Sets, Tuples, and Dictionaries

by Sophia

WHAT'S COVERED

In this lesson, you will learn about other data collections including sets, tuples, and dictionaries. Specifically, this lesson covers:

1. Intro to Other Data Collection Types

As we have learned, there are other data collection types than a list. We are going to go into each in more detail, but let's start with an overview of what they are and why one would be used over another.

The other data collection types are:

- Sets
- Tuples
- Dictionaries

Some past topics to remember:

- Each of these data collection types are **iterable objects**, meaning they are capable of going through each of its members (elements) one at a time using the `iter()` and `next()` functions.
- So far we learned that lists are changeable or **mutable**, meaning we're able to change, reorder, add, or delete elements from a list after it has been created.
- We know that data collection types like lists are **indexed** with an integer, meaning the first element has an index of 0, the second element has an index of 1, and so on. This causes lists to be **ordered using** this

index.

You can see in this table that some of the other data collection types do not follow the same properties as lists.

Iterable Data Collection Types	Creation code	Mutable (changeable)?	Ordered by index
Lists	myList = ["apple","pear","banana"]	Yes	Yes
Sets	mySet = {"apple","pear","banana"}	No, elements within a set are immutable (cannot be changed) however we can add and remove elements from a set	No
Tuples	myTuple = ("apple","pear","banana")	No	Yes
Dictionaries	myDict = {'apple': 'red', 'pear': 'green', 'banana': 'yellow'}	Yes	Yes as keys/value

Note: Remember that you can use single quotes or double quotes around elements and variables; just be consistent and use one or the other—they cannot be combined.

By looking at this table, you may notice that there are some limitations to some of these data collection types, especially sets and tuples. So why would you use them?

Sets are rarely used in Python programming since their elements are not changeable (they are immutable) and are unordered. You will find that you will have a different sequence of elements each time this data collection type is utilized.

Tuples are also not changeable (they are immutable), so once a tuple is created, you're basically stuck with the values (elements). They cannot be changed or replaced. Tuples are rarely used for this very reason as well.

However, you will find that in the Python community, sets and tuples have very specific use cases.

- Sets do not allow duplicates and are the fastest for checking if an element is included. If you need duplicates, use lists or tuples.
- Tuples do not allow any change to elements, so if you believe you have a program that will never, ever need elements to change, a tuple may be the way to go.

Most of what we build upon in this course will be lists and dictionaries, but it is good to know what each of these data collection types can or cannot do.

---

2. Set

Sets are used to store multiple elements in a single variable similar to the other data collection types. Set elements are unordered, so we can’t determine the order in which the elements would appear. Because there is not a defined order, those set elements can appear in a different order every time we use them and cannot be

referenced by an index as we did with a list. A set is generally used specifically for cases where we need to do a lookup of a list that is unique. For example, if you have categories of products that should only be used once, a set would be a good choice, since it wouldn't change and wouldn't need to have an order. This is useful in the case where we have a specific number of defined elements to compare.

Lists are created using square brackets. Sets are created using curly brackets.

EXAMPLE

```
mySet = {"fun","sun","run"}
print(mySet)
```

Notice that with multiple runs of the same code, the output order is different, as the elements are not in any specific order:

```
{'fun', 'run', 'sun'}
```

```
{'sun', 'run', 'fun'}
```

The data in sets are also unchangeable or immutable. This means that we cannot change the elements in the set after the set has been created. However, we can add new elements to the set. In a set within Python, duplicate values are also not permitted. If we attempt to add them in, the set would not display the duplicate value.

EXAMPLE

```
mySet = {"fun","sun","run","fun"}
print(mySet)
```

```
{'fun', 'run', 'sun'}
```

As we can see in this example, we tried to add a second "fun" to the set. But when we ran the code, the second "fun" was just ignored since it was a duplicate element.

Similar to lists, the elements within a set can be of any data type (for example, string, integer, float, etc).

EXAMPLE
This set is valid.

```
mySet = {"fun",10,-2.0,"sun"}
```

2a. .add() Method

Once a set is created, if we wanted to add an element to a set, we would use the **.add() method**. If the element already exists, the `.add()` method will not add the element to the set. When we add to the set, where the element is included is random.

EXAMPLE

```
mySet = {"fun","sun","run"}
mySet.add("bun")
print(mySet)
```

With the following output:

```
{'bun', 'fun', 'run', 'sun'}
```

Note: Even though the screen output has "bun" at the beginning of the set, if we were to rerun this code, "bun" can show up anywhere in the set since there is no indexing structure.

