😊 Collections Module

1. The collections module is a built-in module that implements specialized container data types providing alternatives to Python's general purpose built-in containers. We've already gone over the basics: **dict, list, set, and tuple**.
2. Now we'll learn about the alternatives that the collections module provides.
3. ***Counter*** is a *dict* subclass which helps count hashable objects. Inside of it elements are stored as dictionary keys and the counts of the objects are stored as the value.
4. We start by importing counter from the collections module in our Jupyter Notebook. Then we created a list with some numbers in it.
5. After that we used the counter to see our list and here you can see that we get the values which says that there are 6 instances of 1, 2, then 4 instances of 3 and so on.

```
[2]: from collections import Counter
```

Counter() with lists

```
[5]: lst = [1,2,2,2,2,3,3,3,1,2,1,12,3,2,32,1,21,1,223,1]

     Counter(lst)
```

```
[5]: Counter({1: 6, 2: 6, 3: 4, 12: 1, 32: 1, 21: 1, 223: 1})
```

Counter with strings

```
[8]: Counter('aabsbsbsbhshhbbsbs')
```

```
[8]: Counter({'b': 7, 's': 6, 'h': 3, 'a': 2})
```

6. Now we are looking at the counter with strings here. In code number 8, we looked at the counter for some string, then in the 11th code cell we created a string with a proper sentence, then we split to look at each work, after that we used the counter to see how many times a single word occurs.

Counter with strings

```
[8]: Counter('aabsbsbsbhshhbbsbs')
```

```
[8]: Counter({'b': 7, 's': 6, 'h': 3, 'a': 2})
```

Counter with words in a sentence

```
[11]: s = 'How many times does each word show up in this sentence word times each each word'

words = s.split()

Counter(words)
```

```
[11]: Counter({'each': 3,
         'word': 3,
         'times': 2,
         'How': 1,
         'many': 1,
         'does': 1,
         'show': 1,
         'up': 1,
         'in': 1,
         'this': 1,
         'sentence': 1})
```

7. We can also look at the most common words using counter.

```
[13]: # Methods with Counter()
c = Counter(words)

c.most_common(2)
```

```
[13]: [('each', 3), ('word', 3)]
```

8. **defaultdict** is a dictionary-like object which provides all methods provided by a dictionary but takes a first argument (default_factory) as a default data type for the dictionary. Using **defaultdict** is faster than doing the same using dict.set_default method.
9. The code starts by importing defaultdict from the collections module. First, a normal dictionary is created. Attempting to access a key ('one') in that normal dictionary will cause an error if the key is not present. Then the dictionary is reassigned to a defaultdict with a default factory of object. This means that if you try to access a missing key, instead of raising an error, defaultdict will automatically create a new object instance as the default value for that key.
10. Then, initially d is a dictionary (or possibly a defaultdict with a different default factory) and accessing d['one'] may either raise an error or return a previously set value. The for loop iterates over the keys in d and prints them. Later, d is reassigned to a defaultdict with a lambda that returns 0. With this new default factory, if you try to access d['one'] and the key is missing, it automatically creates the key with a value of 0 and returns 0.

```
[18]: from collections import defaultdict

[20]: d = {}

[22]: d['one']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[22], line 1
----> 1 d['one']

KeyError: 'one'
```

```
[29]: d  = defaultdict(object)

[25]: d['one']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[25], line 1
----> 1 d['one']

KeyError: 'one'
```

```
[27]: for item in d:
          print(item)
```

Can also initialize with default values:

```
[31]: d = defaultdict(lambda: 0)

[33]: d['one']

[33]: 0
```

11. The code starts by creating a simple tuple with three numbers and retrieves its first element using indexing. Then it uses **namedtuple** from the collections module to define a custom data type called "Dog" with fields for age, breed, and name. Two Dog objects are created ("sam" and "frank"). The code demonstrates accessing namedtuple elements either by attribute (for example, sam.age or sam.breed) or by index (sam[0] returns the age).

```
[35]: t = (12,13,14)

[37]: t[0]

[37]: 12
```

[39]:
```
from collections import namedtuple
```

[41]:
```
Dog = namedtuple('Dog',['age','breed','name'])

sam = Dog(age=2,breed='Lab',name='Sammy')

frank = Dog(age=2,breed='Shepard',name="Frankie")
```

[43]:
```
sam
```

[43]:
```
Dog(age=2, breed='Lab', name='Sammy')
```

[45]:
```
sam.age
```

[45]:
```
2
```

[47]:
```
sam.breed
```

[47]:
```
'Lab'
```

[49]:
```
sam[0]
```

[49]:
```
2
```