CIS 130 - Intro to Programming - Spring 2007
Homework Assignment #6 - **INDIVIDUAL** assignment

Homework #6 DUE**: BEGINNING** of class, Wednesday, April 4, 2007

Purpose: get practice with sentinel-controlled loops, file input/output, and lists/arrays

Create a file **hw6.py**, and create the functions described below. Paste evidence of testing them into **hw6.txt**.

1. Consider the function **pig_latinize** provided along with this handout; note that it uses some Python features we haven't discussed, but then you only have to call it, not write it... 8-) . You should copy **pig_latinize.py** into the directory where you put **hw6.py**.

Here is **pig_latinize**'s opening comment block, which should be enough for you to be able to call and use it fruitfully:

```
#-----
# Contract: pig_latinize: string -> string
# Purpose: return a somewhat-pig-latin form of <a_word>
# Examples: pig_latinize("apple") == "apple-ay"
#           pig_latinize("pie") == "ie-pay"
#----
```

Write a function **sent_pig** that prompts the user to interactively enter words that are to be pig-latinized until a string of **quit** is entered; after each word is entered, its pig-latin form should be printed to the screen.

To receive full credit, you must:
* use a properly-structured "classic" sentinel-controlled loop in your function
* call pig_latinize appropriately in your function

2. But, why should the poor word 'quit' not be able to be pig-latinized?

Now write function **sent_pig2**, a variation that uses **answer_y_or_n** (you can find it along with the Not-Really-a-Week-8-Lab-Exercise solution on the course Moodle site, under "Some Solutions") so that the user is asked if he/she wants to enter a word to be pig-latinized before being prompted for that word; he/she should only be prompted for a word if he/she answers 'y' to that question. **sent_pig2** then ends when he/she answers **'n'** to that question.

To receive full-credit, you must:
* use a somewhat-classic-sentinel structure in your function
* call answer_y_or_n and pig_latinize appropriately in your function

Rhetorical questions JUST to think about (and I'm not sure the answers are clear-cut!): which of **sent_pig** and **sent_pig2** would you rather use? When and why might you prefer the style of **sent_pig** (the "classic" sentinel-controlled loop)? When and why might you prefer the style of **sent_pig2**? I am asking this to point out that such decisions are part of the **user interface design** of functions...

3. Then again, what if your user already has a **text file** of words he/she wants to pig-latinize, and he/she doesn't really want to type them in by hand?

First, a useful file-input tidbit: what if you don't want a newline on a string you have read in from a file? (Remember, **readline()** includes a newline at the end, much of the time.) In fact, what if you want ANY

<u>leading</u> or <u>trailing</u> white space (including newline characters) to be chopped off a string?

* To get a version of a string with such white space removed, call the string's **strip** method; here's an example:

```
>>> orig_string = "   hello   \n"
>>> new_string = orig_string.strip()
>>> print "<" + orig_string + ">"
<   hello
>
>>> print "<" + new_string + ">"
<hello>
```

Keeping the string **strip()** method in mind...(and being CAREFUL about when you call it...)

...write a function **file_pig** that expects a file name as its parameter; it is allowed to simply assume that this file exists in its current working directory and is open-able and read-able, and that it contains one word per line. It should open this file and read its contents, trying to pig-latinize each word in that file, and printing the pig-latinized version to the screen.

(Note: your function does not create the input file; it just uses it. When you want to test **file_pig**, you will want to create an example input file with **pico** or the text editor of your choice.)

4. What if you'd like interactive input, BUT you don't want the pig-latinized words to be printed in-between the prompts, but instead all in one "block" of lines at the end?

(That is, you don't want something like:

```
>>> Enter next word: apple
apple-ay
>>> Enter next word: core
ore-cay
>>> Enter next word: ...
```

...but instead you want something like:

```
>>> Enter next word: apple
>>> Enter next word: core
>>> Enter next word: ...
apple-ay
ore-cay
...
```

)

Do you see that you could store the words entered into a **list/array**, and then when all are entered "walk through" that list/array's contents, calling pig_latinize for each?

Write function **list_pig**, which expects as its one parameter the number of words to be entered. It should request that many words, appending each to an list/array; it should then loop through the contents of the resulting list/array, printing to the screen the result of calling pig_latinize for each, one result per line (so that the results are indeed printing all in a block at the end, not interleaved with the word-prompts).

5. Now that we have lists/arrays, it is occasionally useful to have functions that return lists/arrays.

Consider - what if something like **file_pig** created and returned a **list** of the pig-latinized words, instead of printing them to the screen? Then you'd have a function that might be a handy auxiliary function for use in other functions.

So, write function **file_pig2** that expects a file name as its parameter, and it too is allowed to simply assume that this file exists in its current working directory, is open-able and read-able, and contains one word per line. It still opens this file, reads its contents, and pig-latinizes each word in that file, BUT instead of printing the result to the screen, it appends it to a list, and then **returns** the resulting list.

So, if a file **word_list.txt** contains:

```
apple
core
pie
betty
cobbler
turnover
```

...then **`file_pig2("word_list.txt") == ['apple-ay', 'ore-cay', 'ie-pay', 'etty-bay', 'obbler-cay', 'urnover-tay']`**

6. And, we need some file **output** now.

Consider: Python provides some list-goodies that C++ doesn't for arrays, as we have discussed; another example of this is Python's **sort** method for lists:

```
>>> list1 = ["zed", "gamma", "beta", "alpha"]
>>> list1.sort()
>>> list1
["alpha", "beta", "gamma", "zed"]
```

Now, this doesn't always work as well as the above implies -- you can see what I mean if you try to sort a list containing strings and numbers, or containing strings where some begin with uppercase letters and some begin with lowercase letters. (And there *are* Python ways around this issue, but they're outside of scope of CIS 130... 8-)) But, in the meantime, this is still a handy method.

Write a function **sorted_pig** that expects two parameters, the name of an input file expected to contain one word per line, and the name of the desired output file to be created. (Your function may assume that the input file is in the current working directory, is open-able and read-able, and contains one word per line; and, it may assume that it is OK to delete any pre-existing contents in the desired output file, if the output file exists when **sorted_pig** is called.)

Your function should use **file_pig2** to create a pig-latinized list of the words from the input file; it should sort the list and print the sorted result to the output file, one word per line.

When you are happy with your files **hw6.py** and **hw6.txt**, type the following command at the cs-server prompt:

~st10/130submit

Then follow its directions to submit your files.