2b. .update() Method

We can use the **.update() method** to add elements from another set (or iterable data collection type) into the current set. If an element exists in both sets, only one element will be updated to the combined set.

EXAMPLE

```
myPets = {"dog","cat","rabbit"}
yourPets = {"lion","tiger","bear"}
myPets.update(yourPets)
print(myPets)
```

Output using the `.update()` method.

```
{'cat', 'lion', 'dog', 'tiger', 'bear', 'rabbit'}
```

The `.update()` method which adds sets to another set can be used for any iterable object. This includes a list, tuple or a dictionary. For example, if we tried to add a list to a set, here is what would happen:

EXAMPLE

```
myPets = {"dog","cat","rabbit"} #this is the set
yourPets = ["dog","dog","fish"] #this is the list
myPets.update(yourPets)
print(myPets)
```

Here is that output.

```
{'fish', 'dog', 'rabbit', 'cat'}
```

Notice that we have "dog" twice in the list and "dog" already exists in the `myPets` set. As such, when we use the `.update()` method, those elements that are repeated are ignored. The `.update()` method is used to change an existing element in the set, versus trying to add a new element with the `.add()` method.

2c. .remove() and .discard() Methods

We can remove elements from a set (and other data collection types) by using the `.remove()` method we discussed in the last lesson, or with the `.discard()` method. The difference between the `.remove()` method and `.discard()` method is that the `.remove()` method will raise an error if the element does not exist in the set. However, the `.discard()` method will not.

EXAMPLE

```
myPets = {"dog","cat","rabbit"}
myPets.remove("dog")
print(myPets)
myPets.remove("dinosaur")
print(myPets)
```

Output with the use of the `.remove()` method:

```
{'cat', 'rabbit'}
Trackback (most recent call last):
    File "/home/main.py", line 4, in <module>
     myPets.remove("dinosaur")
KeyError: 'dinosaur'
```

Notice that “dog” was found in the set and removed. The updated set does not show “dog”. However, when we tried to remove “dinosaur” using the `.remove()` method, it was not found and gave us the KeyError.

Now let’s try the `.discard()` method. Remember the `.discard()` method does not raise an error if an element in the set is not found.

EXAMPLE

```
myPets = {"dog","cat","rabbit"}
myPets.discard("dog")
print(myPets)
myPets.discard("dinosaur")
print(myPets)
```

Output using the `.discard()` method.

```
{'rabbit', 'cat'}
{'rabbit', 'cat'}
```

As expected, the `.discard()` method ignored the fact that “dinosaur” was not in the set and output the set twice without an error.

Again, as we indicated earlier, sets are not heavily used in a real-world setting as they do not have an ordered structure and each element has to be unique. There are only very specific uses for them. Specifically, sets are very efficient to remove duplicate values from a collection as, by nature, each element in a set is unique. Lists are more widely used.

TERMS TO KNOW

Sets

Sets are a data collection type that is used to store multiple elements in a single variable similar to the other data collection types. Set elements are unordered and unchangeable.

.add()

Once a set is created, if we wanted to add an element to a set, we would use the `.add()` method. If the element already exists, the `.add()` method will not add the element to the set.

.update()

Use the `.update()` method to add elements from another iterable object (data collection type) into the current iterable object. If an element exists in both objects, only one element will be updated to the combined data collection type.

.discard()

The `.discard()` method removes a specified element from the data collection type object. The `.discard()` method will not raise an error if the element does not exist in the object.

---

3. Tuple

The next data collection type we'll look at is a tuple. A tuple is a data collection type that is unchangeable but ordered. The first element has the index value of 0, the second element has the index value of 1, and so on, similar to the list. The order cannot change.

The tuples are completely unchangeable or immutable, meaning we cannot change, add or remove elements after the tuple has been created. This is different from the other collection types. Since the tuples are indexed, they can have elements with the same value similar to a list. Tuples are created using round brackets.

EXAMPLE

```
myTuple = ("ice cream","frozen yogurt","sorbet")
print(myTuple)
```

The output for this tuple.

```
('ice cream', 'frozen yogurt', 'sorbet')
```

Remember that a tuple cannot be changed. If we tried to use the assignment operator to change an element, we'll get an error.

EXAMPLE

```
myTuple = ("ice cream","frozen yogurt","sorbet")
```

```
myTuple[1] = "gelato"
print(myTuple)
```
Output with an error.

```
 Trackback (most recent call last):
    File "/home/main.py", line 2, in <module>
     myTuple[1] = "gelato"
 TypeError: 'tuple' object does not support item assignment
```
Notice we get a TypeError stating that a tuple object does not support item assignment.

