L4

June 12, 2017

1 Lecture 4: Data Structures and Loops

CSCI 1360E: Foundations for Informatics and Analytics

1.1 Overview and Objectives

In this lecture, we'll go over the different collections of objects that Python natively supports. In previous lectures we dealt only with single strings, ints, and floats; now we'll be able to handle arbitrarily large numbers of them, particularly in concert with loops. By the end of the lecture, you should be able to

- Describe the differences between sets, tuples, lists, and dictionaries.
- Perform basic arithmetic operations using arbitrary-length collections.
- Describe the differences between the separate kinds of loops.

1.2 Part 1: Lists

Lists are probably the most basic data structure in Python; they contain ordered elements and can be of abitrary length. Other languages may refer to this basic structure as an "array", and indeed there are similarities. But for our purposes and anytime you're coding in Python, we'll use the term "list."

Lists are the bread-and-butter data structure in Python. If in doubt, 3/4 of the time you'll be using lists.

1.2.1 (Aside)

When I say "data structures," I mean any *type* that is more sophisticated than the `int`s and `float`s we've been working with up until now. I purposefully omitted `str`s, because they are in fact a data structure unto themselves: they build on the single character, and are effective a "list of characters." In much the same way, we'll see in this lecture how to define a "list of `int`s", a "list of `float`s", and even a "list of `str`s"!

Lists in Python have a few core properties:

- **Ordered.** This means the list structure maintains an instrinic ordering of the elements held inside.
- **Mutable.** This means the structure of the list can change; elements can be added, removed, or changed in-place.

```
In [1]: x = list()
```

Here I've defined an empty list, called `x`. Like our previous variables, this has both a name (`x`) and a type (`list`). However, it doesn't have any actual value beyond that; it's just an empty list. Imagine a filing cabinet with nothing in it.

So how do we add things? Lists, as it turns out, have a few *methods* we can invoke (methods are pieces of functionality that we'll cover more when we get to functions). Here's a useful one:

```
In [2]: x.append(1)
```

The `append()` method takes whatever argument I supply to the function, and inserts it into the next available position in the list. Which, in this case, is the very first position (since the list was previously empty, and lists are ordered).

What does our list look like now?

```
In [3]: print(x)

[1]
```

It's tough to tell that there's really anything going on, but those square brackets `[` and `]` are the key: those denote a list, and anything inside those brackets is an element of the list.

Let's look at another list, a bit more interesting this time.

```
In [4]: y = list()
        y.append(1)
        y.append(2)
        y.append(3)
        print(y)

[1, 2, 3]
```

In this example, I've created a new list `y`, initially empty, and added three integer values. Notice the ordering of the elements when I print the list at the end: from left-to-right, you'll see the elements in the order that they were added.

You can put any elements you want into a list. If you wanted, you could add strings

```
In [5]: y.append("this is perfectly legal")
```

and floats

```
In [6]: y.append(4.2)
```

and *even other lists!*

```
In [7]: y.append(list())  # Inception BWAAAAAAAAAA
        print(y)

[1, 2, 3, 'this is perfectly legal', 4.2, []]
```

1.2.2 Indexing

So I have these lists and I've stored some things in them. I can print them out and see what I've stored...but so far they seem pretty unwieldy. How do I remove things? If someone asks me for whatever was added 3^{rd}, how do I give that to them without giving them the whole list?

Glad you asked! The answers to these questions involve *indexing*.

Indexing is what happens when you refer to an existing element in a list. For example, in our hybrid list `y` with lots of random stuff in it, what's the first element?

```
In [8]: first_element = y[1]
        print(first_element)
        print(y)

2
[1, 2, 3, 'this is perfectly legal', 4.2, []]
```

In this code example, I've used the number 1 as an *index* to `y`. In doing so, I took out the value at index 1 and put it into a variable named `first_element`. I then printed it, as well as the list `y`, and voi--

--wait, "2" is the *second* element. o_O

Python and its spiritual progenitors C and C++ are known as *zero-indexed* languages. This means when you're dealing with lists or arrays, the index of the first element is always **0**.

This stands in contrast with languages such as Julia and Matlab, where the index of the first element of a list or array is, indeed, 1. Preference for one or the other tends to covary with whatever you were first taught, though in scientific circles it's generally preferred that languages be 0-indexed[citation needed].

So what is in the 0 index of our list?

```
In [9]: print(y[0])
        print(y)

1
[1, 2, 3, 'this is perfectly legal', 4.2, []]
```

Notice the first thing printed is "1". The next line is the entire list (the output of the second `print` statement).

