BST281
Lab Session 7

Announcement:

- Homework 2 grades will be posted by this Sunday, 3/26.
- **File grading**: we changed our grading methods now to doctest. So (1) please keep everything in your problemset (no need to remove the comment part as we requested before); (2) still remember to change the problemset name before you submit to canvas (e.g. problems03.py)
  Thank you for your patience!
- List comprehension
- More python exercises

TA: Xue Zou Email: xuz943@mail.harvard.edu

Doctest https://docs.python.org/2/library/doctest.html

Say, your py file named: problems02.py

A way to check your answer in terminal:

- python -m doctest problems02.py

Example online:

If you run `example.py` directly from the command line, **`doctest`** works its magic:

```
$ python example.py
$
```

There's no output! That's normal, and it means all the examples worked. Pass `-v` to the script, and **`doctest`** prints a detailed log of what it's trying, and prints a summary at the end:

```
$ python example.py -v
Trying:
    factorial(5)
Expecting:
    120
ok
Trying:
    [factorial(n) for n in range(6)]
Expecting:
    [1, 1, 2, 6, 24, 120]
ok
Trying:
    [factorial(long(n)) for n in range(6)]
Expecting:
    [1, 1, 2, 6, 24, 120]
ok
```

Example in HW2

```
16 (6). Replace the following line with a short function that makes
    fastq_quals_decode work as intended.  You should call fastq_qual_decode on
    each individual quality score character, accumulate the results in a new
    list, and return it.
    """
    return []
```

Most students do the following for loop:

```
    quals=[]
    for str in strQuals:
        quals.append(fastq_qual_decode(str))
    return quals
```

Shortening an if/for loop

Simplify your Python loops

If you're like most programmers, you know that, eventually, once you have an array, you're gonna have to write a loop. Most of the time, this is fine and dandy, but sometimes you just don't want to take up the multiple lines required to write out the full for loop for some simple thing. Thankfully, Python realizes this and gives us an awesome tool to use in these situations. That tool is known as a list comprehension.

http://blog.teamtreehouse.com/python-single-line-loops

What the heck is that?

List comprehensions are lists that generate themselves with an internal for loop. They're a very common feature in Python and they look something like:

```
[thing for thing in list_of_things]
```

Now that I've confused you even more, let's step back. A list has multiple things in it, but it's defined by being between square brackets. Let's say I want to have a function that doubles the values all of the items in a list of numbers. First, let me set up a list of numbers.

```
my_list = [21, 2, 93]
```

Now let's make the function. We'll call it `list_doubler` since that's what it does and it will take an argument that'll be the list we're going to double.

```
def list_doubler(lst):
    doubled = []
    for num in lst:
        doubled.append(num*2)
    return doubled
```

Calling this function would get us a new list with doubled items.

```
my_doubled_list = list_doubler(lst)
```

`my_doubled_list` would now have the values 42, 4, and 186. This function is simple and achieves what we want pretty simply, but it's also five lines, counting the definition line, has a variable that we do nothing but append to and finally return. The only real working part of the function is the for loop. The for loop isn't doing much, either, just multiplying a number by 2. This is an excellent candidate for making into a list comp.

Building the list comprehension

Let's keep it as a function we'll call. We just want to simplify the inside. First, since list comprehensions create lists, and lists can be assigned to variables, let's keep `doubled` but put the list comprehension on the righthand side of it.

```
doubled = [thing for thing in list_of_things]
```

OK, so we need to fill out the right hand side. Just like normal for loops, which the righthand side of the comprehension looks exactly like, we have to name the things in our loop. First, let's name each thing and we'll also use the list variable that's getting passed in.

```
doubled = [thing for num in lst]
```

This won't actually work yet since `thing` isn't a...thing. In our original function, we did `num * 2`, so let's do that again where we have `thing` right now.

```
doubled = [num * 2 for num in lst]
```

Whatever is before the `for` is what actually gets added to the list. Finally, we should return our new list.

```
def list_doubler(lst):
    doubled = [num * 2 for num in lst]
    return doubled
```

Let's test it out.

```
my_doubled_list = list_doubler([12, 4, 202])
```

And, yep, `my_doubled_list` has the expected values of 24, 8, and 404. Great, looks like it worked! But, since we're creating and immediately returning a variable, let's just return the list comprehension directly.

```
def list_doubler(lst):
    return [num * 2 for num in lst]
```

Ok, great, but why would I want to use this?

List comprehensions are great to use when you want to save some space. They're also handy when you just need to process a list quickly to do some repetitive work on that list. They're also really useful if you learn about functional programming, but that's a topic for a later course (hint hint).

But if all you could do is work straight through a list, list comprehensions wouldn't be all that useful. Thankfully, they can be used with conditions.

Example in HW2

```
16 (6). Replace the following line with a short function that makes
    fastq_quals_decode work as intended.  You should call fastq_qual_decode on
    each individual quality score character, accumulate the results in a new
    list, and return it.
    """
    return []
```

Most students do the following for loop:

```
    quals=[]
    for str in strQuals:
        quals.append(fastq_qual_decode(str))
    return quals
```

How should we modify this (based on what we just learned)?

Everything's conditional

Let's make a new function that only gives us the long words in a list. We'll say that any word over 5 letters long is a long word. Let's write it out longhand first.

```
def long_words(lst):
    words = []
    for word in lst:
        if len(word) > 5:
            words.append(word)
    return words
```

We make a variable to hold our words, loop through all of the words in our list, and then check the length of each word. If it's bigger than 5, we add the word to the list and then, finally, we send the list back out. Let's give it a try.

`long_words(['blog', 'Treehouse', 'Python', 'hi'])` gives back `['Treehouse', 'Python']`. That's exactly what we'd expect.

Alright, let's rewrite it to a list comprehension. First, let's build what we already know.

```
def long_words(lst):
    return [word for word in lst]
```

That gives us back all of our words, though, not just the ones that are more than 5 letters long. We add the conditional statement to the end of the for loop.

```
def long_words(lst):
    return [word for word in lst if len(word) > 5]
```

So we used the same exact if condition but we tucked it into the end of the list comprehension. It uses the same variable name that we use for the things in the list, too.

OK, let's try out this version.
`long_words(['list', 'comprehension', 'Treehouse', 'Ken'])` gives back `['comprehension', 'Treehouse']`.

Practices time

- Let's start with an example, say we have 10 numbers, and we want a subset of those that are greater than, say, 5.

```
>>> numbers = [12, 34, 1, 4, 4, 67, 37, 9, 0, 81]
```

Hint:You cab start with your old way first, and then change it to the new way we just learned!

Where to from there?

Explore:

- Dictionary comprehensions

http://legacy.python.org/dev/peps/pep-0274/

- Set comprehensions

http://legacy.python.org/dev/peps/pep-0218/

More python exercise

- Please download and look at the ***python_problemset.pdf***