**Lists**

Earlier when discussing strings we introduced the concept of a *sequence* in Python. Lists can be thought of the most general version of a *sequence* in Python. Unlike strings, they are mutable, meaning the elements inside a list can be changed!

In this section we will learn about:

1.) Creating lists

2.) Indexing and Slicing Lists

3.) Basic List Methods

4.) Nesting Lists

5.) Introduction to List Comprehensions

Lists are constructed with brackets [] and commas separating every element in the list.

Let's go ahead and see how we can construct lists!

In [1]:

*# Assign a list to an variable named my\_list*

my\_list = [1,2,3]

We just created a list of integers, but lists can actually hold different object types. For example:

In [2]:

my\_list = ['A string',23,100.232,'o']

Just like strings, the len() function will tell you how many items are in the sequence of the list.

In [3]:

len(my\_list)

**Indexing and Slicing**

Indexing and slicing work just like in strings. Let's make a new list to remind ourselves of how this works:

In [4]:

my\_list = ['one','two','three',4,5]

In [5]:

*# Grab element at index 0*

my\_list[0]

In [6]:

*# Grab index 1 and everything past it*

my\_list[1:]

In [7]:

*# Grab everything UP TO index 3*

my\_list[:3]

We can also use + to concatenate lists, just like we did for strings.

In [8]:

my\_list + ['new item']

Note: This doesn't actually change the original list!

In [9]:

my\_list

You would have to reassign the list to make the change permanent.

In [10]:

*# Reassign*

my\_list = my\_list + ['add new item permanently']

In [11]:

my\_list

We can also use the \* for a duplication method similar to strings:

In [12]:

*# Make the list double*

my\_list \* 2

In [13]:

*# Again doubling not permanent*

my\_list

**Basic List Methods**

If you are familiar with another programming language, you might start to draw parallels between arrays in another language and lists in Python. Lists in Python however, tend to be more flexible than arrays in other languages for a two good reasons: they have no fixed size (meaning we don't have to specify how big a list will be), and they have no fixed type constraint (like we've seen above).

Let's go ahead and explore some more special methods for lists:

In [14]:

*# Create a new list*

list1 = [1,2,3]

Use the **append** method to permanently add an item to the end of a list:

In [15]:

*# Append*

list1.append('append me!')

In [16]:

*# Show*

list1

Use **pop** to "pop off" an item from the list. By default pop takes off the last index, but you can also specify which index to pop off. Let's see an example:

In [17]:

*# Pop off the 0 indexed item*

list1.pop(0)

In [18]:

*# Show*

list1

In [19]:

*# Assign the popped element, remember default popped index is -1*

popped\_item = list1.pop()

In [20]:

popped\_item

In [21]:

*# Show remaining list*

list1

It should also be noted that lists indexing will return an error if there is no element at that index. For example:

In [22]:

list1[100]

---------------------------------------------------------------------------

IndexError Traceback (most recent call last)

<ipython-input-22-af6d2015fa1f> in <module>()

----> 1 list1[100]

IndexError: list index out of range

We can use the **sort** method and the **reverse** methods to also effect your lists:

In [23]:

new\_list = ['a','e','x','b','c']

In [24]:

*#Show*

new\_list

In [25]:

*# Use reverse to reverse order (this is permanent!)*

new\_list.reverse()

In [26]:

new\_list

In [27]:

*# Use sort to sort the list (in this case alphabetical order, but for numbers it will go ascending)*

new\_list.sort()

In [28]:

new\_list

**Nesting Lists**

A great feature of of Python data structures is that they support *nesting*. This means we can have data structures within data structures. For example: A list inside a list.

Let's see how this works!

In [29]:

*# Let's make three lists*

lst\_1=[1,2,3]

lst\_2=[4,5,6]

lst\_3=[7,8,9]

*# Make a list of lists to form a matrix*

matrix = [lst\_1,lst\_2,lst\_3]

In [30]:

*# Show*

matrix

We can again use indexing to grab elements, but now there are two levels for the index. The items in the matrix object, and then the items inside that list!

In [31]:

*# Grab first item in matrix object*

matrix[0]

In [32]:

*# Grab first item of the first item in the matrix object*

matrix[0][0]

**List Comprehensions**

Python has an advanced feature called list comprehensions. They allow for quick construction of lists. To fully understand list comprehensions we need to understand for loops. So don't worry if you don't completely understand this section, and feel free to just skip it since we will return to this topic later.

But in case you want to know now, here are a few examples!

In [33]:

*# Build a list comprehension by deconstructing a for loop within a []*

first\_col = [row[0] **for** row **in** matrix]

In [34]:

first\_col

We used a list comprehension here to grab the first element of every row in the matrix object. We will cover this in much more detail later on!

For more advanced methods and features of lists in Python, check out the Advanced Lists section later on in this course!