MidtermReview

June 28, 2017

1 Midterm Review

CSCI 1360E: Foundations for Informatics and Analytics

1.1 Material

- Anything in Lectures 1 through 10 are fair game!
- Anything in assignments 1 through 4 are fair game!

1.2 Topics

Data Science - Definition - Intrinsic interdisciplinarity - "Greater Data Science"

Python Language - Philosophy - Compiled vs Interpreted - Variables, literals, types, operators (arithmetic and comparative) - Casting, typing system - Syntax (role of whitespace)

Data Structures - Collections (lists, sets, tuples, dictionaries) - Iterators, generators, and list comprehensions - Loops (`for`, `while`), loop control (`break`, `continue`), and utility looping functions (`zip`, `enumerate`) - Variable unpacking - Indexing and slicing - Differences in indexing between collection types (tuples versus sets, lists versus dictionaries)

Conditionals - `if` / `elif` / `else` structure - Boolean algebra (stringing together multiple conditions with `or` and `and`)

Exception handling - `try` / `except` structure, and what goes in each block

Functions - Defining functions - Philosophy of a function - Defining versus calling (invoking) a function - Positional (required) versus default (optional) arguments - Keyword arguments - Functions that take any number of arguments - Object references, and their behaviors in Python

NumPy - Importing external libraries - The NumPy `ndarray`, its properties (`.shape`), and indexing - NumPy submodules - Vectorized arithmetic in lieu of explicit loops - NumPy array dimensions, or *axes*, and how they relate to the `.shape` property - Array broadcasting, uses and rules - Fancy indexing with boolean and integer arrays

1.3 Midterm Logistics

- The format will be very close to that of JupyterHub assignments (there may or may not be autograders to help).
- It will be **90 minutes**. Don't expect any flexibility in this time limit, so plan accordingly.

- You are **NOT** allowed to use internet resources or collaborate with your classmates (enforced by the honor system), but you **ARE** allowed to use lecture and assignment materials from this course, as well as terminals in the JupyterHub environment or on your local machine.

- I will be available on Slack for questions most of the day tomorrow, from 9am until about 3pm (then will be back online around 4pm until 5pm). Shoot me a direct message if you have a conceptual / technical question relating to the midterm, and I'll do my best to answer ASAP.

1.4 JupyterHub Logistics

- The midterm will be released on JupyterHub at **12:00am on Thursday, June 29**.

- It will be collected at **12:00am on Friday, June 30**. The release and collection will be done by automated scripts, so believe me when I say there won't be any flexibility on the parts of these mechanisms.

- Within that 24-hour window, you can start the midterm (by "Fetch"-ing it on JupyterHub) whenever you like.

- **ONCE YOU FETCH THE MIDTERM, YOU WILL HAVE 90 MINUTES FROM THAT MOMENT TO SUBMIT THE COMPLETED MIDTERM BACK.**

- Furthermore, it's **up to you** to keep track of that time. Look at your system clock when you click "Fetch", or use the timer app on your smartphone, to help you track your time use. Once the 90 minutes are up, the exam is considered late.

- In theory, this should allow you to take the midterm when it is most convenient for you. Obviously you should probably start no later than 10:30PM tomorrow, since any submissions after midnight on Friday will be considered late, even if you started at 11:58PM.

1.5 Tough Assignment Questions and Concepts

1.5.1 From A1

Do NOT hard-code answers!

For example, take the question on taking the square root of a number and converting it to a string:

```
In [1]: number = 3.14159265359
```

Answering this is *not* simply taking what's in the autograder and copy-pasting it into your solution:

```
In [2]: number = "1.7724538509055743"
```

The whole point is that your code should *generalize* to any possible input.

To that end, you want to perform the actual operations required: as stated in the directions, this involves taking the square root and converting the answer to a string:

```
In [3]: number = 3.14159265359
        number = number ** 0.5  # Raise to the 0.5, which means square root.
        number = str(number)    # Cast to a string.
```

With great looping power comes great looping responsibility.

The question that involved finding the first negative number in a list of numbers gave a lot of folks problems. By that, I mean folks combined simultaneous `for` and `while` loops, inadvertently creating more problems with some very difficult-to-follow program behavior.

Thing to remember: **both loops can solve the same problems**, but they lend themselves to different ones. So in almost all cases, you'll only need 1 of them to solve a given problem.

In this case: if you need to perform operations on every element of a list, `for` is your friend. If you need to do something repeatedly *until some condition is satisfied*, `while` is your operator. This question better fits the latter than the former.

```
In [4]: def first_negative(numbers):
            num = 0

            index = 0
            while numbers[index] > 0:
                index += 1
            num = numbers[index]

            return num

In [5]: first_negative([1, 2, 3, -1])

Out[5]: -1

In [6]: first_negative([10, -10, -100, -50, -75, 10])

Out[6]: -10
```

1.5.2 From A2

`zip()` is an amazing mechanism for looping with MULTIPLE collections at once

There were very few students who deigned to use `zip()`; if you can learn how to use it, it will make your life considerably easier whenever you need to loop through multiple lists at the same time.

Take the question on computing magnitudes of 3D vectors.

```
In [7]: def compute_3dmagnitudes(X, Y, Z):
            magnitudes = []

            ### BEGIN SOLUTION

            ### END SOLUTION

            return magnitudes
```

Since all three lists--`X`, `Y`, and `Z`--are the same length, you could run a `for` loop with an index through one of them, and use that index across all three. That would work just fine.

```
In [8]: def compute_3dmagnitudes(X, Y, Z):
            magnitudes = []

            length = len(X)
            for i in range(length):
                # Pull out the corresponding (x, y, z) coordinates.
                x = X[i]
                y = Y[i]
                z = Z[i]

                ### Do the magnitude computation ###

            return magnitudes
```

...but it's very verbose, and can get a bit difficult to follow.

If, instead, you use `zip`, you don't need to worry about using `range` or computing the `len` of a list or even extracting the right `x,y,z` from each index:

```
In [9]: def compute_3dmagnitudes(X, Y, Z):
            magnitudes = []

            for x, y, z in zip(X, Y, Z):
                pass
                ### Do the magnitude computation ###

            return magnitudes
```

Look how much cleaner that is!

The `zip` function is amazing. You can certainly get along without it, as shown in the previous slide, but it handles so much of that work for you, so I encourage you to practice using it.

Indentation in Python may be the most important rule of all.

I cannot overemphasize how important it is for your code indentation to be precise and exact.

Indentation dictates whether a line of code is part of an `if` statement or not, part of a `for` loop or not, part of a `try` block or not, even part of a function or not.

There were quite a few examples of code that was completely correct, but it wasn't indented properly--for example, it wasn't indented under a `for` loop, so the line was executed only once, *after* the `for` loop finished running.

1.5.3 From A3

`if` statements don't always need an `else`.

I saw this a lot:

```
In [10]: def list_of_positive_indices(numbers):
             indices = []
             for index, element in enumerate(numbers):
```

```
            if element > 0:
                indices.append(index)
            else:
                pass  # Why are we here? What is our purpose? Do we even exist?
        return indices
```

`if` statements are adults; they can handle being short-staffed, as it were. If there's literally nothing to do in an `else` clause, you're perfectly able to omit it entirely:

```
In [11]: def list_of_positive_indices(numbers):
             indices = []
             for index, element in enumerate(numbers):
                 if element > 0:
                     indices.append(index)
             return indices
```

An actual example of reference versus value.

This was the bonus question from A3 about building a list-of-lists matrix.

Some of you had a very clever solution that *technically* worked, but would fail spectacularly the moment you actually tried to use the matrix built by the function.

In short: rather than construct a matrix of 0s one element at a time, the strategy was to pre-construct a row of 0s, and then use just 1 loop to append this pre-built list a certain number of times.

