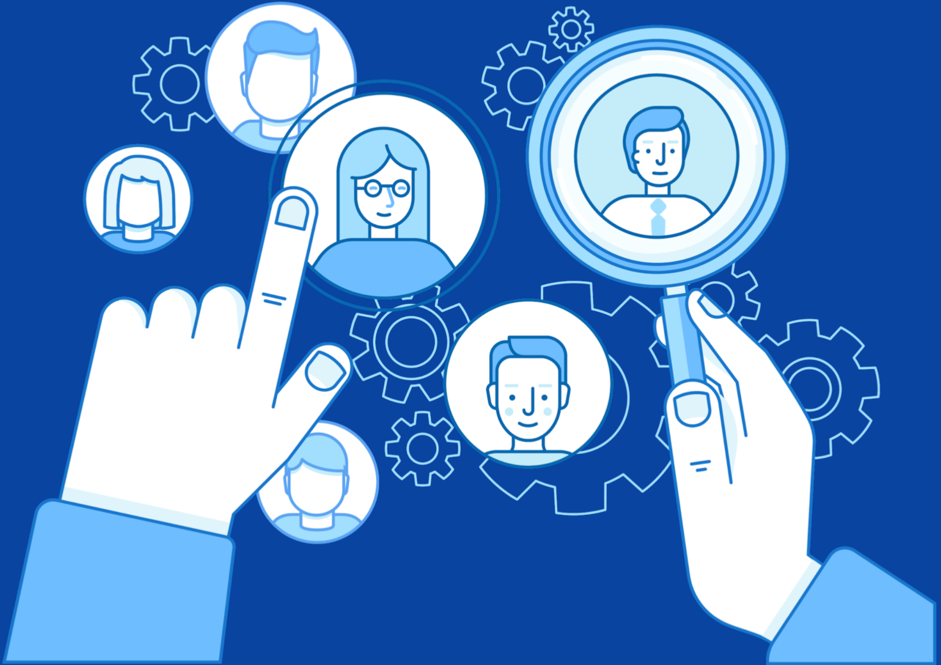
# Digital Career Institute

## Python Course - Collections



Digital Career Institute

# The Collections Module

# The Collections Module

Python has a built-in module specifically for collections.

This module is named `collections`.

This module implements specialized container datatypes providing alternatives to Python’s general purpose built-in containers.

# Collections: Counter

Counting how many occurrences of the same value are found in an iterable requires at least 5 lines of code.

```
>>> fridge = [
...     "Apple", "Apple", "Cabbage",
...     "Steak", "Cheese", "Apple",
...     "Carrot", "Carrot", "Iogurt",
...     "Beer"
... ]
>>> counter = {}
>>> for ingredient in fridge:
...     if ingredient not in counter:
...         counter[ingredient] = 0
...     counter[ingredient] += 1
...
>>> print(counter)
{'Apple': 3, 'Cabbage': 1, 'Steak': 1, 'Cheese': 1,
'Carrot': 2, 'Iogurt': 1, 'Beer': 1}
```

# Collections: Counter

The `Counter` constructor returns a counter object with the same information and requires only one line of code, which means less room for bugs and, therefore, a lower maintenance cost.

The counter object is a dictionary-like object and can be used in the same way.

A counter can also be created empty or passing items as keyword arguments.

```
>>> from collections import Counter
>>> fridge = [
...       "Apple", "Apple", "Cabbage",
...       "Steak", "Cheese", "Apple",
...       "Carrot", "Carrot", "Iogurt",
...       "Beer"
... ]
>>> counter = Counter(fridge)
>>> print(counter)
Counter({'Apple': 3, 'Carrot': 2,
'Cabbage': 1, 'Steak': 1, 'Cheese': 1,
'Iogurt': 1, 'Beer': 1})
>>> counter = Counter(Apple=3, Beer=1)
```

# Collections: Counter

The `Counter` object has additional methods specific to this kind of dataset.

The method `total` will return the sum of all the occurrences, which should be the length of the iterable passed to the `Counter` constructor.

```
>>> from collections import Counter
>>> fridge = [
...         "Apple", "Apple", "Cabbage",
...         "Steak", "Cheese", "Apple",
...         "Carrot", "Carrot", "Iogurt",
...         "Beer"
... ]
>>> counter = Counter(fridge)
>>> print(counter.total())
10
>>> print(len(fridge))
10
```

**New in Python 3.10**

# Collections: Counter

The method `subtract` will **subtract** a counter from another counter.

This will modify the original counter and will return `None`.

