CSC148, Lab #5
week of October 14th, 2013

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 idiom.

List and generator comprehensions capture logical patterns that are often used by programmers. David Goodger describes them:
http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html#list-comprehensions
As part of Python idiom, these forms make a programmer's intention clear. List comprehensions make it clear that you intend to build a new list from an old list (or other iterable) without changing the original iterable. Generator comprehensions make it clear that you want a sequence of values built from an old iterable, but you don't need them all at once.

These also improve performance.

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 complete the definition of `dot_prod()` in idiom.py. We're going to micro-manage a bit and specify **how** you implement it. Please refer to How to code like a Pythonista while you do these.

1. Read the Python docs on zip and figure out how to transform vectors (number lists) **u** and **v** into a single list of pairs — the pair of **u** and **v**'s first elements, then the pair of their second elements, and so on. Please note that **xrange** and **irange** don't exist in Python 3.
2. How do you transform a pair of numbers (n_1, n_2), into the product $n_1 \times n_2$ in Python? You have the option (but are not required) of considering tuple unpacking here
3. Read "How to code like a Pythonista" and figure out how to write a list comprehension of all the products of the pairs generated in Step 1
4. Use the built-in **sum** function to add all the numbers in your list comprehension

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.

Your next task is to complete the definition of `matrix_vector_prod()` in idiom.py. Again we will micro-manage your implementation

1. How do you compute the dot-product of a vector from **M** with **v**? Please don't repeat work you've already done.
2. How do you make a list comprehension for the dot-products from Step 1?

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

list efficiency and sequences

(Student `s2` drives, student `s1` navigates)

List comprehensions are designed to be both clear and efficient. We can test this claim by doing the same thing in two different ways.

Your task is to complete functions `squares_build_list`, `squares_use_comp`, and `squares_use_gen` in idiom.py. Currently these functions run very quickly, but very incorrectly! Again, we're going to micro-manage so that the differences in implementation become obvious.

For `squares_build_list` you are to build up a list of the first n squares of natural numbers by looping over the values [0, ..., n-1] and appending each square to an initially empty list. A handy source of the values [0, ..., n-1] is `range(n)`. This should be a familiar task. Notice that computer scientists are generous enough to include 0 among the natural numbers, hence the $n - 1$ ending value.

For `squares_use_comp` you produce exactly the same list using a list comprehension. Consult Goodger if you are having trouble writing down the list comprehension.

For `squares_use_gen` you produce the corresponding sequence using a generator comprehension (see Goodger again).

The supplied docstrings have a single test case, and it's up to you if you want to provide more.

Evaluate `idiom.py`, so that the timing tests at the end of the file are run. Discuss why the results are different, and under which conditions each variant should be used.

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

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?

Complete the implementation of `pythagorean_triples` in idiom.py. You guessed it: we'll micro-manage and want you to use comprehensions. Here's the idea

1. From the example above, you already know how to produce **all** the triples in the appropriate range. Might as well start there.
2. You should restrict the produced list to just those triples with distinct values. You can add an **if** condition after all the **for foo in bar** clauses.
3. What you really want is to restrict the list so that they are pythagorean triples. Add to your **if** condition.

Notice that the end product is a list. Can you think of any conditions under which you'd want it to be a generator?

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

any and generators

(Student `s1` drives, student `s2` navigates)

Sometimes we write a loop with an early-exit condition — for example, the much-maligned `break` — because we are only interested in the first occurrence of some value in a sequence. Generators, as well as the built-in function any provides an alternative

In a Python shell, experiment with the following expressions:

```
>>> g = (x for x in range(0,1))
>>> any(g)
...
>>> g = (x for x in range(1, 2))
>>> any(g)
```

Explain this result. Now experiment with the three expressions below. Can you explain the difference in performance?

```
>>> [x for x in [1, 2, 3] if any((x == y for y in range(1,4)))]
>>> [x for x in [1, 2, 3] if any((x == y for y in range(1,4000000)))]
>>> [x for x in [1, 2, 3, -1] if any((x == y for y in range(1,4000000)))]
```

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