Dictionaries

We've been learning about *sequences* in Python but now we're going to switch gears and learn about *mappings* in Python. If you're familiar with other languages you can think of these Dictionaries as hash tables.

This section will serve as a brief introduction to dictionaries and consist of:

1.) Constructing a Dictionary

2.) Accessing objects from a dictionary

3.) Nesting Dictionaries

4.) Basic Dictionary Methods

So what are mappings? Mappings are a collection of objects that are stored by a *key*, unlike a sequence that stored objects by their relative position. This is an important distinction, since mappings won't retain order since they have objects defined by a key.

A Python dictionary consists of a key and then an associated value. That value can be almost any Python object.

Constructing a Dictionary

Let's see how we can construct dictionaries to get a better understanding of how they work!

```
In [1]: # Make a dictionary with {} and : to signify a key and a value
        my_dict = {'key1':'value1','key2':'value2'}

In [2]: # Call values by their key
        my_dict['key2']

Out[2]: 'value2'
```

Its important to note that dictionaries are very flexible in the data types they can hold. For example:

```
In [3]: my_dict = {'key1':123,'key2':[12,23,33],'key3':['item0','item1','item2']}

In [4]: # Let's call items from the dictionary
        my_dict['key3']

Out[4]: ['item0', 'item1', 'item2']

In [5]: # Can call an index on that value
        my_dict['key3'][0]

Out[5]: 'item0'

In [6]: # Can then even call methods on that value
        my_dict['key3'][0].upper()

Out[6]: 'ITEM0'
```

We can affect the values of a key as well. For instance:

```
In [7]: my_dict['key1']

Out[7]: 123

In [8]: # Subtract 123 from the value
        my_dict['key1'] = my_dict['key1'] - 123

In [9]: #Check
        my_dict['key1']

Out[9]: 0
```

Python has a built-in method of doing a self subtraction or addition (or multiplication or division). We could have also used += or -= for the above statement. For example:

```
In [10]: # Set the object equal to itself minus 123
         my_dict['key1'] -= 123
         my_dict['key1']
Out[10]: -123
```

We can also create keys by assignment. For instance if we started off with an empty dictionary, we could continually add to it:

```
In [11]: # Create a new dictionary
         d = {}

In [12]: # Create a new key through assignment
         d['animal'] = 'Dog'

In [13]: # Can do this with any object
         d['answer'] = 42

In [14]: #Show
         d
Out[14]: {'animal': 'Dog', 'answer': 42}
```

Nesting with Dictionaries

Hopefully you're starting to see how powerful Python is with its flexibility of nesting objects and calling methods on them. Let's see a dictionary nested inside a dictionary:

```
In [15]: # Dictionary nested inside a dictionary nested inside a dictionary
         d = {'key1':{'nestkey':{'subnestkey':'value'}}}
```

Wow! That's a quite the inception of dictionaries! Let's see how we can grab that value:

```
In [16]: # Keep calling the keys
         d['key1']['nestkey']['subnestkey']
Out[16]: 'value'
```

A few Dictionary Methods

There are a few methods we can call on a dictionary.

```
In [17]: # Create a typical dictionary
         d = {'key1':1,'key2':2,'key3':3}

In [18]: # Method to return a list of all keys
         d.keys()
Out[18]: dict_keys(['key1', 'key2', 'key3'])

In [19]: # Method to grab all values
         d.values()
Out[19]: dict_values([1, 2, 3])

In [20]: # Method to return tuples of all items  (we'll learn about tuples soon)
         d.items()
Out[20]: dict_items([('key1', 1), ('key2', 2), ('key3', 3)])
```

Tuples

In Python tuples are very similar to lists, however, unlike lists they are *immutable* meaning they can not be changed. You would use tuples to present things that shouldn't be changed, such as days of the week, or dates on a calendar.

In this section, we will get a brief overview of the following:

1.) Constructing Tuples

2.) Basic Tuple Methods

3.) Immutability

4.) When to Use Tuples

Constructing Tuples

The construction of a tuples use () with elements separated by commas. For example:

```
In [1]: # Create a tuple
        t = (1,2,3)

In [2]: # Check len just like a list
        len(t)

Out[2]: 3

In [3]: # Can also mix object types
        t = ('one',2)

        # Show
        t

Out[3]: ('one', 2)

In [4]: # Use indexing just like we did in lists
        t[0]

Out[4]: 'one'

In [5]: # Slicing just like a list
        t[-1]

Out[5]: 2
```

Basic Tuple Methods

Tuples have built-in methods, but not as many as lists do.

```
In [6]: # Use .index to enter a value and return the index
        t.index('one')

Out[6]: 0

In [7]: # Use .count to count the number of times a value appears
        t.count('one')

Out[7]: 1
```

Immutability

It can't be stressed enough that tuples are immutable. To drive that point home:

```
In [8]: t[0]= 'change'

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-8-1257c0aa9edd> in <module>()
----> 1 t[0]= 'change'

TypeError: 'tuple' object does not support item assignment
```

Because of this immutability, tuples can't grow. Once a tuple is made we can not add to it.

```
In [9]: t.append('nope')

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-9-b75f5b09ac19> in <module>()
----> 1 t.append('nope')

AttributeError: 'tuple' object has no attribute 'append'
```

When to use Tuples

You may be wondering, "Why bother using tuples when they have fewer available methods?" To be honest, tuples are not used as often as lists in programming, but are used when immutability is necessary. If in your program you are passing around an object and need to make sure it does not get changed, then a tuple becomes your solution. It provides a convenient source of data integrity.

Set and Booleans

There are two other object types in Python that we should quickly cover: Sets and Booleans.

Sets

Sets are an unordered collection of *unique* elements. We can construct them by using the set() function. Let's go ahead and make a set to see how it works.

```
In [1]: x = set()

In [2]: # We add to sets with the add() method
        x.add(1)

In [3]: #Show
        x

Out[3]: {1}
```

Note the curly brackets. This does not indicate a dictionary! Although you can draw analogies as a set being a dictionary with only keys.

We know that a set has only unique entries. So what happens when we try to add something that is already in a set?

```
In [4]: # Add a different element
        x.add(2)

In [5]: #Show
        x

Out[5]: {1, 2}

In [6]: # Try to add the same element
        x.add(1)

In [7]: #Show
        x

Out[7]: {1, 2}
```

Notice how it won't place another 1 there. That's because a set is only concerned with unique elements! We can cast a list with multiple repeat elements to a set to get the unique elements. For example:

```
In [8]: # Create a list with repeats
        list1 = [1,1,2,2,3,4,5,6,1,1]

In [9]: # Cast as set to get unique values
        set(list1)

Out[9]: {1, 2, 3, 4, 5, 6}
```

Booleans

Python comes with Booleans (with predefined True and False displays that are basically just the integers 1 and 0). It also has a placeholder object called None. Let's walk through a few quick examples of Booleans (we will dive deeper into them later in this course).

In [10]:
```
# Set object to be a boolean
a = True
```

In [11]:
```
#Show
a
```

Out[11]:
```
True
```

We can also use comparison operators to create booleans. We will go over all the comparison operators later on in the course.

In [12]:
```
# Output is boolean
1 > 2
```

Out[12]:
```
False
```

We can use None as a placeholder for an object that we don't want to reassign yet:

In [13]:
```
# None placeholder
b = None
```

In [14]:
```
# Show
print(b)
```

```
None
```