Lecture4

August 23, 2018

1 Lecture 4: Data Structures and Loops

CBIO (CSCI) 4835/6835: Introduction to Computational Biology

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 lone strings, ints, and floats; now we'll be able to handle arbitrarily large numbers of them, and how you can do repeated operations on these collections 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 hold any number of elements. 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 `ints` and `floats` we've been working with up until now.

I purposefully omitted `strs`, because they are in fact a data structure unto themselves: they are effectively a "list of characters." In much the same way, we'll see in this lecture how to define a "list of `ints`", a "list of `floats`", and even a "list of `strs`"!

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.

These properties are important; changing one of them will result in a different data structure that we'll see soon!

```
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. If I `print` the list:

```
In [2]: print(x)

[]
```

That's the symbol for "empty".

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 [3]: 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 [4]: 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 [5]: y = list()  # New list: y
        print(y)

[]
```

```
In [6]: y.append(1)
        print(y)

[1]
```

```
In [7]: y.append(2)
        print(y)
```

```
[1, 2]
```

```
In [8]: 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 [9]: y.append("this is perfectly legal")
        print(y)
```

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

and floats

```
In [10]: y.append(4.2)
         print(y)
```

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

and *even other lists!*

```
In [11]: 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 a list is how you retrieve specific elements in a list.

For example, in our hybrid list `y` with lots of random stuff in it, what's the first element?

```
In [12]: 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**.

Let me repeat that: the integer index of the first element is **ZERO**, *not* one.

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 [13]: print(y[0])

1


In [14]: print(y)

[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 [15]: print(y[-1])

[]


In [16]: 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 [17]: print(y[-2]) # second-to-last
```

```
4.2
```

```
In [18]: print(y[-3]) # third-to-last
```

```
this is perfectly legal
```

```
In [19]: print(y) # entire list
```

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

1.2.3 Slicing

Slicing is a type of indexing, where instead of referencing a single position, you *slice* a range of the list.

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 [20]: first_three_elements = y[0:3]  # Slicing!
         print(first_three_elements)
```

```
[1, 2, 3]
```

```
In [21]: print(y)
```

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

That `y[0:3]` notation is the slicing. - The first number, 0, indicates the first index of values we want to keep (the *start* index). - The colon `:` indicates slicing (that we want a range of values from the list). - The second number, 3, indicates the last index of values (the *end* index).

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 [22]: 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.

1.2.4 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 [23]: z = [42, 502.4, "some string", 0]  # This is a list with elements, stored in z

In [24]: 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 [25]: x = [3, 64.2, "some list"]
         print(type(x))

<class 'list'>
```

we use regular parentheses to create a tuple!

```
In [26]: y = (3, 64.2, "some tuple")  # only difference is parentheses instead of brackets
         print(type(y))

<class 'tuple'>
```

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

```
In [27]: print(x)
         x[2] = "a different string"
         print(x)

[3, 64.2, 'some list']
[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 [28]: y[2] = "does this work?"
```

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

<ipython-input-28-a7ecede508b3> 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 [29]: z = tuple()
```

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

```
In [30]: print(y[0:2])
```

```
(3, 64.2)
```

```
In [31]: print(len(y))
```

```
3
```

1.3.2 Sets

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

```
In [32]: x = list()
         x.append(1)
         x.append(2)
         x.append(2)  # Add the same thing twice.

         print(x)
```

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

```
In [33]: s = set()
         s.add(1)
         s.add(2)
         s.add(2)  # Add the same thing twice...again.

         print(s)
```

```
{1, 2}
```

Note how, in the second code block, the set `s` only contains two unique numbers. It discarded the copy it received.

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 [34]: x = [1, 2, 3, 3]
         print(x)

[1, 2, 3, 3]


In [35]: s = set(x)  # Take the list x and "cast" it as a set.
         print(s)

{1, 2, 3}
```

This "enforces" the rules for a `set` on the `list`–in this case, removing the duplicate elements.

Does this remind you of what happens when you cast a `float` to an `int`?

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

```
In [36]: print(s[0])


         ---------------------------------------------------------------------------

         TypeError                                 Traceback (most recent call last)

         <ipython-input-36-5605e14cbe49> 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 [37]: s = set([1, 3, 6, 2, 5, 8, 8, 3, 2, 3, 10])

