Types and Sequences

The absence of static typing in Python doesn't mean that there aren't types. The Python language has a built in function called type which will show you what type a given reference is. Some of the common types includes strings, the none type as we've discussed. Integers and floating point variables. As we've seen you can have reference as to function as well as a function type also exist.

Use `type` to return the object's type.

```
In [2]: type('This is a string')
Out[2]: str

In [3]: type(None)
Out[3]: NoneType

In [4]: type(1)
Out[4]: int

In [5]: type(1.0)
Out[5]: float

In [6]: type(add_numbers)
Out[6]: function
```

Typed objects have properties associated with them, and these properties can be data or functions. A lot of Python's built around different kinds of sequences or collection types. And there's three native kinds of collections that we're going to talk about, tuples, lists, and dictionaries.

Tuples

A tuple is a sequence of variables which itself is immutable. That means that a tuple has items in an ordering, but that it cannot be changed once created. We write tuples using parentheses, and we can mix types for the contents of the tuple. Here's a tuple which has four items. Two are numbers, and two are strings.

Tuples are an immutable data structure (cannot be altered).

```
In [*]: x = (1, 'a', 2, 'b')

        type(x)
```

Note here that I've used single quotes for a string, whereas previously I've used double quotes. In Python, either single or double quotes can be used to denote string values.

Lists:

Lists are very similar, but they can be mutable, so you can change their length, number of elements, and the element values.

Lists are a mutable data structure.

```
In [ ]: x = [1, 'a', 2, 'b']
        type(x)
```

A list is declared using the square brackets. There are a couple of different ways to change the contents of a list. One is through the append function which allows you to append new items to the end of the list.

Use `append` to append an object to a list.

```
In [ ]: x.append(3.3)
        print(x)
```

Both lists and tuples are iterable types, so you can write loops to go through every value they hold. The norm, if you want to look each item in the list is to use a for statement. This is similar to the for each loop in languages like Java and C# but note that there's no typing required.

This is an example of how to loop through each item in the list.

In [ ]:
```
for item in x:
    print(item)
```

Lists and tuples can also be accessed as arrays might in other languages, by using the square bracket operator, which is called the indexing operator. The first item of the list starts at position zero and to get the length of the list, we use the built in len function. There are some other common functions that you might expect like min and max which will find the minimum or maximum values in a given list or tuple.

using the indexing operator:

In [ ]:
```
i=0
while( i != len(x) ):
    print(x[i])

    i = i + 1
```

Python lists and tuples also have some basic mathematical operations that can be allowed on them. The plus sign concatenates lists for instance. And the asterisks repeats the values of a list. A very common operator is the in operator. This looks at set membership and returns a boolean value of true or false depending on whether one item is in a given list. We're going to dive more into operators and special kinds of sequences in a future week when we look at a technique called broadcasting.

Use + to concatenate lists.

In [ ]:
```
[1,2] + [3,4]
```

Use * to repeat lists.

In [ ]:
```
[1]*3
```

Use the `in` operator to check if something is inside a list.

