CSC104, Assignment 2, Summer 2006
Due: Thursday June 29th, 11:59 pm

Anastasia Bezerianos, Danny Heap
June 30, 2006

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 cd command. Start the editor 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.

   Solution: I have done this before, so I go over my notes from A1 and create a new directory.
   
   werewolf:~% mkdir A2
   werewolf:~% cd A2/
   werewolf:~% A2/

2. You will be using the programming language Python. We will introduce some basic concepts in these exercises, and you may certainly post 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 ls and grep in assignment 1). Now type python, and then press enter. You should see something like:

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

   The >>> is indicating where you can type some expression that python will try to understand. Try typing something arithmetic, for example 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.

**Solution:** I first checked if indeed arithmetic operations as I know them work in the python environment.

```python
>>> 1+2
3
>>> 7-4
3
>>> 4-7
-3
>>> 4*2
8
>>> 4*(3+2)
20
```

This is great! Negative numbers and parenthesis also work. I can use python as my calculator ...

I then checked division as the assignment suggested:

```python
>>> 12/4
3
>>> 11/4
2
>>> 11/4.0
2.75
>>> 11*1.0/4
2.75
```

Dividing 12 by 4 gave (not surprisingly) 3. Dividing 11 by 4 gave 2. That doesn't seem right. Dividing 11 by 4.0 gave 2.75, which is exactly right. Also multiplying 11 by 1.0 and then dividing by 4 gives the correct result.

It seems as though when I divide an integer by an integer, I get an integer result. This is a bit like elementary school math: "11 divided by 4 is 2 with a remainder of 3." But where's the remainder? Oh, the prof said in lecture that I could get that using the % operator, but there was some problem with negative

I wanted to also check if regular division happens no matter which of the two numbers is expressed as a having a decimal place:

```python
>>> 11.0/4
2.75
>>> 12.0/4
3.0
```
I couldn’t wait to see the result of division by 0 (computers freak out with that)

```python
>>> 11/0
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
```

Yep, python freaks out appropriately.

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?

**Solution:** Before I appended ",=8", favouriteNumber was undefined. I was a bit worried that it was still undefined after I typed favouriteNumber = 8, but then I noticed that I had typed a lowercase "n" (as far as Python is concerned) is a different identifier. So now favouriteNumber contains 8, whereas favouriteNumber is undefined. I fixed things up by typing favouriteNumber = 8, and then I typed secondFavouriteNumber = 9 (being very obsessive about upper/lower case).

Now when I type favouriteNumber + secondFavouriteNumber I get 17, just as if I had added the numbers stored at those identifiers. favouriteNumber is still 8 and secondFavouriteNumber is still 9. So the addition gave me a result, but did not change the two numbers in any way.

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.

**Solution:** Weird. After I typed secondFavouriteNumber = favouriteNumber both identifiers contained 8, so I figured that now secondFavouriteNumber was just a nick-name for favouriteNumber. However that theory fell apart when I then typed favouriteNumber = 20: now secondFavouriteNumber still contains 8, but favouriteNumber contains 20. I asked the TA, and she suggested the following analogy. You’ve got a messy desk drawer full of numbers, and a couple of post-it notes, with favouriteNumber and secondFavouriteNumber written on them. You open the drawer and stick the favouriteNumber post-it on the number 8. You open the drawer again, and stick the secondFavouriteNumber post-it on whatever number the post-it favouriteNumber is on. You open the drawer a third time, and you move the favouriteNumber post-it to the number 20. The analogy works, but I’m starting to talk to myself and wake up in the middle of the night dreaming about being chased by post-it notes.

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!

SOLUTION: It seems that myWord = 'me' stored the string 'me' at the location myWord. This works equally well (I found out by mistake) if I use double quotes: myWord = "me". I also stored the string 'you' at the location (it seems strange to call a word a location) yourWord, and when I typed yourWord, out popped 'you'. Now I perform "addition" and myWord + yourWord gives 'meyou' (a bit as though python swallowed a cat). So string addition appends one string to another. As advertised, myWord - yourWord generates an error. Playing around with myWord I found out that there is another arithmetic operation that works: myWord*2 gives the egotistical 'meme'.

