The University of Chicago
Department of Computer Science

CMSC 15200 – Introduction to Computer Science 2
Summer Quarter 2007
Lab #4 (08/15/2007)

Name:	

Student ID:		*Lab Instructor:*	Borja Sotomayor

Do not write in this area

1	2	3	4	**TOTAL**

Maximum possible points: 30+ 10

Using Python in the Maclab

Before doing the exercises, verify that you can use the Python interpreter, interactively or by providing a file of Python code. The CS machines have several versions of Python installed, and the default version is 2.4. We will be using Python 2.5, which is available in the following path:

```
/opt/python/python-2.5/bin/python2.5
```

To avoid having to write the entire path, you can modify the PATH environment variable, to make sure your UNIX shell always looks for executables in the above directory.

```
export PATH=/opt/python/python-2.5/bin:$PATH
```

Now you should be able to run Python by running **python2.5**. To verify that you're using the correct version of Python, look at the version number printed when you run the interpreter:

```
Python 2.5 (r25:51908, Feb 23 2007, 12:08:08)
[GCC 3.3.4 (Debian 1:3.3.4-12)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Try to run some basic Python code from the interpreter:

```
>>> fibo = [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
>>> for f in fibo:
...     print f
...
1
1
2
3
5
8
13
21
34
55
```

Now create a file called `fibo.py` with the following contents:

```
def fib(n):
      if n in [1,2]:
             return 1
      else:
             return fib(n-1) + fib(n-2)

fibo = [ fib(x) for x in range(1,11)]
for f in fibo:
      print f
```

Run the file like this:

```
python2.5 fibo.py
```

You should see the first 10 Fibonacci numbers (as in the previous example).

Note that, in a UNIX system, you can make the Python code file itself an executable. First, add the following as the first line of your Python file:

```
#!/opt/python/python-2.5/bin/python2.5
```

Next, mark the file as executable:

```
chmod u+x fibo.py
```

Now, you can run the Python file as an executable, without having to explicitly invoke the Python interpreter:

```
./fibo.py
```

Exercise 1 <<10 points>>

You must write the following function:

```
def clean(s):
      ...
```

The **clean** function expects an "unclean" string of characters. For the purposes of this exercise, an "unclean" string is one where random numbers have been inserted into the words and the case of letters has been jumbled. For example:

```
T87Hi122s 9827i8S1 a82n UN29832c98L98ea156N 8879sTrI978n98g
```

We need to clean this string up by doing the following:

- Remove all numbers from the string
- Lowercase all the letters
- Split the string into the individual words (i.e., we want a list of strings, were each string is a word from the original string). You can assume that words are separated by a single space.

The function must return the list of strings resulting from these three steps. So, the "clean" version of the above string would be:

```
this is an unclean string
```

Or, in a more pythonic notation:

```
[ 'this' , 'is' , 'an' , 'unclean' , 'string']
```

You must implement the function in a file called **unclean.py**. The instructor must be able to test your implementation like this:

```
borja@classes:~$ python2.5
Python 2.5 (r25:51908, Feb 23 2007, 12:08:08)
[GCC 3.3.4 (Debian 1:3.3.4-12)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> s = “T87Hi122s 9827i8S1 a82n UN29832c98L98ea156N 8879sTrI978n98g”
>>> import unclean
>>> unclean.clean(s)
[ 'this' , 'is' , 'an' , 'unclean' , 'string']
```

Exercise 2 <<10 points>>

Implement Exercise 2 from Homework #6 using Python (merging two files of sorted numbers, possibly containing repeated values in the two files). You can use the same sample files provided for Homework #6. The instructor must be able to run your Python script like this:

```
python2.5 merge.py file1 file2 result
```

To do this exercise, you will need to familiarize yourself with Python file I/O, which has not been covered in class. However, since you already know about file I/O with C++, picking up the nuances of file I/O in Python shouldn't be hard. You can read section 7.2 of the Python Tutorial to find out how to open/close/read/write files.

Exercise 3 <<10 points>>

Write a Python program which, given the name of a file, will determine if it is a ZIP file or not. If it is, the program must print out the names of the files contained inside the ZIP file.

For example:

```
borja@classes:~$ python2.5 zip.py list.cpp
list.cpp is not a ZIP file

borja@classes:~$ python2.5 zip.py list.zip
list.cpp is a ZIP file. Its contents are:
list.cpp
list.h
main.cpp
```

To do this exercise, you will have to find a module from the Python Standard Library that allows you to manipulate ZIP files. Note that this module also provides a function to determine if a file is a ZIP file (do *not* determine this by checking if the extension of the file is ".zip").

Exercise 4 <<Extra credit: 10 points>>

Write a postfix notation (also called Reverse Polish Notation) arithmetic evaluator. Your program must ask the user for a postfix expression, evaluate it, and then allow the user to write another expression. The program will run infinitely (i.e., don't include a "Do you want to enter another expression?" prompt).

For example:

```
borja@classes:~$ python2.5 eval.py
Type your expression: 3 5 +
8
Type your expression: 2 3 + 7 *
35
Type your expression: 2 3 + 4 5 + *
45
```

You are allowed to assume that operands and operators are separated by a single space. You only need to support the addition, subtraction, multiplication, and division operators.

How to evaluate a postfix expression was (briefly) discussed in class. Hint: You will need to use a stack.