6.189 Homework 2

Readings

How To Think Like A Computer Scientist: Wednesday - chapter 3 and Appendix A (all), 6.5 - 6.9.
Thursday - chapters 7 & 8 (all).

6.01 Fall 2010 Course Notes: Wednesday - sections 2.1, 2.2, and 2.3 (up to the heading 'Lists').
Thursday - pages 33-37 (sections 'Lists', 'Iterations over lists', 'List Comprehensions').

What to turn in

Turn in a printout of your code exercises stapled to your answers to the written exercises at 3 PM on Friday, January 7th.

Exercise 2.0 – Print vs Return

Note: This exercise is in section 2.2 of the 6.01 Course Notes. If you've already gone through it, you may skip this.

This isn't really an exercise, just an important bit of reading. Download the template file `homework_2.py`. In it these two functions are defined:

```
def f1(x):
     print x + 1
def f2(x):
     return x + 1
```

Run this code in the shell. What happens when we call these functions?

```
>>> f1(3)
4
>>> f2(3)
4
```

It looks like they behave in exactly the same way. But they really don't. Try this:

```
>>> print f1(3)
4
None
>>> print f2(3)
4
```

In the case of `f1`, the function, when evaluated, prints 4; then it returns the value `None`, which is printed by the Python shell. In the case of f2, it doesn't print anything, but it returns 4, which is printed by the Python shell. Finally, we can see the difference here:

```
>>> f1(3) + 1
4
Traceback (most recent call last):
   File "<stdin>", line 1, in ?
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
>>> f2(3) + 1
5
```

In the first case, the function doesn't return a value, so there's nothing to add to 1, and an error is generated. In the second case, the function returns the value 4, which is added to 1, and the result, 5, is printed by the Python read-eval-print loop. The book *How To Think Like A Computer Scientist* was translated from a version for Java, and it has a lot of print statements in it, to illustrate programming concepts. But for just about everything we do, it will be returned values that matter, and printing will be used only for debugging, or to give information to the user.

Print is very useful for debugging. It's important to know that you can print out as many variables and strings as you want in one line, when they are separated by commas. Try this:

```
>>> x = 100
>>> print 'x:', x, 'x squared:', x*x, 'sqrt(x):', x**0.5
x: 100 x squared: 10000 sqrt(x): 10.0
```

Exercise 2.1 – Defining A Function

Recall how we define a function using `def`, and how we pass in parameters. In `homework_2.py` (download this from the website if you haven't yet), transform your code from exercise 1.7 (the rock, paper, scissors game) into a function that takes parameters, *instead* of asking the user for input. Make sure to *return* your answer, rather than printing it.

For this, and all future exercises, include *at least* 3 test cases below your code.

Exercise 2.2 – Writing Simple Methods

In this problem you'll be asked to write two simple methods (method is an interchangeable term for 'function'). Be sure to *test your functions well*, including at least 3 test cases for each method. Do your work in `homework_2.py`.

1. Write a method `is_divisible` that takes two integers, `m` and `n`. The method returns `True` if `m` is divisible by `n`, and returns `False` otherwise. Test cases for this function are included for you; look at the conditions that they test and try to make sure your future test cases are comprehensive.

2. Imagine that Python doesn't have the `!=` operator built in. Write a method `not_equal` that takes two parameters and gives the same result as the `!=` operator. Obviously, you cannot use `!=` within your function! Test if your code works by thinking of examples and making sure the output is the same for your new method as `!=` gives you.

Exercise 2.3 – Math Module

In this exercise, we will play with some of the functions provided in the `math` module. A module is a Python file with a collection of related functions. To use the module, you need to add the following line at the top of your program, right underneath the comments with your name:

```
import math
```

Example: if you want to find out what is $\sin(90°)$, we first need to convert from degrees to radians and then use the sin function in the `math` module:

```
radians = (90.0 / 360.0) * 2 * math.pi
print math.sin(radians)
```

You can do this work in the interpreter by typing `import math` and then these lines.

For mathematical functions, you can generally call `math.func`, where `func` is whatever function you want to call. For example, if you want the sine of an angle `a` (where `a` is in radians), you can call `math.sin(a)`. For logarithms, the function `math.log(n)` calculates the natural logarithm of `n`. You can calculate the log of any base b (as in $\log_b(n)$) using `math.log(n, b)`. The math module even includes constants such as e (`math.e`) and π (`math.pi`). Documentation for the math module is available at `http://docs.python.org/release/2.6.6/library/math.html`

Many computations can be expressed concisely using the "multadd" operation, which takes three operands and computes $a * b + c$. One of the purposes of this exercise is to practice pattern-matching: the ability to recognize a specific problem as an instance of a general category of problems.

In the last part, you get a chance to write a method that invokes a method you wrote. Whenever you do that, it is a good idea to test the first method carefully before you start working on the second. Otherwise, you might find yourself debugging two methods at the same time, which can be very difficult.