It was clever in that it avoided the need for nested loops, which are certainly difficult to write and understand under the best of circumstances! But you'd see some odd behavior if you tried to *use* the matrix that came out...

```
In [12]: def make_matrix(rows, cols):
             pre_built_row = []

             # Build a single row that has <cols> 0s.
             for j in range(cols):
                 pre_built_row.append(0)

             # Now build a list of the rows.
             matrix = []
             for i in range(rows):
                 matrix.append(pre_built_row)

             return matrix
```

```
In [13]: m = make_matrix(3, 4)
         print(m)
```

```
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

Certainly *looks* ok--3 "rows" (i.e. lists), each with 4 0s in them. Why is this a problem?

Let's try changing one of the 0s to something else. Say, change the 0 in the upper left (position 0, 0) to a 99.

```
In [14]: m[0][0] = 99
```

Now if we print this... what do you think we'll see?

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

```
[[99, 0, 0, 0], [99, 0, 0, 0], [99, 0, 0, 0]]
```

cue The Thriller

This is a *pass-by-reference* problem. You've appended three references to the *same object*, so when you update one of them, you actually update them all.

The fix is to use a nested-`for` loop to create everything on-the-fly:

```
In [16]: def make_matrix(rows, cols):
             matrix = []

             for i in range(rows):
                 matrix.append([])  # First, append an empty list for the new row.
                 for j in range(cols):
                     matrix[i].append(0)  # Now grow that empty list.

             return matrix
```

```
In [17]: m = make_matrix(3, 4)
         print(m)
```

```
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

```
In [18]: m[0][0] = 99
         print(m)
```

```
[[99, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

1.5.4 From A4

`len(ndarray)` **versus** `ndarray.shape`

For the question about checking that the lengths of two NumPy arrays were equal, a lot of people chose this route:

```
In [19]: # Some test data
         import numpy as np
         x = np.random.random(10)
         y = np.random.random(10)
```

```
In [20]: len(x) == len(y)
```

```
Out[20]: True
```

which works, but only for *one-dimensional arrays*.

For anything other than 1-dimensional arrays, things get problematic:

```
In [21]: x = np.random.random((5, 5))   # A 5x5 matrix
         y = np.random.random((5, 10))   # A 5x10 matrix

In [22]: len(x) == len(y)

Out[22]: True
```

These definitely are **not** equal in length. But that's because `len` doesn't measure *length* of matrices...it only measures the number of rows (i.e., the first axis--which in this case is 5 in both, hence it thinks they're equal).

You definitely want to get into the habit of using the `.shape` property of NumPy arrays:

```
In [23]: x = np.random.random((5, 5))   # A 5x5 matrix
         y = np.random.random((5, 10))   # A 5x10 matrix

In [24]: x.shape == y.shape

Out[24]: False
```

We get the answer we expect.

1.6 Other Questions from the Google Hangouts Review Session

The Tale of Two `for` Loops

```
In [25]: import numpy as np

         # Generate a random list to work with as an example.
         some_list = np.random.random(10).tolist()
         print(some_list)
```

```
[0.6931832056888297, 0.6021857169653142, 0.03836475358595104, 0.3651604096030209, 0.275536458488
```

1: Looping through *elements*

```
In [26]: for element in some_list:
             print(element)
```

```
0.6931832056888297
0.6021857169653142
0.03836475358595104
0.3651604096030209
0.2755364584885275
0.30045938465583466
0.47102308993526965
0.8970256806073568
0.15810029910258372
0.21366871301862667
```

2: Looping over *indices* of elements

```
In [27]: list_length = len(some_list)
         for index in range(list_length):  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
             element = some_list[index]
             print(element)
```

```
0.6931832056888297
0.6021857169653142
0.03836475358595104
0.3651604096030209
0.2755364584885275
0.30045938465583466
0.47102308993526965
0.8970256806073568
0.15810029910258372
0.21366871301862667
```

In general:

- If you don't care about list order, or where you are in the list--use "loop by element"
- If ordering of elements MATTERS, or where you are in the list during a loop is important--use "loop by index"

Sliding windows for finding substrings in a longer string
From A4, Bonus Part A:

```
In [28]: def count_substring(base_string, substring, case_insensitive = True):
             count = 0

             if case_insensitive == True:
                 base_string = base_string.lower()
                 #base_string = base_string.upper()

             length = len(substring)
             index = 0
             while (index + length) < len(base_string):

                 # Sliding window.
                 substring_to_test = base_string[index : (index + length)]
                 if substring_to_test == substring:
                     count += 1

                 index += 1

             return count
```

Normalization by any other name

This confused some folks, namely because "normalization" can mean a lot of different things. In particular, two different types of normalization were conflated:

1: Rescale vector elements so the vector's magnitude is 1

- NOT the same thing as having all the vector elements SUM to 1

2: Rescale vector elements so they all sum to 1

- What the Bonus, Part B in A4 was actually asking for (even though the autograder was terrible)

tl;dr These are both perfectly valid forms of normalization. It's just that the autograder was horrible. Here's what the spirit of the question was asking for:

```
In [29]: numbers = [10, 20, 30, 40]
         print(sum(numbers))
```

```
100
```

```
In [30]: numbers = [10/100, 20/100, 30/100, 40/100]
         # (0.1 + 0.2 + 0.3 + 0.4)  = 1.0
         print(sum(numbers))
```

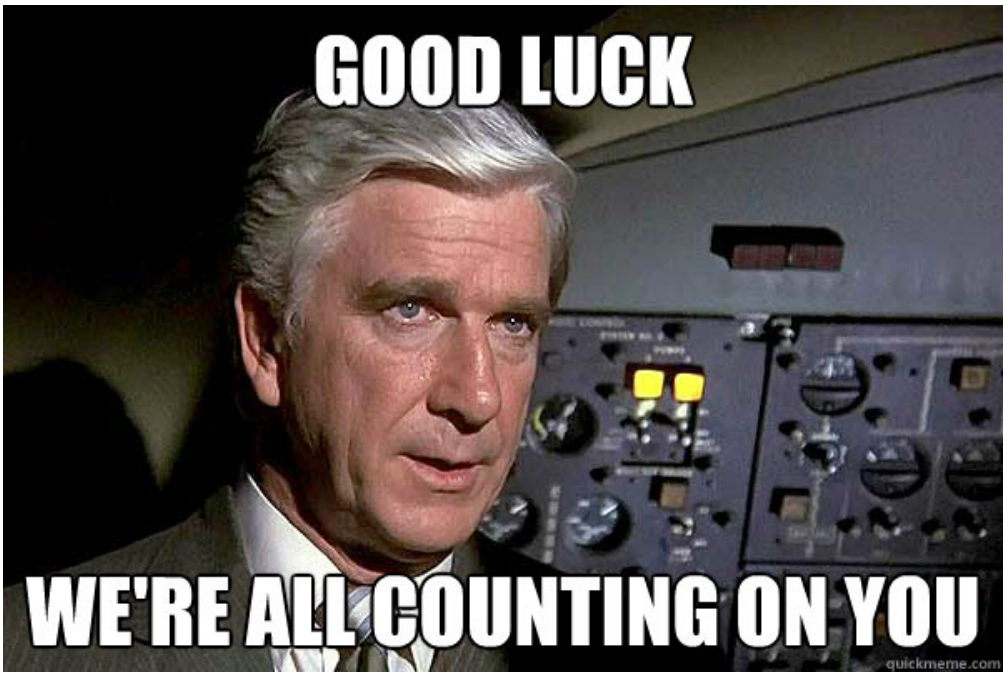
```
1.0
```

```
In [31]: import numpy as np
         def normalize(something):

             # Compute the normalizing constant
             s = something.sum()

             # Use vectorized programming (broadcasting) to normalize each element
             # without the need for any loops
             normalized = (something / s)

             return normalized
```

This can then be condensed into the 3 lines required by the question:

```
In [32]: import numpy as np  # 1
         def normalize(something):  # 2
             return something / something.sum()  # 3
```

1.6.1 Good luck!



goodluck