Lists

پیش‌تر وقتی که رشته‌ها را معرفی کردیم، به آن‌ها یک **دنباله** گفتیم. در حالت کلی در پایتون، به لیست‌ها هم می‌توان به چشم یک دنباله نگاه کرد. اما برخلاف رشته‌ها می‌توانیم عناصر درون یک لیست را تغییر دهیم. چیزهایی که در این آموزش خواهیم دید:

- ایجاد لیست‌ها
- اندیس گذاری و برش لیست
- متدهای پایه‌ای
- لیست‌های تودرتو

لیست‌ها با استفاده از [] ایجاد می‌شوند و هر عنصر درون آن با استفاده از ویرگول جدا می‌شود. بیایید ببینیم چه خبر هست!

```
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)
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',4,5]

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', 4, 5]

In [7]: # Grab everything UP TO index 3
        my_list[:3]
Out[7]: ['one', 'two', 'three']

In [10]: my_list[::-1]
Out[10]: [5, 4, 'three', 'two', 'one']

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

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

```
In [12]: my_list + ['new item']
```

```
Out[12]: ['one', 'two', 'three', 4, 5, 'new item']
```

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

```
In [13]: my_list
```

```
Out[13]: ['one', 'two', 'three', 4, 5]
```

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

```
In [14]: # Reassign
         my_list = my_list + ['add new item permanently']
```

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

```
Out[15]: ['one', 'two', 'three', 4, 5, 'add new item permanently']
```

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

```
In [16]: # Make the list double
         my_list * 2
```

```
Out[16]: ['one',
          'two',
          'three',
          4,
          5,
          'add new item permanently',
          'one',
          'two',
          'three',
          4,
          5,
          'add new item permanently']
```

```
In [17]: # Again doubling not permanent
         my_list
```

```
Out[17]: ['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 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 [18]: # Create a new list
         list1 = [1,2,3]
```

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

In [19]:
```
# Append
list1.append('append me!')
```

In [20]:
```
# Show
list1
```

Out[20]:
```
[1, 2, 3, 'append me!']
```

In [21]:
```
help(list1)
```

```
Help on list object:

class list(object)
 |  list(iterable=(), /)
 |
 |  Built-in mutable sequence.
 |
 |  If no argument is given, the constructor creates a new empty list.
 |  The argument must be an iterable if specified.
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __delitem__(self, key, /)
 |      Delete self[key].
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getitem__(...)
 |      x.__getitem__(y) <==> x[y]
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __iadd__(self, value, /)
 |      Implement self+=value.
 |
 |  __imul__(self, value, /)
 |      Implement self*=value.
 |
 |  __init__(self, /, *args, **kwargs)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __len__(self, /)
 |      Return len(self).
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __mul__(self, value, /)
 |      Return self*value.
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  __reversed__(self, /)
 |      Return a reverse iterator over the list.
```

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

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

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

In [ ]:
```
popped_item
```

In [ ]:
```
# 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 [ ]:
```
list1[100]
```

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

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

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

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

In [ ]:
```
new_list
```

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

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

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

In [ ]:
```
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!