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

```
True
```

```
In [38]: print(11 in s)  # is 11 in s?
```

```
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. With lists and tuples, the indices are integers; with dictionaries, they can be *almost anything*.

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

```
In [40]: print(d)
```

```
{}
```

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

```
In [41]: d["some_key"] = 14.3
         print(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 [42]: d["shannon_quinn"] = ["some", "personal", "information"] # add a list!
         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 [43]: print(d.keys())
```

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

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

```
In [44]: print(d.values())
```

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

Stop for a moment on this slide and convince yourself of what's happening.

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 [45]: 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)

We'll be using dictionaries **a lot** in this course, so go ahead and practice with them!

1.4.1 Membership Testing (in dictionaries... or any other collection)

`in` and `not in` can be used to see if an object or particular value is in a collection.

```
In [46]: # Create our collection.
         collection = [1,2,3]
```

```
In [47]: # You can read this as a question: "Is the number 1 in the variable 'collection'?"
         1 in collection
```

```
Out[47]: True
```

```
In [48]: # Is the number 345 in the variable 'collection'?
         345 in collection

Out[48]: False

In [49]: # Is the STRING "1" NOT IN the variable 'collection'?
         "1" not in collection

Out[49]: True

In [50]: # Is the string "good" in the string "goodness"?

         "good" in "goodness"

         # (yep, strings are considered collections!)

Out[50]: True
```

1.5 PAUSE: Questions on data structures?

- Lists
- Sets
- Tuples
- Dictionaries
- Membership testing

1.6 Part 4: Loops

Looping is an absolutely essential component in general programming and data science. *Loops* allow for concisely-coded, repeated operations.

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 data points? 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 [51]: 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 [52]: number_of_elements = len(ages)  # Length of the list is our denominator
         print(number_of_elements)

8
```

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

```
In [53]: 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.

1.6.1 `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 [54]: the_list = [2, 5, 7, 9]  # Here's our list of things.
         for current_thing in the_list:  # Loop header
             print(current_thing)  # Loop 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., `current_thing`). - The **body** is the chunk of code under the header (*indented!*) that executes on each element of the collection, one at a time.

Back, then, to computing an average:

```
In [55]: age_sum = 0  # Setting up a variable for the numerator.
         ages = [21, 22, 19, 19, 22, 21, 22, 31]  # Here's our list.
         number_of_elements = len(ages)  # Value for the denominator.

In [56]: # Summing up everything in a list is actually pretty easy, code-wise:
         for age in ages:  # For a single element (stored in "age") in the list "ages"
             age_sum += age  # Increment our variable "age_sum" by the value in "age"

In [57]: avg = age_sum / number_of_elements  # Compute the average!
         print("Average age: {:.2f}".format(avg))

Average age: 22.12
```

You can loop through sets and tuples the same way.

```
In [58]: s = set([1, 1, 2, 3, 5])
         for item in s:
             print(item, end = " ")

1 2 3 5

In [59]: 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 [60]: some_list = [3.14159, "random stuff", 4200]
         for item in some_list:
         print(item)


          File "<ipython-input-60-354ceb852b30>", 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!

1.6.2 `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 a list of things, you'll set some condition that evaluates to either `True` or `False`; as long as the condition is `True`, Python executes another loop.

```
In [61]: 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 once, 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 [62]: for i in range(10, 15):
             print(i)
             # 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.

1.6.3 `break` and `continue`

`break` will exit out of the entire loop, while `continue` will skip to the next iteration.

```
In [63]: i = 0
```

```
In [64]: while True:
             i += 1
             if i < 6:
                 continue
             print(i)
             if i > 4:
                 break
```

```
6
```

1.7 Part 5: Conditionals

Up until now, we've been somewhat hobbled in our coding prowess; we've lacked the tools to make different decisions depending on the values our variables take.

For example: how do you find the maximum value in a list of numbers?

```
In [65]: x = [51, 65, 56, 19, 11, 49, 81, 59, 45, 73]
```

Enter `if / elif / else` statements! (otherwise known as "conditionals")

We can use the keyword `if`, followed by a statement that evaluates to either `True` or `False`, to determine whether or not to execute the code. An example:

```
In [66]: x = 5
         if x < 5:
             print("How did this happen?!")  # Spoiler alert: this won't happen.

         if x == 5:
             print("Working as intended.")
```

```
Working as intended.
```

In conjunction with `if`, we also have an `else` clause that we can use to execute whenever the `if` statement doesn't:

```
In [67]: x = 5
         if x < 5:
             print("How did this happen?!")  # Spoiler alert: this won't happen.
         else:
             print("Correct.")
```

```
Correct.
```

This is great! We can finally finish computing the maximum element of a list!