This little caveat is usually the main culprit of errors for new programmers. Give yourself some time to get used to Python's 0-indexed lists. You'll see what I mean when we get to loops.

In addition to elements 0 and 1, we can also directly index elements at the *end* of the list.

```
In [10]: print(y[-1])
         print(y)

[]
[1, 2, 3, 'this is perfectly legal', 4.2, []]
```

Yep, there's our inception-list, the last element of `y`.

You can think of this indexing strategy as "wrapping around" the list to the end of it. Similarly, you can also negate other numbers to access the second-to-last element, third-to-last element...

```
In [11]: print(y[-2])
         print(y[-3])
         print(y)
```

```
4.2
this is perfectly legal
[1, 2, 3, 'this is perfectly legal', 4.2, []]
```

Using more indexing voodoo, you can also index *slices* of lists. Let's say we want to create a new list that consists of the integer elements of `y`, which are the first three. We could pull them out one by one, or use slicing:

```
In [12]: int_elements = y[0:3]  # Slicing!
         print(y)
         print(int_elements)
```

```
[1, 2, 3, 'this is perfectly legal', 4.2, []]
[1, 2, 3]
```

That `y[0:3]` notation is the slicing. The first number, 0, indicates the first index of values we want to keep. The colon `:` indicates slicing, and the second number, 3, indicates the last index of values.

You could even say this out loud: "With list `y`, slice starting at index 0 to index 3." The colon is the "to".

The astute reader will notice that index 3 in `y` is actually the string!

```
In [13]: print(y[3])
```

```
this is perfectly legal
```

So, if we're slicing "from 0 to 3", why is this including index 0 but excluding index 3?

When you slice an array, the first (starting) index is *inclusive*; the second (ending) index, however, is *exclusive*. In mathematical notation, it would look something like this:

$[starting : ending)$

Therefore, the end index is one *after* the last index you want to keep.

1.2.3 One more thing about lists

You don't always have to start with empty lists. You can pre-define a full list; just use brackets!

```
In [14]: z = [42, 502.4, "some string", 0]
```

```
In [15]: print(z)
```

```
[42, 502.4, 'some string', 0]
```

1.3 Part 2: Sets and Tuples

If you understood lists, sets and tuples are easy-peasy. They're both exactly the same as lists...**except:**

Tuples: - **Immutable**. Once you construct a tuple, it cannot be changed.

Sets: - **Distinct**. Sets cannot contain two identical elements. - **Unordered**. Sets don't index the same way lists do.

Other than these two rules, pretty much anything you can do with lists can also be done with tuples and sets.

1.3.1 Tuples

Whereas we used square brackets to create a list

```
In [16]: x = [3, 64.2, "some list"]
         print(type(x))
```

```
<class 'list'>
```

we use regular parentheses to create a tuple!

```
In [17]: y = (3, 64.2, "some tuple")
         print(type(y))
```

```
<class 'tuple'>
```

With lists, if you wanted to change the item at index 2, you could go right ahead:

```
In [18]: x[2] = "a different string"
         print(x)
```

```
[3, 64.2, 'a different string']
```

Can't do that with tuples, sorry. Tuples are *immutable*, meaning you can't change them once you've built them.

```
In [48]: y[2] = "does this work?"
```

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

<ipython-input-48-1347e878f386> in <module>()
----> 1 y[2] = "does this work?"


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

Like `list`, there is a method for building an empty `tuple`. Any guesses?

```
In [20]: z = tuple()
```

And like lists, you have (almost) all of the other methods at your disposal, such as slicing and `len`:

```
In [21]: print(y[0:2])
         print(len(y))
```

```
(3, 64.2)
3
```

1.3.2 Sets

Sets are interesting buggers, in that they only allow you to store a particular element *once*.

```
In [22]: x = list()
         x.append(1)
         x.append(2)
         x.append(2)  # Add the same thing twice.

         s = set()
         s.add(1)
         s.add(2)
         s.add(2)  # Add the same thing twice...again.
```

```
In [23]: print(x)
```

```
[1, 2, 2]
```

```
In [24]: print(s)
```

```
{1, 2}
```

There are certain situations where this can be very useful. It should be noted that sets can actually be built from lists, so you can build a list and then turn it into a set:

```
In [25]: x = [1, 2, 3, 3]
         s = set(x)  # Take the list x and "cast" it as a set.
         print(s)
```

```
{1, 2, 3}
```

Sets also don't index the same way lists and tuples do:

```
In [47]: print(s[0])
```

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

<ipython-input-47-919b5fe16240> in <module>()
----> 1 print(s[0])


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

If you want to add elements to a set, you can use the `add` method.

