

L5

June 13, 2018

1 Lecture 5: Advanced Data Structures

CSCI 1360E: Foundations for Informatics and Analytics

1.1 Overview and Objectives

We've covered list, tuples, sets, and dictionaries. These are the foundational data structures in Python. In this lecture, we'll go over some more advanced topics that are related to these datasets. By the end of this lecture, you should be able to

- Define and use different iterators in loops, such as `range()`, `zip()`, and `enumerate()`
- Use variable unpacking to quickly and elegantly pull data out of lists
- Compare and contrast generators and comprehensions, and how to construct them
- Explain the benefits of generators, especially in the case of huge datasets

1.2 Part 1: Iterators

The unifying theme with all these collections we've been discussing (lists, tuples, sets) in the context of looping, is that they're all examples of *iterators*.

Just think of *Iterators* as "things I can loop over".

Apart from directly iterating over these collections as in the last lecture, the most common iterator you'll use is the `range` function.

1.2.1 `range()`

`range` is a super-handly function for building a list of sequential numbers to loop over. Here's an example:

```
In [1]: for i in range(10):  
        print(i, end = " ")
```

```
0 1 2 3 4 5 6 7 8 9
```

Using just this example, stop and think about how `range` might work.

In short, `range()` generates a list of sequential numbers for you to loop over.

- If you give `range` one number, like in our example (`range(10)`), it will give you back a list of numbers starting at 0 (inclusive) and going up to the number you provided (*exclusive*). So in our example, it gives us all the integers starting at and including 0, and up to (but *excluding*) 10.

- You can also give range *two* numbers: a starting number (inclusive), and an ending number (e.g., 10 like before).

```
In [2]: for i in range(5, 10):
        print(i, end = " ")
```

5 6 7 8 9

1.3 Part 2: List Comprehensions

Here's some good news: if we get right down to it, having done loops and lists already, there's nothing new about list comprehensions.

Here's the bad news: it's a different, and possibly less-easy-to-understand, but much more concise way of creating lists. We'll go over it bit by bit.

Let's look at an example: creating a list of squared values (numbers raised to the second power).

```
In [3]: squares = []
        for element in range(10):
            squares.append(element ** 2)
        print(squares)
```

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

Stop for a moment and convince yourself why this works.

Now, let's break it down, line by line.

```
for element in range(10):
```

- It's a standard "for" loop header.
- The thing we're iterating over is at the end: `range(10)`, or an iterator that contains numbers [0, 10) by 1s (only one number is given, so it starts at 0 inclusive and goes to 10 exclusive).
- In each loop, the current element from the iterator built by `range(10)` is stored in `element`.

So far, so good? Good, next line!

```
squares.append(element ** 2)
```

- First, we compute the square of the current element: `element ** 2`
- We directly append the result of squaring to our list, `squares`, using the `append` method we've seen before

Stop for a minute and consider how that first step could be split into two (think: create a variable, *assign* it the square of the current element, and then append that variable to the list).

Hopefully that's all fairly clear! In a **list comprehension**, we'll see these same pieces show up again, just in a slightly different order.

```
In [4]: squares = [element ** 2 for element in range(10)] # List comprehension!

        print(squares)

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

There it is: a list comprehension. Let's break it down.

- Notice, first, that the entire expression is surrounded by the square brackets [] of a list. This is for the exact reason you'd think: we're building a list!
- The "for" loop is completely intact, too; the entire header appears just as before (albeit at the end of the line).
- The biggest wrinkle is the loop body. It appears right after the opening bracket, *before* the loop header. The rationale for this is that it's easy to see from the start of the line that
 1. We're building a list (revealed by the opening square bracket), and
 2. The list is built by successfully squaring a variable `element`

Here's another example: adding 10 to every element in the `squares` list from before.

```
In [5]: new_counts = [item + 10 for item in squares]
        print(new_counts)

[10, 11, 14, 19, 26, 35, 46, 59, 74, 91]
```

- Lists are iterators by default, so the header (`for item in squares`) goes through the list `squares` one element at a time, storing each one in the variable `item`
- The loop body takes the current element in the list (`item`) and adds 10 to it

Stop here and convince yourself that this is working as expected. For fun, try to rewrite this loop in the more verbose way: by defining an empty list, writing a "regular" loop header, and then appending the elements (plus 10 each) to the list.

1.4 Part 3: Generators

Generators are cool twists on lists. They've been around since Python 2 but took on a whole new life in Python 3.

That said, if you ever get confused about generators, just think of them *as lists*. This can potentially get you in trouble with weird errors, but 90% of the time it'll work every time.

Let's start with an example using `range()`:

```
In [6]: x = range(10)
```

As we know, this will create an iterator with the numbers 0 through 9, inclusive, and assign it to the variable `x`.