```
In [68]: x = [51, 65, 56, 19, 11, 49, 81, 59, 45, 73]
         max_val = 0  # Set a "container" for our maximum.
         for element in x:  # Loop through the list.
             if max_val < element:
                 max_val = element

         print("The maximum element is: {}".format(max_val))
```

```
The maximum element is: 81
```

Let's pause here and walk through that code.

`x = [51, 65, 56, 19, 11, 49, 81, 59, 45, 73]` - This code defines the list we want to look at.

`max_val = 0` - And this is a placeholder for the eventual maximum value (we need a placeholder like this because of how loops work: in a loop, we can only look at **one number at a time**, so on each step of the loop, we update our current "guess" of the maximum).

`for element in x:` - A standard `for` loop header: we're iterating over the list `x`, with the value from the list in the current loop stored in the variable `element`.

`if max_val < element:` - The first line of the loop body is an `if` statement. This statement asks: is the value in our current `max_val` placeholder smaller than the element of the list stored in `element`?

`max_val = element` - If the answer to that `if` statement is `True`, then this line executes: it sets our placeholder equal to the current list element.

Another way to examine this loop would be to walk through the list `x` itself.

- On the first iteration, `max_val = 0` and `element = 51`. The conditional is: `if 0 < 51`. This is `True`! So we set `max_val = 51`. That ends the first iteration.

- On the second iteration, `max_val = 51` and `element = 65`. The conditional is: `if 51 < 65`. This is `True`! So we set `max_val = 65`. That ends the second iteration.

- On the third iteration, `max_val = 65` and `element = 56`. The conditional is: `if 65 < 56`. This is `False`, so the statement underneath doesn't execute, and nothing else happens in the third iteration.

This process continues for every value in the list. Once the final element of `x` is evaluated this way, the loop will end, and whatever the value of the variable `max_val` is will be the maximum value of the list.

Let's look at slightly more complicated but utterly classic example: assigning letter grades from numerical grades.

```
In [69]: student_grades = {
             'Jen': 82,
             'Shannon': 75,
             'Natasha': 94,
             'Benjamin': 48,
         }
```

We know the 90-100 range is an "A", 80-89 is a "B", and so on. How would we build a conditional to assign letter grades?

The third and final component of conditionals is the `elif` statement (short for "else if").

`elif` allows us to evaluate as many options as we'd like, all within *the same conditional context* (this is important). So for our grading example, it might look like this:

```
In [70]: letter = ''

         # Remember the .items() function for dictionaries? Here it is!
         # This loop goes over the dictionary, pulling out each key/value
         # pair, one at a time, storing them in student/grade.
         for student, grade in student_grades.items():
             if grade >= 90:
                 letter = "A"
             elif grade >= 80:
                 letter = "B"
             elif grade >= 70:
                 letter = "C"
             elif grade >= 60:
                 letter = "D"
             else:
                 letter = "F"

             print(student, letter)
```

```
Jen B
Shannon C
Natasha A
Benjamin F
```

Ok, that's neat. But there's still one more edge case: what happens if we want to enforce multiple conditions *simultaneously*?

To illustrate, let's go back to our example of finding the maximum value in a list, and this time, let's try to find the *second*-largest value in the list. For simplicity, let's say we've already found the largest value.

```
In [71]: x = [51, 65, 56, 19, 11, 49, 81, 59, 45, 73]
         max_val = 81  # We've already found it!
         second_largest = 0
```

Here's the rub: we now have *two* constraints to enforce–the second largest value needs to be larger than pretty much everything in the list, but also needs to be *smaller* than the maximum value. Not something we can encode using `if` / `elif` / `else`.

Instead, we'll use two more keywords integral to conditionals: `and` and `or`.

You've already seen `and`: this is used to join multiple boolean statements together in such a way that, if one of the statements is `False`, the *entire* statement is `False`.

