for-loops and range(): some examples (with bonus inclusion of map())

Hi all, some questions have come up repeatedly from students and I thought I'd gather some examples that might help clarify some things. The material in Example 5 illustrates a rather subtle issue with iterating over a list vs. iterating over the indices of the list (i.e., over list(range(len(<the list>))), and is worth being read even if in the case that you feel quite comfortable with for-loops already. (The contents of this post/note have also been posted to the course website at http://www.cs.cornell.edu/courses/cs1110/2018sp/lectures/lecture11/writeup_with_more_loop_examples.pdf)

Let's use Python interactive mode to demonstrate. Try these out yourself if you'd like to give yourself some more practice!

Example 1: for every person *in a given list*, do exactly the same thing --- say "hi". (So, if there are 2 people in the list, say "hi" exactly 2 times; if there are 4 people in the list, say "hi" exactly 4 times).

```
>>> people = ["Hakim", "David", "Anil", "Lillian", "Claire"]
>>> for person in people:
...     print("hi")  #  you have to hit "tab" to indent
... #  you have to hit return again in interactive mode
...
# person has value "Hakim"
hi
# person has value "David"
hi
# person has value "Anil"
hi
# person has value "Lillian"
hi
# person has value "Claire"
hi
# person still has value "Claire", but the loop has stopped
```

Example 2: for every person *in a given list,* do something specific for that person --- say "hi <person name>!".

```
>>> for person in people:
...     print("hi " + person + "!")
...
hi Hakim!
hi David!
hi Anil!
hi Lillian!
hi Claire!
```

Example 3: do something a fixed number of times

Actually, you've already seen this in example 1! The list `people` has five elements in it, so printing "hi" for each item in a 5-element list is going to cause "hi" to be printed 5 times.

But, it's annoying to have to create a dummy 5-item list just to be able to express the notion "5 times". This is where **range()** comes in.

One technical wrinkle: in CS1110, we present for-
loops as always iterating over a list. Technically speaking, the range() function returns something that isn't actually a list, so for conceptually simplicity (even if it takes more typing), w e convert the output of range() via the list() function. [BUT, code is actually correct if you skip listifying the range's return value.]

First, let's just see how the range() function is used:

Example 3a: Print "the ants come marching home" 5 times:

```
>>> for i in list(range(5)):  # OK to also say 'for i in range(5):'
...     print("the ants come marching home")
...
# i has value 0
the ants come marching home
# i has value 1
the ants come marching home
# i has value 2
the ants come marching home
# i has value 3
the ants come marching home
# i has value 4
```

```
the ants come marching home
# i still has value 4, the loop has stopped
```

Example 3b: Print "the ants come marching home" 3 times:

```
>>> for i in list(range(3)):  # OK to also say 'for i in range(3):'
...     print("the ants come marching home")
...
# i has value 0
the ants come marching home
# i has value 1
the ants come marching home
# i has value 2
the ants come marching home
# i still has value 2, the loop has stopped
```

Example 4: what the range function is doing

The syntax of the arguments to range is similar to that of list slicing.

Example 4a: creating a 5-element list that contains the elements 0, 1, 2, 3, and 4.

```
>>> print(list(range(5)))   # saying 'print(range(5))' gives you something different ...
[0, 1, 2, 3, 4]
```

Example 4b: creating a 3 element list that contains the elements 0, 1, 2

```
>>> print(list(range(3)))  # saying 'print(range(3))' gives you something different ...
[0, 1, 2]
```

Example 4c: printing the square of every integer between 1 and 6 inclusive

```
>>> for num in list(range(1,7)):  # OK to say 'for num in range(1,7):'
...     print(str(num**2))
...
# num is 1
1
# num is 2
4
# num is 3
9
# num is 4
16
# num is 5
25
# num is 6
36
# num is still 6, loop has stopped
```

Example 5: if mylist is a list, is there a difference between 'for item in mylist:' and 'for index in list(range(len(mylist)))'?

Answer, *sometimes.* And that sometimes is a source of sometimes mysterious bugs (as well as prelim questions ...) Section 10.3 in the book addresses this topic.

Here's a case where you can get the same result either way, but the bodies of the loops are different.