If you want to remove elements from a set, you can use the `discard` or `remove` methods.

But you *can't index or slice a set.* This is because sets do not preserve any notion of ordering.

Think of them like a "bag" of elements: you can add or remove things from the bag, but you can't really say "this thing comes before or after this other thing."

1.3.3 So why is a `set` useful?

It's useful for checking if you've seen a particular *kind* of thing at least once.

```
In [27]: s = set([1, 3, 6, 2, 5, 8, 8, 3, 2, 3, 10])

         print(10 in s)  # Basically asking: is 10 in our set?

         print(11 in s)
```

```
True
False
```

Aside: the `in` keyword is wonderful. It's a great way of testing if a variable is in your collection (list, set, or tuple) without having to loop over the entire collection looking for it yourself (which we'll see how to do in a bit).

1.4 Part 3: Dictionaries

Dictionaries deserve a section all to themselves.

Are you familiar with key-value stores? Associative arrays? Hash maps?

The basic idea of all these data type abstractions is to map a *key* to a *value*, in such a way that if you have a certain key, you always get back the value associated with that key.

You can also think of dictionaries as unordered lists with more interesting indices.

A few important points on dictionaries before we get into examples:

- **Mutable**. Dictionaries can be changed and updated.
- **Unordered**. Elements in dictionaries have no concept of ordering.

- **Keys are distinct**. The *keys* of dictionaries are unique; no key is ever copied. The *values*, however, can be copied as many times as you want.

Dictionaries are created using the `dict()` method, or using curly braces:

```
In [28]: d = dict()
         # Or...
         d = {}
```

New elements can be added to the dictionary in much the same way as lists:

```
In [29]: d["some_key"] = 14.3
```

Yes, you use strings as keys! In this way, you can treat dictionaries as "look up" tables--maybe you're storing information on people in a beta testing program. You can store their information by name:

```
In [30]: d["shannon_quinn"] = ["some", "personal", "information"]
         print(d)

{'some_key': 14.3, 'shannon_quinn': ['some', 'personal', 'information']}
```

Since dictionaries do not maintain any kind of ordering of elements, using integers as indices won't give us anything useful. However, dictionaries do have a `keys()` method that gives us a **list** of all the keys in the dictionary:

```
In [31]: print(d.keys())

dict_keys(['some_key', 'shannon_quinn'])
```

and a `values()` method for (you guessed it) the values in the dictionary:

```
In [32]: print(d.values())

dict_values([14.3, ['some', 'personal', 'information']])
```

To further induce Inception-style headaches, dictionaries also have a `items()` method that returns a **list** of **tuples** where each tuple is a key-value pair in the dictionary!

```
In [33]: print(d.items())

dict_items([('some_key', 14.3), ('shannon_quinn', ['some', 'personal', 'information'])])
```

(it's basically the entire dictionary, but this method is useful for looping)
Speaking of looping...

1.5 Part 4: Loops

Looping is an absolutely essential component in general programming and data science. Whether you're designing a web app or a machine learning model, most of the time you have no idea how much data you're going to be working with. 100? 1000? 952,458,482,789?

In all cases, you're going to want to run the same code on each one. This is where computers excel: repetitive tasks on large amounts of information. There's a way of doing this in programming languages: *loops*.

Let's define for ourselves the following list:

```
In [34]: ages = [21, 22, 19, 19, 22, 21, 22, 31]
```

This is a list containing the ages of some group of students, and we want to compute the average. How do we compute averages?

We know an average is some *total quantity* divided by *number of elements*. Well, the latter is easy enough to compute:

```
In [35]: number_of_elements = len(ages)
         print(number_of_elements)
```

```
8
```

The total quantity is a bit trickier. You could certainly sum them all manually--

```
In [36]: age_sum = ages[0] + ages[1] + ages[2] # + ... and so on
```

But that's the problem mentioned earlier: how do you even know how many elements are in your list when your program runs on a web server? If it only had 3 elements the above code would work fine, but the moment your list has 2 items or 4 items, your code would need to be rewritten.

1.5.1 `for` loops

One type of loop, the "bread-and-butter" loop, is the `for` loop. There are three basic ingredients:

- some collection of "things" to iterate over
- a placeholder for the current "thing" (because the loop only *works* on 1 thing at a time)
- a chunk of code describing what to do with the current "thing"

Remember these three points *every. time.* you start writing a loop. They will help guide your intuition for how to code it up.

