PYTHON LIST, SET, TUPLE & DICTIONARY OPERATIONS

We learn about different types of operations formed on list, set, tuple and dictionary

ca

LIST

- List is ordered, mutable(elements can be changed),heterogeneous collection of elements in this duplication is allowed.
- List are used to store multiple data at once and also different datatypes like float ,int,string,complex,boolean another list ,tuple etc. .
- List is created by placing items inside [] with separated by commas.
- For e.g. my_sample_numbers = [1,2,3] if we write the code like this print(my_sample_numbers) in console it will display like this [1,2,3].
- To define any empty list you just have to write like this mysample_list = [].
- Another way demovar_samplelist = list(("apple", "banana", "cherry")).

Accessing an Element from List

- **Each item in a list is associated with a number known as list index.**
- **If you count from front side it starts from 0, 1,2,3,4 and so on till numbers of elements present in the list, if you count from back side it start from -1,-2,-3,-4 and so on till no of elements on the list.**
- **For e.g. mysample_list = [“AWS”,”AZURE”,”GCP”,”GOLANG”,”LINUX”,”DOCKER”]. HERE index of AWS is 0 from front side and backside it is -6, similar way inside of Docker from front is 5 while from backside it is -1.**
- **Accessing item from List for e.g. I want to print item present in index number 3 so it write like this print(mysample_list[3]) the output display in console is GOLANG . Similar way if a write print (mysample_list [-5]) the output display in console is AZURE.**
- **Accessing Nested elements from the List demosample_List = [10, [20, 14],"Geeks", ["vs", "Nerds"]]. We can accessing item inside any item print(demosample_List[1][1]) The output display on the console is 14 , try another example print(demosample_List[3][0]) The output on the console displays is vs.**
- **Note if the specified index does not exist in the list, then python throws the Indexerror exception.**
- **To get the size of the python list we can type print(len(mysample_list)) the output will be 6.**

Taking Input of Python List

- Let take us example how we take input from user for list first way

Method 1:

```
Sample_userip = input ("Enter elements (Space-Separated): ")
 # We will split the strings and store it to a list
demosamplelist = Sample_userip.split()
print('The sample list is:', demosamplelist)   # printing the list
```

- In the output console we have to provide Enter elements(Space-Separated): GEEKS VS NERDS The sample list is : ['GEEKS' ,' VS','NERDS']
- Let take us example how we take input from user for list second way

Method 2:

```
# input size of the list
Sample_listsize = int (input ("Enter the size of list: "))
# We store integers in a list using map,
# split and strip functions
samplelst = list (map(int, input("Enter the integer\elements:").strip().split()))[:Sample_listsize]
# printing the list
print ('The list is:', samplelst)
```

- The sample output will be

Enter the size of list: 4

Enter the integer\elements:24 35 343 343

The list is: [24, 35, 343, 343]

Slicing of a Python List

- **It is possible to access a particular section of items from the list using a slicing operator i.e. :**
- **For e.g. take above mysample_list if I want to print element from index 1 to 3 we write like this print(mysample_list[1:3]) then output display on console will be ["AZURE","GCP"]**
- **Taking another example mysample_list [3 :]) then output display on the console will be ["GOLANG","LINUX","DOCKER"].**
- **Suppose we want to print whole list just we have to write like this print(mysample_list[:]) this is slicing way of print.**
- **In similar way we can do negative slicing also i.e. print(mysample_list[-5:--3])**
- **Suppose you skip or jump slicing we can do this way the syntax is [start: end: step] in this order. By default the jump is one even we don't specify. For e.g. print(mysample_list[1:4:2]) the output display on the screen is ['AZURE', 'GOLANG'].**
- **For e.g if you type this print(mysample_list[1:4:-2]) you will get an empty list []**
- **Try e.g print(mysample_list[-6:-1:2]) you will get an ['AWS', 'GCP', 'LINUX'].**

