricky'}
Dictionary after adding 3 elements:
{0: 'Peter', 2: 'Joseph', 3: 'Ricky', 'Emp_ages': (20, 33, 24)}
Updated key value:
{0: 'Peter', 2: 'Joseph', 3: 'Gamaka AI', 'Emp_ages': (20, 33, 24)}

Example - 2:

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGL
E"}
print(type(Employee))
print("printing Employee data .... ")
print(Employee)
print("Enter the details of the new employee....");
Employee["Name"] = input("Name: ");
Employee["Age"] = int(input("Age: "));
Employee["salary"] = int(input("Salary: "));
Employee["Company"] = input("Company:");
print("printing the new data");
print(Employee)
```

Output:

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: 'Gamaka AI', '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.

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGL
E"}
print(type(Employee))
print("printing Employee data .... ")
print(Employee)
print("Deleting some of the employee data")
del Employee["Name"]
del Employee["Company"]
print("printing the modified information ")
print(Employee)
print("Deleting the dictionary: Employee");
del Employee
print("Lets try to print it again ");
print(Employee)
```

Output:

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

Deleting some of the employee data
printing the modified information
{'Age': 29, 'salary': 25000}
Deleting the dictionary: Employee
Lets try to print it again
NameError: name 'Employee' is not defined

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

o **Using pop() method**

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

```
# Creating a Dictionary
Dict = {1: 'Gamaka AI', 2: 'Peter', 3: 'Thomas'}
# Deleting a key
# using pop() method
pop_ele = Dict.pop(3)
print(Dict)
```

Output:

{1: 'Gamaka AI', 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

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

Output:

```
Name
Age
salary
Company
```

Example 2

#for loop to print all the values of the dictionary

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

Output:

```
John
29
25000
GOOGLE
```

Example - 3

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

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

Output:

```
John
29
25000
GOOGLE
```

Example 4

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

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGL
E"}
for x in Employee.items():
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.

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

Output:

```
Name John
Age 29
Salary 25000
Company 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.

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGL
E",[100,201,301]:"Department ID"}
for x,y in Employee.items():
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()	It is used to delete all the items of the dictionary.

2	dict.copy() It returns a shallow copy of the dictionary.	
3	dict.fromkeys(iterable, value = None, /)	Create a new dictionary from the iterable with the values equal to value.
4	dict.get(key, default = "None")	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()	It returns all the key-value pairs as a tuple.
7	dict.keys()	It returns all the keys of the dictionary.
8	dict.setdefault(key,default= "None")	It is used to set the key to the default value if the key is not specified in the dictionary
9	dict.update(dict2)	It updates the dictionary by adding the key-value pair of dict2 to this dictionary.
10	dict.values()	It returns all the values of the dictionary.
11	len()	
12	popItem()	

13	pop()	
14	count()	
15	index()	