CSC104, Assignment 2, Summer 2006
Due: Thursday June 29\textsuperscript{th}, 11:59 pm

Anastasia Bezerianos, Danny Heap
June 13, 2006

\textbf{Finger Exercises}

For this assignment we will be (mainly) working with the programming language Python. The exercises in this section are meant to guide you through learning how to use enough Python to work on the course Pre-req.py and docStats.py programs at the end of this assignment. As always, keep track of things you try (whether they work or not), in a journal.

1. Create a directory called A2 (you might want to look back at journalA1 to remember how to do this). Make A2 your current working directory, using the \texttt{cd} command. Start the editor \texttt{SciTe}, and begin a journal entry of your work on assignment 2 with the current date. Save this file as journalA2. Summarize all your work on assignment 2 in this journal, structured by exercise number and date. Try to follow the problem solving approach taught in class.

2. You will be using the programming language Python. We will introduce some basic concepts in these exercises, and you may certainly pose questions to the course wiki pages. There is also a tutorial on using Python at: http://docs.python.org/tut/tut.html, but you may well find that it is aimed at people with some previous programming experience. The same is true of the help available by typing (of all things) help once you have started up python.

Start out by opening a terminal where you can type commands (as you did for \texttt{ls} and \texttt{grep} in assignment 1). Now type \texttt{python}, and then press enter. You should see something like:

\texttt{Python 2.4.1 (#2, Aug 11 2005, 16:44:28)}
\texttt{[GCC 3.3.3] on linux2}
\texttt{Type "help", "copyright", "credits" or "license" for more information.}
\texttt{>>>}

The \texttt{>>>} is indicating where you can type some expression that python will try to understand. Try typing something arithmetic, for example \texttt{1+2}, and then press enter. Addition and subtraction expressions should take a familiar form. Multiplication uses the * character, and division uses /. Try dividing 12 by 4. Now try dividing 11 by 4. Also, try dividing 11 by 4.0 - that's 4, followed by a decimal point, followed by a zero. Finally, try dividing 11*1.0 by 4. Explain your results in your journal. When you have had enough, hold down the "Ctrl" key with one finger, and then press the "D" key.
3. In assignment 1 you saw that you could store numbers, text, and even formulas in cells, and then refer to those cells by name: A2 or B3. In Python you are given the freedom to decide on how to name "cells", provided you choose a name that begins with an alphabetic character and contains only alphabetic characters, numerals, and the underscore character (uppercase/lowercase is important).

Start up Python (see the previous exercise), and after the >>> type favouriteNumber, and then press the enter key. You should see an error message. Now type favouriteNumber = 8, press the enter key, then type favouriteNumber and press the enter key again. Now store a different number (not 8 again) in a location named secondFavouriteNumber. Type favouriteNumber + secondFavouriteNumber, then press enter. What is stored in favouriteNumber and secondFavouriteNumber?

Here's something a little tricky to puzzle over. Type secondFavouriteNumber = favouriteNumber and press enter. Now check what is stored in favouriteNumber and secondFavouriteNumber (you found out how to check what is in a previous location in the exercise). Now type favouriteNumber = 20 and press enter. Re-check what values are in the locations named favouriteNumber and secondFavouriteNumber. Remember to record all your observations in journalA2.

4. You can also store text, or strings of characters, in Python. After the >>> type myWord = 'me', then press enter. The single quotes (or apostrophes) around 'me' are important, to tip Python off that this is a string of characters, and not the name of some location, such as favouriteNumber. Now store the text 'you' in a location named yourWord. Check what each location contains. Try typing myWord + yourWord, and then pressing enter. Sorry, that's all the arithmetic you can do with text!

5. After the >>> type myNums = [1, 3, 5, 7], then press enter. Check what is contained in the location named myNums. Now type myNums[0], then press enter. What about myNums[1] or myNums[3]? Type myNums[0:3], then press enter. Explain your results as fully as you can.