Adding Element to Python List

Method 1 : Via append method

- Elements can be added to the List by using the built-in append function. Only one element at a time can be added to the list by using the append() method, for the addition of multiple elements with the append() method, loops are used

```
Sample_blanklist=[]
# Adding of Element to list
Sample_blanklist.append(45)
Sample_blanklist.append(78.25)
```

```
Sample_blanklist.append(“Nerds”)
print(“The new list after append”,Sample_blanklist)  # o/p  [45,78.25,”Nerds”]
Sample_blanklist.append((7,4)) # Adding tuple to the list
print(“The new list after adding tuple way”,Sample_blanklist) # o/p [45,78.25,”Nerds”,(7,4)]
Sample_blanklist.append([“GCP”,”K8S”]) # Adding list to the list
print(“The new list after adding list way”,Sample_blanklist) # o/p
[45,78.25,”Nerds”,(7,4),[“GCP”,”K8S”]
```

Method 2 : Via insert method

- Append() method only works for the addition of elements at the end of the list,for addition of element at desired position we use insert() method.It takes two argument(position,value).

```
Sample_blanklist.insert(3, 12)

Sample_blanklist.insert(0, 'Devops')

print("\nList after performing Insert Operation: ",Sample_blanklist)

# o/p  List after performing Insert Operation: ['Devops', 45, 78.25, 'Nerds', 12, (7, 4), ['GCP',
'K8S']]
```

Method 3 : Via extend method

- Other than append() and insert() methods, there’s one more method for the Addition of element ,extend() this method is used to add multiple elements at the same time at the end of the list.

```
Sample_primenos = [2,3,5]

Print(“The sample prime numbers list”,Sample_primenos)

Sample_oddnos =[1,3,5,7,9,11,13,15]

Print(“The sample odd numbers list”,Sample_oddnos)

#Let us join two list

Sample_primenos.extend(Sample_oddnos)

print("The final list after extend operation",Sample_primenos)

#o/p

The sample prime numbers list [2, 3, 5]

The sample odd numbers list [1, 3, 5, 7, 9, 11, 13, 15]

The final list after extend operation [2, 3, 5, 1, 3, 5, 7, 9, 11, 13, 15]
```

Reverse Element to Python List

```
print("The reverse of list",Sample_primenos[::-1])

print("The list:", Sample_primenos)

Sample_primenos.reverse()

print("The reverse of list using reverse method",Sample_primenos)
```

o/p

```
The reverse of list [15, 13, 11, 9, 7, 5, 3, 1, 5, 3, 2]

The list: [2, 3, 5, 1, 3, 5, 7, 9, 11, 13, 15]
```

The reverse of list using reverse method [15, 13, 11, 9, 7, 5, 3, 1, 5, 3, 2]

Changing Element to Python List

print("The list:", Sample_primenos)

Sample_primenos[4] =”Change element”

print("The new list after changing element o n 4th index:", Sample_primenos)

o/p

The list: [15, 13, 11, 9, 7, 5, 3, 1, 5, 3, 2]

The new list after changing element o n 4th index: [15, 13, 11, 9, 'Change element', 5, 3, 1, 5, 3, 2]

Removing Element to Python List

Method1 : Via remove() method

Elements can be removed from the List by using the built-in remove function but an Error arises if the element doesn’t exist in the list. Remove() method only removes one element at a time, to remove a range of elements, the iterator is used. The remove() method removes the specified item.**Note:** Remove method in List will only remove the first occurrence of the searched element.

```
Samplelanguages_list = ['Python', 'Swift', 'C++', 'C', 'Java', 'Rust', 'R']
# remove 'Python' from the list
Samplelanguages_list.remove('Python')
print(“After remove of element from list:”,Samplelanguages_list) # ['Swift', 'C++', 'C', 'Java', 'Rust', 'R']
Second Way
List = [1, 2, 3, 4, 5, 6,7, 8, 9, 10, 11, 12]
# Removing elements from List
# using iterator method
for i in range(1, 5):
   List.remove(i)
print("\nList after Removing a range of elements: ",List)
# o/p List after Removing a range of elements: [5, 6, 7, 8, 9, 10, 11, 12]
```

