

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 [2]: # 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 [4]: 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 [6]: len(my_list)
```

```
Out[6]: 4
```

Indexing and Slicing

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

```
In [7]: my_list = ['one','two','three',4,5]
```

```
In [10]: # Grab element at index 0
my_list[0]
```

```
Out[10]: 'one'
```

```
In [11]: # Grab index 1 and everything past it
my_list[1:]
```

```
Out[11]: ['two', 'three', 4, 5]
```

```
In [13]: # Grab everything UP TO index 3
my_list[:3]
```

```
Out[13]: ['one', 'two', 'three']
```

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

```
In [14]: my_list + ['new item']
```

```
Out[14]: ['one', 'two', 'three', 4, 5, 'new item']
```

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

```
In [15]: my_list
```

```
Out[15]: ['one', 'two', 'three', 4, 5]
```

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

```
In [16]: # Reassign
my_list = my_list + ['add new item permanently']
```

```
In [18]: my_list
```

```
Out[18]: ['one', 'two', 'three', 4, 5, 'add new item permanently']
```

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

```
In [20]: # Make the list double
my_list * 2
```

```
Out[20]: ['one',
          'two',
          'three',
          4,
          5,
          'add new item permanently',
          'one',
          'two',
          'three',
          4,
          5,
          'add new item permanently']
```

```
In [23]: # Again doubling not permanent
my_list
```

```
Out[23]: ['one', 'two', 'three', 4, 5, 'add new item permanently']
```

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 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 [42]: # Create a new list
l = [1,2,3]
```

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

```
In [43]: # Append
l.append('append me!')
```

```
In [44]: # Show
l
```

```
Out[44]: [1, 2, 3, 'append me!']
```

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 [45]: # Pop off the 0 indexed item
l.pop(0)
```

```
Out[45]: 1
```

```
In [46]: # Show
l
```

```
Out[46]: [2, 3, 'append me!']
```

```
In [47]: # Assign the popped element, remember default popped index is -1
popped_item = l.pop()
```

```
In [48]: popped_item
```

```
Out[48]: 'append me!'
```

```
In [49]: # Show remaining list
1
```

```
Out[49]: [2, 3]
```

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

```
In [50]: l[100]
```

```
-----
IndexError                                Traceback (most recent call last)
<ipython-input-50-3e7ce3111e95> in <module>()
----> 1 l[100]

IndexError: list index out of range
```

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

```
In [51]: new_list = ['a','e','x','b','c']
```

```
In [52]: #Show
new_list
```

```
Out[52]: ['a', 'e', 'x', 'b', 'c']
```

```
In [53]: # Use reverse to reverse order (this is permanent!)
new_list.reverse()
```

```
In [54]: new_list
```

```
Out[54]: ['c', 'b', 'x', 'e', 'a']
```

```
In [55]: # Use sort to sort the List (in this case alphabetical order, but for numbers it
new_list.sort())
```

```
In [56]: new_list
```

```
Out[56]: ['a', 'b', 'c', 'e', 'x']
```

Nesting Lists

A great feature 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 [57]: # 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 [60]: # Show
matrix
```

```
Out[60]: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Now 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 [61]: # Grab first item in matrix object
matrix[0]
```

```
Out[61]: [1, 2, 3]
```

```
In [62]: # Grab first item of the first item in the matrix object
matrix[0][0]
```

```
Out[62]: 1
```

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 [63]: # Build a list comprehension by deconstructing a for loop within a []
first_col = [row[0] for row in matrix]
```

```
In [64]: first_col
```

```
Out[64]: [1, 4, 7]
```

We used 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 list section later on in this course!

In []: