namedtuple( )

- It returns a tuple with a named entry, which means there will be a name assigned to each value in the tuple.
- It overcomes the problem of accessing the elements using the index values. With namedtuple( )

In [ ]:

```
from collections import namedtuple
```

In [ ]:

```
a = 'Gowtham', 'State Street'
print(type(a))
print(a)
```

In [ ]:

```
a = namedtuple('courses' , 'name, tech, year')
s = a('data science' , 'python', 2020)
print(s)
```

In [ ]:

```
# How To Create A namedtuple Using A List?
s._make(['data science' , 'python',2020])
```

In [ ]:

```
# Accessing the values
Student = namedtuple('Student', 'fname, lname, age')
s1 = Student('John', 'Clarke', '13')
print(s1.fname)
print(s1.lname)
print(s1.age)
```

In [ ]:

```
# With List
s2 = Student._make(['Adam','joe','18'])
print(s2)
```

In [ ]:

```
# Create a New Instance Using Existing Instance

s2 = s1._asdict()
print(s2)
```

In [ ]:

```
# Changing Field Values with _replace() Function

s2 = s1._replace(age='14')
print(s1)
print(s2)
```

deque

- deque pronounced as ‘deck’ is an optimized list to perform insertion and deletion easily.

In [ ]:

```
#creating a deque
from collections import deque

a = ['S' , 'T' , 'A' , 'T' , 'E']
a1 = deque(a)
print(a1)
```

In [ ]:

```
a = ["a","b","c"]
deq = deque(a)
print(deq)
```

In [ ]:

```
b = ["a",'a',"b","c",'a', 'b']
```

```
deq = deque(b)
print(deq.count("a"))
```

In [ ]:

```
# Now lets take a look at how we will insert and remove items from deque.

a1.append('a')
print(a1)

a1.appendleft('e')
print(a1)
```

In [ ]:

```
# inserting a component is enhanced utilizing deque, also you can remove components as well

a1.pop()
print(a1)


a1.popleft()
print(a1)
```

ChainMap

- It is a dictionary like class which is able to make a single view of multiple mappings.
- It basically returns a list of several other dictionaries. Suppose you have two dictionaries with several key value pairs,
- in this case ChainMap will make a single list with both the dictionaries in it.

In [ ]:

```
from collections import ChainMap
a = {1: 'Programming' , 2: 'python'}
b = {3: 'data science', 4: 'Machine learning'}
c = ChainMap(a,b)
print(c)
```

In [ ]:

```
dict1 = { 'a' : 1, 'b' : 2 }
dict2 = { 'c' : 3, 'b' : 4 }
chain_map = ChainMap(dict1, dict2)
print(chain_map.maps)
```

In [ ]:

```
print(chain_map['a'])
```

In [ ]:

```
dict2['c'] = 5
print(chain_map.maps)
```

In [ ]:

```
import collections

# initializing dictionaries
dic1 = { 'a' : 1, 'b' : 2 }
dic2 = { 'b' : 3, 'c' : 4 }

# initializing ChainMap
chain = collections.ChainMap(dic1, dic2)

# printing chainMap using maps
print ("All the ChainMap contents are : ")
print (chain.maps)

dic3 = { 'f' : 5 }

# using new_child() to add new dictionary
chain1 = chain.new_child(dic3)

# printing chainMap using map
print ("Displaying new ChainMap : ")
print (chain1.maps)
```

In [ ]:

```
dict3 = {'e' : 5, 'f' : 6}#Adding a New Dictionary to ChainMap
new_chain_map = chain_map.new_child(dict3)
print(new_chain_map)
```

- To access or insert elements we use the keys as index.
- But to add a new dictionary in the ChainMap we use the following approach.

In [ ]:

```
a1 = { 5: 'AI' , 6: 'neural networks'}
c1 = c.new_child(a1)
print(c1)
```

Counter

- It is a dictionary subclass which is used to count objects.

In [ ]:

```
from collections import Counter
```

In [ ]:

```
a = [1,1,1,1,2,3,3,4,3,3,4]
c = Counter(a)
print(c)
```

In [ ]:

```
a = [1,2,3,4,1,2,6,7,3,8,1]
Counter(a)
```

In [ ]:

```
Counter({1:3,2:4}) #the Counter() function can take a dictionary as an argument.
# In this dictionary, the value of a key should be the 'count' of that key.
```

In [ ]:

```
list = [1,2,3,4,1,2,6,7,3,8,1]
cnt = Counter(list)
print(cnt[1])
```

Apart from that, Counter has three additional functions:

- Elements
- Most_common([n])
- Subtract([interable-or-mapping])

In [ ]:

```
# The element() Function - returns an iterator object for the values in the Counter.

x = Counter("State Street")

# printing the elements of counter object
for i in x.elements():
    print( i, end = " ")
```