Method 2 : Via pop() method

pop() can also be used to remove and return an element from the list, but by default it removes only the last element of the list, to remove an element from a specific position of the List, the index of the element is passed as an argument to the pop() method.

```
List = [1, 2, 3, 4, 5]
# Removing element from the
# Set using the pop() method
List.pop()
print("\nList after popping an element: ",List)
# Removing element at a specific location from the
```

```
# Set using the pop() method
List.pop(2)
print("\nList after popping a specific element: ",List)
#o/p List after popping an element: [1, 2, 3, 4]
List after popping a specific element: [1, 2, 4]
```

Method 3: Via del () method

del() can also be used to remove one or more items from list

```
Sapmplanguages = ['Python', 'Swift', 'C++', 'C', 'Java', 'Rust', 'R']
# deleting the second item
del Sapmplanguages [1]
print("The list after deleting second index item",Sapmplanguages) # ['Python', 'C++', 'C', 'Java',
'Rust', 'R']
# deleting the last item
del Sapmplanguages [-1]
print("The list after deleting item via negative index",Sapmplanguages) # ['Python', 'C++', 'C',
'Java', 'Rust']
# delete first two items
del Sapmplanguages [0 : 2]
print("The list after deleting item within the range",Sapmplanguages) # ['C', 'Java', 'Rust']
```

Tuple

- Tuples are used to hold together multiple objects. Think of them as similar to lists, but without the extensive functionality that the list class gives you. One major feature of tuples is that they are *immutable* like strings i.e. we cannot modify tuples.
- Tuples are defined by specifying items separated by commas within an optional pair of parentheses.
- Tuples are usually used in cases where a statement or a user-defined function can safely assume that the collection of values (i.e. the tuple of values used) will not change.
- An empty tuple is constructed by an empty pair of parentheses such as myempty = (). However, a tuple with a single item is not so simple. we have to specify it using a comma following the first (and only) item so that Python can differentiate between a tuple and a pair of parentheses surrounding the object in an expression i.e. you have to specify singleton = (2 ,) if you mean you want a tuple containing the item 2.
- A tuple can have any number of items and they may be of different types (integer, float, list, string, etc.).

```
# Different types of tuples
  # Empty tuple
   mysample_tuple = ()
   print("This display empty tuple",mysample_tuple)
 #  singleton tuple
  Mysample_var1 = (1,)
print("This display singleton tuple",Mysample_var1)
  # tuple with mixed datatypes
my_tuple = (1, "Hello", 3.4)
print("Tuple can have multiple datatypes",my_tuple)
```

```
# nested tuple
mysample_var2= ("mouse", [8, 4, 6], (1, 2, 3))
print("Tuples can store data in nested form",mysample_var2)
# Another way to create singleton tuple
Mysample_var3 = "hello",
print(type(Mysample_var3))  # <class 'tuple'>
```

Access Python Tuple Elements

- Like a list each element of a tuple is represented by index numbers **(0, 1, ...)** where the first element is at index **0**.
- We use the index number to access tuple elements. We can use the index operator [] to access an item in a tuple, where the index starts from 0.
- So, a tuple having **6** elements will have indices from **0** to **5**. Trying to access an index outside of the tuple index range(**6,7,...** in this example) will raise an IndexError.The index must be an integer, so we cannot use float or other types. This will result in Type Error.

```
# accessing tuple elements using indexing
Mysample_letters = ("p", "r", "o", "g", "r", "a", "m")
print(Mysample_letters[0])   # prints "p"
print(Mysample_letters[5])   # prints "a"
```

Negative Indexing

- Python allows negative indexing for its sequences.
- The index of **-1** refers to the last item, **-2** to the second last item and so on. For example,

```
# accessing tuple elements using indexing
Mysample_letters = ("p", "r", "o", "g", "r", "a", "m", "m","e","r")
print(Mysample_letters[-1])   # prints "r"
print(Mysample_letters[-4])   # prints "m"
```

