4 Lists

Python has a built-in list data structure that is easy to use and incredibly convenient. So, for instance, you can say

```
>>> y = [1, 2, 3]
>>> y[0]
1
>>> y[2]
3
>>> len(y)
3
>>> y.append(4)
>>> y
[1, 2, 3, 4]
>>> y[1] = 100
>>> y
[1, 100, 3, 4]
>>>
```

A list is written using square brackets, with entries separated by commas. You can get elements out by specifying the index of the element you want in square brackets, but *note that the indexing starts with 0*!

You can add elements using `append` (and in other ways, which we'll explore in the exercises). The syntax that is used to append an item to a list, `y.append(4)`, is a little bit different than anything we've seen before. It's an example of Python's *object-oriented programming* facilities, which we'll study in detail later. Roughly, you can think of it ask asking the object `y` for its idea of how to append an item, and then applying it to the integer 4.

You can change an element of a list by assigning to it: on the left-hand side of the assignment statement is `y[1]`. Something funny is going on here, because if it were on the right-hand side of an assignment statement, the expression `y[1]` would have the value 2. But it means something different on the left-hand side: it names a place where we can store a value (just as using the name of a variable `x` on the left-hand side is different from using it on the right-hand side). So, `y[1] = 100` changes the value of the second element of `y` to be 100.

In Python, you can also make something like a list, but called a *tuple*, using round parentheses instead of square ones: `(1, 2, 3)`. Tuples cannot have their elements changed, and also cause some syntactic confusion, so we'll mostly stick with lists.

4.1 Iteration over lists

What if you had a list of integers, and you wanted to add them up and return the sum? Here are a number of different ways of doing it.[1]

First, here is something like you might have learned to write in a Java class (actually, you would have used `for`, but Python doesn't have a `for` that works like the one in C and Java).

```
def addList1(l):
    sum = 0
    listLength = len(l)
    i = 0
    while (i < listLength):
        sum = sum + l[i]
        i = i + 1
    return sum
```

It increments the index `i` from 0 through the length of the list - 1, and adds the appropriate element of the list into the sum. This is perfectly correct, but pretty verbose and easy to get wrong.

```
def addList2(l):
    sum = 0
    for i in range(len(l)):
        sum = sum + l[i]
    return sum
```

Here's a version using Python's `for` loop. The `range` function returns a list of integers going from 0 to up to, but not including, its argument. So `range(3)` returns `(0, 1, 2)`. A loop of the form

```
for x in l:
   something
```

will be executed once for each element in the list `l`, with the variable `x` containing each successive element in `l` on each iteration. So,

```
for x in range(3):
    print x
```

will print `0 1 2`. Back to `addList2`, we see that `i` will take on values from 0 to the length of the list minus 1, and on each iteration, it will add the appropriate element from `l` into the sum. This is more compact and easier to get right than the first version, but still not the best we can do!

```
def addList3(l):
    sum = 0
    for v in l:
        sum = sum + v
    return sum
```

This one is even more direct. We don't ever really need to work with the indices. Here, the variable `v` takes on each successive value in `l`, and those values are accumulated into `sum`.

For the truly lazy, it turns out that the function we need is already built into Python. It's called `sum`:

```
def addList4(l):
    return sum(l)
```

Now, what if we wanted to increment each item in the list by 1? It might be tempting to take an approach like the one in `addList3`, because it was so nice not to have to mess around with indices. Unfortunately, if we want to actually change the elements of a list, we have to name them explicitly on the left-hand side of an assignment statement, and for that we need to index them. So, to increment every element, we need to do something like this:

```
def incrementElements(l):
    for i in range(len(l)):
        l[i] = l[i] + 1
```

We didn't return anything from this procedure. Was that a mistake? What would happen if you evaluated the following three expressions in the Python shell?

```
>>> y = [1, 4, 6]
>>> incrementElements(y)
```

```
>>> y
```

Later, when we look at higher-order functions, we'll see another way to do both `addList` and `incrementElements`, which many people find more beautiful than the methods shown here.

Footnotes:

[1] For any program you'll ever need to write, there will be a huge number of different ways of doing it. How should you choose among them? The most important thing is that the program you write be correct, and so you should choose the approach that will get you to a correct program in the shortest amount of time. That argues for writing it in the way that is cleanest, clearest, shortest. Another benefit of writing code that is clean, clear and short is that you will be better able to understand it when you come back to it in a week or a month or a year, and that other people will also be better able to understand it. Sometimes, you'll have to worry about writing a version of a program that runs very quickly, and it might turn out that in order to make that happen, you'll have to write it less cleanly or clearly or briefly. But it's important to have a version that's correct before you worry about getting one that's fast.

6.01SC Introduction to Electrical Engineering and Computer Science

Spring 2011