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

```
# Assign a list to an variable named my_list
my_list = [1,2,3,4]
```

```
print(my_list)
```

```
[1, 2, 3, 4]
```

```
type(my_list)
```

```
list
```

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

```
my_list = ['A string',23,100.232,'o',True]
```

```
my_list
```

```
['A string', 23, 100.232, 'o', True]
```

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

```
len(my_list)
```

```
5
```

List Indexing

- Indexing work just like in strings.
- A list index refers to the location of an element in a list.

- Remember the indexing begins from 0 in Python.
- The first element is assigned an index 0, the second element is assigned an index of 1 and so on and so forth.

```
# Consider a list of strings
loss_functions = ['Mean Absolute Error','Mean Squared Error','Huber Loss','Log Loss','Hinge
print(loss_functions)
```

```
['Mean Absolute Error', 'Mean Squared Error', 'Huber Loss', 'Log Loss', 'Hinge Loss']
```

```
# Grab the element at index 0, which is the FIRST element
print(loss_functions[0])
```

```
Mean Absolute Error
```

```
len(loss_functions)
```

```
5
```

```
# Grab the element at index 3, which is the FOURTH element
print(loss_functions[3])
```

```
Log Loss
```

```
print(loss_functions[6])
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-11-36fa04b37479> in <module>()
----> 1 print(loss_functions[6])

IndexError: list index out of range
```

```
# Grab the element at the index -1, which is the LAST element
print(loss_functions[-1])
```

```
Hinge Loss
```

```
# Grab the element at the index -3, which is the THIRD LAST element
print(loss_functions[-3])
```

```
Huber Loss
```

List Slicing

- We can use a `:` to perform *slicing* which grabs everything up to a designated point.
- The starting index is specified on the left of the `:` and the ending index is specified on the right of the `:`.
- Remember the element located at the right index is not included.

```
# Print our list
print(loss_functions)
```

```
['Mean Absolute Error', 'Mean Squared Error', 'Huber Loss', 'Log Loss', 'Hinge Loss']
```

```
# Grab the elements starting from index 1 and everything past it
loss_functions[1:4]
```

```
['Mean Squared Error', 'Huber Loss', 'Log Loss']
```

- If you do not specify the ending index, then all elements are extracted which comes after the starting index including the element at that starting index. The operation knows only to stop when it has run through the entire list.

```
# Grab everything starting from index 2
loss_functions[2:]
```

```
['Huber Loss', 'Log Loss', 'Hinge Loss']
```

- If you do not specify the starting index, then all elements are extracted which comes befores the ending index excluding the element at the specified ending index. The operation knows only to stop when it has extracted all elements before the element at the ending index.

```
# Grab everything before the index 4
loss_functions[:4]
```

```
['Mean Absolute Error', 'Mean Squared Error', 'Huber Loss', 'Log Loss']
```

- If you do not specify the starting and the ending index, it will extract all elements of the list.

```
# Grab everything
print(loss_functions[:])
```

```
['Mean Absolute Error', 'Mean Squared Error', 'Huber Loss', 'Log Loss', 'Hinge Loss']
```

- We can also extract the last four elements. Remember we can use the index -4 to extract the FOURTH LAST element

```
# Grab the LAST FOUR elements of the list
loss_functions[-4:]
```

```
['Mean Squared Error', 'Huber Loss', 'Log Loss', 'Hinge Loss']
```

```
loss_functions[-10]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-20-8a581720341d> in <module>()
----> 1 loss_functions[-10]

IndexError: list index out of range
```

- It should also be noted that list indexing will return an error if there is no element at that index.

```
len(loss_functions)
```

```
5
```

```
loss_functions[8]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-21-4de3193e7629> in <module>()
----> 1 loss_functions[8]

IndexError: list index out of range
```

List Operations

- Remember we said that lists are mutable as opposed to strings. Lets see how can we change the elements of a list

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

```
print(loss_functions)
```

```
['Mean Absolute Error', 'Mean Squared Error', 'Huber Loss', 'Log Loss', 'Hinge Loss']
```

```
loss_functions + ['Kullback-Leibler']
```

```
['Mean Absolute Error',
 'Mean Squared Error',
 'Huber Loss',
 'Log Loss',
 'Hinge Loss',
 'Kullback-Leibler']
```

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

```
print(loss_functions)
```

```
['Mean Absolute Error', 'Mean Squared Error', 'Huber Loss', 'Log Loss', 'Hinge Loss']
```

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

```
# Reassign
loss_functions = loss_functions + ['Kullback-Leibler']
```

```
loss_functions
```

```
['Mean Absolute Error',
 'Mean Squared Error',
 'Huber Loss',
 'Log Loss',
 'Hinge Loss',
 'Kullback-Leibler']
```

