CSC148, Lab #5

This document contains the instructions for lab number 5 in CSC148H. 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 previous labs (including pair programming).

Overview

In this lab you will learn and reason with some Python idioms.

Comprehensions, **zip**, and **filter** capture logical patterns that are often used by programmers, and can sometimes provide more-readable and internally-optimized code for these patterns. These forms make a programmer's intention clear: that they are intending 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. They support several operations that we will study here.

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:

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

That is, 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`. The function uses list comprehensions and `zip` that you should understand before continuing. Consult the slides and other materials for this lab on the labs page of the website and play around with these tools in the shell!

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. start a total at 0
2. loop over the provided lists
3. inside the loop, update your total
4. when you're done, return your total

Also, in **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**, passes appropriate tests.

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 — by a vector **v**, resulting in a new vector. The idea is to take the dot-product of each vector in **M** with the vector **v** 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 **dot_prod** in order to implement **matrix_vector_prod**.

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

(Student s2 drives, student s1 navigates)

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

```
[(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 up on `filter` in the provided lab slides. Once you're done, re-implement **pythagorean_triples** in the file **loop.py**, without using comprehensions or 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

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 slides for `any` and `all`. Once you're done, re-implement **any_pythagorean_triples** in **loop.py**, without using 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.