Slicing

We can access a range of items in a tuple by using the slicing operator colon :.

```
# elements 2nd to 4th index
print(Mysample_letters [1:4])  #  prints ('r', 'o', 'g')
# elements beginning to 2nd
print(Mysample_letters [:-7]) # prints ('p', 'r','o')
# elements 8th to end
print(Mysample_letters [7:]) # prints ('m', 'e','r')
# elements beginning to end
print(Mysample_letters [:]) # Prints ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'm', 'e','r')
```

Python Tuple Methods

In Python ,methods that add items or remove items are not available with tuple. Only the following two methods are available.

```
sampley_tuple = ('a', 'p', 'p', 'l', 'e',)
print(sampley_tuple.count('p'))  # prints 2
print(sampley_tuple.index('l'))  # prints 3
```

Advantages of Tuple over List in Python

Since tuples are quite similar to lists, both of them are used in similar situations.

However, there are certain advantages of implementing a tuple over a list:

- We generally use tuples for heterogeneous (different) data types and lists for homogeneous (similar) data types.
- Since tuples are immutable, iterating through a tuple is faster than with a list. So there is a slight performance boost.
- Tuples that contain immutable elements can be used as a key for a dictionary. With lists, this is not possible.
- If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

Set

- Sets are unordered collections of simple objects. These are used when the existence of an object in a collection is more important than the order or how many times it occurs.
- Using sets, you can test for membership, whether it is a subset of another set, find the intersection between two sets, and so on.
- In Python, we create sets by placing all the elements inside curly braces *{}*, separated by comma.
- A set can have any number of items and they may be of different types (integer, float, tuple, string etc.). But a set cannot have mutable elements like lists, sets or dictionaries as its elements.

```
# create an empty set
empty_set = set()
# create an empty dictionary
empty_dictionary = { }
# check data type of empty_set
print('Data type of empty_set:', type(empty_set))
# check data type of dictionary_set
print('Data type of empty_dictionary', type(empty_dictionary))
# create a set of integer type
Samplestudent_rollno = {112, 114, 116, 118, 115}
print('Student rollnos :',Samplestudent_rollno)
# create a set of string type
samplevowel_letters = {'a', 'e', 'i', 'o', 'u'}
print('Vowel Letters:',samplevowel_letters)
```

```
# create a set of mixed data types
samplemixed_set = {'Hello', 101, -2, 'Bye'}
print('Set of mixed data types:', samplemixed_set)
#check the length of the set
samplenumbers = {2, 4, 6, 6, 2, 8}
print(samplenumbers)   # {8, 2, 4, 6}
print(len(samplenumbers)) # 4
```

Add Items to a Set in Python

- In Python, we use the add() method to add an item to a set. For example,
- Sets are mutable. However, since they are unordered, indexing has no meaning.
- We cannot access or change an element of a set using indexing or slicing. Set data type does not support it.

```
samplenumbers = {21, 34, 54, 12}
print('Initial Set:',samplenumbers) # o/p {34,12,21,54}
# using add() method
samplenumbers.add(32)
print('Updated Set:', samplenumbers)  # o/p {32,34,12,21,54}
```

Update Python Set

The update() method is used to update the set with items other collection types (lists, tuples, sets, etc). For example,

```
companies = {'Lacoste', 'Ralph Lauren'}
tech_companies = ['apple', 'google', 'apple']
#let us  update operation
companies.update(tech_companies)
print("The final set :-",companies) # Output: {'google', 'apple', 'Lacoste', 'Ralph Lauren'}
```

Remove an Element from a Set

We use the discard() method to remove the specified element from a set. For example,

```
samplelanguages = {'Swift', 'Java', 'Python'}
print('Initial Set:',samplelanguages)
# remove 'Java' from a set
removedValue = samplelanguages.discard('Java')
print('Set after remove():', samplelanguages)
```

Python Set Operations

Python Set provides different built-in methods to perform mathematical set operations like union, intersection, subtraction, and symmetric difference.

