Notebook

March 27, 2019

1 Complex Data Types

Some kinds of data can store other kinds of data.

1.1 Lists

We've actually seen the most common complex data type a few times before, I just haven't pointed it out to you. We make a list by enclosing the elements of a list in square brackets.

```
In [1]: list_of_numbers = [1, 2, 3]
        list_of_strings = ["cat", "dog", "bird"]
```

we can have lists of lists.

```
In [2]: for sublist in [list_of_numbers, list_of_strings]:
            for item in sublist:
                print(item)
```

```
1
2
3
cat
dog
bird
```

As you've seen, you can iterate over lists with a loop. Indeed, lists are the most common data structure in Python for such tasks.

One extremely important fact about lists in Python is that they are *mutable*. By this, I mean that you can have a list, assigned to a name, and you can change its contents without changing its name or the relationship between that list and the variable. Indeed, lists have a bunch of methods that are intended to mutate them. Let's see some, eh?

```
In [3]: example = list_of_strings.append("fish")
        counterexample = "cat".upper()
```

You might expect the `.append()` method to work sort of like the `.upper()` method for strings, where it returns a new instance of the data type. But, as you'll see, it doesn't:

```
In [4]: print(example)

None


In [5]: print(counterexample)

CAT
```

The reason that the `append()` method doesn't return anything is because it mutates the original list: now the original list has a new word in it!

```
In [6]: print(list_of_strings)

['cat', 'dog', 'bird', 'fish']


In [7]: list_of_strings.sort()
        print(list_of_strings)

['bird', 'cat', 'dog', 'fish']
```

Mutation is useful, but it's also perilous. In Jupyter Notebooks, in particular, it's easy to mess things up. For example, suppose I had realized I'd made a mistake in some complex code cell where I'd run a bunch of mutating functions, and so I tweaked something and ran it again. Then, way down the line, like 50 cells later, I might find that I had, say, the word `fish` twice in my list rather than just once! This can be a very difficult problem to debug. (The easiest solution is usually to go up to the top of the screen and click "Kernel" and then "Restart & Run All"

Another trap with mutable lists is that you can actually change them in functions without using the `global` keyword. Why? Well, because scope is about controlling which variable name refers to what, not what's inside them!

```
In [8]: def add_cow_to_list(animals):
            animals.append("cow")
            return animals

        is_this_a_second_list = add_cow_to_list(list_of_strings)

In [9]: print(is_this_a_second_list)

['bird', 'cat', 'dog', 'fish', 'cow']


In [10]: print(list_of_strings)

['bird', 'cat', 'dog', 'fish', 'cow']
```

The original list got changed inside the function! But it gets even crazier.

```
In [11]: is_this_a_second_list.append("mouse")
         print(list_of_strings)

['bird', 'cat', 'dog', 'fish', 'cow', 'mouse']
```

That's right. We didn't actually make a second list when we returned the list out of the function. We actually just assigned the same list to a second variable!

This is a general principle of Python lists: when you assign them to a new variable, it doesn't make a copy, it just adds another variable pointing to the same list; when you change one, you change the other.

This can be really annoying behavior, it's actually one of my least favorite things about Python. It means it's easy to screw up your lists accidentally. And it's a big way that beginners get tripped up. A good rule of thumb for avoiding problems is to never mutate the original list in either a function or a loop. Instead, create a new blank list (which you can create with `[]`) and then add things to that. For example:

```
In [12]: def safe_add_horse(animals):
             newlist = []
             for x in animals:
                 newlist.append(x)
             newlist.append("horse")
             return newlist

In [13]: neigh = safe_add_horse(list_of_strings)

In [14]: print(list_of_strings)

['bird', 'cat', 'dog', 'fish', 'cow', 'mouse']


In [15]: print(neigh)

['bird', 'cat', 'dog', 'fish', 'cow', 'mouse', 'horse']
```

There are more sophisticated ways of dealing with these problems, which we can talk about down the road.