Here is the output of my terminal:

>>> myWord = 'me'
>>> yourWord = 'you'
>>> myWord
'me'
>>> yourWord
'you'
>>> myWord + yourWord
'meyou'
>>> myWord-yourWord
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: unsupported operand type(s) for -: 'str' and 'str'

My identifier myNums labels the list [1, 3, 5, 7] (I'm "sticking" with the post-it note analogy, for now). I can extract elements of the list, which (for some reason) begin with element 0, so myNums[0] is the first (zeroth?) element: 1. The second element is at myNums[1]: the number 3. The, uh, fourth element is at myNums[3], the number 7.

By putting a colon into the brackets, I get a list of the consecutive elements from element 0 up to (but not including) element 3: [1, 3, 5]. Originally I figured this was the same as "the three consecutive elements, starting at element 0," but myNums[1:3] returns [3, 5] scratched that theory. The way I eventually figured it out was by trying several combinations of numbers before and after the colon, and seeing the unexpected results of myNums[3:1] (an empty list).

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.

SOLUTION: I labeled the list ['One', 'Ring', 'to', 'rule', 'them', 'all'] with my identifier myWords, and now when I type myWords python responds with the list. The elements can be accessed by
their position, in much the same manner as numbers, so when I label the consecutive elements of
myWords from 1 up to (but not including) 3 with fewerWords, I find that fewerWords labels the
list [‘Ring’, ‘to’]. When I check out myWords it still contains [‘One’, ‘Ring’, ‘to’, ‘rule’, ‘them’,
‘all’], so it is unchanged by that operation with the brackets and the colon.

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.

SOLUtion: I labelled the string ‘gollum’ with the identifier oneWord, but then when I typed few-
erWords + oneWord, I got an error saying that I cannot ”concatenate” a string with a list.
That bothered me, but I continued to type fewerWords.append(‘hobbit’), and suddenly the list
labelled by fewerWords had a new element: ‘hobbit’. I guess this is the prof’s heavy-handed way
of showing that you need this ”append” incantation to add things to a list.

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).

SOLUtion: I then labeled the string ‘elf’ with the identifier anotherWord=[‘elf’]. I noticed the brackets
around the word [‘elf’]. It seems that this anotherWord is actually a list containing one word.
After I type fewerWords.append(anotherWord) I see that fewerWords actually contains [‘Ring’,
‘to’, ‘hobbit’, [‘elf’]]. The word ‘elf’ seems to be inside a list (a list inside a list, this is wild!).
After I typed finalWord = [‘dwarf’] and fewerWords + finalWord, I no longer get an error the same
way I did when I tried to add ‘gollum’ before. I get [‘Ring’, ‘to’, ‘hobbit’, [‘elf’], ‘dwarf’],
and I notice that there are no longer brackets around ‘dwarf’. It seems that the brackets that
put the word ‘dwarf’ in a list (labeled finalWord) allow it to be ”added” to another list. Unlike
append, the brackets get removed and only the contents of the list finalWord are added to the list.

When I check the contents of the list fewerWords the word ‘dwarf’ is nowhere to be seen. It seems
that unlike append the addition does not change the original list (in the same way that adding
favouriteNumber + secondFavouriteNumber in Ex 3 did not change what is favoriteNumber). So
the last part of the exercise (moreWords = fewerWords + finalWord) is there to label the result
of the addition. Now I have:

>>> fewerWords
[‘Ring’, ’to’, ’hobbit’, [‘elf’]]
>>> moreWords
[‘Ring’, ’to’, ’hobbit’, [‘elf’], ‘dwarf’]

When I type len(fewerWords) and len(moreWords) I get the length of the two lists, so 4 and 5
respectively.

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.
**Solution:** Now there's more list arithmetic: moreWords = moreWords * 2 first "doubles" the list labelled by moreWords (it adds a copy of itself to the end), and then labels that new list with moreWords itself. So now moreWords labels ['Ring', 'to', 'hobbit', 'elf', 'dwarf', 'Ring', 'to', 'hobbit', 'elf', 'dwarf'] (I wonder where Gandalf is ...)

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.