Union of Two Sets

The union of two sets **A** and **B** include all the elements of set **A** and **B**.

```
# first set
A = {1, 3, 5}
# second set
B = {0, 2, 4}
```

```
# perform union operation using |
print('Union using operator way:', A | B) # o/p  {0,1,2,3,4,5}
# perform union operation using union()
print('Union using method function:', A.union(B)) # o/p  {0,1,2,3,4,5}
```

Set Intersection

The intersection of two sets A and B include the common elements between set A and B.

```
# first set

A = {1, 3, 5}

# second set

B = {1, 2, 3}

# perform intersection operation using &

print('Intersection using  operator way:', A & B) # o/p  {1,3}


# perform intersection operation using intersection()

print('Intersection using method function:', A.intersection(B)) # o/p  {1,3}
```

Difference between Two Sets

The difference between two sets A and B include elements of set A that are not present on set B.

```
# first set
A = {2, 3, 5}
# second set
B = {1, 2, 6}
# perform difference operation using &
print('Difference using operator way:', A - B) # o/p {1,6,5}

# perform difference operation using difference()
print('Difference using method function:', A.difference(B)) # o/p {3,5,6}
```

Set Symmetric Difference

The symmetric difference between two sets A and B includes all elements of A and B without the common elements.

```
# first set
A = {2, 3, 5}
# second set
B = {1, 2, 6}
# perform difference operation using &
print('using ^:', A ^ B)
# using symmetric_difference()
print('using symmetric_difference():', A.symmetric_difference(B))
```

DICTIONARY

It is a set of *key: value* pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: `{}`. Placing a comma-separated list of key:value pairs within the braces adds initial key:value pairs to the dictionary; this is also the way dictionaries are written on output.

```
samplecapital_city = {"Nepal": "Kathmandu", "India": "Delhi", "Bhutan": "Thimpu"}
print("Sample way create dict",samplecapital_city)
```

```
sampledict = dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])

print("Sample diffrent way create dict",sampledict)

# create an empty dictionary
empty_dictionary = { }
```

Add Elements

```
samplecapital_city = {"Nepal": "Kathmandu", "India": "Delhi", "Bhutan": "Thimpu"}
print("Initial sample city:-",samplecapital_city)
print("length of sample city :- ", len(samplecapital_city)) # 3
print("Sort the sample city",sorted(samplecapital_city, reverse=False))#Sort the sample city ['Bhutan',
'India', 'Nepal']
samplecapital _city["Japan"] = "Tokyo"
print("Updated sample city:- ", samplecapital _city)
# updated sample city :-{"Nepal": "Kathmandu", "India": "Delhi", "Bhutan": "Thimpu","Japan":"Tokyo"}
```

Update Elements

```
samplecapital_city = {"Nepal": "Kathmandu", "India": "Delhi", "Bhutan": "Thimpu","Japan":"Tokyo"}
print("Initial sample city:-",samplecapital_city)
samplecapital _city["Japan"] = "Kyoto"
print("Updated new sample city:- ", samplecapital _city)
# Updated new sample city:- {"Nepal": "Kathmandu", "India": "Delhi", "Bhutan":
"Thimpu","Japan":"Kyoto"}
samplecapital_city.update({"India":"DELHI"})
print("New updated sample city:-",samplecapital_city)
# New updated sample city:- {"Nepal": "Kathmandu", "India": "DELHI", "Bhutan":
```

Accessing Elements

```
samplecapital_city = {"Nepal": "Kathmandu", "India": "Delhi", "Bhutan": "Thimpu","Japan":"Kyoto"}
print("Initial sample city:-",samplecapital_city)
print("Getting value of particular key:- ", samplecapital_city["Bhutan"]) # Thimpu as
outputprint("Getting value of particular key not present :-",samplecapital_city["Thimpu"])#
KeyError as output

samplevarx = samplecapital_city.items()

samplevary = samplecapital_city.keys()

samplevarz = samplecapital_city.values()

print("Accessing entire of dict:-",samplevarx)

print("Accessing entire keys  of dict:-",samplevary)

print("Accessing entire values of dict:-",samplevarz)
```