Let's start simple: looping through a list, printing out each item one at a time.

```
In [37]: the_list = [2, 5, 7, 9]
         for N in the_list:       # Header
             print(N, end = " ")  # Body
```

```
2 5 7 9
```

There are two main parts to the loop: the **header** and the **body**.

- The **header** contains 1) the collection we're iterating over (e.g., the list), and 2) the "placeholder" we're using to hold the current value (e.g., `N`).

- The **body** is the chunk of code under the header (*indented!*) that executes on each element of the collection, one at a time.

Go over these last two slides until you understand what's happening! These are absolutely critical to creating arbitrary loops.

Back, then, to computing an average:

```
In [38]: age_sum = 0  # Setting up a variable to store the sum of all the elements in our list.
         ages = [21, 22, 19, 19, 22, 21, 22, 31]  # Here's our list.

In [39]: # Summing up everything in a list is actually pretty easy, code-wise:
         for age in ages:  # For each individual element (stored as "age") in the list "ages"...
             age_sum += age  # Increment our variable "age_sum" by the quantity stored in "age"

In [40]: avg = age_sum / number_of_elements  # Compute the average using the formula we know and
         print("Average age: {:.2f}".format(avg))

Average age: 22.12
```

If what's happening isn't clear, go back over it. If it still isn't clear, go over it again. If it still isn't clear, please ask!

You can loop through sets and tuples the same way.

```
In [41]: s = set([1, 1, 2, 3, 5])
         for item in s:
             print(item, end = " ")

1 2 3 5

In [42]: t = tuple([1, 1, 2, 3, 5])
         for item in t:
             print(item, end = " ")

1 1 2 3 5
```

Important: INDENTATION MATTERS.

You'll notice in these loops that the *loop body* is distinctly indented relative to the *loop header*. This is intentional and is indeed how it works! If you fail to indent the body of the loop, Python will complain:

```
In [46]: some_list = [3.14159, "random stuff", 4200]
         for item in some_list:
         print(item)
```

```
  File "<ipython-input-46-e6ab552bd1f0>", line 3
    print(item)
        ^
IndentationError: expected an indented block
```

With loops, whitespace in Python *really* starts to matter. If you want many things to happen inside of a loop (and we'll start writing longer loops!), you'll need to indent every line!

1.5.2 `while` loops

"While" loops go back yet again to the concept of boolean logic we introduced in an earlier lecture: loop until some condition is reached.

The structure here is a little different than `for` loops. Instead of explicitly looping over an iterator, you'll set some condition that evaluates to either `True` or `False`; as long as the condition is `True`, Python executes another loop.

```
In [44]: x = 10

         while x < 15:
             print(x, end = " ")
             x += 1  # TAKE NOTE OF THIS LINE.
```

```
10 11 12 13 14
```

`x < 15` is a boolean statement: it is either `True` or `False`, depending on the value of `x`. Initially, this number is 10, which is certainly `< 15`, so the loop executes. 10 is printed, `x` is incremented, and the condition is checked again.

A potential downside of `while` loops: **forgetting to update the condition inside the loop.**

It's easy to take for granted; `for` loops implicitly handle updating the loop variable for us.

```
In [45]: for i in range(10, 15):
             print(i, end = " ")
             # No update needed!
```

```
10 11 12 13 14
```

Use `for` loops frequently enough, and when you occasionally use a `while` loop, you'll forget you need to update the loop condition.

1.6 Review Questions

Some questions to discuss and consider:

1: Without knowing the length of the list `some_list`, how would you slice it so only the first and last elements are removed?

2: Provide an example use-case where the properties of sets and tuples would come in handy over lists.

3: Would it be possible to convert a list to a dictionary? How? Would anything change?

4: Create a dictionary of lists, where the lists contain numbers. For each key-value pair, compute an average.

5: Using the awful matrix construct of a "list of lists," show how you could write loops that double the value of each element of the matrix.

6: `for` and `while` loops may have different syntax and different use cases, but you can often translate the same task between the two types of loops. Show how you could use a `while` loop to iterate through a list of numbers from `range()`.

1.7 Course Administrivia

- **How is Assignment 0 going?** Post any questions in Slack to `#questions`!
- **Assignment 1 will be out on tomorrow!**

1.8 Additional Resources

1. Matthes, Eric. *Python Crash Course*. 2016. ISBN-13: 978-1593276034
2. Grus, Joel. *Data Science from Scratch*. 2015. ISBN-13: 978-1491901427