1. Write a function `multadd` that takes three parameters, `a, b` and `c`. Test your function well before moving on.
2. Underneath your function definition, compute the following values using `multadd` and print out the result:
   - `angle_test` $= \sin\left(\frac{\pi}{4}\right) + \frac{\cos(\frac{\pi}{4})}{2}$
   - `ceiling_test` $= \left\lceil \frac{276}{19} \right\rceil + 2\log_7(12)$

     Hint: If you are unfamiliar with the notation $\lceil\rceil$, this represents the *ceiling* of a number. The *ceiling* of some float x means that we always "round up" `x`. For example, $\lceil 2.1 \rceil = \lceil 2.9 \rceil = 3.0$. Look at the math module documentation for a way to do this!

   If everything is working correctly, your output should look like:

```
sin(pi/4) + cos(pi/4)/2 is:
1.06066017178
ceiling(276/19) + 2 log_7(12) is:
17.5539788165
```

3. Write a new function called `yikes` that has one argument and uses the `multadd` function to calculate the following:

$$xe^{-x} + \sqrt{(1 - e^{-x})}$$

There are two different ways to raise e to a power- check out the math module documentation. Be sure to return the result! Try `x=5` as a test; your answer should look like:
`yikes(5) is 1.0303150673`.

Exercise 2.4 – More Functions

Here's two more functions to try your hand at...

1. Write a method `rand_divis_3` that takes no parameters, generates and prints a random number, and finally returns `True` if the randomly generated number is divisible by 3, and `False` otherwise. For this method we'll use a new module, the `random` module. At the top of your code, underneath `import math`, add the line `import random`. We'll use this module to generate a random integer using the function `randint`, which works as follows:

   ```
   random.randint(lo, hi)
   ```

   where `lo` and `hi` are integers that tell the code the range in which to generate a random integer (this range is inclusive). 0 to 100 is probably a decent range.

2. Write a method `roll_dice` that takes in 2 parameters - the number of sides of the die, and the number of dice to roll - and generates random roll values for each die rolled. Print out each roll and then return the string "That's all!" An example output:

   ```
   >>> roll_dice(6, 3)
   4
   1
   6
   That's all!
   ```

Exercise 2.5 – Quadratic Formula

Write a function `roots` that computes the roots of a quadratic equation. Check for complex roots and print an error message saying that the roots are complex.

Hint 1: Your function should take three parameters- what are they?

Hint 2: We know the roots are complex when what condition about the discriminant is met?

Be sure to use a variety of test cases, that include complex roots, real roots, and double roots.

Optional: For an extra challenge, compute and print out the complex roots (Python can natively handle complex numbers - here's a good reference: `http://infohost.nmt.edu/tcc/help/pubs/python/web/complex-type.html`).

Exercise 2.6 – The game of Nims/Stones

In this game, two players sit in front of a pile of 100 stones. They take turns, each removing between 1 and 5 stones (assuming there are at least 5 stones left in the pile). The person who removes the last stone(s) wins.

Download `nims.py` from the website and open it up. Check out the lines of text in between the sets of `'''`, underneath the definition of `play_nims`. This is called a docstring, and is handy to use to tell users of your program what parameters to pass in, and what your program does.

In this problem, you'll write a function to play this game; we've outlined it for you. It may seem tricky, so break it down into parts. Like many programs, we have to use nested loops (one loop inside another). In the outermost loop, we want to keep playing until we are out of stones. Inside that, we want to keep alternating players. You have the option of either writing two blocks of code, or keeping a variable that tracks the current player. The second way could be slightly trickier, but it's definitely do-able!

Finally, we might want to have an innermost loop that checks if the user's input is valid. Is it a number? Is it a valid number (e.g. between 1 and 5)? Are there enough stones in the pile to take off this many? If any of these answers are no, we should tell the user and re-ask them the question.

As always, feel free to ask the LAs for help on any part of this problem.

If you choose to write two blocks of code, the basic outline of the program should be something like this:

```
while [pile is not empty]:
   while [player 1's answer is not valid]:
      [ask player 1]
   [execute player 1's move]

   [same as above for player 2]
```

Be careful with the validity checks. Specifically, we want to keep asking player 1 for their choice as long as their answer is not valid, BUT we want to make sure we ask them at least ONCE. So, for example, we will want to keep a variable that tracks whether their answer is valid, and set it to False initially.

When you're finished, test each other's programs by playing them!

This is the last problem covering Wednesday's material. Before moving on to Exercise 2.7, we suggest you do the written exercises.

Exercise 2.7 – Working With Lists

Download `strings_and_lists.py` from the course website. Study the function `sum_all`.

`sum_all` takes a list of numbers as a parameter (note how we specify, with a comment, what the type of the parameter must be), and returns the sum of all the numbers in the list.

Now make a new function `cumulative_sum` that modifies `sum_all` so that instead of returning the sum of all the elements, it returns the cumulative sum; that is a new list where the i^{th} element is the sum of the first $i+1$ elements from the original list. For example, the cumulative sum of `[4, 3, 6]` is `[4, 7, 13]`.

