Lecture 12 — Controlling Loops

Restatement of the Basics

- `for` loops tend to have a fixed number of iterations computed at the start of the loop
- `while` loops tend to have an indefinite termination, determined by the conditions of the data
- Most Python `for` loops are easily rewritten as `while` loops, but not vice-versa.
    - In other programming languages, `for` and `while` are almost interchangeable, at least in principle.

Overview of Today

- Ranges and control of loop iterations
- Nested loops
- Lists of lists
- Contolling loops through `break` and `continue`

Reading: *Practical Programming*, rest of Chapter 9.

Part 1: Ranges and For Loops— A Review

- A range is a function to generate a sequence of integers:

```
for i in range(10):
    print(i)
```

  outputs the digits 0 through 9 in succession, one per line.

    - Remember that this is up to and **not including** the end value specified!
- A range is not quite a list — instead it generates values for each successive iteration of a `for` loop.
    - For now we will convert each range to a list as the basis for studying them.
- If we want to start with something other than 0, we provide two integer values

```
>>> list(range(3,8))
[3, 4, 5, 6, 7]
```

- With a third integer values we can create increments. For example,

```
>>> list(range(4,20,3))
[4, 7, 10, 13, 16, 19]
```

  starts at 4, increments by 3, stops when 20 is reached or surpassed.
- We can create backwards increments

```
>>> list(range(-1, -10, -1))
[-1, -2, -3, -4, -5, -6, -7, -8, -9]
```

Using Ranges in For Loops

- We can use the `range` to generate the sequence of loop variable values in a for loop. Our first example is printing the contents of the `planets` list

```
planets = [ 'Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter',
    'Saturn', 'Uranus', 'Neptune', 'Pluto' ]
for i in range(len(planets)):
    print(planets[i])
```

  (In this case we don't need a index variable - we can just iterate over the values in the list.)
- The variable `i` is variously known as the *index* or the *loop* index variable or the subscript.
- We will modify the loop in class to do the following:
  - Print the indices of the planets (starting at 1!)
  - Print the planets backward.
  - Print every other planet.

Loops That Do Not Iterate Over All Indices

- Sometimes the loop index should not go over the entire range of indices, and we need to think about where to stop it *early*, as the next example shows.
- Example: Returning to our example from Lecture 1, we will briefly re-examine our solution to the following problem: Given a string, how can we write a function that decides if it has three consecutive double letters?

```
def has_three_doubles(s):
    for i in range(0, len(s)-5):
        if s[i] == s[i+1] and s[i+2] == s[i+3] and s[i+4] == s[i+5]:
            return True
    return False
```

- We have to think carefully about where to start our looping and where to stop!
- Refer back to Lecture 10 for further examples

Part 1 Practice

We will only go over a few of these in class, but you should be sure you can handle all of them

1. Generate a range for the positive integers less than 100. Use this to calculate the sum of these values, with and without (i.e. use `sum`) a for loop.
2. Use a range and a for loop to print the positive, even numbers less than the integer value associated with `n`.
3. Suppose we want a list of the squares of the digits 0..9. The following does NOT work

   ```
   squares = list(range(10))
   for s in squares:
       s = s*s
   ```

   Why not? Write a different for loop that uses indexing into the `squares` list to accomplish our goal.
4. The following code for finding out if a word has two consecutive double letters is wrong. Why? When, specifically, does it fail?

   ```
   def has_two_doubles(s):
       for i in range(0, len(s)-5):
           if s[i] == s[i+1] and s[i+2] == s[i+3]:
               return True
       return False
   ```

Part 2: Nested Loops

- Some problems require *iterating* over either
  - two dimensions of data, or
  - all pairs of values from a list
- As an example, here is code to print all of the products of digits:

```
digits = range(10)
for i in digits:
    for j in digits:
        print("{} x {} = {}".format(i,j,i*j))
```

- How does this work?
  - for each value of `i` the variable in the first, or "outer", loop, Python executes the *entire* second, or `inner`, loop
  - Importantly, `i` stays fixed during the entire inner loop.
- We will look at finding the two closest points in a list.

