

# Digital Career Institute

## Python Course - Collections



# Goal of the Submodule

The goal of this submodule is to help the learners work with collections in Python. By the end of this submodule, the learners should be able to understand

- What is a collection
- Which are the built-in collection types in Python
- The characteristics and differences between each collection type.
- How to choose the right data type for each situation.
- The additional data types provided by the `collections` Python module.
- How to use counters, ordered dictionaries, chain maps and named tuples.

# Topics

- Collections and iterables.
  - Types of collections: linear, associative and graphs.
  - Linear collections: types and I/O methods.
  - Associative collections: sets and arrays.
- Lists.
- Tuples.
- Sets.
- Dictionaries.
- Operations with iterables.
  - Iterating iterables.
  - Iterable functions.
- The `collections` Python module.
  - Counters.
  - Ordered dictionaries.
  - Chain maps.
  - Named tuples.

# Collections

# What is a Collection?



# What is a Collection?



The screenshot shows the Merriam-Webster website with the word "collection" entered in the search bar. The page displays the word "collection" as a noun, along with its pronunciation and a "Save Word" button. The definition is provided in two main parts: 1. The act or process of collecting, with examples like "the collection of data" and "the collection of taxes". 2. A group of collected items, with examples like "an accumulation of objects gathered for study, comparison, or exhibition or as a hobby", "a collection of poems", "a collection of photographs", and "a baseball card collection". It also includes sub-definitions for "GROUP, AGGREGATE" (e.g., "a collection of symptoms") and "a set of apparel designed for sale usually in a particular season" (e.g., "the designer's spring collection of dresses").

merriam-webster.com/dictionary/collection

Merriam-Webster SINCE 1828

collection

Dictionary Thesaurus

**collection** noun

Save Word

col·lec·tion | \ kə-'lek-shən \

**Definition of collection**

1 : the act or process of collecting  
// the *collection* of data  
// the *collection* of taxes

2 a : something collected  
*especially* : an accumulation of objects gathered for study, comparison, or exhibition or as a hobby  
// a *collection* of poems  
// a *collection* of photographs  
// a baseball card *collection*

b : GROUP, AGGREGATE  
// a *collection* of symptoms, such as fatigue, headaches, and joint pain

c : a set of apparel designed for sale usually in a particular season  
// the designer's spring *collection* of dresses

An **accumulation** of objects.

A **group** of objects.

An **aggregate** of objects.



# What is a Collection?

A collection of **groceries**



A collection of **computers**



A collection of **customers**



# What is a Collection?

H	e	l	l	o		W	o	r	l	d
---	---	---	---	---	--	---	---	---	---	---

A text string is also a collection, of characters.



# What is a Collection?

In computer programming a collection is a type of **iterable**.

An iterable is any type, which value can be decomposed into different members, and its members can be returned one at a time (i.e. in `for` loops).

# Storing Collections: Linear

Objects of a collection can be stored in a line, one after the other.



These are called **Linear Collections**.

## Examples:

- A queue of customers.
- A list of students.
- The steps in a recipe.
- The names of one's children.

# Storing Linear Collections

H e l l o   w o r l d

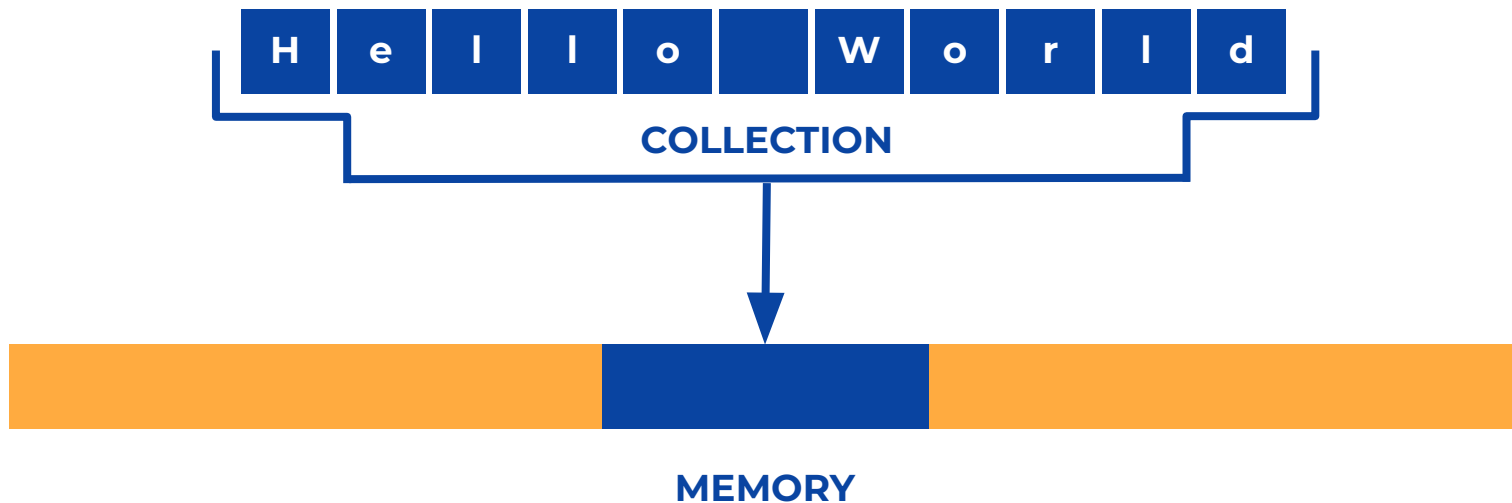
Placing the objects in line implicitly defines an order.  
A different order of the same elements is a different sequence.

r o l l e d   w o H l

A linear collection is called  
a **sequence** In Python.

# Storing Linear Collections

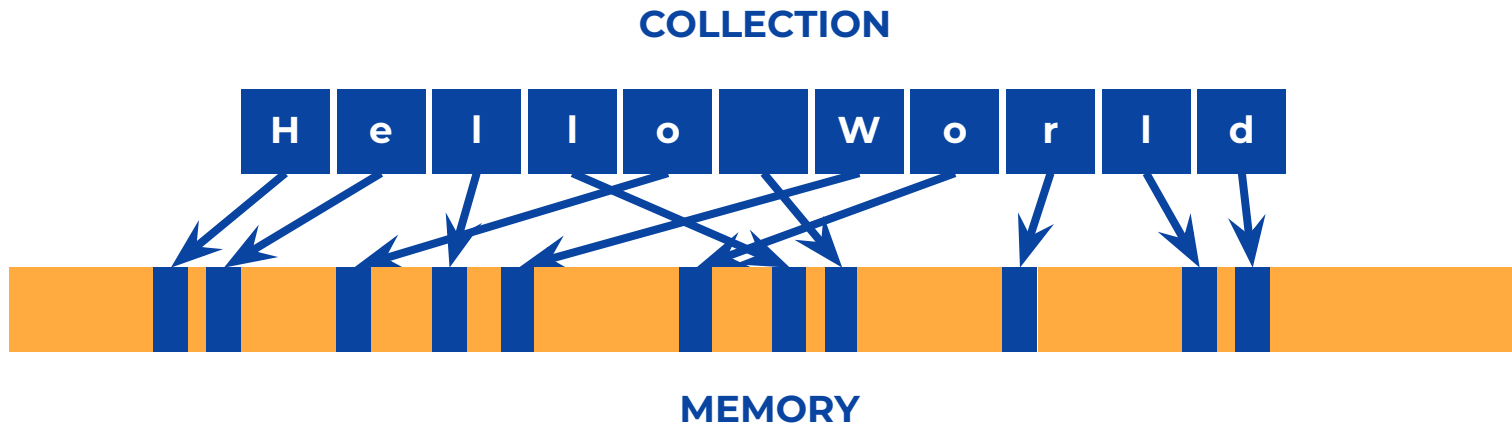
To identify the order, the objects are often stored using contiguous spaces in memory. Then, the order of the sequence is **implicit** in memory.



Most language data types use this approach.

# Storing Linear Collections

Items can also be stored in different locations in memory.  
Then, the order of the sequence must be made **explicit**.



To identify the order, each item will hold its value  
and a link to the next item in the sequence.

These are called **Linked Lists** and are not natively implemented in Python.

# Using Collections

In computer science, collections are objects with properties and methods.

READ

ADD

REMOVE

SIZE

Different types of collections may have different methods, but reading, adding and removing elements are common.