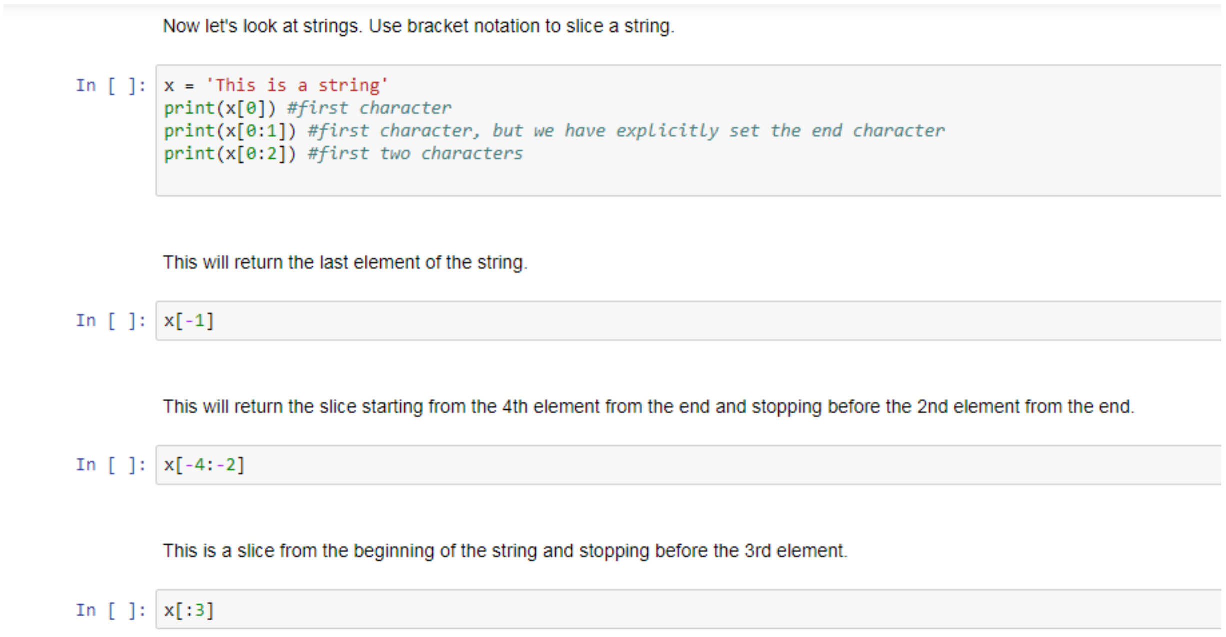
```
In [ ]: 1 in [1, 2, 3]
```

Perhaps the most interesting operations you can do with lists are called slicing. Where the square bracket array syntax for accessing an element might look fairly similar to that which you've seen in other languages. In Python, the indexing operator allows you to submit multiple values. The first parameter is the starting location, if this is the only element then one item is return from the list. The second parameter is the end of the slice. It's an exclusive end so if you slice with the first parameter being zero the next parameter being one, then you only get back one item.

This is much easier to explain with an example. One handy aspect of Python is that all strings are actually just lists of characters so slicing works wonderfully on them. Here's an example. When we run x[0] or x[0:1] we get just the first character of the string. But when we run x[0:2], we get the first two characters of the string.

Our indexing values can also be negative which is really cool. And this means to index from the back of the string. So x[-1] gets us the last letter of the string, and x[-4:-2] reads in all of the characters from the 4th last to the 2nd last positions.

Finally if we want to reference the start or the end of the string implicitly, we can by just leaving the parameter empty. So x[:3] starts with the first character and goes until position three. And the x[3:] starts with the fourth character because indexing always begins with zero and goes to the end of the list.



Now let's look at strings. Use bracket notation to slice a string.

```
In [ ]: x = 'This is a string'
        print(x[0]) #first character
        print(x[0:1]) #first character, but we have explicitly set the end character
        print(x[0:2]) #first two characters
```

This will return the last element of the string.

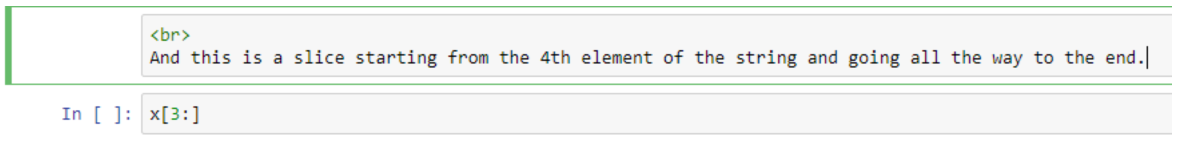
```
In [ ]: x[-1]
```

This will return the slice starting from the 4th element from the end and stopping before the 2nd element from the end.

```
In [ ]: x[-4:-2]
```