In [ ]:

```
# Example - 2
b = Counter({'State' : 4, 'Street' : 1,
            'Bellandur' : 2, 'python' : 3})

for i in b.elements():
    print ( i, end = " ")
print()

# Example - 3
c = Counter([1, 2, 21, 12, 2, 44, 5, 13, 15, 5, 19, 21, 5])

for i in c.elements():
    print ( i, end = " ")
print()
```

```
# Example - 4
d = Counter( a = 2, b = 3, c = 6, d = 1, e = 5)

for i in d.elements():
    print ( i, end = " ")
```

In [ ]:

```
# The most_common() Function

list = [1,2,3,4,1,2,6,7,3,8,1]
cnt = Counter(list)
print(cnt.most_common())
```

In [ ]:

```
coun = Counter(a=1, b=2, c=3, d=120, e=1, f=219)

# This prints 3 most frequent characters
for letter, count in coun.most_common(3):
    print('%s: %d' % (letter, count))
```

In [ ]:

```
# Arithmetic operations in counter
import collections

c1 = collections.Counter(['a', 'b', 'c', 'a', 'b', 'b'])
c2 = collections.Counter('alphabet')

print ('C1:', c1)
print ('C2:', c2)

print ('\nCombined counts:')
print (c1 + c2)

print ('\nSubtraction:')
print (c1 - c2)

print ('\nIntersection (taking positive minimums):')
print (c1 & c2)

print ('\nUnion (taking maximums):')
print (c1 | c2)
```

OrderedDict

- It is a dictionary subclass which remembers the order in which the entries were added.
- Basically, even if you change the value of the key,
- the position will not be changed because of the order in which it was inserted in the dictionary.

In [ ]:

```
from collections import OrderedDict
od = OrderedDict()
od[1] = 's'
od[2] = 't'
od[3] = 'a'
od[4] = 't'
od[5] = 'e'
print(od)
```

In [ ]:

```
od = OrderedDict()
od['a'] = 1
od['b'] = 2
od['c'] = 3
print(od)
```

In [ ]:

```
od.keys()
```

In [ ]:

```
od.values()
```

In [ ]:

```
for key, value in od.items():
    print(key, value)
```

In [ ]:

```
q = ["a","c","c","a","b","a","a","b","c"]
cnt = Counter(q)
od = OrderedDict(cnt.most_common())
for key, value in od.items():
    print(key, value)
```

Defaultdict

- Defaultdict is a sub-class of the dict class that returns a dictionary-like object.
- The functionality of both dictionaries and defualtdict are almost same except for the fact that defualtdict never raises a KeyError.
- It provides a default value for the key that does not exists.

In [ ]:

```
from collections import defaultdict
d = defaultdict(int)
#we have to specify a type as well.
d[1] = 'State'
d[2] = 'python'
print(d[3])
#it will give the output as 0 instead of keyerror.
```

In [ ]:

```
nums = defaultdict(int)
nums['one'] = 1
nums['two'] = 2
print(nums['three'])
```

In [ ]:

```
from collections import defaultdict

count = defaultdict(int)
names_list = "Mike John Mike Anna Mike John John Mike Mike Britney Smith Anna Smith".split()
for names in names_list:
    count[names] +=1
print(count)
```

In [ ]:

```
# Function to return a default values for keys that is not present
def def_value():
    return "Not Present"

# Defining the dict
d = defaultdict(def_value)
d["a"] = 1
d["b"] = 2

print(d["a"])
print(d["b"])
print(d["c"])
```

In [ ]:

```
# Defining the dict and passing
# lambda as default_factory argument
d = defaultdict(lambda: "Not Present")
d["a"] = 1
d["b"] = 2

print(d["a"])
print(d["b"])
print(d["c"])
```

Sets

- A Set is an unordered collection data type that is iterable, mutable and has no duplicate elements.
- The major advantage of using a set, as opposed to
- a list, is that it has a highly optimized method for checking whether a specific element is contained in the set.

In [ ]:

```
Set = set(["a", "b", "c"])

print("Set: ")
print(Set)

# Adding element to the set
Set.add("d")

print("\nSet after adding: ")
print(Set)
```

Frozen sets in Python are immutable objects that only support methods and operators that produce a result without affecting the frozen set or sets to which they are applied.

In [ ]:

```
normal_set = set(["a", "b","c"])

print("Normal Set")
print(normal_set)

# A frozen set
frozen_set = frozenset(["e", "f", "g"])

print("\nFrozen Set")
print(frozen_set)

# we are trying to add element to a frozen set
# frozen_set.add("h")
```

In [ ]:

```
# Adding Elements
people = {"Jay", "Idrish", "Archi"}

print("People:", end = " ")
print(people)

# This will add Daxit in the set
people.add("Daxit")

# Adding elements to the set using iterator
for i in range(1, 6):
    people.add(i)

print("\nSet after adding element:", end = " ")
print(people)
```