Adding is commonly referred as **Push**, removing as **Pop**, reading as **Lookup**, and the size as **Length**.

# Reading Linear Collections

The objects in a linear collection can be accessed using numeric indexes that identify their position in the sequence.

```
collection[0] = Box 1
```

Indexes start at 0 and increase by 1.

They are not permanently associated to the object. If `Box 1` is removed, the rest of the boxes drop one position and the object associated with the index 0 will be `Box 2`.

INDEX	OBJECT
[ 3 ]	BOX 4
[ 2 ]	BOX 3
[ 1 ]	BOX 2
[ 0 ]	BOX 1

COLLECTION

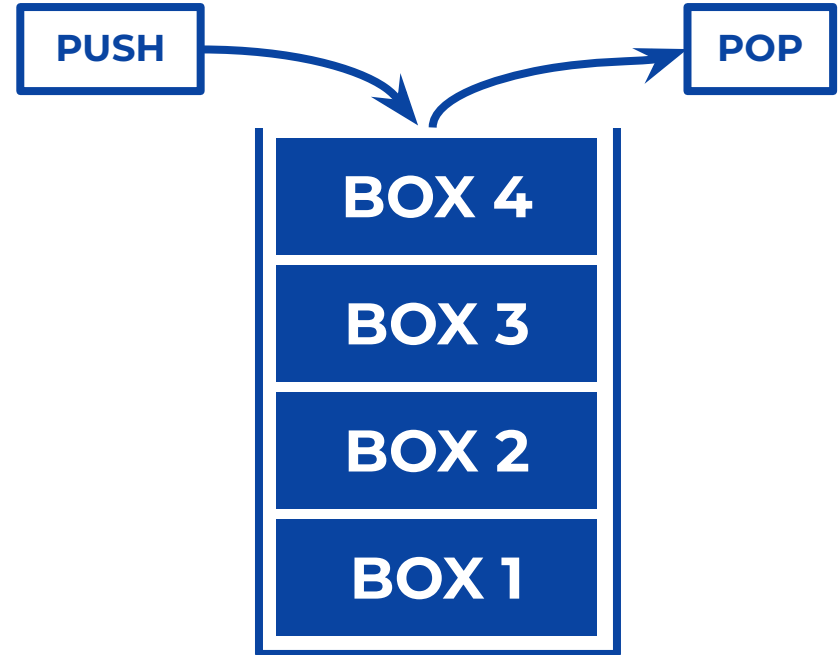


# Types of Linear Collections: Stacks

A pile of items is called a **stack** in computer science.

Stacks are linear collections where items can be added at one end and removed from the same end.

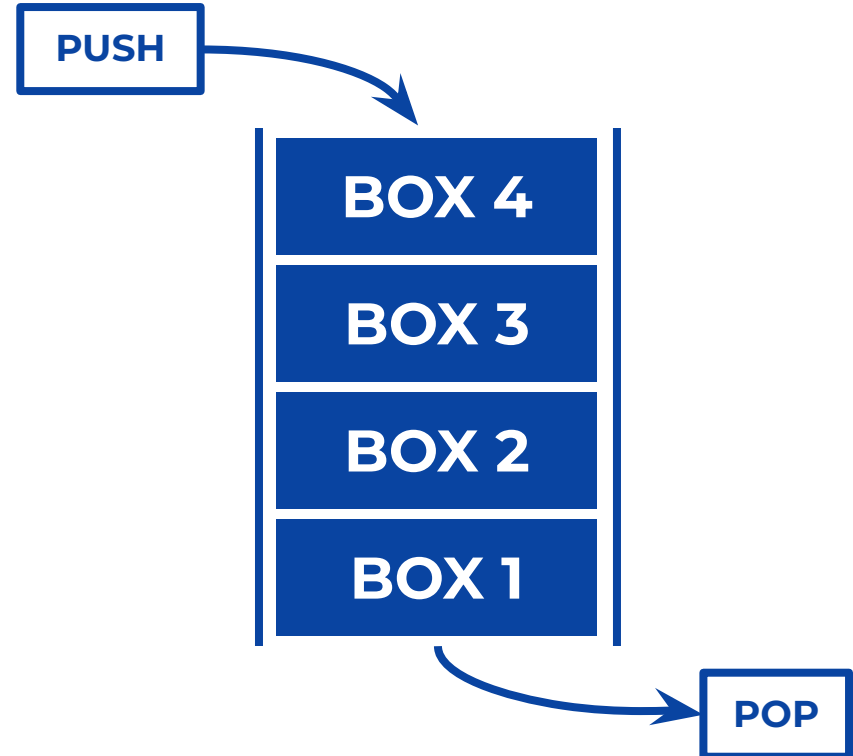
This method of manipulating linear collections is called **LIFO** (Last In, First Out).



# Types of Linear Collections: Queues

A **queue** is a linear collection that, instead, adds the new items on one end and removes them from the other.

This method of manipulating linear collections is called **FIFO** (First In, First Out).

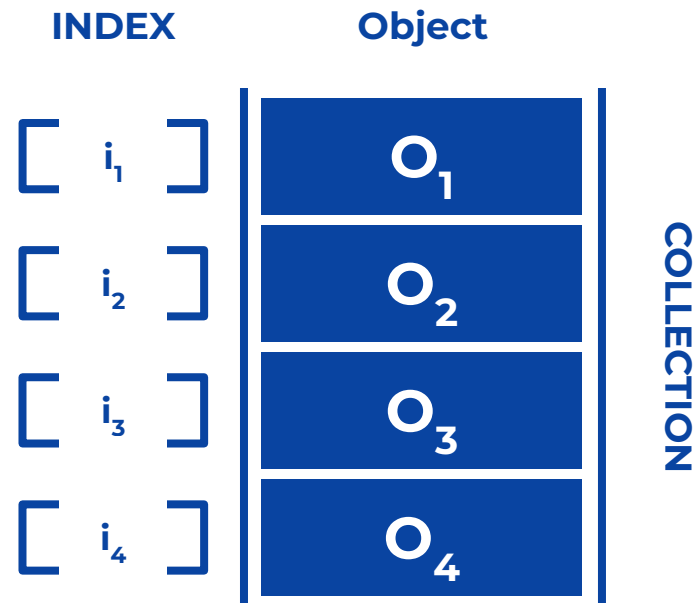


# Storing Collections: Associative

Using linear collections implicitly defines an order, and adding or removing objects may change the index associated to each value.

Associative Collections provide a method to assign each index to the same value, even if the rest of the objects are removed.

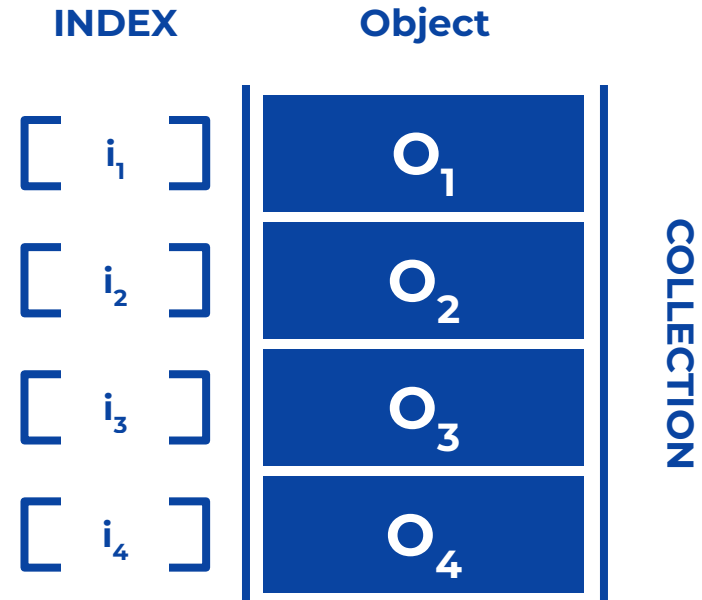
The index  $i_2$  will always link to the object  $o_2$ , even if the object  $o_1$  is removed from the collection.



# Storing Collections: Associative

**Examples:**

- A dictionary.
- The shopping list.
- The ingredients in a recipe.
- A user profile.



# Types of Associative Collections: Arrays

Associative arrays are collections that use a key to index each one of its elements.

This key is usually a string that defines the value associated with it. It is also often called a collection of **key-value** pairs.