This is a slice from the beginning of the string and stopping before the 3rd element.

```
In [ ]: x[:3]
```



```
<br>
And this is a slice starting from the 4th element of the string and going all the way to the end.
```

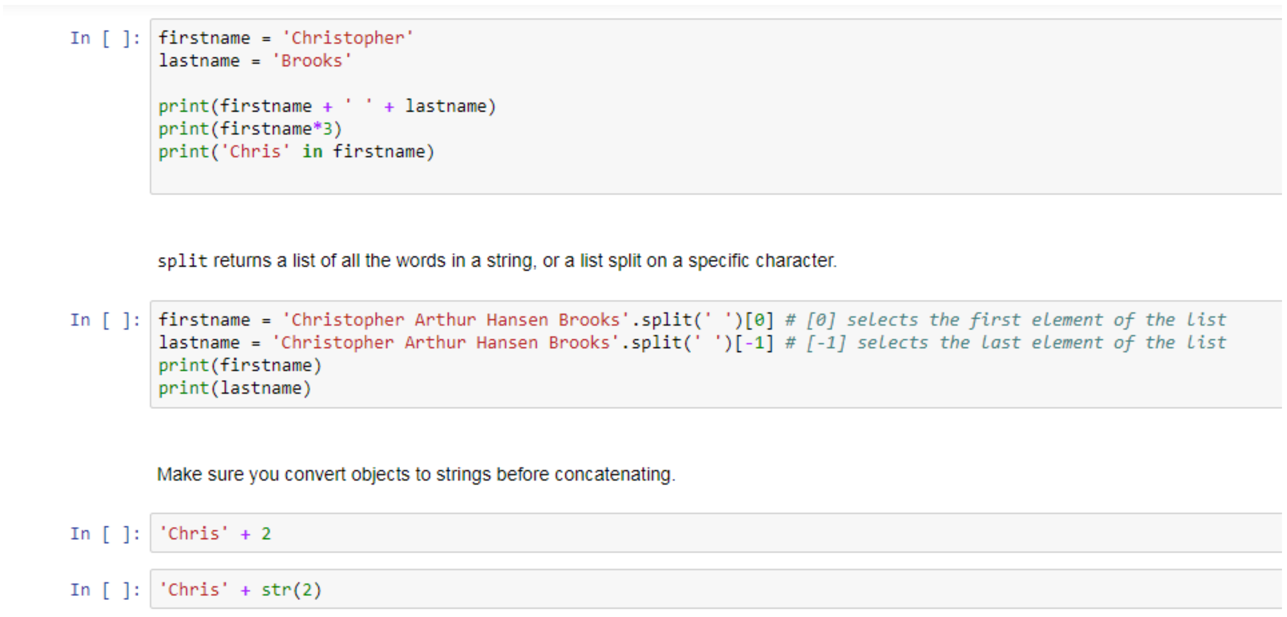
```
In [ ]: x[3:]
```

Now, I'm taking a bit of an aside here to talk about manipulating strings. Slicing isn't the only way to manipulate strings. And a common activity is to split strings based on substrings. That is, to go through the string looking for patterns, and segmenting it as appropriate. This is called regular expression evaluation, and we're going to cover this in detail in the section of the specialization which deals with text mining since it's a very common operation. But Python has some basic tools for text analysis. And I'm going to show you them here.

As we saw, strings are just lists of characters. So operations you can do on a list, you can do on a string. This means that you can concatenate two strings together using the plus operator. And multiplying strings will repeat a given string. You can also search for strings using the in operator.

The string type has an associated function called split.