**Solution:** When I type color['apple'] I get 'red', and (similarly), when I type color['banana'] I get 'yellow'. It seems like I'll be able to match colors to fruit. I can retrieve things on the left side of the colon in color by providing the things on the right side of the colon.

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.

**Solution:** As expected, I get back ['CSC258', 'CSC209', 'CSC207'] when I type courseTable['CSC369']. I'm not sure why I'd want this, but there it is. I also checked out what's in courseTable, and it contains 'CSC369': ['CSC258', 'CSC209', 'CSC207'] — the thing to the left of the colon is a string, and the thing to the right that I can access using the thing on the left is a list. The prof showed us in lecture that you can't switch things around - you can't look up the key string by using the list, and you can't even create a lookup-table where lists are on the left-hand side of the colon (you get an error about lists not being "hashable").

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.

**Solution:** I typed the python code I was asked to, and the got back nextNumber = 1, followed by sum = 1, and then nextNumber = 3, followed by sum = 4, followed by nextNumber = 5, followed by sum = 7, followed by nextNumber = 7, followed by sum = 16, followed by nextNumber = 9, followed by sum = 25. Staring at the code I typed, I figure that the label nextNumber was applied to each number in numberList in turn (which explains how they each get printed out. sum begins at zero, and then has, in turn, 1, 3, 5, 7, 9 added to it, and those intermediate sums are printed out: 1, 4, 9, 16, 25.
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` 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
```

**Solution:** I typed what was asked, and found that, indeed, `nextWord` was not, at first, defined. After I typed the for loop, I got back `nextWord` is “my”, followed by `nextWord` is “dog”, followed by `nextWord` is “has”, `nextWord` is “fleas”. So the label `nextWord` is put on each string in `wordList`, and then that label is used to print out what the string is. I initially made a mistake in indentation, and got a message nagging me about it. As with the sum example above, another sum is calculated here at each step in the for loop. It is the sum of the characters of all the words. So the `numChars` was initially 0, it became 2 for “my”, 5 for “my” and “dog” etc. The label `numChars` correctly points to the value 13 (I verified this is correct by counting all the characters myself). That “+=” in `numChars += len(nextWord)` gave me some grief at first, but the TA said it is a fancy way of saying `numChars = numChars + len(nextWord)`.

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
```

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

**Solution:** I know from previous exercises that the for loop goes through all the elements in the `coursesToTake` list. It seems that for each one of the courses in that list, I create an entry in the dictionary where both the key and the value it points to is the same, the course itself. This really seems silly. I check `courseDictionary` and not surprisingly I see courses pointing to themselves. But wait, something is going on. I have 3 entries in the `courseDictionary`, but I had 5 courses in `coursesToTake`. I look at `coursesToTake` carefully and see that some courses are actually repeated. These double courses are not present in the `courseDictionary`. This might have something to do with dictionaries not having double keys (even if their order is a bit weird). Neat! The `coursesToTake2 = courseDictionary.keys()` command seems to take all the keys of `courseDictionary` (all items before the “,” ) and put them in a list that is labeled `coursesToTake2`. So now `coursesToTake` still had duplicate entries, but `coursesToTake2` has no duplicate courses (a good thing too, I would hate to repeat a course by accident!).
10. Exit Python (by holding down the Ctrl key and pressing the letter D). Make sure that A2 is the current working directory (type \texttt{pwd} and then press \texttt{enter} to verify this). Now type \texttt{cp -r} \texttt{pub/cscCourseList.txt} ... and press \texttt{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. \texttt{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 \texttt{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).

\begin{verbatim}
>>> def readSomeCourseLines(numLines):
...     courseFile = file('cscCourseList.txt', 'r')
...     for nextLine in courseFile:
...         numLines = numLines - 1
...         print nextLine
...         print nextLine.split(), '\n' * 2
...         if numLines <= 0: break
...     courseFile.close()
\end{verbatim}

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

**Solution:** I had some trouble with copying the file to my home directory, I was messing up the spaces in the command I guess. The comment by the instructor clarified things. I got a bit confused as to where to type the definition but I finally understood that I needed to run python again.