Associative arrays often have methods to see if a key exists, to remove a key, access all keys, ...

**Example:** a user profile.

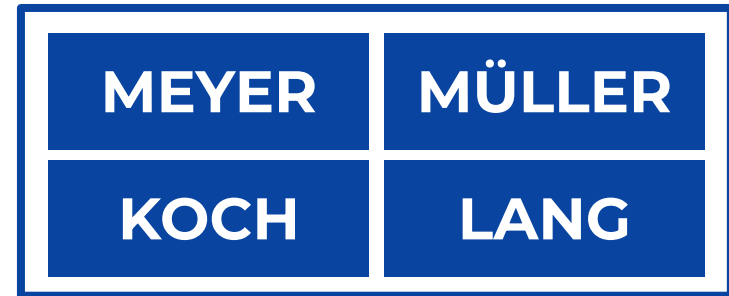
INDEX	Object	COLLECTION
[name]	MARY	
[address]	STREET...	
[age]	25	
[phone]	61691...	

# Types of Associative Collections: Sets

A **Set** is an unordered collection where duplicates are not allowed. They replicate the behavior of sets from **Set Theory**.

Sets often have methods to see if a value exists in the set and also provide specific methods to add and remove objects.

**Example:** german family names.

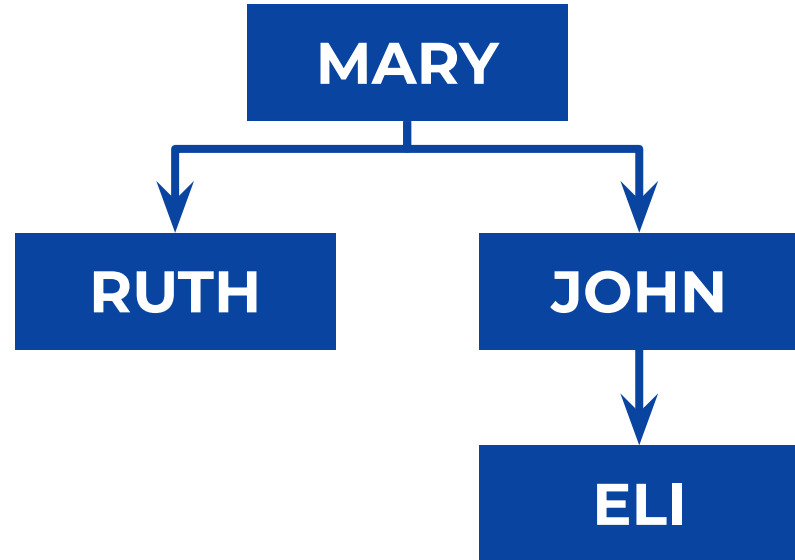


# Storing Collections: Graphs

Graphs are collections of node objects, each of which refers to one or more other objects and has its own properties and methods to add, remove and read its elements.

**Examples:**

- The road network.
- The fiber network.
- The employees in a company.
- The genealogy.





# Types of Collections: Summary

**LINEAR**

**ASSOCIATIVE**

**GRAPHS**

**Linked Lists**

**Stacks**

**Queues**

**Arrays**

**Sets**

# We learned ...

- What is a collection.
- That there are different types of collections: linear, associative and graphs.
- That linear collections have an order, which can be implicit or explicit (linked lists).
- How to access and manipulate stacks and queues.
- That removing the first object of a linear collection will change the index associated to the rest of the objects.
- That associative collections use other kinds of indexes to access their objects.
- That sets do not allow duplicate values in the collection.

# Self Study



- Explore the different types of Python built-in collections:
  - Strings
  - Lists
  - Tuples
  - Sets
  - Dictionaries

# Collections in Python

# Collections in Python

## LINEAR

**LIST**

**TUPLE**

## ASSOCIATIVE

**DICTIONARY**

**SET**

# Lists

## Collections in Python

**Lists** are Python's basic implementation of linear collections.

They:

- Have an **order**.
- **Allow duplicate** values.
- Allow their objects to be **changed**.
- Allow objects of **different types**.



# Defining Lists

They are defined using square brackets `[]`.

Any **iterable** can be converted into a list by using the `list` constructor.

The `list` constructor can also be used to create empty lists if no argument is given.

The output is the same as using empty square brackets `[]`.

```
>>> fridge = [
...     "Apple", "Apple",
...     "Cabbage", "Steak",
...     "Cheese", "Apple",
...     "Carrot", "Carrot",
...     "Iogurt", "Beer"
... ]
>>> hello = list("hello")
>>> print(hello)
['h', 'e', 'l', 'l', 'o']
>>> empty_fridge = list()
>>> print(empty_fridge)
[]
```

# Defining Lists

Lists can contain values of any type.

Items in the list can be of mixed types.

The items themselves can also be lists.

Lists of integers can be generated using `range(start, end, step)`.


```
>>> fridge = ["Apple", "Apple"]
>>> letters = list("hello")
>>> ages = [32, 45, 42, 12, 34, 57]
>>> dates = [datetime, datetime]
>>> data = ["John", 32, datetime]
>>> lists = [
...     ["John", "Mary", "Amy"],
...     [32, 43, 51]
... ]
>>> sequence = range(2, 10, 3)
>>> print(sequence)
[2, 5, 8]
```

# Python Lists: Accessing Values

Printing the list will show its values in the exact **same order** used when the list was defined.

Each value in the list can be accessed using its numeric **position** in the list (starting at zero). This is named the **index**.

Slicing can be used to access parts of the list or reverse the order of the list.



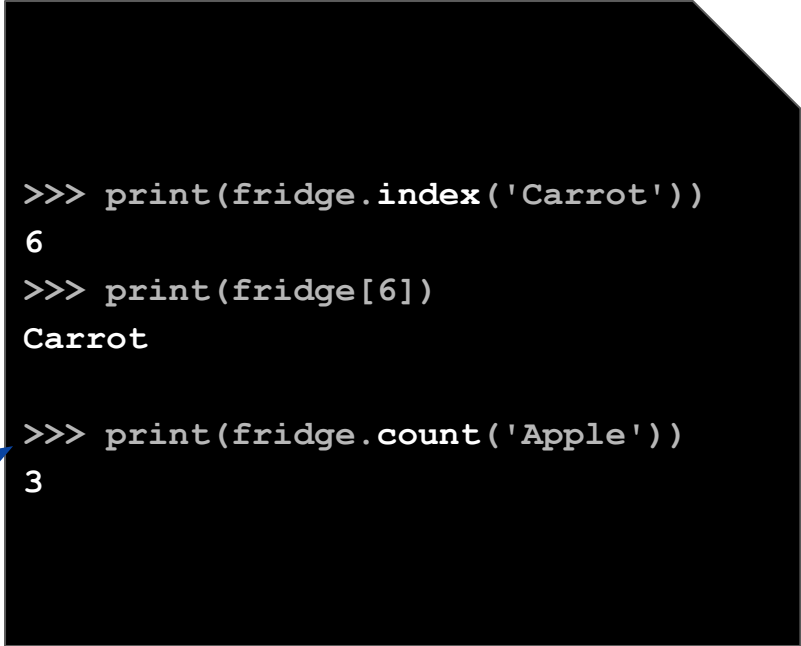
```
>>> print(fridge)
['Apple', 'Apple', 'Cabbage', ...]
>>> print(fridge[0])
Apple
>>> print(fridge[4])
Cheese
>>> print(fridge[0:2])
['Apple', 'Apple']
>>> print(fridge[-2:])
['Iogurt', 'Beer']
>>> print(fridge[::-1])
['Beer', 'Iogurt', 'Carrot', ...]
```

# Python Lists: Accessing Values

Using indexes to access or slice lists is useful when we know them, but this is often not the case and a mechanism to search the values must be available.

This can be done with the method **index**, that returns the index of the first occurrence in the list.