Example: Finding the Two Closest Points

- Suppose we are given a list of point locations in two dimensions, where each point is a tuple. For example,

```
points = [ (1,5), (13.5, 9), (10, 5), (8, 2), (16,3) ]
```

- Our problem is to find the two points that are closest to each other.
  - We started working on a slightly simpler version of this problem at the end of Lecture 10.
- The natural idea is to compute the distance between any two points and find the minimum.
  - We can do this with and without using a list of distances.
- Let's work through the approach to this and post the result on the course website.

Part 3: Lists of Lists

- In programming you often must deal with data much more complicated than a single list. For example, we might have a list of lists, where each list might be temperature (or pH) measurements at one location of a study site:

```
temps_at_sites = [ [ 12.12, 13.25, 11.17, 10.4],
                   [ 22.1, 29.3, 25.3, 20.2, 26.4, 24.3 ],
                   [ 18.3, 17.9, 24.3, 27.2, 21.7, 22.2 ],
                   [ 12.4, 12.5, 12.14, 14.4, 15.2 ] ]
```

- Here is code to find the site with the maximum average temperature; note that no indices are used.

```
averages = []
for site in temps_at_sites:
    avg = sum(site) / len(site)
    averages.append(avg)

max_avg = max(averages)
max_index = averages.index(max_avg)
print("Maximum average of {:.2f} occurs at site {}".format(max_avg, max_index))
```

- Notes:
  - for loop variable `site` is an **alias** for each successive list in `temps_at_sites`
  - A separate list is created to store the computed averages
  - We will see in class how this would be written without the separate `averages` list.

Part 4: Controlling Execution of Loops

- We can control loops through use of
  - `break`
  - `continue`
- We need to be careful to avoid infinite loops

Using a Break

- We can terminate a loop immediately upon seeing the 0 using Python's `break`:

```
sum = 0
while True:
    x = int(input("Enter an integer to add (0 to end) ==> "))
    if x == 0:
        break
    sum += x

print(sum)
```

- `break` sends the flow of control immediately to the first line of code outside the current loop, and
- The while condition of `True` essentially means that the only way to stop the loop is when the condition that triggers the `break` is met.

Continue: Skipping the Rest of a Loop Iteration

- Suppose we want to skip over negative entries in a list. We can do this by telling Python to `continue` when the loop variable, taken from the list, is negative:

```
for item in mylist:
    if item < 0:
        continue
    print(item)
```

- When it sees `continue`, Python immediate goes back to the "top" of the loop, skipping the rest of the code, and initiates the next iteration of the loop with a new value for `item`.
- Any loop that uses `break` or `continue` can be rewritten without either of these.
    - Therefore, we choose to use them only if they make our code clearer.
    - A loop with more than one `continue` or `break` is rarely well-structured, so if you find that you have written such a loop you should stop and rewrite your code.
- The example above, while illustrative, is probably better without the `continue`.
    - Usually when we use `continue` the rest of the loop is relatively long. The condition that triggers the `continue` is tested at or near the top of the loop.

Termination of a While Loop

- When working with a while loop one always needs to ensure that the loop will terminate! Otherwise we have an *infinite loop*.
- Sometimes it is easy to decide if a loop will terminate. Sometimes it is not.
- Do either of the following examples cause an infinite loop?

```
import math
x = float(input("Enter a positive number -> "))
while x > 1:
    x = math.sqrt(x)
    print(x, flush=True)
```

```
import math
x = float(input("Enter a positive number -> "))
while x >= 1:
    x = math.sqrt(x)
    print(x, flush=True)
```

Summary

- `range` is used to generate a sequence of indices in a `for` loop.
- Nested for loops may be needed to iterate over two dimensions of data.

- Lists of lists may be used to specify more complex data. We process these using a combination of `for` loops, which may need to be nested, and Python's built-in functions. Use of Python's built-in functions, as illustrated in the example here in these notes, is often preferred.
- Loops (either for or while) may be controlled using `continue` to skip the rest of a loop iteration and using `break` to terminate the loop altogether. These should be used sparingly!