The minus operator `–` can also be used to obtain a similar result but returning a new counter, and not changing the original counter.

This operator also removes any elements with a counter of 0.

```
>>> from collections import Counter
>>> fridge = [
...       "Apple", "Apple", "Cabbage",
...       "Steak", "Cheese", "Apple",
...       "Carrot", "Carrot", "Iogurt",
...       "Beer"
... ]
>>> counter = Counter(fridge)
>>> lunch = Counter(Cabbage=1, Carrot=2)
>>> counter.subtract(lunch)
>>> print(counter)
Counter({'Apple': 3, 'Steak': 1,
'Cheese': 1, 'Iogurt': 1, 'Beer': 1,
'Cabbage': 0, 'Carrot': 0})
```

# Collections: Counter

The method `update` will **add** a counter to another counter.

This will modify the original counter and will return `None`.

The plus operator `+` can also be used to obtain a similar result but returning a new counter, and not changing the original counter.

```
>>> from collections import Counter
>>> fridge = [
...       "Apple", "Apple", "Cabbage",
...       "Steak", "Cheese", "Apple",
...       "Carrot", "Carrot", "Iogurt",
...       "Beer"
... ]
>>> counter = Counter(fridge)
>>> shop = Counter(Carrot=6, Beer=6)
>>> counter.update(shop)
>>> print(counter)
Counter({'Carrot': 8, 'Beer': 7, 'Apple':
3, 'Cabbage': 1, 'Steak': 1, 'Cheese': 1,
'Iogurt': 1})
```

# Collections: Counter

Notice that even though printing the counter shows the keys sorted by occurrence, when it is iterated it does not follow the same order.

```
>>> counter = Counter(fridge)
>>> print(counter)
Counter({'Apple': 3, 'Carrot': 2,
'Cabbage': 1, 'Steak': 1, 'Cheese': 1,
'Iogurt': 1, 'Beer': 1})
>>> for item in counter:
...     print(item)
...
Apple
Cabbage
Steak
Cheese
Carrot
Iogurt
Beer
```

# Collections: Counter

The method `most_common` will return a list sorted by occurrence in descending order that can be iterated in the same order.

The items of the list will be tuples containing both the value and the counter of each item.

This method accepts a positional argument to limit the amount of items returned (i.e: get the `n` most repeated items).

```
>>> counter = Counter(fridge)
>>> print(counter)
Counter({'Apple': 3, 'Carrot': 2,
'Cabbage': 1, 'Steak': 1, 'Cheese': 1,
'Iogurt': 1, 'Beer': 1})
>>> for item in counter.most_common():
...     print(item)
...
('Apple', 3)
('Carrot', 2)
('Cabbage', 1)
('Steak', 1)
('Cheese', 1)
('Iogurt', 1)
('Beer', 1)
```