The number of occurrences of a value can be obtained with the **count** method.



```
>>> print(fridge.index('Carrot'))
6
>>> print(fridge[6])
Carrot

>>> print(fridge.count('Apple'))
3
```

# Python Lists: Order

The order of the list can be reversed with slicing if we don't want to make the change permanent.



```
>>> print(fridge[::-1])
['Beer', 'Iogurt', 'Carrot',...]
>>> print(fridge)
['Apple', 'Apple', 'Cabbage',...]
```

The method **reverse** will make the change permanent on the original variable name and will return **None**.



```
>>> print(fridge.reverse())
None
>>> print(fridge)
['Beer', 'Iogurt', 'Carrot',...]
```

# Python Lists: Sorting

The list can also be sorted ascending or descending. The **sort** method can be used to change the order of the items according to their values.



```
>>> print(fridge.sort())
None
>>> print(fridge)
['Apple', 'Apple', 'Apple', ...]
```

The argument **reverse** can be used to reverse the order of the sorting.



```
>>> fridge.sort(reverse=True)
>>> print(fridge)
['Steak', 'Iogurt', 'Cheese', ...]
>>> fridge.sort()
>>> fridge.reverse()
>>> print(fridge)
['Steak', 'Iogurt', 'Cheese', ...]
```

This is a shortcut for using first **sort()** and then **reverse()**.



# Python Lists: Changing Values

The values in the collection can be changed by using indexes.

A set of contiguous values can be changed in a single instruction as well.

Setting a value to **None** will not remove the value from the list.

Setting a value to an empty list will not remove the value either.



```
>>> fridge[0] = 'Beer'
>>> print(fridge)
['Beer', 'Apple', 'Cabbage',...]
>>> fridge[0:2] = ['Juice']
>>> print(fridge)
['Juice', 'Cabbage', 'Steak',...]
>>> fridge[0] = None
>>> print(fridge)
[None, 'Cabbage', 'Steak',...]
>>> fridge[0] = []
>>> print(fridge)
[[], 'Cabbage', 'Steak',...]
```

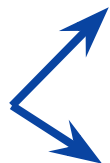


# Python Lists: Adding Values

Adding new values can be done with the method **append**. Values are added at the end.

Adding new values using **append** does not change the index of the other values.

Two or more lists can also be concatenated using the **+** operator. This is equivalent to using the **extend** method.



```
>>> fridge.append('Soda')
>>> print(fridge[-2:])
['Beer', 'Soda']

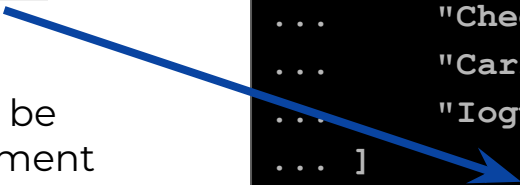
>>> eggs = ['Egg', 'Egg']
>>> fridge = fridge + eggs
>>> print(fridge[-4:])
['Beer', 'Soda', 'Egg', 'Egg']

>>> fridge = fridge.extend(eggs)
>>> print(fridge[-4:])
['Egg', 'Egg', 'Egg', 'Egg']
```

# Python Lists: Adding Values

New values can be added also in other positions of the list with the **insert** method.

The position where the value will be inserted is indicated as first argument and then the value to be inserted as second argument.



```
>>> fridge = [  
...     "Apple", "Apple",  
...     "Cabbage", "Steak",  
...     "Cheese", "Apple",  
...     "Carrot", "Carrot",  
...     "Iogurt", "Beer"  
... ]  
>>> fridge.insert(1, 'Pie')  
>>> print(fridge)  
['Apple', 'Pie', 'Apple', ...]
```

# Python Lists: Removing Values

Removing values can be done with the method **pop**. Values are removed from the end and returned, which can then be assigned to another variable name.

The **pop** method accepts an argument that will be the index of the value to be removed and returned.

The **remove** method finds the first occurrence of the given value and removes it from the list. It returns **None**.

```
>>> last_item = fridge.pop()
>>> print(fridge[-2:])
['Carrot', 'Iogurt']
>>> print(last_item)
Beer

>>> first_item = fridge.pop(0)
>>> print(fridge)
['Pie', 'Apple', 'Cabbage',...]

>>> print(fridge.remove('Apple'))
None
>>> print(fridge)
['Pie', 'Cabbage', 'Steak',...]
```

# Comparing Python Lists

Comparing two lists will return **True** if the following statements are true for the elements in it:

- They are the same
- They are in the same order

```
>>> list1 = [1, 2, 3]
>>> list2 = [1, 2, 3]
>>> list1 == list2
True
>>> list1 is list2
False
>>> list3 = [3, 2, 1]
>>> list1 == list3
False
>>> list1 == list3[::-1]
True
>>> list4 = [1, 1, 2, 2, 3, 3]
>>> list1 == list4
False
```

# Python Lists: Use Case Examples

## FRIDGE

- Apple
- Carrot
- logurt
- Apple
- logurt
- Beer
- Steak
- Cabbage
- Eggplant
- Orange Juice

## SENSOR DATA

*For instance, temperature.*

- 15
- 15
- 16
- 17
- 19
- 20
- 22
- 22
- 20
- 21

## GOLD MEDALS

- M. Mayer
- D. Hermann
- J. Ludwig
- Z.W. Ren
- Z.W. Ren
- U. Bogataj
- U. Bogataj
- A. Fontana
- J.T. Boe
- B. Karl

# Python List Methods: Summary

Lists have a variety of methods:

## **Add**

- Append
- Extend
- Insert

## **Remove**

- Pop
- Remove

## **Search & Analyze**

- Index
- Count

## **Sort**

- Sort
- Reverse

```
>>> print(dir(fridge))  
[..., 'append', 'count', 'extend',  
'index', 'insert', 'pop', 'remove',  
'reverse', 'sort']
```

# Tuples

## Collections in Python

**Tuples** are like lists but they cannot be changed.

They are the equivalent of constants for collections and are faster than lists.

They:

- Have an **order**.
- **Allow duplicate** values.
- **Do not allow** their objects to be **changed**.
- Allow objects of **different types**.



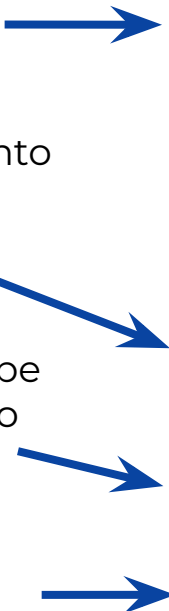
# Defining Tuples

They are defined using parentheses `()`.

Any **iterable** can be converted into a tuple by using the **tuple** constructor.

The **tuple** constructor can also be used to create empty tuples if no argument is given.

The output is the same as using empty parentheses `()`.



```
>>> days = (  
...     "Monday",  
...     "Tuesday",  
...     "Wednesday",  
...     "Thursday",  
...     "Friday",  
... )  
>>> hello = tuple("hello")  
>>> print(hello)  
('h', 'e', 'l', 'l', 'o')  
>>> empty_tuple = tuple()  
>>> print(empty_tuple)  
()
```

# Defining Tuples

Tuples can contain values of any type.



Items in the tuple can be of mixed types.



The items themselves can also be tuples.



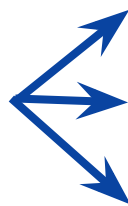
```
>>> fridge = ("Apple", "Apple")
>>> letters = tuple("hello")
>>> ages = (32, 45, 42, 12, 34, 57)
>>> dates = (datetime, datetime)
>>> data = ("John", 32, datetime)
>>> tuples = (
...     ("John", "Mary", "Amy"),
...     (32, 43, 51)
... ]
```

# Python Tuples: Accessing Values

Printing the tuple will show its values in the exact **same order** used when the tuple was defined.

Each value in the tuple can be accessed using its numeric **position** in the tuple (starting at zero). This is named the **index**.

Slicing can be used to access parts of the tuple or reverse its order.



```
>>> print(days)
('Monday', 'Tuesday', ...)
>>> print(days[0])
Monday
>>> print(days[4])
Friday
>>> print(days[0:2])
('Monday', 'Tuesday')
>>> print(days[-2:])
('Thursday', 'Friday')
>>> print(days[::-1])
('Friday', 'Thursday', ...)
```

# Python Tuples: Accessing Values

Tuples also have the method **index** implemented to return the index of the first occurrence of the given value in the tuple.

The number of occurrences of a value can also be obtained with the **count** method.