You can also make lists of words. After the >>> type myWords = ['One', 'Ring', 'to', 'rule', 'them', 'all'], then press enter. Check what the location named myWords contains. Type fewerWords = myWords[1:3] and see what the location named fewerWords contains. Also check what the location named myWords contains. Experiment until you can explain what's going on.

6. Try typing oneWord = 'gollum', then press enter, followed by fewerWords + oneWord, then press enter. Type fewerWords.append('hobbit'), then press enter, and check the value of stored at the location named fewerWords.

Try typing anotherWord = ['elf'], then press enter, followed by fewerWords.append(anotherWord) and check the value of fewerWords.

Then type finalWord = ['dwarf'] and then press enter. What does fewerWords + finalWord give? Check the value of stored at the location named fewerWords. Then type moreWords = fewerWords + finalWord and see what is the value of moreWords.

You can also try len(fewerWords) and len(moreWords).

Type moreWords = moreWords * 2, then press enter. Check what is stored at the location called moreWords. At this point, you might want to chat with a TA or your instructor about what's going on. All observations should of course go in journalA2.
7. With lists you could get an element if you knew its position, for example, in Ex. 5 myNums[0] held the element '1'. Sometimes it is more convenient to access elements without knowing their position, but some other key. Try typing `color = {'apple': 'red', 'banana': 'yellow', 'pear': 'green'}`, then press enter. Now type `color['apple']`, then press enter. Try some other fruit.

In the example above, color is a look-up table: we look up the character string for a color by providing the character string for a fruit. Look-up tables are even more flexible than that. After the `>>>` type `courseTable = {'CSC369': 'CSC258', 'CSC209', 'CSC207'}` and then press enter. In the look-up table `courseTable` you can retrieve the list of prerequisite course codes ['CSC258', 'CSC209', 'CSC207'] using the key 'CSC369'. Try typing `courseTable['CSC369']` and then pressing enter.

8. Time for some repetition. After the `>>>` type: `numberList = [1, 3, 5, 7, 9]` and then press enter. Type `sum = 0` and then press enter. Type `nextNumber` and then press enter (what do you see?). Now type the following, being sure to leave the same spaces on the left-hand side as shown below (press enter twice to get back to the left-hand margin after the . . .):

```python
>>> for nextNumber in numberList:
    ...     print 'nextNumber = ', nextNumber
    ...     sum = sum + nextNumber
    ...     print 'sum = ', sum
```

You may want to chat with a TA or instructor about this. A key observation is that all three lines that are indented under for `nextNumber` in `numberList` are repeated.

If you're comfortable with the previous example, try typing `wordList = ['my', 'dog', 'has', 'fleas']` and then press enter. Just to convince yourself that there is no location (yet) named `nextWord`, type `nextWord` and then press enter. Type `numChars = 0` and then press enter. Now type the following, being sure to leave the same spaces on the left-hand side as shown below (press enter twice to get back to the left-hand margin after the . . .). The . . . are generated automatically by Python, just as the `>>>` are, so you don't type them.:

```python
>>> for nextWord in wordList:
    ...     print 'nextWord is', nextWord
    ...     numChars += len(nextWord)
    ...     print 'numChars is', numChars
```

Try to explain as best as you can what happened in all the previous examples.

9. Let us make things more interesting by combining the things we learned about dictionaries and repetition using "for". Type `coursesToTake = ['CSC104', 'CSC324', 'CSC465', 'CSC104', 'CSC465']`. Then type the following:

```python
>>> courseDictionary = {}
>>> for nextCourse in coursesToTake:
    ...     courseDictionary[nextCourse] = nextCourse
```

```python
3
```
Then type coursesToTake2 = courseDictionary.keys(). Look at what is stored in courseDictionary, coursesToTake and coursesToTake2 and try to explain it.