Like strings, we can slice (subset) lists.

```
In [16]: list_of_strings[1]

Out[16]: 'cat'

In [17]: list_of_strings[-1]

Out[17]: 'mouse'
```

(Negative numbers index strings from the end, but start at 1.)

```
In [18]: list_of_strings[-1:-5:-1]
```

```
Out[18]: ['mouse', 'cow', 'fish', 'dog']
```

One nice trick about slicing lists is that slices return new lists, so they can help you get around mutability problems. A secret trick to copy a list is to use an empty slice with all the elements represented by blank spots between the colons.

```
In [19]: newlist = list_of_strings[::]
         newlist.append("bear")
         print(newlist)
         print(list_of_strings)
```

```
['bird', 'cat', 'dog', 'fish', 'cow', 'mouse', 'bear']
['bird', 'cat', 'dog', 'fish', 'cow', 'mouse']
```

But be careful. This just makes what's known as a *shallow* copy: it doesn't copy lists within lists, so you can still accidentally mutate those.

```
In [20]: biglist = [list_of_numbers, list_of_strings]
         new_big_list = biglist[::]
         new_big_list[0].append(4)
         print(biglist)
         print(new_big_list)
```

```
[[1, 2, 3, 4], ['bird', 'cat', 'dog', 'fish', 'cow', 'mouse']]
[[1, 2, 3, 4], ['bird', 'cat', 'dog', 'fish', 'cow', 'mouse']]
```

1.2 Tuples

Tuples are surrounded by parentheses, and are like lists, except they're *immutable*—what that means is that you can't change them. There isn't an append or a sort or anything like that.

However, that doesn't mean they're useless, because you can concatenate them with a plus sign (you can do that with lists too) and assign them to new variables.

```
In [21]: mytupe = (1, 2)
         myothertupe = (3, 4)
         mytupe + myothertupe
```

```
Out[21]: (1, 2, 3, 4)
```

Also, even though something's immutable, it doesn't mean that you can't reassign the name, it just means you can't muck around with the contents. So this works:

```
In [22]: mytupe = mytupe + myothertupe

         print(mytupe)
```

```
(1, 2, 3, 4)
```

But you can't do things like assigning to slices—you could do that with lists, but not with tuples. (You also can't do it with strings.)

```
In [23]: list_of_numbers[0] = 99
         print(list_of_numbers)
```

```
[99, 2, 3, 4]
```

```
In [24]: mytupe[0] = 99
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)

<ipython-input-24-7dc7a24e9236> in <module>
----> 1 mytupe[0] = 99


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

1.3 Dictionaries

The other main complex data type is the dictionary, or "dict". That's how Python stores associations between names and other data in a list-like form. For example, think of a telephone directory, that associates names with numbers. You could easily represent that with a Python dict.

```
In [25]: phonebook = {"Joe Schmoe": 5551212, "Jill Schmill": 911, "Jack Schmack": 8675309}
```

The syntax should be self-explanatory: curly brackets around the edges, then pairs of a name, or *key*, followed by a colon, followed by a *value* for that key, where all the pairs are separated by commas.

Phonebooks, unlike lists and tuples, don't have an order to them. Instead, you access them by their keys.

```
In [26]: print(phonebook["Jack Schmack"])
```

```
8675309
```

Dictionaries are also mutable, and you can add a new key or change the value attached to a key just by assigning to it.

```
In [27]: phonebook["Paul Gowder"] =  3843202
```

```
In [28]: print(phonebook)
```

```
{'Joe Schmoe': 5551212, 'Jill Schmill': 911, 'Jack Schmack': 8675309, 'Paul Gowder': 3843202}
```

You can iterate over a list's keys, values, or items (as key-value tuples) with, respectively, the `keys()`, `values()`, and `items()` methods.

```
In [30]: for item in phonebook.items():
             print(item)
```

```
('Joe Schmoe', 5551212)
('Jill Schmill', 911)
('Jack Schmack', 8675309)
('Paul Gowder', 3843202)
```

The results of those methods act like lists, but they aren't really lists—they have a special property called *laziness* which means that they don't actually exist in memory until you use them.

```
In [31]: lazykeys = phonebook.keys()
```

```
In [32]: print(lazykeys)
```

```
dict_keys(['Joe Schmoe', 'Jill Schmill', 'Jack Schmack', 'Paul Gowder'])
```

```
In [33]: print(lazykeys[0])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)

<ipython-input-33-01ad6b36bc56> in <module>
----> 1 print(lazykeys[0])


TypeError: 'dict_keys' object does not support indexing
```

It doesn't support indexing because it's not a real list. But you can make it one by slapping a call to a `list()` constructor function around it:

```
In [34]: eagerkeys = list(lazykeys)
```

```
In [35]: print(eagerkeys)
```

```
['Joe Schmoe', 'Jill Schmill', 'Jack Schmack', 'Paul Gowder']
```

```
In [36]: eagerkeys[0]
```

```
Out[36]: 'Joe Schmoe'
```