Outputs

Accessing entire of dict:- dict_items([('Nepal', 'Kathmandu'), ('India', 'Delhi'), ('Bhutan', 'Thimpu'), ('Japan', 'Kyoto')])

Accessing entire keys of dict:- dict_keys(['Nepal', 'India', 'Bhutan', 'Japan'])

Accessing entire values of dict:- dict_values(['Kathmandu', 'Delhi', 'Thimpu', 'Kyoto'])

Deleting Elements

samplecapital_city = {"Nepal": "Kathmandu", "India": "Delhi", "Bhutan": "Thimpu","Japan":"Kyoto"}

print("Initial sample city:-",samplecapital_city)

del samplecapital_city["Nepal"]

print("Getting value after deleting of particular key:- ", samplecapital_city) #{"India": "Delhi", "Bhutan": "Thimpu","Japan":"Kyoto"}

del samplecapital_city

print("Getting output after deleting whole dictionary:-",samplecapital_city)# NameError: name ' samplecapital_city ' is not defined

samplecapital_city = {"Nepal": "Kathmandu", "India": "Delhi", "Bhutan": "Thimpu","Japan":"Kyoto"}

samplecapital_city.popitem()

print("New updated after pop sample city:-",samplecapital_city)

samplecapital_city.clear()

print("New updated after clear sample city:-",samplecapital_city)

 # O/p

New updated after pop sample city:- {'Nepal': 'Kathmandu', 'India': 'Delhi', 'Bhutan': 'Thimpu'}
New updated after clear sample city:- {}

List, Set, Dictionary Comprehensions

- The comprehension consists of a single expression followed by at least one for clause and zero or more for or if clauses
- List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable or to create a subsequence of those elements that satisfy a certain condition.

Syntax:

[expression for item in iterable if conditional]

The expression can be any arbitary expression, complex expressions, tuple, nested functions, or another list comprehension.This is equivalent to

for item **in iterable**:
  if conditional:
    expression

Return Type: List

Using List Comprehension:A list comprehension consists of brackets[] containing an expression followed by a for clause, then zero or more for or if clauses. The result will be a new list resulting from evaluating the expression in the context of the for and if clauses that follow it.

List comprehension vs for loop.

```
#Using List Comprehension
Samplevar1=[i*i for i in range(1,11)]
print ("The output is using list comprehension:-",Samplevar1) #Output: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

# Using for loop
Samplevarl1=[ ] # empty list
for i in range(1,11):
a=i*i
Samplevarl1.append(a)
print ("This is output from for loop",Samplevarl1) #Output:[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

2. List comprehension vs filter.

filter:

Returns an iterator from those elements of *iterable* for which *function* returns true. iterable may be either a sequence, a container which supports iteration, or an iterator

```
#Using filter function
Sample_evenos=filter(lambda x: x%2==0,range(1,11))
#filter() returns an iterator.
print (Sample_evennos) #Output:<filter object at 0x0144EC10>
print (list(Sample_evennos))#Output:[2, 4, 6, 8, 10]

#Using List Comprehension
Demoevenno=[n for n in range(1,11) if n%2==0]
print ("Ouptut using list comprehension:-",Demoevenno) #Output:[2, 4, 6, 8, 10]
```

List Comprehension vs map.

map:Return an iterator that applies a *function* to every item of *iterable*, yielding the results.

```
#Using map() function
samplel1=map(lambda x:x*x,range(1,11))
#Returns an iterator(map object)
print (sapmlel1)#Output:<map object at 0x00C0EC10>
print (list(samplel1))#Output:[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

#Using List Comprehension
samplel2=[x*x for x in range(1,11)]
print ("Output display via list comprehension",samplel2)#Output:[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Nested loops in List Comprehension.

Flatten a list using List Comprehension with two 'for' clause:

```
demol1=[[1,2,3],[4,5,6],[7,8,9]]
demol2=[num2 for num1 in demol1 for num2 in num1]
print (demol2) #Output:[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Multiple if condition in List Comprehension.

