CIS 480 - Python - Fall 2005
WEEK 2 LAB EXERCISE and Homework #1

week 2 lab exercise due: Thursday, September 1st , END of lab
HW #1 due: Thursday, September 8th, 12:00 noon

Until I develop more formal opening comment block standards (which I plan to do), begin **each python module** that you write with at least the following opening comments:

* a comment containing the name of the module,
* a comment containing your name, and
* a comment containing the date that your module was last modified

WEEK 2 LAB EXERCISE

1. Create a Python module **lab01.py**. In this module, let's go with a simple, and classic, numeric function: write a Python function **fahr_to_cels** that will accept a Fahrenheit temperature as its single parameter, and **return** (not print!) a Celsius temperature as output. Be careful --- it should behave itself whether the argument is an integer or a floating point value. (You do not have to actually test the argument type for this one; just make sure you devise a computation that works for either integers or floating point values.)

(Not sure of the formula? Google works quite nicely for looking up such things...)

Now, you can and should test your function within the **python** interpreter until you are confident that it works.

2. ...and now, you are going to create file output that demonstrates that it works. In a separate module **lab01_test.py**:

* write the proper command that it would take to allow you to call your **fahr_to_cels** function within this module (without redefining the function in this file!)

* write a print statement that prints a string that includes the testing call you are about to make along with the value you expect it to return, and then write another print statement that prints the result of actually making that call. (That is, in your output you'll see one line that says what you are about to call and what it should return, and in the next line you'll see what that call actually returned.)

  Do this for at least 3 different calls of your function. (That will mean at least 6 lines in your output.)

* Then, at the command line, try:

```
python lab01_test.py
```

  Don't like what you see? Then tweak lab01.py/lab01_test.py until you do.

  Then, create an output file by doing:

```
python lab01_test.py > lab01_test.out
```

3. Now, write you name on the "Next:" list on the board. When I reach your name, I will come and check your lab01.py, lab01_test.py, and lab01_test.out files.

All of the above must be completed by the end of your lab hour.

HOMEWORK #1

Create a Python module **hw01.py**. Within it, include the following:

1. Let's start with an (admittedly bizarre) function **multiples_in_range** that takes a numeric value, a beginning value, and and ending value as arguments. It prints, one per line, each of the values of the numeric value that fall strictly between the beginning value and the ending value. You can assume --- without adding code to check --- that all three arguments are positive numbers.

For example, if **multiples_in_range** is called with the arguments **(2, 7, 13)**, the function would print the following:

```
8
10
12
```

And, of course, test this function within the **python** interpreter until you are confident that it works.

2. Now, we're going to create a module that will eventually be used to create file output that demonstrates that **multiples_in_range** works. In a separate module **hw01_test.py**:

* write the proper command that it would take to allow you to call your **multiples_in_range** function within this module (without redefining the function in this file!)

* write a print statement that prints a string that includes the testing call you are about to make and a list of the values that you expect it to print, and then simply make that call. (NOTE that this testing call doesn't need to be part of a print statement --- this is a function that happens to have printing as a side effect!)

  Do this for at least 3 different calls of your function.

You should, of course, test **hw01_test.py** until you are comfortable with it. You should wait to create an output file, however; we're going to add more functions and more tests to **hw01.py** and **hw01_test.py**.

3. Here's another "classic", to test your branching chops in Python. Within the module **hw01.py**, write a function **grade** which takes a numeric grade as argument, and returns (NOT prints!) the corresponding letter grade **A**, **B**, **C**, **D**, or **F** as its result.

You are **required** to use a **single** if statement in determining the letter grade --- you may **not** use 5 separate if statements. You do not have to check grade's argument for reasonableness, either. (Use the classic grading scale --- >= 90 is an A, >=80 and <90 is B, >=70 and <80 is C, >=60 and <70 is D, and <60 is F).

And, of course, test this function within the **python** interpreter until you are confident that it works.

FOR a 5 POINT BONUS: within your **grade** function, **after** you've determined the base letter

(NOT as part of that determination), **concatenate** a plus or minus to the base letter if it is appropriate, assuming that >= X7 is a plus version of that grade, and >= X0 and < X3 is a minus version of that grade, returning the resulting letter grade as **grade**'s result. Note that you must meet the additional requirements mentioned in #4 below to receive all of the 5 points.

4. Now, we're going to add to **hw01_test.py** to demonstrate that **grade** works, too. To **hw01_test.py**, add the following:

* write a print statement that prints a string that includes the testing call you are about to make and the letter grade you expect it to return, and then print the results of making that call. (Rhetorical question: do you understand why grade's call needs to be part of a **print** statement here, but multiples_in_range's call in #2 did not need to be?)

  Do this for at least 5 different calls of your function (making sure that each possible letter grade results at least once, so that you've tested each branch).

* (**if** you are trying for the 5 point bonus in #3, make sure at least one of those tests results in a grade that ends in +, at least one results in a grade that ends in -, and at least one results in a grade that ends in neither + not -.)

You should, of course, test **hw01_test.py** again to try out your new tests until you are comfortable with it. But, don't create an output file, yet, because there's a bit more to add.

5. Let's conclude HW #1 with a function that takes better advantage of Python's dynamic typing.

First, a note: I strongly suspect that we'll have a more elegant way of testing for this soon, but note that both of the following allow me to test if a value is an integer:

```
val = 3
if type(val) == type(0):
    print "val is an integer!"
else:
    print "val is not an integer!"

val = 7
if str(type(val)) == "<type 'int'>":
    print "val is an integer!"
else:
    print "val is not an integer!"
```

The built-in function **str()** returns a string version of its argument. So, for example, **str(37)** would return **'37'** (the string containing the characters 3 and 7). Also, note that + can be used to concatenate two strings... (but, unlike Java, they really have to be strings --- string + doesn't do automatic string conversion like Java does...)

Keeping the above in mind, in **hw01.py**, write a function **bump_it** that, if given an integer, returns a value that is one more than that integer, if given a floating point number, returns a value that is **.1** more than that floating point number, and if given a string, returns a string that is the same except that it has had an exclamation point concatenated to the end of it. For an argument of any other type passed to it, it should return the argument passed to it.

As always, test this function within the **python** interpreter until you are confident that it works.

6. Finally, add to **hw01_test.py** to demonstrate that **bump_it** works, too. To **hw01_test.py**, add the following:

* write a print statement that prints a string that includes the testing call you are about to make and the value you expect it to return, and then **print** the results of making that call.

  Do this for at least 4 different calls of your function (making sure that you include at least one test each with an integer argument, a floating-point argument, a string argument, and some other kind of argument. (hint: if you add an L to the end of a sequence of digits, you get a literal that is of type long...)

You should, of course, test **hw01_test.py** again to try out your new tests until you are comfortable with it.

And, when you are satisfied, now you should create a submittable output file **hw01_test.out**:

```
python hw01_test.py > hw01_test.out
```

By the due date and time given at the beginning of this handout, use **~st10/480submit** to submit your final versions of **hw01.py**, **hw01_test.py**, and **hw01_test.out**

(And remember --- you can submit more than one version of these before the deadline, if inspiration strikes after a submission. As the syllabus notes, I'll simply grade the latest version that was submitted before the deadline.)