Lesson 3: Lists & Tuples

There's many different data types available for manipulation in Python, and we'll talk about two ways of organizing that data into *sequences of objects* within this bootcamp.

Lists

First: lists. In programming, it is common to want to work with collections of data. In Python, a **list** is one of the many built-in data structures that allows us to work with a collection of data in sequential order.

In Python, **lists** are ordered collections of items that allow for easy use of a set of data. List values are placed in between square brackets `[ ]`, separated by commas. It is good practice to put a space between the comma and the next value. The values in a list do not need to be unique (taht is, the same value can be repeated). Lists are a versatile data type that can contain multiple different data types within the same square brackets. The possible data types within a list include numbers, strings, other objects, and even other lists.

Empty lists do not contain any values within the square brackets.

In [1]:
```
new_list = [1, -1, 2, -76.2, 'apple']
type(new_list)
```

Out[1]:
```
list
```

In [2]:
```
empty_list = []
empty_list
```

Out[2]:
```
[]
```

Notice how the list can be composed of any combination of types, but the type of the list itself is just `list`. A list can contain Boolean values, unevaluated arithmetic operations, or even another list as an element!

In [3]:
```
math_list = [2*3, 5-19, 5-4//2]
math_list
```

Out[3]:
```
[6, -14, 3]
```

> **Note:** Did you calculate that last element in the list above correctly? Make sure you understand the order of operations.

In [4]:
```
weird_list = [-32, 4.3, "apple", [42, "the meaning of life"]]
weird_list
```

Out[4]:
```
[-32, 4.3, 'apple', [42, 'the meaning of life']]
```

We can also create a list from the characters of a string.

In [5]:
```
sentence = 'a string'
list(sentence)
```

Out[5]:
```
['a', ' ', 's', 't', 'r', 'i', 'n', 'g']
```

List Operators

Operators on lists behave much like operators on strings. The `+` operator on lists means list concatenation.

In [6]:
```
[1, 2, 3] + [4, 5, 6]
```

Out[6]:
```
[1, 2, 3, 4, 5, 6]
```

> **Note:** This will not work for adding one item at a time (use `.append()` method, see far below). In order to add one item, create a new list with a single value and then use the plus symbol to add the list.

The `*` operator on lists means list replication and concatenation.

In [7]:
```
[1, 2, 3] * 3
```

Out[7]:
```
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

In [8]:
```
[1, 2, 3] * 4
```

Out[8]:
```
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Membership operators

Membership operators are used to determine if an item is in a list. There are two membership operators: `in` and `not in`, and both result in Boolean values, `True` or `False`.

Python	English
`in`	is a member of
`not in`	is not a member of

In [9]:
```
ex_list_2 = [1.1, -4.1, 'peach', [42, "the meaning of life"]]
1 in ex_list_2
```

Out[9]:
```
False
```

In [10]:
```
[42, "the meaning of life"] in ex_list_2
```

Out[10]: True

In [11]:
```
"the meaning of life" in ex_list_2
```

Out[11]: False

In [12]:
```
"the meaning of life" in ex_list_2[-1]
```

Out[12]: True

Notice that the string "the meaning of life" is not considered within the list `ex_list_2` because it is an element of a list within a list.

List Indices

We've covered how to find an element in a list, but what about grabbing an element out of a list via it's location in that list? For example, what if I want the 5th entry in a list? That's what the *index* is for.

Python list elements are ordered by *index*, a number referring to an element's placement in the list. List indices start at `0` and increment by one for every subsequent element in the list. To access a list element by index, square bracket notation is used: `list[index]`.

> **Note: Zero-Indexing** In Python, list index begins at zero and ends at the length of the list minus one. For example, in the following list, 'apple' is found at index `2`.