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

```
# Make the list double
len(loss_functions * 6)
```

```
36
```

List Functions

`len()`

- `len()` function returns the length of the list

```
print(loss_functions)
```

```
['Mean Absolute Error', 'Mean Squared Error', 'Huber Loss', 'Log Loss', 'Hinge Loss', 'K
```

```
len(loss_functions)
```

```
6
```

min()

- `min()` function returns the minimum element of the list
- `min()` function only works with lists of similar data types

```
new_list = [6, 9, 1, 3, 5.5]
```

```
min(new_list)
```

```
1
```

```
my_new_list = ['a','b', 'z' ,'y' , 'm']
```

```
min(my_new_list)
```

```
'a'
```

max()

- `max()` function returns the maximum element of the list
- `max()` function only works with lists of similar data types

```
new_list = ['Argue','Burglar','Parent','Linear','shape']
```

```
max(new_list)
```

```
'shape'
```

`sum()`

- `sum()` function returns the sum of the elements of the list
- `sum()` function only works with lists of numeric data types

```
new_list = [6, 9, 1, 3, 5.5]
```

```
sum(new_list)
```

```
24.5
```

`sorted()`

- `sorted()` function returns the sorted list
- `sorted()` function takes reverse boolean as an argument
- `sorted()` function only works on a list with similar data types

```
new_list
```

```
[6, 9, 1, 3, 5.5]
```

```
sorted(new_list)
```

```
[1, 3, 5.5, 6, 9]
```

```
print(new_list)
```

```
[6, 9, 1, 3, 5.5]
```

```
new_list = ['Argue','Burglar','Parent','Linear','shape']
```

```
sorted(new_list)
```

```
['Argue', 'Burglar', 'Linear', 'Parent', 'shape']
```

```
print(new_list)
```

```
[6, 9, 1, 3, 5.5]
```

```
sorted(new_list,reverse=True)
```

```
[9, 6, 5.5, 3, 1]
```

```
sorted(new_list)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-45-99e16da477d4> in <module>()
----> 1 sorted(new_list,False)

TypeError: sorted expected 1 arguments, got 2
```

- `sorted()` function does not change our list

```
new_list
```

```
[6, 9, 1, 3, 5.5]
```

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:

```
# Create a new list
my_list = [1,2,3,1,1,1,3,10,5,8]
```

`append()`

- Use the `append()` method to permanently add an item to the end of a list.
- `append()` method takes the element which you want to add to the list as an argument

```
# Print the list
my_list
```

```
[1, 2, 3, 1, 1, 1, 3, 10, 5, 8]
```

```
len(my_list)
```

```
10
```

```
# Append to the end of the list
my_list.append('Append me!')
```

- Ah darn it! I was expecting some output. Lets see what happened to `my_list`

```
print(my_list)
```

```
[1, 2, 3, 1, 1, 1, 3, 10, 5, 8, 'Append me!']
```

```
len(my_list)
```

```
11
```

- Woah! Calling the `append()` method changed my list? Yes, the `append()` method changes your original list!

```
# Show
my_list.append(2.73)
```

```
print(my_list)
```

```
[1, 2, 3, 1, 1, 1, 3, 10, 5, 8, 'Append me!', 2.73]
```

- We can in fact add a list object to our `my_list` object

```
my_list.append([1,2,3])
```

```
my_list
```

```
[1, 2, 3, 1, 1, 1, 3, 10, 5, 8, 'Append me!', 2.73, [1, 2, 3]]
```

```
len(my_list)
```

```
13
```

```
my_list.append([10,[19,20],30])
```

```
len(my_list)
```

```
14
```

extend()

- Use the `extend()` method to merge a list to an existing list
- `extend()` method takes a list or any iterable(don't worry about it now) as an argument.
- Quite helpful when you have two or more lists and you want to merge them together

```
# Print the list
print(my_list)
```

```
[1, 2, 3, 1, 1, 1, 3, 10, 5, 8, 'Append me!', 2.73, [1, 2, 3], [10, [19, 20], 30]]
```

```
my_list.extend(['Wubba','Lubba','Dub Dub'])
```

```
print(my_list)
```

```
[1, 2, 3, 1, 1, 1, 3, 10, 5, 8, 'Append me!', 2.73, [1, 2, 3], [10, [19, 20], 30], 'Wubb
```

```
len(my_list)
```

```
17
```

pop()

- Use `pop()` to "pop off" an item from the list.
- By default `pop()` takes off the element at the last index, but you can also specify which index to pop off.
- `pop()` takes the index as an argument and returns the elenent which was popped off.

```
# Print the list
print(my_list)
```

```
[1, 2, 3, 1, 1, 1, 3, 10, 5, 8, 'Append me!', 2.73, [1, 2, 3], [10, [19, 20], 30], 'Wubb
```

```
# Pop off the 0 indexed item
my_list.pop()
```

```
'Dub Dub'
```