```
In [7]: print(x)
range(0, 10)
```

In the same way that *lists* are iterators, so are *generators*; which is what `x` is here.

So `range()` gives us a generator! Great! ...what does that mean, exactly?

For *most* practical purposes, generators and lists are indistinguishable. However, there are some key differences to be aware of:

- **Generators are "lazy"**. This means when you call `range(10)`, not all 10 numbers are immediately computed; in fact, none of them are. They're computed on-the-fly in the loop itself! This really comes in handy if, say, you wanted to loop through 1 trillion numbers, or call `range(1000000000000)`. With vanilla lists, this would immediately create 1 trillion numbers in memory and store them, taking up a *whole lot* of space. With generators, only 1 number is ever computed at a given loop iteration. Huge memory savings!

This "laziness" means you cannot directly *index* a generator, as you would a list, since the numbers are *generated* on-the-fly during the loop.

The other point of interest with generators:

- **Generators only work *once***. This is where you can get into trouble. Let's say you're trying to identify the two largest numbers in a generator of numbers. You'd loop through once and identify the largest number, then use that as a point of comparison to loop through *again* to find the second-largest number (you could do it with just one loop, but for the sake of discussion let's assume you did it this way). With a list, this would work just fine. Not with a generator, though. You'd need to explicitly recreate the generator.

How do we build generators? Aside from `range()`, that is.

Remember list comprehensions? Just replace the brackets of a list comprehension `[]` with parentheses `()`.

```
In [8]: x = [i for i in range(10)] # Brackets -> list
print(x)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [9]: x = (i for i in range(10)) # Parentheses -> generator
print(x)
```

```
<generator object <genexpr> at 0x10976b410>
```

In sum, use **lists** if:

- you're working with a relatively small amount of elements
- you want to add to / edit / remove from the elements
- you need direct access to arbitrary elements, e.g. `some_list[431]`

On the other hand, use **generators** if:

- you're working with a giant collection of elements
- you'll only loop through the elements once or twice
- when looping through elements, you're fine going in sequential order

1.5 Part 4: Other looping mechanisms

There are a few other advanced looping mechanisms in Python that are a little complex, but can make your life a lot easier when used correctly (especially if you're a convert from something like C++ or Java).

1.5.1 Variable unpacking

This isn't a looping mechanism *per se*, but it is incredibly useful and is often used in the context of looping.

Imagine you have a tuple of a handful of items; for the sake of example, we'll say this tuple stores three elements: first name, last name, and favorite programming language.

The tuple might look like this:

```
In [10]: t = ("shannon", "quinn", "python")
```

Now I want to pull the elements out of the tuple and work with them independently, one at a time. You already know how to do this:

```
In [11]: first_name = t[0]
         last_name = t[1]
         language = t[2]
```

```
print(language)
```

python

...but using three lines of code, one per variable, to extract all the elements from the tuple into their own variables, is kind of clunky.

Luckily, there's a method called "variable unpacking" that allows us to compress those three lines down to one:

```
In [12]: fname, lname, language = t # THREE variables undergoing assignment!
         print(fname)
```

shannon

This does *exactly* the same thing as before. By presenting three variables on the left hand side, we're telling Python to pull out elements of the tuple at positions 0, 1, and 2.

(variable unpacking is always assumed to start at position 0 of the structure on the right hand side)

We'll see more examples of this in practice using the looping tools later in the lecture.

1.5.2 zip()

zip() is a small method that packs a big punch. It "zips" multiple lists together into something of one big mega-list for the sole purpose of being able to iterate through them all simultaneously.

Directly as a result, variable unpacking can play a huge role in using this function! So if you didn't quite follow variable unpacking, please ask!

Here's an example: first names, last names, and favorite programming languages.

```
In [13]: # Three lists, each with four elements.

# First element of "first_names" corresponds to
# the first element of "last_names" and the first
# element of "fave_langs".

first_names = ['Shannon', 'N.K.', 'Natasha', 'Benjamin']
last_names = ['Quinn', 'Jemisin', 'Romanov', 'Button']
fave_langs = ['Python', 'Java', 'Assembly', 'Go']
```

I want to loop through these three lists simultaneously, so I can print out the person's first name, last name, and their favorite language on the same line. Since I know they're the same length, I can zip them together and, combined with a neat use of variable unpacking, do all of this in two lines:

```
In [14]: zipped_lists = zip(first_names, last_names, fave_langs)
        for fname, lname, lang in zipped_lists:
            print(fname, lname, lang)
```

```
Shannon Quinn Python
N.K. Jemisin Java
Natasha Romanov Assembly
Benjamin Button Go
```

```
In [15]: zipped_lists = zip(first_names, last_names, fave_langs)
        for fname, lname, lang in zipped_lists:
            print(fname, lname, lang)
```

```
Shannon Quinn Python
N.K. Jemisin Java
Natasha Romanov Assembly
Benjamin Button Go
```

There's a lot happening here, so take it in chunks:

- `zip(first_names, last_names, fave_langs)`: This zips together the three lists, so that the elements at position 0 all line up, then the elements at position 1, then position 2, and so on.
- Each iteration of the loop handles one of those zipped positions.
- Since we know one of those zipped positions contains one element from each of the three lists (and therefore three total elements), we can use variable unpacking to extract each one of the individual elements into individual variables.

This is tricky! It's totally ok if you don't fully absorb it just yet. Open up a blank notebook on JupyterHub or fire up this lecture in Binder and experiment with these examples!

1.5.3 enumerate()

Of course, there are always those situations where it's really, really nice to have an index variable in the loop. Let's take a look at that previous example:

```
In [16]: zipped_lists = zip(first_names, last_names, fave_langs)
        for fname, lname, lang in zipped_lists:
            print(fname, lname, lang)
```

```
Shannon Quinn Python
N.K. Jemisin Java
Natasha Romanov Assembly
Benjamin Button Go
```

This is great if all I want to do is loop through the lists simultaneously. But what if the *ordering* of the elements matters? For example, I want to prefix each sentence with the line number. How can I track what index I'm on in a loop if I don't use `range()`?

`enumerate()` handles this. By wrapping the object we loop over inside `enumerate()`, on each loop iteration we not only get the next object of interest, but also the index of that object. To wit:

```
In [17]: x = ['a', 'list', 'of', 'strings']
        for index, element in enumerate(x):
            print(index, element) # prints the number first, list element second
```

```
0 a
1 list
2 of
3 strings
```

This comes in handy anytime you need to loop through a list or generator, but also need to know what index you're on.

And note again: we're using variable unpacking in the loop header. `enumerate` essentially performs an "invisible" `zip()` on the iterator you supply, and zips it up with numbers, one per element of the iterator.

1.5.4 break and continue

These are two commands that give you much greater control over loop behavior, beyond just what you specify in the header.

- With `for` loops, you specify how many times to run the loop.
- With `while` loops, you iterate until some condition is met.

For the vast majority of cases, this works well. But sometimes you need just a *little* more control for extenuating circumstances.

There are some instances where, barring some external intervention, you really do want to just loop forever:

```
In [ ]: while True:
        # "True" can't ever be "False", so this is quite literally an infinite loop!
```

How do you get out of this infinite loop? With a break statement.

```
In [25]: while True:
        print("Help! I'm stuck in an infinite loop!")
        break

        print("Whew, now I'm outside the loop!")
```

```
Help! I'm stuck in an infinite loop!
Whew, now I'm outside the loop!
```

Just break. That will snap whatever loop you're currently in and immediately dump you out just after it.

Same thing with for loops:

```
In [23]: for i in range(100000): # Loop 100,000 times!
        print(i, end = " ")
        if i == 5:
            break
        print(i)
```

```
0 1 2 3 4 5 5
```

Don't worry about that weird if there--we'll cover that soon enough--but briefly, it tests if our loop variable `i` has the value 5. If so, then that `break` underneath is executed.

That's why, even though `range` clearly says it's supposed to loop until 100,000, it breaks after only 5.

Similar to `break` is `continue`, though you use this when you essentially want to "skip" certain iterations.

`continue` will also halt the current iteration, but instead of ending the loop entirely, it basically skips you on to the *next* iteration of the loop without executing any code that may be below it.

```
In [24]: for i in range(100):
        continue
        print("This will never be printed, because 'continue' skips it.")
        print(i)
```

```
99
```

Notice how the `print` statement inside the loop is never executed, but our loop counter `i` is still incremented through the very end.

1.6 Review Questions

Some questions to discuss and consider:

1: I want a list of all possible combinations of (x, y) values for the range $[0, 9]$. Show how this can be done with a list comprehension using *two* for-loops.

2: Without consulting Google University, consider how generators might work under the hood. How do you think they're implemented?

3: Go back to the example with three lists (first names, last names, and programming languages). Show how you could use `enumerate` to prepend a line number (the current index of the lists) to the sentence printed for each person, e.g.: "17: Joe Schmo's favorite language is C++."

4: Show how you would use variable unpacking to extract all the values from a list with 5 elements.

5: We've seen how `range` works by giving it 1 or 2 numbers, but it can also take a 3rd number: an *increment amount*. This specifies how much the value should change between each number; the default is 1 but by specifying a third number you can change that. Show how to generate a list of all the *even numbers* between 4 and 40 using `range`.

1.7 Course Administrivia

- **"Cell was changed and shouldn't have" errors on your assignments.** If you're getting these errors, it's because you put your code in the wrong cell. Make sure you edit *only* the cells that say # YOUR CODE HERE or YOUR ANSWER HERE. Also, be sure to delete or comment out the line that says `raise NotImplementedError()`.
- **If you need to re-fetch an assignment, you have to delete the entire directory of the old version.** For example, in the case where errors are found in the assignment and a new version needs to be pushed, you'll have to delete your current version *as well as the folder it's in*--so, everything--in order to re-fetch a new version.
- **How is A1 going?** A2 will be out on Thursday.
- **We are having our first biweekly live review session on Friday!** Details will be announced in the Slack chat. This is completely unstructured time--basically, office hours--where you can show up to the Google Hangout anytime during the time interval and ask questions. *Please come with questions!*

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