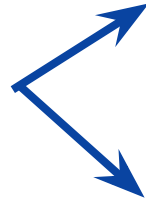
```
>>> print(days.index('Thursday'))  
3  
>>> print(days[3])  
Thursday
```



```
>>> print(days.count('Monday'))  
1
```

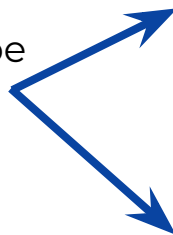
# Python Tuples: Changing Values

The items inside a tuple cannot be changed.



```
>>> days[0] = "Sunday"
Traceback (most recent call last):
  File "/home/DCI/test.py", line 71,
in <module>
    days[0] = "Sunday"
TypeError: 'tuple' object does not
support item assignment
```

If an item in the tuple is another type of iterable, its contents can still be changed.



```
>>> days = ("Monday", ["Tuesday"])
>>> days[0][0] = "Sunday"
>>> days[1].pop()
>>> days[1].append("Monday")
>>> print(days)
(['Sunday'], ['Monday'])
```

# Python Tuples: Sort, Add & Remove

As opposed to lists, tuples cannot be changed.

Therefore, **they do not have** any of the methods that can be used in lists to manipulate its contents:

- Append
- Extend
- Insert
- Pop
- Remove
- Sort
- Reverse

```
>>> days.append("Saturday")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
>>> days.pop()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'pop'
>>> days.sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'sort'
```

# Comparing Python Tuples

Comparing two tuples will return **True** if the following statements are true for the elements in it:

- They are the same
- They are in the same order

```
>>> list1 = (1, 2, 3)
>>> list2 = (1, 2, 3)
>>> list1 == list2
True
>>> list1 is list2
False
>>> list3 = (3, 2, 1)
>>> list1 == list3
False
>>> list1 == list3[::-1]
True
>>> list4 = [1, 1, 2, 2, 3, 3]
>>> list1 == list4
False
```

# Python Tuples: Use Case Examples

## WEEK DAYS

- Monday
- Tuesday
- Wednesday
- Thursday
- Friday
- Saturday
- Sunday

## MONTHS

- January
- February
- March
- April
- May
- June
- July
- August
- September
- October
- November
- December

## MEDAL TYPES

- Gold
- Silver
- Bronze



# Python Tuple Methods: Summary

Tuples only have methods to search and analyze the values:

- Index
- Count

```
>>> print(dir(days))  
[..., 'count', 'index']
```

# Sets

## Collections in Python

**Sets** distinctive feature is that they do not allow duplicate values.

They:

- Have an **no order**.
- **Do not store duplicate** values.
- **Do not allow** their objects to be **changed**.
- Allow objects of **different types**.

# Defining Sets

They are defined using curly brackets `{}`.



```
>>> fruits = {  
...     "Apple",  
...     "Orange",  
...     "Pear",  
...     "Banana",  
...     "Apricot"  
... }
```

Any **iterable** can be converted into a set by using the `set` constructor.



```
>>> hello = set("hello")  
>>> print(hello)  
{'o', 'e', 'h', 'l'}  
>>> empty_set = set()  
>>> print(empty_set)  
set()
```

The `set` constructor can also be used to create empty sets if no argument is given.



# Defining Sets

Sets can contain values of any type. Repeated values can be added, but they will only be stored once.

Items in the set can be of mixed types.

But the items themselves can not be sets.

A set can also be created using the `copy` method of another set.



```
>>> fruits = {"Apple", "Apple"}
>>> letters = set("hello")
>>> ages = {32, 45, 42, 12, 34, 57}
>>> dates = {datetime, datetime}
>>> data = {"John", 32, datetime}
>>> sets = {
...     {"John", "Mary", "Amy"},
...     {32, 43, 51}
... }
TypeError: unhashable type: 'set'
>>> copy = data.copy()
```

# Python Sets: Accessing Values

Printing the set will show its values in a **different order** than the one used when the set was defined.



The values in a set cannot be accessed directly using indexing.



Therefore, there is no **index** method. Because the set has no order, it has no method **sort** or **reverse**. And because the set has no repeated values, it does not have the **count** method either.

```
>>> print(fruits)
{'Apricot', 'Banana', 'Pear', ...}

>>> print(fruits[0])
Traceback (most recent call
last):
  File "<stdin>", line 1, in
<module>
TypeError: 'set' object does not
support indexing
```

# Python Sets: Adding Values

Adding new values can be done with the method **add**. The new value may show anywhere, as there is no order.



```
>>> fruits.add('Pineapple')
>>> print(fruits)
{'Apricot', 'Pineapple', ...}
```

The **update** method can be used to add multiple values at once. It accepts any iterable as an argument.



```
>>> fruits.update(
...     ['Mango', 'Mango']
... )
```

Adding a value twice throws no error, but only stores the value once.



```
>>> fruits
{'Apricot', 'Pineapple',
 'Banana', 'Mango', 'Pear',
 'Apple', 'Orange'}
```

# Python Sets: Removing Values

The method **pop** will remove a random element and will return it. It accepts no argument.



```
>>> random = fruits.pop()
>>> print(random)
```

Apricot

The **discard** method removes the given value and returns nothing.



```
>>> print(fruits.discard('Mango'))
```

None

```
>>> print(fruits)
```

```
{'Pineapple', 'Banana', 'Pear',
 'Apple', 'Orange'}
```

```
>>> fruits.remove('Mango')
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

KeyError: 'Mango'

```
>>> fruits.clear()
```

```
>>> print(fruits)
```

set()

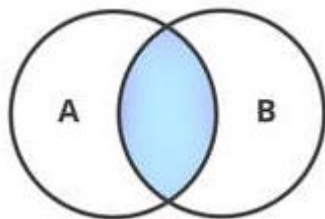
The **clear** method will remove all items in the set.





# Python Sets: Set Operations

The set type includes methods to perform the standard operations between sets of Set Theory and return new sets.

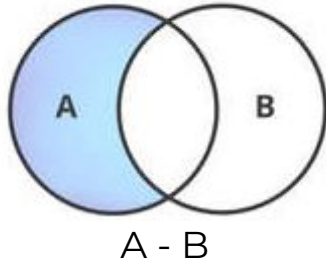


A and B

The method **intersection** returns a set containing the values present in both sets. The original sets remain unchanged.

```
>>> fruits = {  
...     "Pear",  
...     "Mango",  
...     "Apple",  
...     "Strawberry"  
... }  
>>> smoothie = {  
...     "Mango",  
...     "Orange",  
...     "Pear"  
... }  
>>> fruits.intersection(smoothie)  
{ 'Mango', 'Pear' }
```

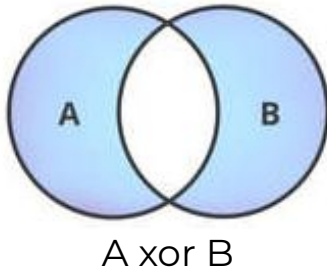
# Python Sets: Set Operations



The method **difference** returns a set containing the values present in A that don't exist in B. The original sets remain unchanged.

```
>>> fruits = {  
...     "Pear",  
...     "Mango",  
...     "Apple",  
...     "Strawberry"  
... }  
>>> smoothie = {  
...     "Mango",  
...     "Orange",  
...     "Pear"  
... }  
>>> fruits.difference(smoothie)  
{'Strawberry', 'Apple'}
```

# Python Sets: Set Operations



The method **`symmetric_difference`** returns a set containing the values present in A or B that don't exist at the same time in both. The original sets remain unchanged.

```
>>> fruits = {  
...     "Pear",  
...     "Mango",  
...     "Apple",  
...     "Strawberry"  
... }  
>>> smoothie = {  
...     "Mango",  
...     "Orange",  
...     "Pear"  
... }  
>>> fruits.symmetric_difference(smoothie)  
{'Orange', 'Strawberry', 'Apple'}
```

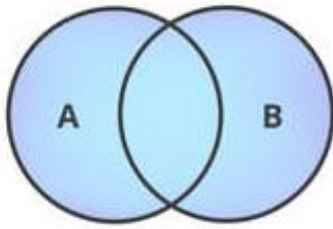
# Python Sets: Set Operations

The `difference_update`, `intersection_update` and `symmetric_difference_update` methods will calculate the same thing as do the `difference`, `intersection` and `symmetric_difference` methods.

The difference is that the first methods will overwrite the original set (fruits, in this case) instead of returning the result.

```
>>> fruits = {  
...     "Pear",  
...     "Mango",  
...     "Apple",  
...     "Strawberry"  
... }  
>>> smoothie = {  
...     "Mango",  
...     "Orange",  
...     "Pear"  
... }  
>>> fruits.difference_update(smoothie)  
>>> print(fruits)  
{'Strawberry', 'Apple'}
```

# Python Sets: Set Operations



A or B

The method **union** returns a set containing the values present in A or B (or both). The original sets remain unchanged.