```
#Finding numbers that are divisible by 2 and 3.

demol1=[1,2,3,4,5,6,7,8,9,10,11,12]
demol2=[n for n in demol1 if n%2==0 if n%3==0]
print (demol2)#Output:[6, 12]
```

We can mention expression as a tuple in a list comprehension. It should be written within parentheses. Otherwise, it will raise Error. The result will be a list of tuples.

```
# Creating a list of tuples using List Comprehension with two 'for' clause:

a1=['red','green','blue']
b1=[0,1,2]
a2=[(a,b) for a in a1 for b in b1]
print (a2)#Output:[('red', 0), ('red', 1), ('red', 2), ('green', 0), ('green', 1), ('green', 2), ('blue', 0), ('blue', 1), ('blue', 2)]
```

If the expression is a tuple and if not enclosed within parentheses, it will raise SyntaxError.

```
a1=['red','green','blue']
b1=[0,1,2]
a2=[a,b for a in a1 for b in b1]

#SyntaxError: invalid syntax
```

#Using zip() function in List Comprehension:

```
l1=['red','green','blue']
l2=[0,1,2]
l3=[(n1,n2) for n1,n2 in zip(l1,l2)]
print (l3)#Output:[('red', 0), ('green', 1), ('blue', 2)]
```

Set Comprehensions:

Set comprehension is written within curly braces{}.Returns new set based on existing iterables.

Return Type: Set

Syntax:{expression for item in iterable if conditional}

 # find even numbers using set Comprehension:

```
Demovar1={n for n in range(1,11) if n%2==0}
print (Demovar1) #Output:{2, 4, 6, 8, 10}
```

#find the square of numbers using Set Comprehension.

```
Demos1={n*n for n in range(1,11)}
#Sets are unordered.
print (Demos1) #Output: {64, 1, 4, 36, 100, 9, 16, 49, 81, 25}
```

If condition in Set comprehension:

Below example, we are calculating the square of even numbers.The expression can be a tuple, written within parentheses. We are using if condition to filter the even numbers.Sets are unordered. So elements are returned in any order.

```
sampleset={(n,n*n) for n in range(1,11) if n%2==0}
#Sets are unordered
print (sampleset)#Output:{(6, 36), (4, 16), (10, 100), (2, 4), (8, 64)}
print (type(sampleset))#Output:<class 'set'>
```

Dict comprehension

A dict comprehension, in contrast, to list and set comprehensions, needs two expressions separated with a colon followed by the usual "for" and "if" clauses. When the comprehension is run, the resulting key and value elements are inserted in the new dictionary in the order they are produced.

Dictionary comprehension is written within curly braces{}.In Expression key and value are separated by :

Syntax{key:value for (key,value) in iterable if conditional}

Return Type: dict

#square of numbers using Dictionary Comprehension.

```
d1={n:n*n for n in range(1,11)}
print (d1) #Output:{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81, 10: 100}
```

#two sets using dictionary comprehension:

```
d1={'color','shape','fruit'}
d2={'red','circle','apple'}
d3={k:v for (k,v) in zip(d1,d2)}
print (d3) #Output: {'fruit': 'circle', 'shape': 'red', 'color': 'apple'}
```

#calculating the square of even numbers.

```
d={n:n*n for n in range(1,11) if n%2==0}
print (d)#Output:{2: 4, 4: 16, 6: 36, 8: 64, 10: 100}
print (type(d))#Output:<class 'dict'>
```

#number of occurrences of each word in the list by calling count()on each word in the list.

```
#finding the number of occurrences of each character in the string
s="dictionary"
d={n:s.count(n) for n in s}
print (d)#Output:{'d': 1, 'i': 2, 'c': 1, 't': 1, 'o': 1, 'n': 1, 'a': 1, 'r': 1, 'y': 1}
```

```
# number of occurrences of each word in list using dict comprehension
l=["red","green","blue","red"]
d={n:l.count(n) for n in l}
print (d)#Output:{'red': 2, 'green': 1, 'blue': 1}
```