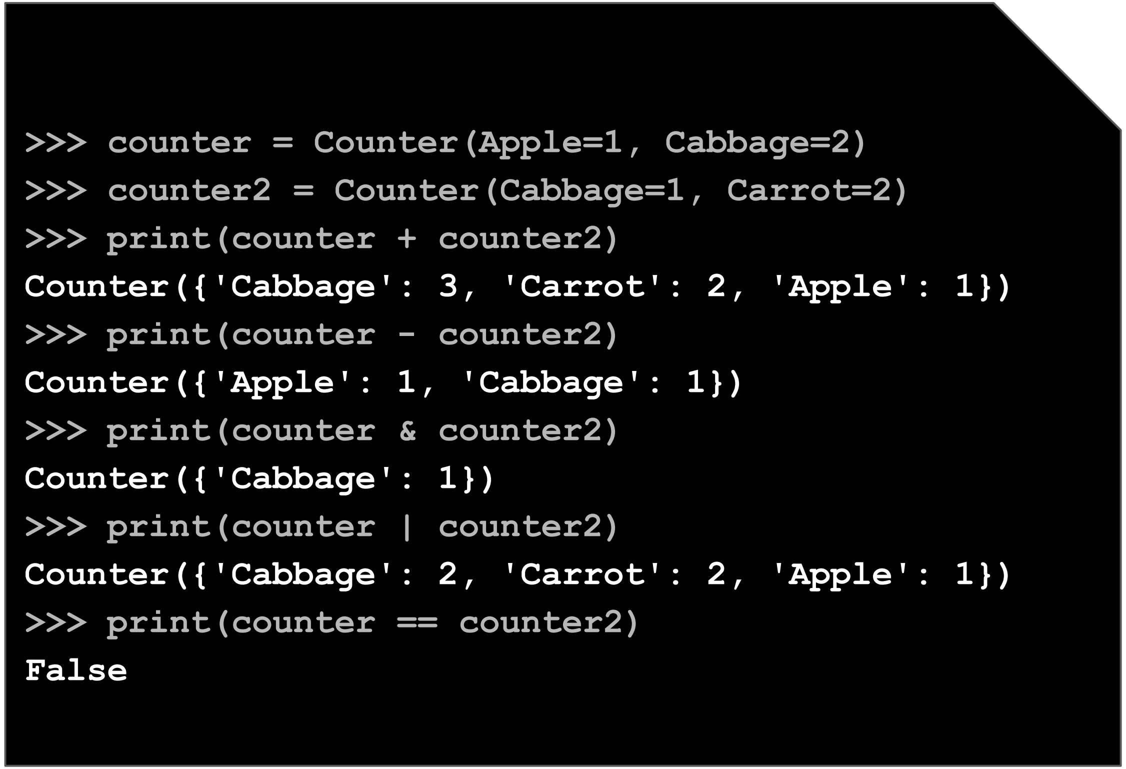
# Collections: Counter

The method **`clear`** will empty the counter and remove all the items.

```
>>> counter = Counter(fridge)
>>> print(counter)
Counter({'Apple': 3, 'Carrot': 2,
'Cabbage': 1, 'Steak': 1, 'Cheese': 1,
'Iogurt': 1, 'Beer': 1})
>>> counter.clear()
>>> print(counter)
Counter()
```

# Collections: Counter

Counters can also be used with some binary operators.

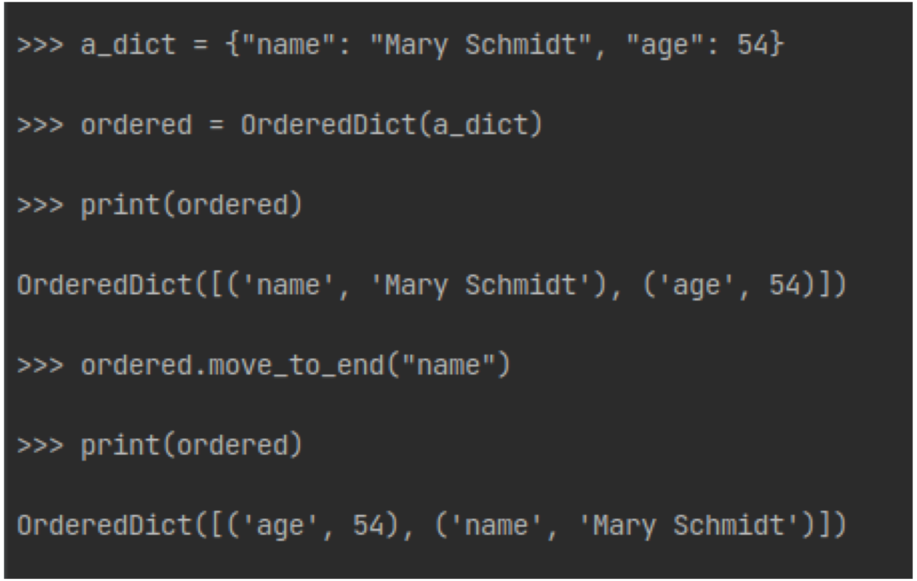


```
>>> counter = Counter(Apple=1, Cabbage=2)
>>> counter2 = Counter(Cabbage=1, Carrot=2)
>>> print(counter + counter2)
Counter({'Cabbage': 3, 'Carrot': 2, 'Apple': 1})
>>> print(counter - counter2)
Counter({'Apple': 1, 'Cabbage': 1})
>>> print(counter & counter2)
Counter({'Cabbage': 1})
>>> print(counter | counter2)
Counter({'Cabbage': 2, 'Carrot': 2, 'Apple': 1})
>>> print(counter == counter2)
False
```

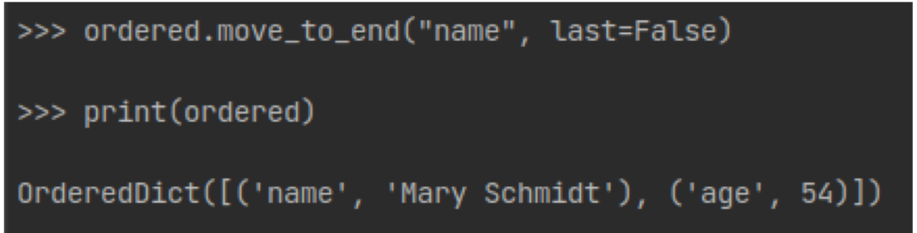
# Collections: OrderedDict

The type OrderedDict is like a standard dict in most Python versions, but it packs some additional methods specifically for managing the order.