```
>>> fruits = {  
...     "Pear",  
...     "Mango",  
...     "Apple",  
...     "Strawberry"  
... }  
>>> smoothie = {  
...     "Mango",  
...     "Orange",  
...     "Pear"  
... }  
>>> fruits.union(smoothie)  
{'Strawberry', 'Orange', 'Mango', 'Pear',  
'Apple'}
```

# Python Sets: Set Operations

The previous methods return a set with the result. Sometimes the script only requires to know if some kind of relationships exists between the sets, not the particular values.

The method `isdisjoint` returns `True` if the two sets have no item in common and `False` if they share at least one value.

```
>>> fruits = {  
...     "Pear",  
...     "Mango",  
...     "Apple",  
...     "Strawberry"  
... }  
>>> smoothie = {  
...     "Mango",  
...     "Orange",  
...     "Pear"  
... }  
>>> fruits.isdisjoint(smoothie)  
False
```

# Python Sets: Set Operations

The method **issubset** returns **True** if the first set is completely contained in the set passed as argument.

The method **issuperset** returns **True** if the first set completely contains the set passed as argument.

```
>>> fruits = {  
...     "Pear",  
...     "Mango",  
...     "Apple",  
...     "Strawberry"  
... }  
>>> smoothie = {  
...     "Mango",  
...     "Pear"  
... }  
>>> fruits.issubset(smoothie)  
False  
>>> smoothie.issubset(fruits)  
True
```

# Comparing Python Sets

Comparing two sets will return **True** if the following statement is true for the elements in it:

- They are the same

*A set is very useful when we need to know if the items in two iterables are the same.*

*Using **set()** on both iterables before comparing them will remove duplicates and ignore the order.*

```
>>> set1 = {1, 2, 3}
>>> set2 = {1, 2, 3}
>>> set1 == set2
True
>>> set1 is set2
False
>>> set1 == {3, 2, 1}
True
>>> set1 == {1, 1, 2, 2, 3, 3}
True
>>> list1 = [1, 2, 3]
>>> list2 = [3, 1, 2, 1, 3, 2]
>>> set(list1) == set(list2)
True
```



# Python Sets: Use Case Examples

## CITIES VISITED

- Berlin
- Barcelona
- Stockholm
- Trondheim
- Salzburg
- Brno
- Girona
- Manchester
- Ljubljana
- Tijuana

## REGISTERED

- Mary
- John
- Eva
- Susie
- Peter
- Lucy
- Ronnie
- Gerald
- Anna
- Anthony

## SPORTS

- Badminton
- Tennis
- Athletics
- Swimming
- Basketball
- Football
- Baseball
- Table-tennis
- Skiing
- Curling

# Python Set Methods: Summary

Sets have a long number of methods:

## Add

- Add
- Update

## Remove

- Pop
- Remove
- Discard
- Clear

## Manage

- Copy

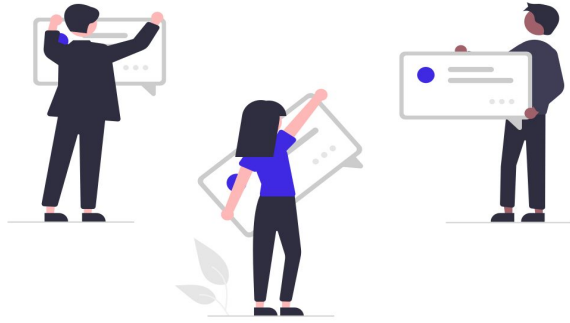
## Set Operations

- Intersection
- Difference
- ...

```
>>> print(dir(fruits))
[... , 'add', 'clear', 'copy',
'difference', 'difference_update',
'discard', 'intersection',
'intersection_update',
'isdisjoint', 'issubset',
'issuperset', 'pop', 'remove',
'symmetric_difference',
'symmetric_difference_update',
'union', 'update']
```

# We learned ...

- That one of the most common type of collections in Python is the list. Lists have an implicit order according to the position of the value in the list.
- That lists can be manipulated to add, remove and changes its elements.
- That tuples are very similar to lists, but they are immutable. Their elements cannot be changed.
- That comparing between lists works the same as comparing between tuples. They are only the same if they have the same values and in the same order.
- That sets, as opposed to lists and tuples, do not allow repeated values and their values have no order.
- That two sets are the same if they have the same values, no matter the order in which they were added to the set.



- How does comparison work in detail?
- What's the difference between a set and a list?
- Why are sets considered associative collections?
- What are the advantages of using tuples over lists?

# Expert Round

# Dictionaries

## Collections in Python

**Dictionaries** are associative arrays.  
They:

- Have an **order**.
- **Allow duplicate** values.
- **Allow** their objects to be **changed**.
- Allow objects of **different types**.

# Defining Dictionaries

They are defined using curly brackets `{}`. Each key-value pair is separated by a comma and every key is separated by a colon from the value.



Some **iterables** can be converted into a dictionary by using the `dict` constructor.



The `dict` constructor can also be used to create empty tuples if no argument is given.



```
>>> address = {
...     "name": "Harry Potter",
...     "street": "Private Drive",
...     "number": 4,
...     "city": "Little Whinging",
...     "county": "Surrey",
... }
>>> choice_dict = dict(choices)
>>> print(choice_dict)
{'Mon': 'Monday', 'Tue': 'Tuesday', ...}
>>> empty_dict = dict()
>>> print(empty_dict)
()
```

# Defining Dictionaries

Dictionaries can contain values of any type.

Dictionary indices (keys) can also be of any type.

Items in the dictionary can be of mixed types and values can be repeated.

The items themselves can also be dictionaries.

```
>>> values = {"a": 1, "b": "string"}
>>> keys = {1: "one", date: "day"}
>>> repeated = {1: "hi", 2: "hi"}
>>> users = {
...     "john": {
...         "name": "John Doe",
...         "age": 30
...     },
...     "jane": {
...         "name": "Jane Doe",
...         "age": 40
...     }
... }
```



# Defining Dictionaries

Dictionaries can also be initialized with the **fromkeys** class method.



This method takes a first argument as a sequence of keys and an optional second argument as the default value to initialize the values of each key.

Dictionaries can also be created by copying other dictionaries, using **copy**.



```
>>> template = dict.fromkeys(
...     ("street", "number", "zip",
...     "city", "country"),
...     "Unknown"
... )
>>> print(template)
{'street': 'Unknown', 'number': 'Unknown',
'zip': 'Unknown', 'city': 'Unknown',
'country': 'Unknown'}
>>> address = template.copy()
>>> address.update({"street": "Hogwarts"})
{'street': 'Hogwarts', 'number':
'Unknown', 'zip': 'Unknown', 'city':
'Unknown', 'country': 'Unknown'}
```


# Python Dictionaries: Accessing Values

Printing the dictionary will show its values in the **same order** used when the list was defined.

*!! This is only true in versions >3.6 of the Python interpreter. In Python 3.6 dictionaries do not have an order.*

Each value in the dictionary can be accessed indexing its **key** or using the **get** method.

The **get** method accepts a second argument that will be used as default value if the given key does not exist.



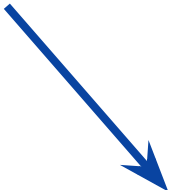

```
>>> print(users)
{'john': {'name': 'John Doe', ...
>>> print(users["john"])
{'name': 'John Doe', 'age': 30}
>>> print(users.get("jane"))
{'name': 'Jane Doe', 'age': 40}
>>> print(users.get(
...     "mario",
...     {"name": "Unknown"}
... ))
{"name": "Unknown"}
```

# Python Dictionaries: Accessing Values

The **setdefault** method works similarly to the **get** method. It returns the value if the key exists, and if not it returns the default value given.

The difference is that if the given key does not exist in the dictionary, this key is created with the given default value.

If no default value is passed, it is created with the value **None**.



```
>>> print(users.setdefault(
...     "mario",
...     {"name": "Mario Sanz",
...      "age": 25}
... ))
{"name": "Mario Sanz", "age": 25}
>>> print(users)
{'john': {'name': 'John Doe',
'age': 30},
'jane': {'name': 'Jane Doe',
'age': 40},
'mario': {'name': 'Mario Sanz',
'age': 25}}
```

# Python Dictionaries: Changing Values

The values in the dictionary can be changed by using the keys as indexes.

The `get` method can not be used to accomplish this.