In [13]:
```
ex_list = [0, 'kiwi', 'apple', -5.1, 10.02, 12.34, -0.56, 'pickles', 2, 'ten', 11.0]
ex_list[2]
```

Out[13]: 'apple'

> **Note:** Negative indices for lists in Python can be used to reference elements in relation to the end of a list (think of it as counting backwards instead of forwards). This can be used to access single list elements or as part of defining a list range. For example,
>
> - `my_list[-1]` to select the last element
> - `my_list[-3:]` to select the last three elements
> - `my_list[:-2]` to select everything *except* the last two elements

In [14]:
```
ex_list[1:4]
```

Out[14]: ['kiwi', 'apple', -5.1]

In [15]:
```
ex_list[-3:]
```

Out[15]: [2, 'ten', 11.0]

In [16]:
```
ex_list[:-3]
```

Out[16]: `[0, 'kiwi', 'apple', -5.1, 10.02, 12.34, -0.56, 'pickles']`

In [17]:
```
list_example = [0, 1, 2, 3, 4, 5, 6, 7, 8 , 9, 10]
```

Values	0	1	2	3	4	5	6	7	8	9	10
Forward indices	0	1	2	3	4	5	6	7	8	9	10
Backward indices	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

In [18]:
```
list_example[10]
```

Out[18]: `10`

In [19]:
```
list_example[-1]
```

Out[19]: `10`

In [20]:
```
list_example[1]
```

Out[20]: `1`

In [21]:
```
list_example[-10]
```

Out[21]: `1`

In [22]:
```
len(list_example) #Length function in Python
```

Out[22]: `11`

List slicing

What if we want to find a range of elements in a list? That would be a **slice**, or a sub-list of Python list elements which can be selected from a list using a colon-separated starting and ending point `(:)`. The syntax pattern is `List[start:end]`. The slice will include the `start` index, and everything until but excluding the `end` element. When slicing a list, a new list is returned, so if the slice is saved and then altered, the original list remains the same.

In [23]:
```
list_example[0:5]
```

Out[23]: `[0, 1, 2, 3, 4]`

Then `[7:10]` will pull out the last three elements from a list.

In [24]:
```
list_example[7:10]
```

Out[24]: `[7, 8, 9]`

Using negative indices will instruct the machine what range to leave off the list instead of what to include.

In [25]:
```
list_example[0:-3]
```

Out[25]:
```
[0, 1, 2, 3, 4, 5, 6, 7]
```

In the case of the `list_example`, each value is incremented by one for each index. Instead of incrementing by one, we could increment by 2. This would be changing the **stride** of the list:

In [26]:
```
list_example[0:10:3]
```

Out[26]:
```
[0, 3, 6, 9]
```

> **Note:** If the end is left blank, the default is to include the entire string. Similarly, we can leave out the start index, as its default is zero.

In [ ]:
```
list_example[start:end:stride]
```

In general, the indexing scheme follows the properties:

- If there are no colons, a single element is returned.
- If there are any colons, we are slicing the list, and a list is returned.
- If there is one colon, stride is assumed to be `1`.
- If start is not specified, it is assumed to be zero.
- If end is not specified, the interpreted assumed you want the entire list.
- If stride is not specified, it is assumed to be `1`.

Mutability

Unlike the other data types we've so far encountered (`int`, `float`, `str`), lists are **mutable**. Mutable means we can change their values without creating a new list. However, we cannot change the data type or identity.

In [28]:
```
new_list = [0, 1, 2, 3, 4, 5] # a fresh new list for this example
new_list[4] = 'four' # reassign the element at index 4 to be 'four'
new_list # output the new new_list
```

Out[28]:
```
[0, 1, 2, 3, 'four', 5]
```

List Methods

As a quick aside, let's talk about all the ways we can restructure a list without actually having to make a new list.

- **`.append()`:** As mentioned previous, in Python you can add values to the end of a list using the `.append()` method. This will place the object passed in as a new element at the very end of the list. Printing the list afterwards will visually show the appended value. This `.append()` method is not to be confused with returning an entirely new list with the passed object.
- **`.remove()`:** The `.remove()` method in Python is used to remove an element from a list by

passing in the value of the element to be removed as an argument. In the case where two or more elements in the list have the same value, the first occurrence of the element is removed.

- **`.insert()`:** The Python list method `.insert()` allows us to add an element to a specific index in a list. It takes in two inputs:
  - The index that you want to insert into.
  - The element that you want to insert at the specified index.
- **`.pop()`:** The `.pop()` method allows us to remove an element from a list while also returning it. It accepts one optional input which is the index of the element to remove. If no index is provided, then the last element in the list will be removed and returned.
- **`.count()`:** The `.count()` Python list method searches a list for whatever search term it receives as an argument, then returns the number of matching entries found.
- **`.sort()`:** The `.sort()` Python list method will sort the contents of whatever list it is called on. Numerical lists will be sorted in ascending order, and lists of strings will be sorted into alphabetical order. It modifies the original list, and has no return value.