10. Exit Python (by holding down the Ctrl key and pressing the letter D). Make sure that A2 is the current working directory (type pwd and then press enter to verify this). Now type cp -hp /pub/cscCourseList.txt . and press enter. This copies the text of a list of CS course prerequisites to your directory A2. You can examine it with the text editor Scite. The first word in every row is a course number, and everything after the "P:" are course numbers of its prerequisites. Now we'll do a function definition. Type the following, being sure to leave the same spaces on the left-hand side as shown below. Press enter twice to get back to the left-hand margin after the . . . (the . . . are generated automatically by Python, just as the >>> are, so you don't type them).

```python
>>> def readSomeCourseLines(numLines):
...     courseFile = file('cscCourseList.txt', 'r')
...     for nextLine in courseFile:
...         numLines = numLines - 1
...         print nextLine
...         print nextLine.split(), '
' * 2
...         if numLines <= 0: break
...     courseFile.close()
```

After the >>> type readSomeCourseLines(5). Of course you don't need to stop at 5. This creates a function (program) called readSomeCourseLines and the number you type in the parenthesis is stored in the location named numLines for the indented lines of Python. What is the difference between what's stored in nextLine and nextLine.split() at each step?

That's probably plenty of exploring for now. The next two sections explain the motivation for the prerequisite and doc modules respectively, and how to complete the files prerequisite.py and doc.py so that you can run the modules.

**Course Prerequisites**

You are checking the list of Computer Science courses and see the course "CSC324 Principles of Programming Languages". You are immediately intrigued and wonder what are the course prerequisites for taking that course. You flip through the student handbook and see you need to take at least three other courses. Each of them seems to have their own prerequisites. You check these out and they also have prerequisites. At this point you have forgotten what were the intermediate courses you had to take and need to start all over again . . . Wouldn't it be great if you could ask someone to give you all the courses you ever had to take in order to register to CSC324 so you can plan accordingly? Well, now you can!

Here is how the program will work. In a previous exercise we copied the file cscCourseList.txt to your home directory. If you haven't seen this file already, you can examine it with the text editor scite. The first word in every row is a course number, and everything after the "P:" are course numbers of its prerequisites. Our program will go through the entire file and create a dictionary connecting each course with a list of its prerequisite courses.

Once your program has recorded all the possible courses, we begin our course hunting adventure by starting out with a desired course number and looking for its prerequisites. We print those out and then look for their prerequisites. We print those out as well and look for their prerequisites in turn, and so on, until we reach a point where there are no prerequisites left (some first year courses). We now have printed every possible course you ever had to take to be able to register in your desired course.
YOUR CONTRIBUTION TO COURSE PREREQUISITES

First of all emulate the finger exercise for copying cscCourseList.txt to your A2 directory, to copy prerequisite.py to your A2 directory.

Change your current working directory to A2. In your terminal window type idle prerequisite.py and then press enter. Two windows will pop up: one where you can edit the definitions of functions buildCourseTable and getPrerequisite, the other where you can experiment with your creations. You can switch between these windows by clicking them with your mouse.

The function definitions for buildCourseTable and getPrerequisite are incomplete. Above each function definition are a few lines of text beginning with a single # character that describe the function that follows. These are comments: python ignores the line following the # character, but human readers (you, for example) should not.

The first line of each function definition gives it a name (for example buildCourseTable, and a set of parameters. When somebody uses the function they store actual values (e.g. numbers or character strings) in the locations named by those parameters.

The remaining lines of each function definition are indented. Some of the lines beginning with a double ##. These are comments that tell you what python code you need to put immediately underneath the ##. Make sure the code you put in lines up horizontally with the left end of the ## (you should be able to reach this with tabs).

If you complete the definitions correctly (you’ll probably want to review the finger exercises a lot, and ask your TA and instructor lots of questions), you have a working duo of prerequisite functions. Switch to the other window and try:

>>> import prerequisite
>>> courseTable = {}
>>> prerequisite.buildCourseTable( 'cscCourseList.txt', courseTable)
>>> prerequisite.getPrerequisites( 'CSC324', courseTable)

If your functions are working, you’ll get several rows of text. The first will be the code of the course you want the prerequisites of (in this example ‘CSC324’). The next line will be indented and will contain all prerequisites of ‘CSC324’. The third line will be further indented and contain all the prerequisites of the second line (that is all the prerequisites of the prerequisites of ‘CSC324’) and so on. Otherwise, you can return to the function definitions, fix things up, type F5 and click save, return to the window for experimentation and repeat the above commands. Whatever happens, record your observations and explanations in journalA2. If there is some part of the task that is stumpng everyone, we will consider giving some judicious hints.