Example 5a: Let's take the list people from above again, and where we want to greet each person individually.

In this case, I would personally prefer to loop directly over the list, because the code is simpler --- this was given in Example 2 above. Here is is again:

```
>>> for person in people:
...     print("hi " + person + "!")
...
```

```
hi Hakim!
hi David!
hi Anil!
hi Lillian!
hi Claire!
```

Here's how one way we could get the same printout, but looping over the sequence [0, 1, 2, 3, 4] and treating each item in it as a index into the list people:

```
>>> n = len(people)
>>> print(n)
5
>>> for i in list(range(n)):
...     print("hi " + people[i] + "!")
...
hi Hakim!
hi David!
hi Anil!
hi Lillian!
hi Claire!
```

Alternately, you can skip the intermediate step of creating "n", and just do this:

```
>>> for i in list(range(len(people))):
...     print("hi " + people[i] + "!")
...
hi Hakim!
hi David!
hi Anil!
hi Lillian!
hi Claire!
```

Important example 5b: Suppose we needed to update a list of data observations so that every True became a False and every False needed to be changed to a True. (Maybe we're flipping an image from black-on-white to white-on-black. I'm using True and False so we don't get confused between list indices and values in the list.)

Here, a natural way to do this would be to use iteration over the indices -- for each position in the list, "flip' (do a "not" on) the item at that position:

```
>>> data = [True, False, True, False]
>>> for i in range(len(data)):  # OK to also do 'for i in list(range(len(data))):'
...     data[i] = not data[i]
...
>>> data
[False, True, False, True]
```

This supposed alternate for-loop on the list itself instead of the list indices does NOT work, because I don't know what variable to put in the place of ???

```
>>> for item in data:
...     ??? = not item
```

You might respond, "couldn't I just get the position by using index, like data[data.index(item)] = not item?"

There's nothing syntactically wrong with that, but you don't get the result you expect.

```
>>> data = [True, False, True, False]
>>> for item in data:  # adding print statements for clarity
...     i = data.index(item)
...     print("\titem is: " + str(item) + "; i is: " + str(data.index(item)))
...     data[i] = not data[i]   # changing this to 'data[i] = not item' will not solve the problem
...     print("\tNow data is: " + str(data))
...
        item is: True; i is: 0
        Now data is: [False, False, True, False]   # Note that the first False is now at 0!
        item is: False; i is: 0
        Now data is: [True, False, True, False]   # Note that the first True is now once again at 0!
        item is: True; i is: 0
        Now data is: [False, False, True, False]  # Note that the first False is now once again at 0!
        item is: False; i is: 0
```

```
        Now data is: [True, False, True, False]
```

It looks like data didn't change. Actually, it did change during the loop: the item at 0 was "reversed" several times.

General principle: using index() to get the index of a particular item is tricky when there might be repeated items in your list.

Now, if you are the contrarian type, you might go away and think about this for a while, and then come back and say, "but couldn't I just do this: make a *new list* and then assign data to be that new list?"

```
>>> data = [True, False, True, False]
>>> alteredcopy = []
>>> for item in data:
...     alteredcopy.append(not item)
...
>>> data = alteredcopy
>>> data[False, True, False, True]
```

To which I would say, "Very clever!" What you have done may not have been anticipated by the textbook author when they wrote 10.3.

But a general problem with this approach of making a new list and then reassigning the old list variable to store the new list is, if you were doing this inside a function and data was a parameter to your function, then after the function call finished, the parameter data
disappears, and if the function had been passed a global variable's value as input, the global variable's list wouldn't have been changed.

Incidentally, if you didn't want to change the original list, one way to make the alteredcopy above without an explicit for-loop is to use map(), which makes one look super-cool (in a CS1110 kind of way):

```
data = [True, False, True, False]
>>> def not_it(b):
...     """Returns: not b, where b is a boolean"""
...     return not b
...
>>> alteredcopy = list(map(not_it, data))
>>> alteredcopy
[False, True, False, True]
```

#pin

lecture

~ An instructor (Yiting Wang) thinks this is a good note ~

Updated Just now by Prof. Lee

followup discussions *for lingering questions and comments*