> **`sorted()` Function:** The Python `sorted()` function accepts a list as an argument, and will return a new, sorted list containing the same elements as the original. Numerical lists will be sorted in ascending order, and lists of Strings will be sorted into alphabetical order. It does not modify the original, unsorted list.

In [ ]:
```
new_list.append(6)
new_list
```

In [ ]:
```
new_list.remove(3)
new_list
```

In [ ]:
```
new_list.insert(6,'six')
new_list
```

In [ ]:
```
new_list.insert(5,4)
new_list
```

In [ ]:
```
new_list.sort()
```

In [ ]:
```
sorted(new_list)
```

Tuples

A **tuple** is just like a list, except it's *immutable* (basically a read-only list). It's also created just like a list, except we use parentheses instead of brackets. The only thing to watch out for is that a tuple with a single item needs to include a comma after the item.

In [ ]:
```
a_tuple = (0,)
not_a_tuple = (0) # this is just the number 0 (normal use of parantheses)
type(a_tuple), type(not_a_tuple)
```

Lists can be converted into tuples with the `tuple()` function acting on a list, as follows.

In [ ]:
```
a_list = [-32, 4.3, "apple", [42, "the meaning of life"]]
list_to_tuple = tuple(a_list)
list_to_tuple
```

> **Note:** The list within `a_list` is still a list, even as an element within `list_to_tuple`.

In [ ]:
```
type(list_to_tuple[3])
```

And if we try to change any element of this tuple, we'll get an error.

In [ ]:
```
list_to_tuple[3] = 7
```

Take a good look at that error:

```
TypeError: 'tuple' object does not support item assignment
```

This is an example of the tuple's immutable property in action. In fact, the immutability of tuples makes them a rather safe and convenient way to store data, making them very useful. So, unless you really need mutability for something, use tuples instead of lists.

Tuple Slicing

Slicing of tuples is the same as lists, except a tuple is returned from the slicing operation instead of a list.

In [31]:
```
new_tup = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
new_tup[::-1] # reverses order of entries
```

Out[31]:
```
(10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0)
```

In [32]:
```
new_tup
```

Out[32]:
```
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

In [33]:
```
new_tup[1::2] # extracts only odd-valued entries
```

Out[33]:
```
(1, 3, 5, 7, 9)
```

In [34]:
```
new_tup
```

Out[34]:
```
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

Operator `+` on Tuples

Same as lists, the operator `+` can concatenate tuples, as seen in the example below.

In [35]:
```
diff_tup = new_tup + (11, 12, 13, 14, 15)
diff_tup
```

Out[35]:
```
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)
```

In [36]:
```
new_tup
```

Out[36]:
```
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

Membership operators on Tuples

Membership operators work the same on tuples as on lists, as well.

In [37]:
```
9 in new_tup # "is the value 9 within the tuple new_tup?"
```

Out[37]:
```
True
```

In [38]:
```
'apple' not in new_tup # "is 'apple' not within new_tup?"
```

Out[38]:
```
True
```

Unpacking Tuples

Now let's talk about something slightly different: *unpacking* tuples. This action means taking the elements of a tuple and assigning them to another variable to keep track of. *Unpacking* is particularly useful when we want to return more than one value from a function and further using the values as stored in different variables. We'll make use of this later on when we talk about functions in a few lessons.

In [39]:
```
tiny_tup = (1, 2, 3)
(x, y, z) = tiny_tup
# (x, y, z) = (1, 2, 3)
```

In [40]:
```
x
```

Out[40]:
```
1
```

In [41]:
```
y
```

Out[41]:
```
2
```

In [42]:
```
z
```

Out[42]:
```
3
```

> **Note:** The paranthesis around the newly assigned variables are dispensible, as noted below.

In [43]:
```
a, b, c = tiny_tup
print(a,b,c)
```

```
1 2 3
```

Please proceed to Assignment 3.