If we create a tuple with only one element, we have to add a comma after the element. Otherwise, Python won't recognize it as a tuple. In our first example below, we will set up a tuple with only one element. Then we will print it to the screen and use the `type()` function to return the variable type.

EXAMPLE

```
lonelyTuple = ("ice cream")
print(lonelyTuple)
print(type(lonelyTuple))
```
Output showing the tuple.

```
ice cream
<class 'str'>
```
The `type()` function returns that this is a string.

With the comma, we can see it is recognized as a tuple.

EXAMPLE

```
lonelyTuple = ("ice cream",)
print(lonelyTuple)
print(type(lonelyTuple))
```
Showing this as output:

```
('ice cream',)
<class 'tuple'>
```
Similar to the list and set, we can use any data type within a tuple (for example, string, integer, float, etc).

EXAMPLE
For example, this tuple is valid.

```
myTuple = ("fun",10,-2.0,"sun")
print(myTuple)
print(type(myTuple))
```
With the following output:

```
('fun', 10, -2.0, 'sun')
<class 'tuple'>
```
To access an element in a tuple, we can use the index number inside of the square brackets like this:

EXAMPLE

```
myTuple = ("ice cream","frozen yogurt","sorbet")
print(myTuple[0])
```
The output of the element.

```
ice cream
```
Ice cream, as expected, since that is in the index 0 position.

Note that we can also use negative values, with -1 being the last element in the tuple, -2 being the second to last element, and so forth.

EXAMPLE

```
myTuple = ("ice cream","frozen yogurt","sorbet")
print(myTuple[-1])
```
The output shows sorbet.

```
sorbet
```
Again, tuples are unchangeable, which means that we cannot add, change or remove elements. We can, however, use a workaround to convert the tuple to a list, change the list, and then convert the list back to a tuple. This isn't something that's used often, but it can be for situations where new items need to be added into the tuple. Take categories of products, for example, which may need additions or changes. The conversion would be useful.

EXAMPLE

```
myTuple = ("ice cream","frozen yogurt","sorbet")
myList = list(myTuple)
myList[1] = "gelato"
myTuple = tuple(myList)
print(myTuple)
```

With the following output:

```
('ice cream', 'gelato', 'sorbet')
```

That conversion workaround works but is not really ideal. Tuples are not supposed to change. Just like with sets, tuples are not widely used, and if they are, it is expected that the elements remain constant.

TERM TO KNOW

Tuple

A tuple is a data collection type that is ordered and unchangeable. This means that a tuple uses an indexing structure for its order and its elements can't be changed after it has been defined.

4. Dictionary

The last data collection type to discuss is a dictionary. A **dictionary** is like a list, but more generalized. A list uses index positions which have to be integers; for a dictionary, the index can be (almost) any type. Besides this, dictionaries store data values in key:value pairs.

What is a key:value pair? You can think of a dictionary as a mapping between the index positions (which are called the keys) and a set of values (elements). Each key maps to a value. The association of a key and a value (element) is called a **key:value pair**. An easy way to think about it is like an actual dictionary, where you have a word paired with its definition. The word would be the key and the definition would be the value.

So how do lists or tuples, which are ordered by index positions, and a dictionary, which is ordered by this key:value pairing, differ from one another? Remember, a set is not ordered.

Data Collection Type	Code	Notes
List	myPets = [‘dog’,]	Dog is in index [0] position, cat in index 1, and monkey in index 2. We would just need to know which index number an element is.
Tuple	myPets = ("dog","cat","monkey")	Same as list type where we can access each item using the index position.
Dictionary	myPets = {‘canine’: ‘dog’, ‘feline’: ‘cat’, ‘primate’ : ‘monkey’}	Instead of index position, dictionaries use keys so the key canine’s value is dog, the key feline’s value is cat, and the key primate’s value is monkey.

So, the dictionary has a key that we associate each element to, instead of an index position.

In addition, dictionaries are changeable, meaning that we can change, add, or remove elements after the dictionary has been created. Dictionaries cannot have two elements of the same key, so no duplication of keys.

As an example, we'll build a dictionary that maps from English to Spanish words, so the keys and the values are all strings. The **dict() function** creates a new dictionary with no elements. Because `dict()` is the name of a built-in function, you should avoid using it as a variable name.

```
eng2span = dict()
print(eng2span)
```

The output shows an empty dictionary.

```
{}
```

The curly brackets, {}, represent an empty dictionary.

To add elements to this new empty dictionary, you can use square brackets.

EXAMPLE

```
eng2span['one'] = 'uno'
```

This creates an element that maps from the key "one" to the value "uno". If we print the dictionary again, we see a key-value pair with a colon between the key and value.