```
>>> addr = {  
...     "name": "Harry Potter",  
...     "street": "Private Drive",  
...     "number": 4,  
...     "city": "Little Whinging",  
...     "county": "Surrey",  
... }  
>>> addr["street"] = "Hogwarts End"  
>>> print(addr["street"])  
Hogwarts End
```

# Python Dictionaries: Adding Values

New values can be added to the dictionary the same way they are changed.

If the key does not exist, it will be created.

The **update** method can be used to merge a dictionary into another.

Values will be overwritten or created.

```
>>> addr = {  
...     "name": "Harry Potter",  
...     "street": "Private Street",  
...     "number": 4,  
...     "city": "Little Whinging"  
... }  
>>> addr["country"] = "UK"  
>>> print(addr["country"])  
UK  
>>> fix = {"country": "UK", "continent": "Europe"  
...       "street": "Hogwarts End"}  
>>> addr.update(fix)  
>>> print(addr)  
{'name': 'Harry Potter', 'street': 'Hogwarts End',  
'number': 4, 'city': 'Little Whinging', 'country':  
'UK', 'continent': 'Europe'}
```

# Python Dictionaries: Removing Values

The **popitem** method removes and returns the last item in the dictionary. It does not accept any argument.

The **pop** method removes the item with the given key and returns its value. The argument is required.

Notice the **popitem** method returns a tuple containing both the key and the value. The **pop** method only returns the value.

```
>>> addr = {  
...     "name": "Harry Potter",  
...     "street": "Private Drive",  
...     "number": 4,  
...     "city": "Little Whinging",  
...     "county": "Surrey",  
... }  
>>> addr.popitem()  
( 'county', 'Surrey' )  
>>> addr.popitem()  
( 'city', 'Little Whinging' )  
>>> addr.pop("street")  
Private Drive
```

# Python Dictionaries: Removing Values

The **clear** method removes all the items in the dictionary.

```
>>> addr = {  
...     "name": "Harry Potter",  
...     "street": "Private Drive",  
...     "number": 4,  
...     "city": "Little Whinging",  
...     "county": "Surrey",  
... }  
>>> addr.clear()  
>>> print(addr)  
{}
```

# Python Dictionaries: Other Methods

Dictionaries have three additional methods that are used often.

The method **keys** returns an iterable with all the keys (and no values) of the dictionary.

The method **values** returns an iterable with all the values (and no keys) of the dictionary.

The method **items** returns an iterable with all the keys and values of the dictionary.

```
>>> addr = {  
...     "name": "Harry Potter",  
...     "street": "Hogwarts",  
...     "number": 4,  
...     "city": "Little Whinging"  
... }  
>>> addr.keys()  
dict_keys(['name', 'street', 'num...  
>>> addr.values()  
dict_values(['Harry Potter', 'Hog...  
>>> addr.items()  
dict_items([('name', 'Harry Potte...
```




# Comparing Python Dictionaries

Comparing two dictionaries will return **True** if the following statements are true for the elements in it:

- They have the same keys
- For each key the value is the same

*Notice the order of the keys in the dictionary is not considered.*



```
>>> dict1 = {"a": 1, "b": 2}
>>> dict1 = {"a": 1, "b": 2}
>>> dict1 == dict2
True
>>> dict1 is dict2
False

>>> dict1 == {"b": 2, "a": 1}
True

>>> dict1 == {"a": 2, "b": 1}
False
```

# Python Dictionaries: Use Case Examples

## USER PROFILE

- First name
- Family name
- Date of birth
- City of residence
- Country of residence
- Sex
- Job title
- Company
- Interests

## REGISTRATION

- Student
- Course
- Tutor
- Date of registration
- Passed (Yes/No)
- Date of finalization

## ADDRESS

- Type of street
- Street name
- Street number
- Door number
- Postal code
- District
- City
- Country

# Python Dictionary Methods: Summary

Dictionaries have the following methods:

## **Add & Create**

- Setdefault
- Update
- Copy
- Fromkeys

## **Remove**

- Pop
- Popitem
- Clear

## **Access**

- Get

## **Other**

- Items
- Keys
- Values

```
>>> print(dir(address))  
[..., 'clear', 'copy', 'fromkeys',  
'get', 'items', 'keys', 'pop',  
'popitem', 'setdefault', 'update',  
'values']
```

# We learned ...

- That Python's associative arrays are called dictionaries.
- That dictionaries, as opposed to lists and tuples, use keys instead of indices to refer to each of the values inside them.
- That template dictionaries can be created with the **fromkeys** method using a custom default value.
- That dictionaries have specific methods to return different types of iterables: **keys**, **values** and **items**.
- That two dictionaries are considered the same if they have the same keys and values, even if they are in different order.

# Using Iterables

## Collections in Python

# Using Iterables

Iterables, as the name suggests, are used for one main purpose: **iteration**.

There are also a variety of built-in functions that require iterables as arguments.

A list of strings can be iterated to access each one of its values and the `len` function can be used to obtain the number of items in the passed iterable (in this case, a string).

The same applies to every type of iterable.

```
>>> iterables_list = [
...     "string", "list", "set",
...     "tuple", "dictionary"
... ]
>>> for item in iterables_list:
...     print(item, len(item))
...
string 6
list 4
set 3
tuple 5
dictionary 10
```

# Iterating Dictionaries

Dictionaries are a special case, because each element is a composite of a key and a value.

The dictionary iterator yields only the key.

It has the same effect as using the **keys** method of the dictionary. This method yields the key of each item in the dictionary.

Dictionaries also have other methods to yield the values.

```
>>> profile = {
...     "name": "Mary Schmidt",
...     "age": 54
... }
>>> for key in iterables:
...     print(key)
...
name
age
>>> for key in iterables.keys():
...     print(key)
...
name
age
```

# Iterating Dictionaries

The method `values` will yield the values of each item.



```
>>> for value in iterables.values():
...     print(value)
...
Mary Schmidt
54
```

The method `items` will yield a tuple containing the key and value of each item.



```
>>> for item in iterables.items():
...     print(item)
...
('name', 'Mary Schmidt')
('age', 54)
```

The tuple can be unpacked in the same `for` instruction to make the code more readable.



```
>>> for key, value in iterables.items():
...     print(key, "=>", value)
...
name => Mary Schmidt
age => 54
```



# Iterating Tuples

Another common pattern is to use tuples instead of dictionaries to store **key-value** pairs that need to remain constant. This can be done defining a bi-dimensional tuple (a tuple of tuples).



The iteration will yield a tuple that can be unpacked like we do with the **items** method of a dictionary.




```
>>> days = (
...     ('Mon', 'Monday'),
...     ('Tue', 'Tuesday'),
...     ('Wed', 'Wednesday'),
...     ('Thu', 'Thursday'),
...     ('Fri', 'Friday'),
...     ('Sat', 'Saturday'),
...     ('Sun', 'Sunday')
... )
>>> for key, value in days:
...     print(key, "=>", value)
...
Mon => Monday
Tue => Tuesday
# continues
```

# Functions with Iterables: Enumerate

Python has some built-in functions that require or accept iterables and can be very useful in common situations. `list`, `tuple`, `dict` and `set` are some of them.

Another one of the most used functions is `enumerate`.

The `enumerate` function takes any iterable and for each value yields a tuple containing the position of that value and the value itself.



```
>>> list = [
...     'Monday',
...     'Tuesday',
...     'Wednesday',
...     'Thursday',
...     'Friday',
...     'Saturday',
...     'Sunday'
... ]
>>> for position, value in enumerate(list):
...     print(position, "=>", value)
...
0 => Monday
1 => Tuesday
# continues
```

# Functions with Iterables: Enumerate

The `enumerate` function allows for a simpler and more readable code to the alternative of adding a counter manually.



```
>>> list = [  
...     'Monday',  
...     'Tuesday',  
...     'Wednesday',  
...     'Thursday',  
...     'Friday',  
...     'Saturday',  
...     'Sunday'  
... ]  
>>> position = 0  
>>> for value in list:  
...     print(position, "=>", value)  
...     position += 1  
...  
0 => Monday  
1 => Tuesday  
# continues
```

# Functions with Iterables: Enumerate

The `enumerate` function can even be used on iterables that yield iterables, like the `items` method of a dictionary.



This iterable can be further unpacked using parentheses.



