ICS3U: Tuple and List Basics

Tuples

While a string is a sequence of characters, a **tuple** is a collection of objects. Each object in a tuple is called an **element**. These could be integers, floating-point values, strings, or even other tuples. A tuple is created when two or more objects (separated by commas) are assigned to a variable. Round brackets, `()`, can also be included to make it clear that a tuple is being created.

```
>>> coordinates = (5, 3)
>>> type(coordinates)
<class 'tuple'>
```

Tuples may be composed of elements of any data type, and need not be of a single data type.

```
>>> strings = ("we", "love", "Python")
>>> mixed = (9.8, "hello", -3, True)
```

Indexing and slicing operate in the same manner with tuples as they do with strings. Instead of returning individual characters, however, they return objects.

```
>>> coordinates[0]
5
>>> strings[1:]
('love', 'Python')
```

Like strings, tuples are immutable. Attempting to change a value will result in an error.

```
>>> coordinates[0] = 7
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Lists

While tuples are good for storing collections of values that do not need to change, in many cases we will want to alter the contents as a program runs. Python provides **lists** for this. To create a list, use square brackets, `[]`.

```
>>> integers = [5, 9, 1, 3, 7]
>>> type(integers)
<class 'list'>
```

It is possible to create an empty list – one with no elements inside of it – using an empty set of square brackets. We will use this a lot when dynamically adding and deleting elements in later lessons.

```
>>> empty = []
>>> type(empty)
<class 'list'>
>>> empty
[]
```

Like tuples and strings, indexing and slicing work with lists.

```
>>> integers[0]
5
>>> integers[2:]
[1, 3, 7]
```

Sometimes we want to create a list with a certain number of place-holder elements in it. These values may be overridden later in a program. To set the number of elements in a list, include the place-holder value as an element and use the `*` operator to duplicate it.

```
>>> zeroes = [0]*5
>>> zeroes
[0, 0, 0, 0, 0]
```

As mentioned earlier, lists are mutable and their contents can be changed.

```
>>> zeroes[1] = 4
>>> zeroes
[0, 4, 0, 0, 0]
```

It is also possible to replace a slice of a list with the contents of another list.

```
>>> zeroes[1:4] = [9, 7, 3]
>>> zeroes
[0, 9, 7, 3, 0]
>>> zeroes[:3] = [2, 1]
>>> zeroes
[2, 1, 0]
```

Note that the list may adjust its size as necessary to contain the new elements.

Functions Used With Tuples and Lists

When `len` is used with a tuple or list, it returns the number of elements in it. It does not count the number of characters in strings contained in a list, or the number of digits in any numeric values. Using `len` on an empty list will return a value of zero.

```
>>> len(integers)
5
>>> len(empty)
0
```

The `max` and `min` functions will find the largest and smallest elements in a tuple or list. For numeric data types, this works as expected. For strings, `min` returns the string that would appear first alphabetically (with all uppercase letters preceding lowercase ones), while `max` returns the string that would appear last.

```
>>> max(numbers)
9
>>> max(strings)
'we'
>>> min(strings)
'Python'
```

The last example is often surprising to programmers who are new to Python, since 'l' appears before 'P' alphabetically, but since the 'P' is uppercase, it comes before lowercase 'l'.

For numeric data types, `sum` will determine the sum of all values in a list. Behind-the-scenes, this is done via a simple loop that iterates over all of the elements.

```
>>> sum(integers)
25
```

This is equivalent to the code below.

```
total = 0
for i in integers:
   total += 1
```

Attempting to use `sum` on a tuple or list that contains non-numeric values will result in an error.

```
>>> sum(mixed)
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

One important thing to note about lists is that variables that are assigned to lists behave a little bit differently than other simple data types like integers and strings. Consider the following example, where the variables `x` and `y` are assigned different values.

```
>>> x = 5
>>> y = x
>>> x
5
>>> y
5
>>> y = 7
>>> x
5
>>> y
7
```

When `y` is assigned the value of `x`, both variables have the same value. Later, when `y` is reassigned a new value, `x` retains its value. Now, let's repeat the same example with lists.

```
>>> L1 = [1, 2, 3]
>>> L2 = L1
>>> L1
[1, 2, 3]
>>> L2
[1, 2, 3]
>>> L2 = [4, 5, 6]
>>> L1
[1, 2, 3]
>>> L2
[4, 5, 6]
```

Like the integers `x` and `y`, when `L1` is assigned `L2`, the values of each list are the same. Later, when `L2` is assigned a new value, `L1` retains its value. Now, look carefully at the following example.

```
>>> L1 = [1, 2, 3]
>>> L2 = L1
>>> L1
[1, 2, 3]
>>> L2
[1, 2, 3]
>>> L2[0] = 9
>>> L1
[9, 2, 3]
>>> L2
[9, 2, 3]
```

Note that for *both* `L1` and `L2`, the first element had its value changed. This may come as a *big* surprise for novice Python programmers. The reason why this occurs is because both variables, `L1` and `L2`, point to the same list in main memory. Lists can contain a large number of elements, so Python attempts to conserve memory by maintaining only a single instance of the list. Since L1 and L2 refer to the same list, any changes to `L2` will result in changes to `L1`, and *vice versa*. In the previous example, we reassigned a completely new list to `L2`, so this did not occur.

There are a several ways to make a completely independent copy of a list. One of them is to use the built-in `list` function.

```
>>> L1 = [1, 2, 3]
>>> L2 = list(L1)
>>> L2[0] = 9
>>> L1
[1, 2, 3]
>>> L2
[9, 2, 3]
```

Alternatively, you can create a new instance of a list by using a slice that includes the entire list. This is the more “Pythonic” way of duplicating a list.

```
>>> L1 = [1, 2, 3]
>>> L2 = L1[::]
>>> L2[0] = 9
>>> L1
[1, 2, 3]
>>> L2
[9, 2, 3]
```