The method **`move_to_end`** will move the item with the given key to one end of the dictionary. By default it will be the right end, but the argument **`last=False`** will move it to the left.



```
>>> a_dict = {"name": "Mary Schmidt", "age": 54}

>>> ordered = OrderedDict(a_dict)

>>> print(ordered)

OrderedDict([('name', 'Mary Schmidt'), ('age', 54)])

>>> ordered.move_to_end("name")

>>> print(ordered)

OrderedDict([('age', 54), ('name', 'Mary Schmidt')])
```



```
>>> ordered.move_to_end("name", last=False)

>>> print(ordered)

OrderedDict([('name', 'Mary Schmidt'), ('age', 54)])
```

# Collections: ChainMap

The **`ChainMap`** type is very similar to the **`update`** method in dictionaries.

It merges together a series of dictionaries, but instead of updating one of them it just returns a new object. Items are merged from right to left.

The **`ChainMap`** object is a dictionary-like object and its items can be accessed using the keys.

```
>>> from collections import ChainMap
>>> root = {"a": 1, "b": 2}
>>> adjust1 = {"b": 3, "c": 4}
>>> chain = ChainMap(adjust1, root)
>>> print(chain)
ChainMap({'b': 3, 'c': 4}, {'a': 1, 'b': 2})
>>> print(dict(chain))
{'a': 1, 'b': 3, 'c': 4}
>>> print(chain['a'])
1
>>> print(chain['b'])
3
>>> print(chain['c'])
4
```

# Collections: ChainMap

The **`ChainMap`** is more than just a method of updating dictionaries, it has memory. It remembers which dictionaries were merged to produce the object.

It has a property **`maps`**, as a list of input elements. This list can be manipulated and the main object will be changed.

```
>>> root = {"a": 1, "b": 2}
>>> adjust1 = {"b": 3, "c": 4}
>>> adjust2 = {"c": 5, "d": 6}
>>> chain = ChainMap(adjust2, adjust1, root)
>>> print(chain)
ChainMap({'c': 5, 'd': 6}, {'b': 3, 'c': 4},...
>>> print(chain.maps)
[{'c': 5, 'd': 6}, {'b': 3, 'c': 4}, {'a': 1,...
>>> chain.maps[0]['a'] = 100
>>> print(chain.maps)
[{'c': 5, 'd': 6, 'a': 100}, {'b': 3, 'c': 4},...
>>> print(chain['a'])
100
```

# Collections: ChainMap

A **`ChainMap`** is a dictionary-like object and can also be used as an argument of another **`ChainMap`**.

This is used often to provide configuration objects that can work in different contexts. Each **`ChainMap`** in the tree represents a context that inherits from another parent or default context.

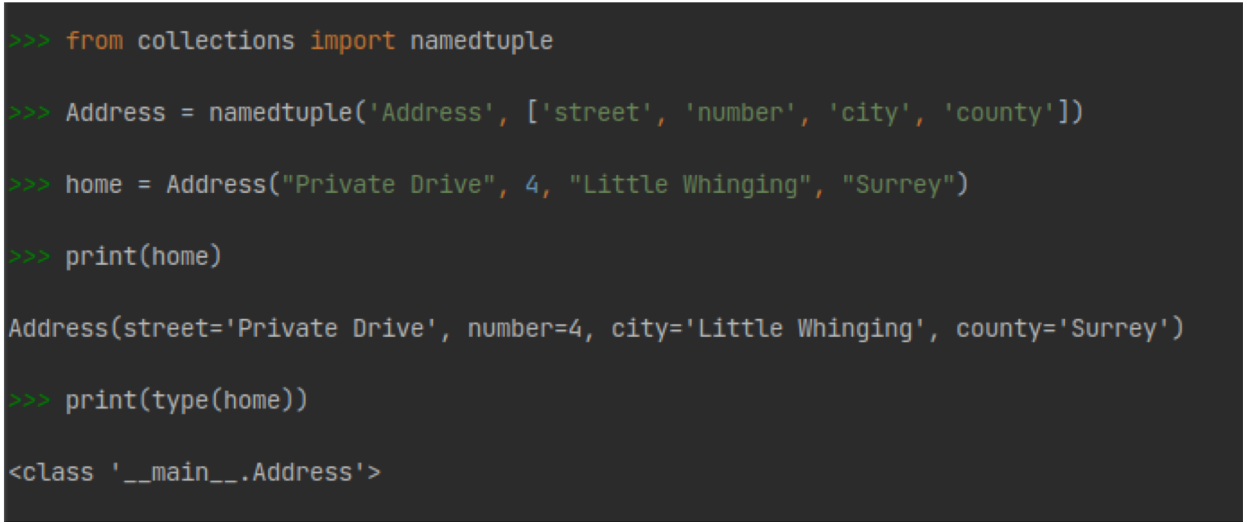
```
>>> root = {"a": 1, "b": 2}
>>> adjust1 = {"b": 3, "c": 4}
>>> adjust2 = {"c": 5, "d": 6}
>>> chain1 = ChainMap(adjust1, root)
>>> chain2 = ChainMap(adjust2, chain1)
>>> print(chain2)
ChainMap({'c': 5, 'd': 6}, ChainMap({'b': 3, 'c':
4}, {'a': 1, 'b': 2}))
>>> print(chain2.maps[0])
{'c': 5, 'd': 6}
>>> print(chain2.maps[1])
ChainMap({'b': 3, 'c': 4}, {'a': 1, 'b': 2})
>>> print(chain2['c'])
5
>>> print(chain2.maps[1]['c'])
4
```