Exercise 2.8 – Report Card with GPA

Write a function `report_card` where the user can enter each of his grades, after which the program prints out a report card with GPA. Remember to ask the user how many classes he took (*think* - why would we need to ask this? Could we write the program a different way, which wouldn't need that info?). Example output is below.

```
>>> report_card()
How many classes did you take? 4
What was the name of this class? 18.02
What was your grade? 94
...
REPORT CARD:
18.02 - 94
21H.601  96
8.01  91
5.111 - 88
Overall GPA  92.25
```

Hints: You'll want to use a for loop, and you'll probably want to keep track of names and grades seperately; there are a couple ways to do this. Remember, add to lists with `my_list.append(elt)`.

Exercise 2.9 – Pig Latin

Write a function `pig_latin` that takes in a single word, then converts the word to Pig Latin. To review, Pig Latin takes the first letter of a word, puts it at the end, and appends "ay". The only exception is if the first letter is a vowel, in which case we keep it as it is and append "hay" to the end.

E.g. "boot" → "ootbay", and "image" → "imagehay".

It will be useful to define a list at the top of your code file called `VOWELS`. This way, you can check if a letter `x` is a vowel with the expression `x in VOWELS`. Remember - to get a word except for the first letter, you can use `word[1:]`.

Be sure to look at the first optional problem for ways to improve on your Pig Latin converter.

Exercise 2.10 – List Comprehensions

List comprehensions follow naturally from set builder notation and lambda calculus. They are *very* cool and make your life a lot easier. Don't worry if you don't get them; however, you will be seeing them in 6.01.

Read about list comprehensions in chapter 2 of the 6.01 course notes; the Wikipedia article on them are good, and this site is concise and good:
`http://www.secnetix.de/olli/Python/list_comprehensions.hawk` - or just Google "Python list comprehensions" and find a site that makes sense to you.

Problems: Put these exercises in `strings_and_lists.py`.

1. Write a list comprehension that prints a list of the cubes of the numbers 1 through 10.
2. Write a list comprehension that prints out the possible results of two coin flips (one result would be `'ht'`). (*Hint* - how many results should there be?)
3. Write a function that takes in a string and uses a list comprehension to return all the vowels in the string.

4. Run this list comprehension in your prompt:

   ```
   [x+y for x in [10,20,30] for y in [1,2,3]]
   ```

   Figure out what is going on here, and write a nested for loop that gives you the same result. Make sure what is going on makes sense to you!

Exercise OPT.1 – Pig Latin Sentences

This optional problem builds on the work you did in Exercise 2.9. Save your work for this problem in a new file, `pig_latin.py`; you may wish to reuse the code you wrote before to help with this exercise.

Converting one word to Pig Latin is okay, but it would be more useful to be able to convert whole sentences; so for this exercise, we'll use `raw_input` to ask the user for a full sentence and translate it, word by word. It's tricky for us to deal with punctuation and numbers with what we know so far, so instead, ask the user to enter only words and spaces. You can convert their input from a string to a list of strings by calling `split` on the string; also, you can use `lower` to make a string all lowercase:

```
>>> phrase = 'My namE is JohN SmIth'
>>> word_list = phrase.split()
>>> print word_list
['My', 'namE', 'is', 'JohN', 'SmIth']
>>> lowercase_phrase = phrase.lower()
>>> print lowercase_phrase
'my name is john smith'
```

Using a list of words, you can go through each word and convert it to Pig Latin.

Hint: It will make your life much easier - and your code much better - if you separate tasks into functions, e.g. have a function that converts one word to Pig Latin rather than putting it into your main program code.

More extensions: Once you have your program working, make it interactive such that it keeps translating phrases into pig latin until the user enters in the phrase `QUIT`. Or, you can add in some more complex Pig Latin rules - for example, words that start with "th", "st", "qu", "pl", or "tr" should move both of those letters to the end.

Eg, "stop" → "opstay", and "there" → "erethay"

There are many other Pig Latin rules that you can find online if you want a true converter. Finally, you could try and deal with punctuation by looking for it within a string and moving it to the end of the word (the solutions I wrote only handle commas, periods, `!`, `?`, `:` and `;` that appear at the ends of words, as they are pretty simple to handle).

Exercise OPT.2 – List Comprehension Challenges (tricky!)

1. Write a function that takes in a list of elements of different types and uses a list comprehension to return all the elements of the list of type `int`. **Note:** The function `isinstance` will be of help here. Google "Python isinstance" and see if you can figure out what it does, or type `help(isinstance)` at the Python shell.
2. Write a list comprehension which solves the equation $y = x^2 + 1$. Your solution should print out a list of $[x, y]$ pairs; use the domain $x \in [-5, 5]$ and the range $y \in [0, 10]$.
3. Similarly, write a list comprehension that finds the integer solutions `[x, y]` for a circle of radius 5.
4. Make your own list comprehension challenge! Write a comment of what you're trying to do in your code, then put the list comprehension below the comment.

6.189 A Gentle Introduction to Programming

January IAP 2011