CSE231 - Lab 06

Lists, Tuples, Mutability

Finally, a real data structure

After all this time, we finally get to use a real data structure. Like we've talked about before, strings could be considered a fairly primitive data structure since it's essentially a container that holds ASCII characters.

Lists, however, can hold ***anything***. They can be built and edited in a similar fashion to strings, just that now we can hold a variety of other data types. When talking generally about programming, you might see people online talk about "arrays", and so it's probably helpful to make the connection that "lists" are the "arrays" of Python, just with a different name.

Tuples are identical to lists, but are *immutable*. We'll talk about what this means in a bit.

What can you do with a list?

Like strings, they share *a lot* of the same functionality. You're able to slice and iterate through them like you can strings. You can use the `in` keyword, the `len()` function to determine the number of objects within it, etc..

```
my_list = [1, 2, 3, 4, 5]

slice = my_list[0:3]    # my_slice = [1, 2, 3]

length = len(my_list)    # length = 5

my_cool_bool = 2 in my_list    # my_cool_bool = True
```

What can you do with a list? (cont.)

Like I said, lists can hold *anything*, and that includes other data structures. You can have lists of strings and *lists of lists*, which is a very common structure used to simulate matrices.

```
my_list = [“Arthur”, “King”, “of”, “the”, “Britons”]

A = my_list[1]    # A = “King”

B = my_list[1][2]    # B = “n”

my_list = [ [1, 2], [3, 4] ]

C = my_list[1]    # C = [3, 4]

D = my_list[1][0]    # D = 3
```

Creating Lists

You've seen me initialize lists quite a bit now, which you use the square brackets for. You can also initialize an empty list and append to it later, or create one from an already existing data structure using a type-cast.

```
my_list = []

for i in range(5):    # my_list = [0, 1, 2, 3, 4]

    my_list.append( i )

my_list = list("Hello")    # my_list = ["H", "e", "l", "l", "o"]

my_list = ["string", 1, 3.14, True]    # Mixing types
```

List Methods

Most of the method functions we'll be talking about are referred to as *in-place* methods. Meaning that you *do not* re-assign the variable that holds your list.

`.append()` - Takes any element and concatenates it to the back of the list

`.pop()` - Removes last element of the list, unless given an index argument. Returns the element that was just popped.

`.sort()` - Sorts the list, ascending order by default

`.insert()` - Takes `` `index` ``, `` `value` `` as arguments, inserts value into the position index

`.remove()` - Removes the first instance of a specified value

.append()

`.append(obj)`

`obj` - any object to be appended to an already existing list

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

my_list.append(5)    # my_list = [1, 2, 3, 4, 5]
```

.pop()

```
.pop([index])
```

`index` - Optional. Index of the object you want to remove, default is the -1.

Returns the object that was just removed.

```
my_list = [1, 2, 3, 4, 5]

elem = my_list.pop()    # my_list is now [1, 2, 3, 4]

# and elem now holds the value: 5
```

.sort()

`.sort([key, reverse])`

Sorts a list of objects in ascending order. You can supply ‘`reverse=True`’ to make it sort in descending order. We’ll discuss the ‘`key`’ parameter at a later date.

```
my_list = [5, 2, 3, 1, 4]

my_list.sort()    # my_list = [1, 2, 3, 4, 5]

my_list.sort(reverse=True)    # my_list = [5, 4, 3, 2, 1]
```

.insert()

`.insert(index, obj)`

`index` - the index of the list you want `obj` to be inserted into

`obj` - the object you want inserted

```
my_list = [1, 2, 4, 5]

my_list.insert(2, 3)    # my_list = [1, 2, 3, 4, 5]
```

.remove()

`.remove(obj)`

`obj` - the object you want removed from your list

```
my_list = [1, 2, 3, 4, 5]

my_list.remove(3)    # my_list = [1, 2, 4, 5]
```

`.join()` / `.split()`

These two methods are extremely useful to jump back and forth between strings and lists. Both of these *are string methods.*

While you *can* use the list-cast, `list()`, on a string to convert it to a list, this will split the string into a list by character, which isn't *usually* what we want.

`.join( list_obj )` - joins the elements in a list by delimitation of the string being acted on by the method function

`.split( [delim] )` - splits a string into separate objects of a list by the `` `delim` `` string. If no `` `delim` ``, it will split by *any* amount of whitespace in between characters. (Important for the lab today)

`.join()` / `.split()` Example

Note that these are *not* in-place method functions. They do not edit the original value, they return copies since strings are immutable.

```
my_list = [“I”, “am”, “very”, “tired”]

out_str = “--”.join(my_list)  # out_str = “I--am--very--tired”

out_list = out_str.split(“--”) # out_list = [“I”, “am”, “very”, “tired”]

whacky = out_str.split(“e”)  # whacky = [“I--am--v”, “ry--tir”, “d”]
```

Tuples

At the beginning, I briefly mentioned tuples. They’re like lists, but are instead *immutable*, meaning you cannot edit their contents.

```
my_tuple = (“I”, “am”, “making”, “this”, “at”, “2am”)

my_tuple.append(“hahaha”)

# AttributeError: 'tuple' object has no attribute 'append'
```

Tuples *do not* have any method functions that can change the contents of the tuple. You can “look” at them, meaning you can do things like take a copy, access elements, loop through it, etc..

Mutability

"Objects of built-in types like (int, float, bool, str, tuple) are immutable. Objects of built-in types like (list, set, dict) are mutable."

Any time you've been "changing" an object in Python like an int, float or string, you've actually been taking a copy of the original value and changing the *variable's* held value*. You've not been changing the value itself.*

This is why lists are different. Lists have method functions (the ones we've talked about) that go into where the memory for the list objects are being held on your computer, and edits the value of them.

We'll talk more about the consequences of this later.

List Comprehension

If you’re looking to reduce some lines or get spicy with your Python, you can quickly generate lists in one line with list comprehension.

```
my_list = [i for i in range(3)]    # [0, 1, 2]

my_list = [i**2 for i in range(1, 4)]    # [1, 4, 9]

my_list = [i**3 for i in range(10) if i % 2 == 0]

# ^ this equates to: [0, 8, 64, 216, 512]

my_list = [0 for i in range(5)]    # [0, 0, 0, 0, 0]
```

List Comprehension (cont.)

Here's a small example that shows an equivalency between the traditional for-loop appendation, and a list comprehension.

```
my_list = [i for i in range(3)]

# [0, 1, 2], the example from earlier

my_list = []

for i in range(3):    # Equivalent methodology

   my_list.append( i )
```