TRY IT

Directions: Try using the `dict()` function, the first pairing, and output to screen code below.

```
eng2span = dict()
eng2span['one'] = 'uno'
print(eng2span)
```

The output shows one: uno.

```
{'one': 'uno'}
```

This output format is also an input format that you can use. You don't have to create an empty dictionary using the `dict()` function. You can, for example, create a new dictionary with three elements using the curly brackets and the pairings separated by a comma.

TRY IT

Directions: Try creating a dictionary with three elements using the code below.

```
eng2span = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
print(eng2span)
```

Output with three elements.

```
{'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

Instead of index positions, you use the keys to look up the corresponding values.

TRY IT

Directions: Try returning the element attached to the ‘two’ key using the code below.

```
eng2span = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
print(eng2span['two'])
```

The output shows dos.

```
dos
```

The key "two" always maps to the value "dos". If the key isn’t in the dictionary, we’ll get an exception.

TRY IT

Directions: Try the code below to look for a key that is not present in the dictionary.

```
eng2span = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
print(eng2span['four'])
```

The output shows KeyError.

```
Trackback (most recent call last):
   File "/home/main.py", line 2, in <module>
     print(eng2sp['four'])
KeyError: 'four'
```

We see the KeyError since the key "four" does not exist in our dictionary.

The `len()` function works on dictionaries to return the number of key-value pairs.

EXAMPLE

```
myPets = {'caine': 'dog', 'feline': 'cat', 'primate' : 'monkey'}
print (len(myPets))
```

The output shows 3.

```
3
```

In the code above, we can see that there are three key-value pairs.

The `in` operator works on dictionaries too; it tells you whether something appears as a key in the dictionary. The `in` operator only looks at the keys, not the values that are in a dictionary. To see whether something appears as a value in a dictionary, you can use the `.values()` method. The **.values() method** returns an object that contains the values of the dictionary as a list. Once this is in list form we could use the `in` operator. If we do not convert it to a list, it will simply have the type of dictionary values with a key and value for each element. The code below prints out the boolean value if the string "uno" appears in the `myVals` dictionary.

EXAMPLE

```
eng2span = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
myVals = list(eng2span.values())
print ('uno' in myVals)
```

The output shows True.

```
True
```

TERMS TO KNOW

Dictionary

A dictionary is like a list, but more generalized. Dictionaries store data values in key:value pairs that are changeable, meaning that we can change, add or remove elements after the dictionary has been created. Dictionaries cannot have two elements of the same key.

key:value pair

Used with the dictionary data collection type. Think of a dictionary as a mapping between the index positions (which are called the keys) and a set of values (elements). Each key maps to a value. The association of a key and a value (element) is called a key:value pair.

dict()

The `dict()` function creates a new dictionary with no elements.

.values()

The `.values()` method returns an object that contains the values of the dictionary as a list.

SUMMARY

In this lesson, we learned about the **other data collection types**: **sets**, **tuples** and **dictionaries**, and compared them to each other and to lists. We discussed that sets are unordered, immutable (unchangeable), not indexed, and cannot have duplicates. We also were introduced to tuples, which are index-ordered like lists, but are not mutable once created and cannot include duplicates. Finally, we

learned that dictionaries are mutable and use keys and values instead of index positions to reference elements.

Best of luck in your learning!

TERMS TO KNOW

.add()
Once a set is created, if we wanted to add an element to a set, we would use the .add() method. If the element already exists, the .add() method will not add the element to the set.

.discard()
The .discard() method removes a specified element from the data collection type object. The .discard() method will not raise an error if the element does not exist in the object.

.update()
Use the .update() method to add elements from another iterable object (data collection type) into the current iterable object. If an element exists in both objects, only one element will be updated to the combined data collection type.

.values()
The .values() method returns an object that contains the values of the dictionary as a list.

Dictionary
A dictionary is like a list, but more generalized. Dictionaries store data values in key:value pairs that are changeable, meaning that we can change, add, or remove elements after the dictionary has been created. Dictionaries cannot have two elements of the same key.

Sets
Sets are a data collection type that is used to store multiple elements in a single variable similar to the other data collection types. Set elements are unordered and unchangeable.

Tuple
A tuple is a data collection type that is ordered and unchangeable. This means that a tuple uses an indexing structure for its order and its elements can’t be changed after it has been defined.

dict()
The dict() function creates a new dictionary with no elements.

key:value pair

A key:value pair is used with the dictionary data collection type. Think of a dictionary as a mapping between the index positions (which are called the keys) and a set of values (elements). Each key maps to a value. The association of a key and a value (element) is called a key:value pair.