- `pop()` method changes the list by popping off the element at the specified index

```
print(my_list)
```

```
[1, 2, 3, 1, 1, 1, 3, 10, 5, 8, 'Append me!', 2.73, [1, 2, 3], [10, [19, 20], 30], 'Wubt
```

```
# Assign the popped element, remember default popped index is -1
my_list.pop(-1)
```

```
'Lubba'
```

```
len(my_list)
```

```
14
```

```
print(my_list)
```

```
[1, 2, 3, 1, 1, 1, 3, 10, 5, 8, 'Append me!', 2.73, [1, 2, 3], [10, [19, 20], 30], 'Wubt
```

`remove()`

- Use `remove()` to remove an item/element from the list.
- By default `remove()` removes the specified element from the list.
- `remove()` takes the element as an argument.

```
# Print the list
print(my_list)
```

```
[1, 2, 3, 1, 1, 1, 3, 10, 5, 8, 'Append me!', 2.73, [1, 2, 3], [10, [19, 20], 30], 'Wubt
```

```
# Remove the element which you want to
my_list.remove([10, [19, 20], 30])
```

```
# Show
print(my_list)
```

```
[1, 2, 3, 1, 1, 1, 3, 10, 5, 8, 'Append me!', 2.73, [1, 2, 3], 'Wubba']
```

```
my_list.remove([1,2,3])
```

```
print(my_list)
```

```
[1, 2, 3, 1, 1, 1, 3, 10, 5, 8, 'Append me!', 2.73, 'Wubba']
```

```
len(my_list)
```

```
12
```

```
my_list.remove(1)
```

```
print(my_list)
```

```
[2, 3, 1, 1, 3, 10, 5, 8, 1.43, 'Wubba', 'Lubba']
```

```
len(my_list)
```

```
11
```

```
my_list.clear
```

count()

- The `count()` method returns the total occurrence of a specified element in the list

```
print(my_list)
```

```
[1, 2, 3, 1, 1, 1, 3, 10, 5, 8, 'Append me!', 2.73, 'Wubba']
```

```
# Count the number of times element 1 occurs in my_list
my_list.count('Wubba')
```

```
1
```

```
my_list.count(1)
```

```
4
```

index()

- The `index()` method returns the index of a specified element.

```
print(my_list)
```

```
[1, 2, 3, 1, 1, 1, 3, 10, 5, 8, 'Append me!', 2.73, 'Wubba']
```

```
my_list.index('Wubba')
```

```
12
```

```
my_list.index(3)
```

```
2
```

sort()

- Use `sort()` to sort the list in either ascending/descending order
- The sorting is done in ascending order by default
- `sort()` method takes the reverse boolean as an argument
- `sort()` method only works on a list with elements of same data type

```
new_list = [6, 9, 1, 3, 5]
```

```
# Use sort to sort the list (this is permanent!)
new_list.sort()
```

```
print(new_list)
```

```
[1, 3, 5, 6, 9]
```

```
# Use the reverse boolean to set the ascending or descending order
new_list.sort(reverse=True)
print(new_list)
```

```
[9, 6, 5, 3, 1]
```

```
boolean_list = [True, False]
```

```
boolean_list.sort(reverse=True)
```

```
print(boolean_list)
```

```
[True, False]
```

reverse()

- `reverse()` method reverses the list

```
my_list = [1, 1, 1, 1, 1.43, 2, 3, 3, 5, 8, 10, 'Lubba', 'Dub Dub']
```

```
print(my_list)
```

```
[1, 1, 1, 1, 1.43, 2, 3, 3, 5, 8, 10, 'Lubba', 'Dub Dub']
```

```
my_list.reverse()
```

```
print(my_list)
```

```
['Dub Dub', 'Lubba', 10, 8, 5, 3, 3, 2, 1.43, 1, 1, 1, 1]
```

Nested 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 make three lists
lst_1=[1,2,3]
lst_2=['b','a','d']
lst_3=[7,8,9]

# Make a list of lists to form a matrix
list_of_lists = [lst_1,lst_2,lst_3]
```

```
print(list_of_lists)
```

```
[[1, 2, 3], ['b', 'a', 'd'], [7, 8, 9]]
```

```
# Show
type(list_of_lists)
```

```
list
```

```
# Grab first item in matrix object
list_of_lists[1]
```

```
['b', 'a', 'd']
```

```
# Grab first item of the first item in the matrix object
list_of_lists[1]
```

```
['b', 'a', 'd']
```

```
list_of_lists[1][1][][][]
```

```
'a'
```

Start coding or generate with AI.