In [ ]:

```
# initialize my_set
my_set = {1,3}
print(my_set)

# my_set[0] # you will get an error TypeError: 'set' object does not support indexing

# add an element
my_set.add(2)
print(my_set)

# add multiple elements
my_set.update([2,3,4])
print(my_set)

# add list and set
my_set.update([4,5], {1,6,8})
print(my_set)
```

In [ ]:

```
# Union

people = {"Jay", "Idrish", "Archil"}
vampires = {"Karan", "Arjun"}
dracula = {"Deepanshu", "Raju"}

# Union using union() function
population = people.union(vampires)

print("Union using union() function")
print(population)

# Union using "|" operator
population = people|dracula

print("\nUnion using '|' operator")
print(population)
```

In [ ]:

```
# Intersection

set1 = set()
set2 = set()

for i in range(5):
    set1.add(i)

for i in range(3,9):
    set2.add(i)

# Intersection using
# intersection() function
set3 = set1.intersection(set2)

print("Intersection using intersection() function")
print(set3)

# Intersection using "&" operator
set3 = set1 & set2

print("\nIntersection using '&' operator")
print(set3)
```

In [ ]:

```
# Difference

set1 = set()
set2 = set()

for i in range(5):
    set1.add(i)

for i in range(3,9):
    set2.add(i)

# Difference of two sets
# using difference() function
set3 = set1.difference(set2)

print(" Difference of two sets using difference() function")
print(set3)

# Difference of two sets
# using '-' operator
set3 = set1 - set2

print("\nDifference of two sets using '-' operator")
print(set3)
```

In [ ]:

```
# Clear() method empties the whole set.
```

```
set1 = {1,2,3,4,5,6}

print("Initial set")
print(set1)

# This method will remove all the elements of the set
set1.clear()

print("\nSet after using clear() function")
print(set1)
```

In [ ]:

```
# Operations in set

# Creating two sets
set1 = set()
set2 = set()

# Adding elements to set1
for i in range(1, 6):
    set1.add(i)

# Adding elements to set2
for i in range(3, 8):
    set2.add(i)

print("Set1 = ", set1)
print("Set2 = ", set2)
print("\n")

# Union of set1 and set2
set3 = set1 | set2# set1.union(set2)
print("Union of Set1 & Set2: Set3 = ", set3)

# Intersection of set1 and set2
set4 = set1 & set2# set1.intersection(set2)
print("Intersection of Set1 & Set2: Set4 = ", set4)
print("\n")

# Checking relation between set3 and set4
if set3 > set4: # set3.issuperset(set4)
    print("Set3 is superset of Set4")
elif set3 < set4: # set3.issubset(set4)
    print("Set3 is subset of Set4")
else : # set3 == set4
    print("Set3 is same as Set4")

# displaying relation between set4 and set3
if set4 < set3: # set4.issubset(set3)
    print("Set4 is subset of Set3")
    print("\n")

# difference between set3 and set4
set5 = set3 - set4
print("Elements in Set3 and not in Set4: Set5 = ", set5)
print("\n")

# checkv if set4 and set5 are disjoint sets
if set4.isdisjoint(set5):
    print("Set4 and Set5 have nothing in common\n")

# Removing all the values of set5
set5.clear()

print("After applying clear on sets Set5: ")
print("Set5 = ", set5)

# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

# use ^ operator for symmetric difference
print('\nThe elements from both the sets without anything in common', A ^ B)
```

In [ ]:

```
# initialize my set
# initialize my_set
my_set = {1, 3, 4, 5, 6}
print(my_set)

# discard an element
my_set.discard(4)
print(my_set)

# remove an element
my_set.remove(6)
print(my_set)

# discard an element
# not present in my_set
my_set.discard(2)
print(my_set)

# remove an element not present in my_set you will get an error Output: KeyError: 2

# my_set.remove(2)
```

Global and Local Variables in Python

- Global variables are the one that are defined and declared outside a function and we need to use them inside a function.

In [ ]:

```
# This function uses global variable s
def f():
    print(s)

# Global scope
s = "State Street"
f()
```

If a variable with same name is defined inside the scope of function as well then it will print the value given inside the function only and not the global value.

In [ ]:

```
# This function has a variable with name same as s.
def f():
    s = "Bellandur"
    print(s)

# Global scope
s = "State Street"
f()
print(s)
```

In [ ]:

```
def f():
    print(s)
    s = "State Street"


# Global scope
s = "Bellandur"
f()
print(s)
```

In [ ]:

```
a = 1 # a global Variable

# Uses global because there is no local 'a'
def f():
    print ('Inside f() : ', a)

# Variable 'a' is redefined as a local
def g():
    a = 2
    print ('Inside g() : ',a)

# Uses global keyword to modify global 'a'
def h():
    global a
    a = 3
    print ('Inside h() : ',a)

# Global scope
print ('global : ',a)
f()
print ('global : ',a)
g()
print ('global : ',a)
h()
print ('global : ',a)
print('\nThe new value of a is', a)
```