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!

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 = ['string',23,12.3,'abhi']
```

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

In [3]:
```
len(my_list)
```

Out[3]: 4

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',1,2,3]
```

In [5]:
```
# Grab element at index 0
my_list[0]
```

Out[5]: 'one'

In [6]:
```
# Grab index 1 and everything past it
my_list[1:]
```

Out[6]: ['two', 'three', 1, 2, 3]

In [7]:
```
# Grab everything UP TO index 3
my_list[:3]
```

Out[7]: ['one', 'two', 'three']

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

In [8]:
```
my_list + ['abhi']
```

Out[8]: ['one', 'two', 'three', 1, 2, 3, 'abhi']

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

In [9]:
```
my_list
```

Out[9]:
```
['one', 'two', 'three', 1, 2, 3]
```

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

In [10]:
```
# Reassign
my_list = my_list + ['abhishek']
```

In [11]:
```
my_list
```

Out[11]:
```
['one', 'two', 'three', 1, 2, 3, 'abhishek']
```

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

In [12]:
```
# Make the list double
my_list * 2
```

Out[12]:
```
['one',
 'two',
 'three',
 1,
 2,
 3,
 'abhishek',
 'one',
 'two',
 'three',
 1,
 2,
 3,
 'abhishek']
```

In [13]:
```
# Again doubling not permanent
my_list
```

Out[13]:
```
['one', 'two', 'three', 1, 2, 3, 'abhishek']
```

Basic List Methods

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
```

Out[16]:
```
[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 [17]:
```
# Pop off the 0 indexed item
list1.pop(0)
```

Out[17]:
```
1
```

In [18]:
```
# Show
list1
```

Out[18]:
```
[2, 3, 'append me!']
```

In [19]:
```
# Assign the popped element, remember default popped index is -1
popped_item = list1.pop()
```

In [20]:
```
popped_item
```

Out[20]:
```
'append me!'
```

In [21]:
```
# Show remaining list
list1
```

Out[21]:
```
[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 [22]:
```
list1[100]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-22-b581ef4fac5e> 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','b','h','i']
```

In [24]:
```
#Show
new_list
```

Out[24]:
```
['a', 'b', 'h', 'i']
```

In [25]:
```
# Use reverse to reverse order (this is permanent!)
new_list.reverse()
```

In [26]:
```
new_list
```

Out[26]:
```
['i', 'h', 'b', 'a']
```

In [27]:
```
# Use sort to sort the list (in this case alphabetical order, but for numb
ers it will go ascending)
new_list.sort()
```

In [28]:
```
new_list
```

Out[28]:
```
['a', 'b', 'h', 'i']
```

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
```

Out[30]:
```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

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]
```

Out[31]:
```
[1, 2, 3]
```

In [32]:
```
# Grab first item of the first item in the matrix object
matrix[0][0]
```

Out[32]:
```
1
```