This function breaks the string up into substrings based on a simple pattern. Here for instance, I'll just split my full name based on the presence of a space character. The result is a list of four elements. We can choose the first element with the indexing operator to be the first name, and the last element to be my last name.



```
In [ ]: firstname = 'Christopher'
        lastname = 'Brooks'

        print(firstname + ' ' + lastname)
        print(firstname*3)
        print('Chris' in firstname)
```

split returns a list of all the words in a string, or a list split on a specific character.

```
In [ ]: firstname = 'Christopher Arthur Hansen Brooks'.split(' ')[0] # [0] selects the first element of the list
        lastname = 'Christopher Arthur Hansen Brooks'.split(' ')[-1] # [-1] selects the last element of the list
        print(firstname)
        print(lastname)
```

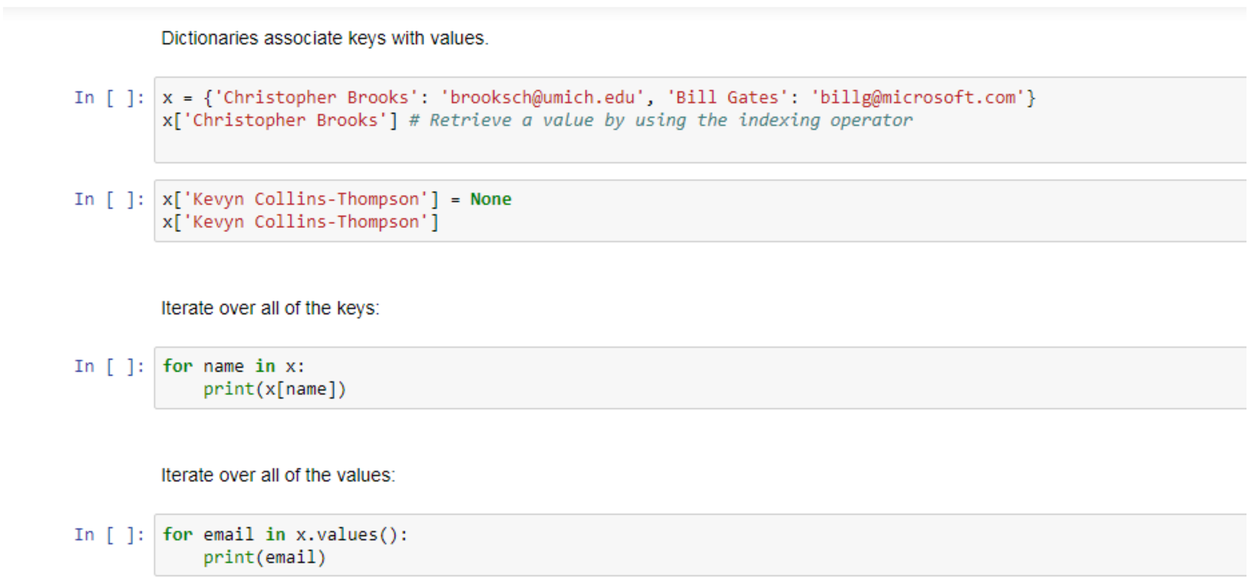
Make sure you convert objects to strings before concatenating.

```
In [ ]: 'Chris' + 2
```

```
In [ ]: 'Chris' + str(2)
```

We'll touch on strings just a bit more but before we do, I want to talk about dictionaries. Dictionaries are similar to lists and tuples in that they hold a collection of items, but they're labeled collections which do not have an ordering. This means that for each value you insert into the dictionary, you must also give a key to get that value out. In other languages the structure is often called a map. And in Python we use curly braces to denote a dictionary. Here is an example where we might link names to email addresses. You can see that we indicate each item of the dictionary when creating it using a pair of values separated by colons. Then you can retrieve a value for a given label using the indexing operator.

The types you use for indices or values in the dictionary can be anything. And this could be a mixture of types if you prefer.



Dictionaries associate keys with values.

```
In [ ]: x = {'Christopher Brooks': 'brooksch@umich.edu', 'Bill Gates': 'billg@microsoft.com'}
        x['Christopher Brooks'] # Retrieve a value by using the indexing operator
```

```
In [ ]: x['Kevyn Collins-Thompson'] = None
        x['Kevyn Collins-Thompson']
```

Iterate over all of the keys:

```
In [ ]: for name in x:
            print(x[name])
```

Iterate over all of the values:

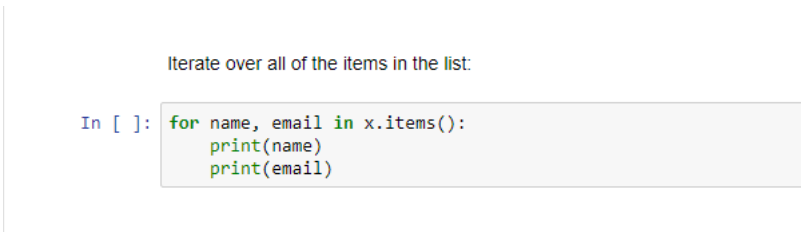
```
In [ ]: for email in x.values():
            print(email)
```

We can add new items to the dictionary using the same indexing operator we are used to. Just on the left hand side of a statement.

You an iterate over all of the items in a dictionary in a number of ways. First you can iterate over all of the keys and just pull the contents out as you see fit.

Or you can iterate over the values and just ignore the keys.

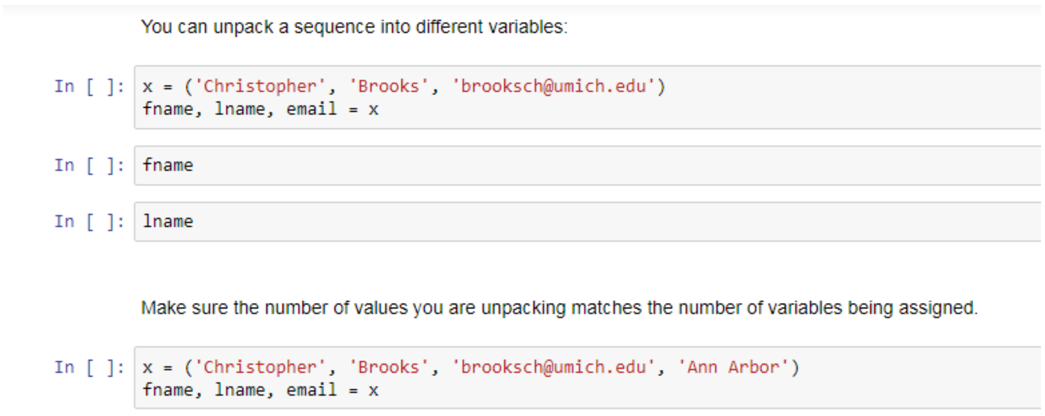
Finally you can iterate over both the values and the keys at once using the item's function.



Iterate over all of the items in the list:

```
In [ ]: for name, email in x.items():
            print(name)
            print(email)
```

This last example is a little bit different, and it's an example of something called unpacking. In Python you can have a sequence. That's a list or a tuple of values, and you can unpack those items into different variables through assignment in one statement. Here's another example of that, where we have a tuple that has my first name, last name, and email address.


You can unpack a sequence into different variables:

```
In [ ]: x = ('Christopher', 'Brooks', 'brooksch@umich.edu')
        fname, lname, email = x
```

```
In [ ]: fname
```

```
In [ ]: lname
```

Make sure the number of values you are unpacking matches the number of variables being assigned.

```
In [ ]: x = ('Christopher', 'Brooks', 'brooksch@umich.edu', 'Ann Arbor')
        fname, lname, email = x
```

I declare three variables and assign them to the tuple.

Underneath, Python has unpacked the tuple, and assigned each of these variables in order. We can see that if we add a fourth item to the tuple, Python isn't sure how to unpack that, so we have an error. That's an overview of built in types with Python. In the next lecture, we're going to revisit strings briefly, then start working with some data files.