In general it is a good practice to check the contents of courseTable and see what you have stored there. It should be a dictionary with entries like the second example in Ex 7.

Note that our program does not deal with cases of OR (alternative) course prerequisites (it becomes really complicated to display all such courses).

Doc

Have you ever noticed how some writers tend to write longwinded prose, with big words, while others tend to use simple words and short sentences? Wouldn’t it be nice to be able to compare different types of prose based on their sentence number, sentence length and word length and decide if there is a type of written literature you prefer? Or to see if the style of writing of an author changes between her different books? The doc module will allow you to get statistics on different aspects of a written text.
Here is how the program will work. We will count the paragraphs in the text. For each paragraph, we will count the number of full sentences and clauses (more on this later). For each sentence we will count the number of clauses, for each clause the number of words and for each word the number of characters. We will make sure we sum up all this information and at the end we will print the number of paragraphs, sentences, clauses, words and characters, as well as the average number of sentences per paragraph, clauses per sentence, words per clause and characters per word.

We mentioned before that each paragraph will be split into full sentences and each sentence split into clauses. We split sentences generally when we find one of . ? ! or a combination of these punctuation marks followed by any type of quotes (otherwise we would be counting these quotes as part of our words). Full sentences are usually followed by one or more spaces.

Similarly we can split clauses when we find the , ; punctuation marks or a combination of these and any type of quotes. Clauses are also followed by one or more spaces. You will see the above splitting criteria expressed as regular expressions in the doc.py file.

There is the question as to how to split text into paragraphs. In the text we’re using paragraphs are separated by \r\n\r\n
YOUR CONTRIBUTION TO DOC

First of all emulate the finger exercise for copying oscCourseList.txt to your A2 directory, to first copy alice30.txt, and then doc.py to your A2 directory. The file alice30.txt is actually a copy of Lewis Carroll’s “Alice in Wonderland”.

Change your current working directory to A2. In your terminal window type idle doc.py and then press enter. Again two windows will pop up: one with the definition of the function doc, the other where you can experiment with your creations. You can switch between these windows by clicking them with your mouse.

The function definition for doc is again incomplete. As before, above the function definition are a few lines of text beginning with a single # character that describe the function that follows. Again the ## comments tell you what python code you need to put immediately underneath the ##. Don’t forget that your code needs to line up horizontally with the left end of the ## (you reach this with tabs).

If you complete the definitions correctly (you’ll probably want to review the finger exercises a lot, and ask your TA and instructor lots of questions), you have a working report function. Switch to the other window and try:

```python
>>> import doc
>>> doc.report('alice30.txt')
```

If your functions are working, you’ll get several rows regarding the alice30.txt file. The first five rows will give you the number of paragraphs, sentences, clauses, words and characters in your text. The next four lines will be the average number of sentences per paragraph, clauses per sentence, words per clause and characters per word for the alice30.txt file. Otherwise, you can return to the function definitions, fix things up, type F5 and click save, return to the window for experimentation, and repeat the above commands. Whatever happens, record your observations and explanations in journalA2. Again, for parts of the task that are stumping everyone, we will consider giving hints.

For this section you had to copy "Alice in Wonderland" from the instructor’s home directory to your directory A2, in order to experiment with your program. You may also download lots of public-domain literature from http://www.gutenberg.org.
WHAT TO HAND IN

Under Assignments on the course web page you will find a link to the CDF submit facility. Submit the following files:

- journalA2
- prerequisite.py
- doc.py

You should submit your files early and often. The first time you create a file with meaningful content, submit it. You may re-submit the same file as many times as you wish, and only the last submission is stored. A good habit is to re-submit your files each time you improve them.