# Collections: NamedTuple

A `namedtuple` creates a new custom data type with the name and attributes indicated.

It requires a string for the name of the type and a list of attributes.

Creating new objects of that type can be done by using the constructor and passing the data as positional arguments.



```
>>> from collections import namedtuple

>>> Address = namedtuple('Address', ['street', 'number', 'city', 'county'])

>>> home = Address("Private Drive", 4, "Little Whinging", "Surrey")

>>> print(home)

Address(street='Private Drive', number=4, city='Little Whinging', county='Surrey')

>>> print(type(home))

<class '__main__.Address'>
```

# Collections: NamedTuple

The elements in a `namedtuple` have an implicit order and can be accessed using indices.

They can also be accessed using **dot notation**. This often provides a more readable code.

It is often used when dealing with CSV files and other tabular read-only data.

```
>>> home = Address(
...     "Private Drive", 4, "Little Whinging",
...     "Surrey"
... )
...
>>> print(home[0])
Private Drive
>>> print(home.street)
Private Drive
```

# Collections: NamedTuple

The objects created this way are tuples, they do not allow changing the values or adding new keys.

```
>>> home[0] = "Somewhere else"
TypeError: 'Address' object does not support item
assignment
>>> home.street = "Somewhere else"
AttributeError: can't set attribute
>>> home.country = "Neverland"
AttributeError: 'Address' object has no attribute
'country'
```

All fields must have values.

```
>>> home = MyAddress("Private Drive", 4)
TypeError: <lambda>() missing 2 required positional
arguments: 'city' and 'county'
```

# Collections: NamedTuple

A **`namedtuple`** has some useful methods.

The **`_asdict`** method will return a dictionary with the data.

The **`_replace`** method will return a new object with the new values given.

The original object is still read-only and does not change.

```
>>> print(home._asdict())
{'street': 'Private Drive', 'number': 4, 'city':
'Little Whinging', 'county': 'Surrey'}

>>> print(home._replace(number=6))
Address(street='Private Drive', number=6,
city='Little Whinging', county='Surrey')

>>> print(home)
Address(street='Private Drive', number=4,
city='Little Whinging', county='Surrey')
```

# We learned ...

- That Python has various built-in types that are a bit more complex and addressed to be used in more specific situations.
- That there is a `Counter` type that is a type of dictionary specific for dealing with counters.
- That a counter can be created passing an iterable and the object will contain the number of occurrences of the same value in the iterable.
- That there is a `ChainMap` type that is used to merge different dictionaries and remember each of the inputs.
- That there is a `namedtuple` type that is used to store read-only data that can be accessed using both integer indices and named keys.

# Self Study

- Explore methods of sorting the different types of collections.
- Include the `collections` module datatypes.

Digital Career Institute

# Documentation

# Documentation

- Python List (With Examples) - programiz
- Python Set (With Examples) - programiz
- Python Dictionary - Geeks for Geeks
- collections — Container datatypes — Python documentation
- Write Pythonic and Clean Code With namedtuple – Real Python

# THANK YOU

**Contact Details**
**DCI Digital Career Institute gGmbH**

Digital Career Institute
DCI