CSC148, Lab #5
week of February 10th, 2014

This document contains the instructions for lab number 5 in CSC148H1. To earn your lab mark, you must actively participate in the lab. We mark you in order to ensure a serious attempt at learning, NOT to make careful critical judgments on the results of your work.

General rules

We will use the same general rules as for the first lab (including pair programming). See the instructions at the beginning of Lab 1 to refresh your memory.

Overview

In this lab you will learn, and reason about, some python idioms, as well as develop some unit testing skills.

Comprehensions, **zip**, and **filter** capture logical patterns that are often used by programmers, and provide an occasionally-more-readable and internally-optimized form for them, see Python tips on loops, and Goodger on comprehensions, filter documentation, and Python zip documentation. These forms make a programmer's intention clear — that they are intendeding to produce a new list or iterable from an old one. In this lab, you will re-implement some functions written using comprehensions and **filter** to use loops instead, and verify that you have consistent implementations using unit tests.

Vector and matrix operations

(Student s1 drives, student s2 navigates)

Vectors can be represented as python lists of numbers. You may have encountered them, but in any case they support some operations peculiar to themselves.

dot product

One such operation is the **dot-product** — a way of multiplying two vectors to get a single number (rather than a list of numbers). Here's an example, where the symbol $\cdot$ represents the dot-product operation:

$$[1, 2, 3] \cdot [4, 5, 6] = (1 \times 4) + (2 \times 5) + (3 \times 6) = 4 + 10 + 18 = 32$$

Basically, we multiply the corresponding elements of the two vectors together, and then sum those products.

Your first task is to read the definition of **dot_prod()** in comprehension.py. You may also look over Python tips on loops, tuple unpacking, and zip to see how this solution works.

Now, re-implement this function (write your own implementation) in a new file called **loop.py** without using a list comprehension. Use exactly the same function name and parameters, just change the body. Your basic approach will be:

1. create an empty list (if you're producing a list), or a variable initially set to 0
2. loop over the iterables (lists in this case) provided
3. inside the list, update your list or variable
4. when you're done, return your list or variable

Also, in a new file called **tester.py**, create new test cases in the TestCase **DotProductTester**. Increase your confidence that your implementation of **dot_prod()**, as well as the one in **comprehension.py**, pass appropriate tests. To get an idea of what tests you should consider, review Choosing test cases. You should strive to test one case at a time with small **test...** methods, rather than lumping all your tests together.

If you're stuck, talk to your TA. If you're not stuck, show your TA your work.

matrix-vector product

Another operation multiplies a **matrix M** — essentially a list of vectors — times a vector **v**, resulting in a new vector. The idea is to take the dot-product of each vector in the matrix with the vector you are multiplying it with to yield the corresponding entry in the new vector. An example should make this more concrete (here we indicate the matrix-vector product by $\times$)

$$[[1,2],[3,4]] \times [5,6] = [[1,2] \cdot [5,6], [3,4] \cdot [5,6]] = [17, 39]$$

Notice that we recycle the **dot_product** in order to implement the **matrix_vector** product.

Again, your first task is to read the definition of **matrix_vector_prod()** in comprehension.py. . Then re-implement **matrix_vector_prod()** in the file **loop.py**, using a loop or loops, rather than a comprehension. You should certainly use **dot_prod()** in your implementation. Once you are done, create some new test cases in the TestCase **MatrixVectorProductTester** in the file **tester.py**. Increase your confidence that both implementations pass appropriate tests.

If you're stuck, talk to your TA. If you're not stuck, show your TA your work.

Pythagorean triples

List comprehensions aren't just limited to iterating over a single iterable. Try out the following example:

```
[(i, j, k) for i in range(3) for j in range(3) for k in range(3)]
```

Pythagorean triples are triples of integers (x, y, z) where $x^2 + y^2 = z^2$ (representing the sides of special right-angle triangles). These can be discovered analytically, but why not let a computer do the work?

Read over the implementation of **pythagorean_triples()** in **comprehension.py**. You may first want to read documentation for filter. Once you're done, re-implement **pythagorean_triples()** in the file **loop.py**, using (of course!) neither comprehensions nor the built-in **filter** function. Add test methods to the TestCase PythagoreanTripleTester, to increase your confidence that both implementations pass appropriate tests.

If you get stuck, call over your TA. If you don't get stuck, show your completed work to your TA.

any and all

(Student s1 drives, student s2 navigates)

Sometimes we want to apply a boolean function to a list — do **any** list elements satisfy a given condition, or do **all** list elements satisfy a given condition.

Read over the implementation of **any_pythagorean_triples()** in **comprehension.py**. You may first want to look over the documentation for all, any, as well as generator comprehensions. Once you're done, re-implement **any_pythagorean_triples()** in **loop.py**, without using a comprehensions, **any**, **all**, **filter()**, or **pythagorean_triples()**. Then add test methods to the TestCase AnyPythagoreanTripleTester, to increase your confidence that both implementations pass the appropriate tests.

If you get stuck, talk to your TA. If you don't get stuck, also talk to your TA.