```
In [72]: # An example where only 1 thing is False -- makes the whole thing False.
         True and True and True and True and True and True and False

Out[72]: False
```

One `False` ruins the whole thing.

However, we haven't encountered `or` before. How do you think it works?

Here's are two examples:

```
In [73]: True or True or True or True or True or True or False

Out[73]: True

In [74]: False or False or False or False or False or False or True

Out[74]: True
```

Figured it out?

Where `and` needs *every* statement it joins to be `True` in order for the whole statement to be `True`, only one statement among those joined by `or` needs to be `True` for everything to be `True`.

How about this example?

```
In [75]: (True and False) or (True or False)

Out[75]: True
```

(Order of operations works the same way!)

Getting back to conditionals, then: we can use this *boolean logic* to enforce multiple constraints simultaneously.

```
In [76]: for element in x:
             if second_largest < element and element < max_val:
                 second_largest = element

         print("The second-largest element is: {}".format(second_largest))
```

```
The second-largest element is: 73
```

Let's step through the code.

```
for element in x:
    if second_largest < element and element < max_val:
        second_largest = element
```

- The first condition, `second_largest < element`, is the same as before: if our current estimate of the second largest element is smaller than the latest element we're looking at, it's definitely a candidate for second-largest.
- The second condition, `element < max_val`, is what ensures we don't just pick the largest value again. This enforces the constraint that the current element we're looking at is also *less than* the maximum value.
- The `and` keyword glues these two conditions together: it *requires that they BOTH* be `True` before the code inside the statement is allowed to execute.

It would be easy to replicate this with "nested" conditionals:

```
In [77]: second_largest = 0
         for element in x:
             if second_largest < element:
                 if element < max_val:
                     second_largest = element

         print("The second-largest element is: {}".format(second_largest))
```

```
The second-largest element is: 73
```

...but your code starts getting a little unwieldy with so many indentations.

You can glue as many comparisons as you want together with `and`; the whole statement will only be `True` if *every single condition* evaluates to True. This is what `and` means: *everything* must be True.

The other side of this coin is `or`. Like `and`, you can use it to glue together multiple constraints. Unlike `and`, the whole statement will evaluate to True *as long as at least ONE condition is True*. This is far less stringent than `and`, where *ALL* conditions had to be True.

```
In [78]: numbers = [1, 2, 5, 6, 7, 9, 10]
         for num in numbers:
             if num == 2 or num == 4 or num == 6 or num == 8 or num == 10:
                 print("{} is an even number.".format(num))

2 is an even number.
6 is an even number.
10 is an even number.
```

In this contrived example, I've glued together a bunch of constraints. Obviously, these constraints are mutually exclusive; a number can't be equal to both 2 and 4 at the same time, so `num == 2 and num == 4` would never evaluate to True. However, using `or`, only one of them needs to be True for the statement underneath to execute.

There's a little bit of intuition to it.

- "I want this AND this" has the implication of both at once.
- "I want this OR this" sounds more like either one would be adequate.

1.8 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: Write a `while` loop that never terminates (ends).

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.

7: Go back to the `if` / `elif` / `else` example about student grades. Let's assume, instead of `elif` for the different conditions, you used a bunch of `if` statements, e.g. `if grade >= 90`, `if grade >= 80`, `if grade >= 70`, and so on; effectively, you didn't use `elif` at all, but just used `if`. What would the final output be in this case?

8: There's a whole field of "Boolean Algebra" that is not to different from the "regular" algebra you're familiar with. In this sense, rather than floating-point numbers, variables are either `True` or `False`, but everything still works pretty much the same. Take the following equation: `a(a + b)`. Let's say `a = True` and `b = False`. If multiplication is the `and` operator, and addition is the `or` operator, what's the final result?

9: How would you write the following constraint as a Python conditional: $10 < x \leq 20$, and x is even.

1.9 Course Administrivia

- **How is Assignment 1 going?** Post any questions in Slack to `#questions`! It's due next Thursday by 11:59pm!

- **We will have a guest lecturer next Tuesday!** It would be a good idea to go over the command-line lectures. I will be back for next Thursday's lecture.

- **Since I won't be available next Tuesday, I'll have office hours right after lecture next Thursday.** Same time as the normal Tuesday slot (11:30 - 1pm), in case you had any last-minute questions on Assignment 1.

1.10 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