After numerous complaints that I had messed up the indentation (I had), or "invalid syntax" (the TA pointed out that I had missed the colon), I finally got some output. I get a line of text from the course prerequisite file, followed by the individual words (strings) put in a list. I would get as many lines/strings as the number I put in the parenthesis for \texttt{readSomeLines(...)}. It looks as though \texttt{nextLine} holds a string, but \texttt{nextLinesplit()} breaks that string up into a list of strings, making new entries where there is a space character. It certainly looks clumsy.

It seems as though \texttt{courseFile} ends up accessing the file \texttt{'cscCourseList.txt'}, so I suppose the command file does that. The TA confirmed my conjecture, and pointed out that the 'r' meant that \texttt{courseFile} was allowed to read things from the file. Whatever number I type between the parentheses gets stored in \texttt{numLines} and this is decreased by 1 each time the loop repeats, until finally it will be no more than 0, when the loop breaks. I guess I close the file out of tidiness at the end.

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 \texttt{prerequisite.py} and \texttt{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 `vi`. 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:

```python
>>> 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 stumping 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).

**SOLUTION:** Copying prerequisite.py was identical to copying cscCourseList.txt, except replacing cscCourseList.txt with prerequisite.py. Same, only different. The two windows popped after I typed idle prerequisite.py, but the font was unreadably small. I finally managed to choose a larger font under the "Options" menu.

Since I had been previously burned by bad indentation, I was extremely careful to make my lines have the same indentation as the ## symbols. A lot of the lines to fill in were similar to previous exercises, for example level = level + 1 was pretty similar to sum = sum + nextNumber. There were a couple of weird ones, though.

The comment " ## add to the elements of the list newPre, the elements of the list addPreCourses. Make sure you don't get lists inside lists " was followed by the python command: newPre = newPre + addPreCourses. That's exactly the command I think I was supposed to fill in! The prof said in class that he had inadvertently left in one line of his solution (no extra charge, I guess).

I am really glad I understood all the indexing and getting elements from lists exercises because I had to use them a few times.

Finally I got it working (see below). When I typed prerequisite.buildCourseTable( 'cscCourseList.txt', courseTable), I got a dictionary in courseTable full of entries of the sort seen in Ex 7. In getPrerequisites('CSC324',courseTable) the program finds the entry of the dictionary courseTable that has 'CSC324' as a key. When it finds it it keeps all the stuff the key points to (the prerequisites) in a list called Pre and prints them one by one. It also looks for the prerequisites of these Pre entries and stores them in newPre. After it has gone through all the entries in Pre, it replaces Pre with newPre (the next level of prerequisites) and goes through the process again.

I test my program by typing the command the set of commands the assignment says. I get:

```python
>>> prerequisite.getPrerequisite('CSC324',courseTable)
CSC324
  CSC207  CSC236
  CSC148  CSC148  CSC165
  CSC108  CSC108  CSC108
```

No way I am taking 8 extra courses from the Computer Science department! But wait, I look closer and see that some of these repetitions. I decide to use the code from Ex 9 to remove duplicates from my prerequisite lists. That is better, all the multiple entries for 'CSC148' and 'CSC108' are removed, it turns out I (only) need 5 courses.

Although by using the dictionary trick I remove duplicates at each row, I can still have duplicate courses across rows (for example for 'CSC443'). I have no idea how this could be solved.

Also as the assignment mentions, none of these courses in the list provide alternatives to the courses listed, but it is still a handy little program (if someone bothers to copy all course codes and their prerequisites in a nice file like 'cscCourseList.txt' ...)

10
# buildCourseTable names a function. You provide values for the parameters
# filename (a string of characters naming a file)
# courseTable (a lookup table)
def buildCourseTable(filename, courseTable):
    courseFile = file(filename, 'r')
    for nextLine in courseFile:
        wordList = nextLine.split()
        courseName = wordList[0]
        courseTable[courseName] = wordList[2:]
    courseFile.close()
    print 'All done building courseTable.'

# getPrerequisite names a function. You provide values for the parameters
# course (a string of characters naming the course whose prerequisites you need)
# courseTable (a lookup table)
def getPrerequisite(course, courseTable):
    Pre = courseTable[course]
    level = 1
    while Pre:
        newPre = []
        for preCourse in Pre:
            newPre = newPre + addPreCourses
        level = level + 1
        courseDictionary = {}
        for nextCourse in newPre:
            courseDictionary[nextCourse] = nextCourse
        newPre = courseDictionary.keys()
# sorts alphabetically the elements in newPre
newPre.sort()

## Set Pre to be equal to newPre
Pre = newPre
print ' ',

Doc

Have you ever noticed how some writers tend to write longwinded prose, with big words, while others tend to use simple words and small 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 questions 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 osCourseList.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](http://www.gutenberg.org).

**SOLUTION:** As before I copied the required files and typed *idle doc.py* to start working with the file. Most of the things I had to fill in had long descriptions, but where things I could find out from the finger exercises. For example I got the length of *sentenceList* by doing `len(sentenceList)`. I increased the *sentenceCount* by the length of *sentenceList*, by typing `sentenceCount += len(sentenceList)` (the fancy way of adding something to an existing label). I got a bit confused there as to what needs to be added where, but I went through the ## comments very carefully and figured things out.

I ran my program as the assignment asked, and although the number of paragraphs, sentences, etc seemed logical, my averages were all whole numbers. I puzzled a bit over this and finally figured out I was not allowing for decimal numbers (I was doing integer arithmetic). I fixed this by using what I learned in the finger Ex 2, by multiplying my division result by 1.0.

I noticed from these averages (look below) that this text of "Alice in Wonderland" probably has very few sentences per paragraph, which have long clauses (unlike how I write). I checked the *alice30.txt* file using *scite* to verify this and it is indeed the case. Indeed, the first and second paragraph is only one(!) sentence. I was also a bit surprised that the number of words in each clause was so small, but by scrolling through the file I found several very small clauses (for example "Down, down, down.") which obviously brings the average down. It also makes sense that the characters per word is small, because of all the "it, the, that, he, she" words that appear all the time.

Here is my code:

```python
# calculate some statistics about a text document
import re # regular expression (RE) tools

# report(fileName, paraMark)
# fileName is a string naming the file
# paraMark is a string that marks the break between paragraphs
def report(fileName, paraMark):
    fileString = file(fileName, 'r').read() # read the file into a string
    # split text based on paragraph ending mark
    parList = fileString.split(paraMark)
    # define what characters determine a full sentence using RE
    fullStop = re.compile(r"\n\n(?!\")"+\s+")
    # define what characters determine a clause sentence using RE
    shortStop = re.compile(r"(?!\n\n)(?!\")"+\s+")
    # set my counting variables to 0
    charCount = 0
    wordCount = 0
```
sentenceCount = 0
clauseCount = 0
for par in parList:
    sentenceList = fullStop.split(par)  # break into sentences
    ## increase sentenceCount by the length of sentenceList
    sentenceCount += len(sentenceList)
    for sentence in sentenceList:
        clauseList = shortStop.split(sentence)  # break into clauses
        ## increase clauseCount by the length of clauseList
        clauseCount += len(clauseList)
        for clause in clauseList:
            wordList = clause.split()  # break into words
            ## increase wordCount by the length of wordList
            wordCount += len(wordList)
            for word in wordList:
                ## increase charCount by the length of word
                charCount += len(word)

    ## print the length of parList, then a "," then the word "paragraphs" (with the quotes)
print len(parList), "paragraphs"
print sentenceCount, "sentences"
print clauseCount, "clauses"
print wordCount, "words"
print charCount, "characters"

## In the command below, replace sentenceCount with the result of dividing sentenceCount by ... 
## ... the length of parList. Make sure you allow results with decimal points.
print 1.0 * sentenceCount / len(parList), "sentences per paragraph"
## In the command below, replace clauseCount with the result of dividing clauseCount by ... 
## ... sentenceCount. Make sure you allow results with decimal points.
print 1.0 * clauseCount / sentenceCount, "clauses per sentence"
## In the command below, replace wordCount with the result of dividing wordCount by ... 
## ... clauseCount. Make sure you allow results with decimal points.
print 1.0 * wordCount / clauseCount, "words per clause"
## In the command below, replace charCount with the result of dividing charCount by .... 
## ... wordCount. Make sure you allow results with decimal points.
print 1.0 * charCount / wordCount, "characters per word"

**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.