```
>>> dict = {
...     "name": "Mary Schmidt",
...     "age": 54
... }
>>> for position, value in enumerate(dict.items()):
...     print(position, "=>", value)
...
0 => ('name', 'Mary Schmidt')
1 => ('age', 54)
>>> for pos, (key, val) in enumerate(dict.items()):
...     print(pos, ".", key, "=>", val)
...
0 . name => Mary Schmidt
1 . age => 54
```

# Functions with Iterables: Zip

The **zip** function takes any number of iterables and returns a **zip object**.

This object is an iterator that yields a tuple with the value in each position of each passed iterable.

```
>>> nums = [1, 2, 3]
>>> eng = ("one", "two", "three")
>>> zip1 = zip(nums, eng)
>>> print(zip1)
<zip object at 0x7ff4f285cd80>
>>> for item in zip(nums, eng):
...     print(item)
...
(1, 'one')
(2, 'two')
(3, 'three')
```

# Functions with Iterables: Zip

Any kind of iterable (and any amount) can be passed to the function.

Notice that sets can be passed too, but since they have no order the items are fetched randomly.

Every time we execute this code the german words will appear in a different order.

```
>>> nums = "123"
>>> eng = ("one", "two", "three")
>>> deu = {"ein", "zwei", "drei"}
>>> cat = {"un": 1, "dos": 2, "tres": 3}
>>> fra = ["un", "deux", "trois"]

>>> for item in zip(nums, eng, deu, cat, fra):
...     print(item)
...
('1', 'one', 'zwei', 'un', 'un')
('2', 'two', 'drei', 'dos', 'deux')
('3', 'three', 'ein', 'tres', 'trois')
```

# Functions with Iterables: Zip

Items can also be unpacked to provide a more readable code inside the loop.



```
>>> nums = "123"
>>> eng = ("one", "two", "three")
>>> deu = {"ein", "zwei", "drei"}
>>> cat = {"un": 1, "dos": 2, "tres": 3}
>>> fra = ["un", "deux", "trois"]
>>> for num, en, de, ca, fr in zip(nums, eng, deu, cat, fra):
...     nums, eng, deu, cat, fra):
...     print(num + ". " + en, de, ca, fr)
...
1. one drei un un
2. two zwei dos deux
3. three ein tres trois
```

# Functions with Iterables: Sum, Max & Min

Some of the built-in mathematical functions accept iterables as arguments, so long as the iterable yields numeric values.

**Max** and **min** also accept string iterables, as the **>** operand also exists for this type. →

```
>>> nums = [1, 2, 3, 4, 5]
>>> print(sum(nums), max(nums), min(nums))
15 5 1
>>> nums = {1, 2, 3, 4, 5}
>>> print(sum(nums), max(nums), min(nums))
15 5 1
>>> nums = [1, 2, 3, 4, "5"]
>>> print(sum(nums), max(nums), min(nums))
Traceback (most recent call last):
  File "/home/DCI/test.py", line 30, in <module>
    print(sum(nums), max(nums), min(nums))
TypeError: unsupported operand type(s) for +:
'int' and 'str'
>>> nums = ["a", "b", "c", "d", "e"]
>>> print(max(nums), min(nums))
e a
```



# Functions with Iterables: Sorted

The **sorted** function returns a new list in alphabetical or numerical order.



It does not change the existing iterable.



Sorting a dictionary with **sorted** will return a list with the keys in alphabetical order, because the default iterable of a dictionary is by key.



Getting the **sorted** list of values can be done using the **values** method.



It accepts a keyword argument named **reverse** as a boolean that defaults to **False**.



```
>>> nums = [4, 3, 5, 2, 1]
>>> print(sorted(nums))
[1, 2, 3, 4, 5]

>>> print(nums)
[4, 3, 5, 2, 1]

>>> dict1 = {"c": 2, "b": 1, "a": 3}
>>> print(sorted(dict1))
['a', 'b', 'c']


>>> print(sorted(dict1.values()))
[1, 2, 3]

>>> print(sorted(dict1, reverse=True))
['c', 'b', 'a']
```

# Functions with Iterables: Sorted

The **sorted** function can also be used on lists of dictionaries to sort the list based on the values of one of the keys in these dictionaries.

This can be done passing a function as the **key** argument. This function should return the value to be used for sorting.



```
>>> dict1 = [  
...     {"name": "John", "age": 31},  
...     {"name": "Mary", "age": 46},  
...     {"name": "Lucy", "age": 25}  
... ]  
...  
>>> by_age = lambda user: user["age"]  
>>> by_name = lambda user: user["name"]  
  
>>> print(sorted(dict1, key=by_age))  
[{'name': 'Lucy', 'age': 25}, {'name': 'John',  
'age': 31}, {'name': 'Mary', 'age': 46}]  
  
>>> print(sorted(dict1, key=by_name))  
[{'name': 'John', 'age': 31}, {'name': 'Lucy',  
'age': 25}, {'name': 'Mary', 'age': 46}]
```

# Functions with Iterables: Any & All

Some functions use iterables to return a boolean value indicating if the iterable matches a certain condition.

The function **any** will return **True** only if **any** of the values in the iterable is *truthy*.

The function **all** will return **True** only if **all** the values in the iterable are *truthy*.

```
>>> a_list = [1, True, "Mary", {1, 2}]
>>> print(bool(a_list), any(a_list), all(a_list))
True True True
```

```
>>> a_list = [1, True, "Mary", {}]
>>> print(bool(a_list), any(a_list), all(a_list))
True True False
```

```
>>> a_list = [0, False, "", {}]
>>> print(bool(a_list), any(a_list), all(a_list))
True False False
```

# Functions with Iterables: Any & All

Some functions use iterables to return a boolean value indicating if the iterable matches a certain condition.

The function **any** will return **True** only if **any** of the values in the iterable is *truthy*.

The function **all** will return **True** only if **all** the values in the iterable are *truthy*.

```
>>> a_list = [1, True, "Mary", {1, 2}]
>>> print(bool(a_list), any(a_list), all(a_list))
True True True
```

```
>>> a_list = [1, True, "Mary", {}]
>>> print(bool(a_list), any(a_list), all(a_list))
True True False
```

```
>>> a_list = [0, False, "", {}]
>>> print(bool(a_list), any(a_list), all(a_list))
True False False
```

# Functions with Iterables: Map

The most common functions in functional programming require both an iterable and a function.

This is the case of the **map** function, that applies the given function to each element of the given iterable.

It returns a **map object** that is an iterable containing the output of that process.

```
>>> a_list = [1, 2, 3, 4, 5]
>>> by_two = lambda num: num * 2
>>> a_list_by_two = map(by_two, a_list)
>>> print(a_list_by_two)
<map object at 0x7f11957546d0>

>>> print(list(a_list_by_two))
[2, 4, 6, 8, 10]
```

# Functions with Iterables: Filter

The **filter** function returns an iterable with the elements of the given iterable that match a condition defined in a function.

It returns a **filter object** that is an iterable containing the output of that process.

```
>>> nums = [1, 2, 3, 4, 5]
>>> is_odd = lambda num: (num % 2) != 0
>>> odds = filter(is_odd, nums)

>>> print(odds)
<filter object at 0x7fced01036d0>

>>> print(list(odds))
[1, 3, 5]
```

# Iterable Functions: Summary

## CONSTRUCTORS

- list()
- tuple()
- set()
- dict()

## PACKING

- enumerate()
- zip()

## REDUCE

- sum()
- max()
- min()

## ORDER

- sorted()

## BOOLEAN

- any()
- all()

## FUNCTIONAL

- map()
- filter()

Many built-in modules also provide functions that use iterables, like `random.shuffle` or `functools.reduce`.

# We learned ...

- That Python has an abstract type named **iterable** that represents an object that can store different objects inside, which can be provided one at a time.
- That there are many built-in utility functions that use iterables to do various operations.
- That some functions, like **enumerate** and **zip** provide packing and unpacking functionalities to produce new iterables.
- Some of these functions are used to obtain boolean values (**any**, **all**) or a reduced number or string (**sum**, **max**, **min**).
- That some functions use both an iterable and a function, like **sorted**, **map** and **filter**.



# 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.

New in Python 3.10

```
>>> 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
```

# 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.

All fields must have values.

```
>>> 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'

>>> 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.

# 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