CSC104, Assignment 2, Winter 2006
Sample solution

Danny Heap

Finger exercises

Here's a sample solution

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.

   Solution: Between my *journalA1* and the handout for assignment 1, I was able to do this pretty easily. Creating directories for each assignment is probably as feasible a system as any for organizing my course work.

2. You will be using the programming language Python. I 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 *df* 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 10 by 5. Now try dividing 9 by 5. Finally, try dividing 9 by 5.0 — that's 5, followed by a decimal point, followed by a zero. 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: Dividing 10 by 5 gave (not surprisingly) 2. Dividing 9 by 5 gave 1. That doesn't seem right. Dividing 9 by 5.0 gave 1.8, which is exactly right. 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: "9

divided by 5 is 1 with a remainder of 4." 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 numbers. Anyways, 9 % 5 is 4. The prof also said that Python might be changing, and in order to guarantee what he calls "integer division" I should use double slashes, as in: 9 // 5 is 1.

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: *D2* 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 = 7*, press the *enter* key, then type *favouriteNumber* and press the *enter* key again. Now store a different number (not 7 again) in a location named *secondFavouriteNumber*. Type *favouriteNumber + secondFavouriteNumber*, then press *enter*.

   SOLUTION: Before I appended "= 7" *favouriteNumber* was undefined. I was a bit worried that it was still undefined after I typed *favouritenumber = 7*, but then I noticed that I had typed a lowercase "n" which (as far as Python is concerned) is a different identifier. So now *favouritenumber* contains 7, whereas *favouriteNumber* is undefined. I fixed things up by typing *favouriteNumber = 7*, and then I typed *secondFavouriteNumber = 8* (being very obsessive about upper/lower case). Now when I type *favouriteNumber + secondFavouriteNumber* I get 15, just as if I had added the numbers stored at those identifiers.

   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 location in previous exercises). Now type *favouriteNumber = 19* 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 7, so I figured that now *secondFavouriteNumber* was just a nick-name for *favouriteNumber*. However that theory fell apart when I then typed *favouriteNumber = 19*: now *secondFavouriteNumber* still contains 7, but *favouriteNumber* contains 19.

   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 7. 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 19. 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.

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

   SOLUTION: My identifier *myList* 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 *myList[0]* is the first (zeroeth?) element: 1. The second element is at *myList[1]*: the number 3. The, uh, fourth element is at *myList[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 *myList[1:3]* returns *[3,5]* scratched that theory.

   You can also makes list of words. After the >>> type *myWords = ['Double', 'double', 'toil', 'and', 'trouble.']*, 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 labelled the list *['Double', 'double', 'toil', 'and', 'trouble']* 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 *['double', 'toil']*. When I check out *myWords* it still contains *['Double', 'double', 'toil', 'and', 'trouble']*, so it is unchanged by that operation with the brackets and the colon.

6. Try typing *oneWord = 'hand'*, then press *enter*, followed by *fewerWords + oneWord*, then press *enter*. Type *fewerWords.append('finger')*, then press *enter*, and check the value of stored at the location named *fewerWords*. Type *fewerWords = fewerWords * 2*, then press *enter*. Check what is stored at the location called *fewerWords*. At this point, you might want to chat with a TA or your instructor about what's going on.

   SOLUTION: I labelled the string 'hand' with the identifier *oneWord*, but then when I typed *fewerWords + oneWord*, I got an errors saying that I can "concatenate" a string with a list. That bothered me, but I continued to type *fewerWords.append('finger')*, and suddenly the list labelled by *fewerWords* had a new element: 'finger'. I guess this is the prof's heavy-handed way of showing that you need this "append" incantation to add things to a list. Now there's more arithmetic: *fewerWords = fewerWords * 2* first "doubles" the list labelled by *fewerWords* (it adds a copy of itself to the end), and then labels that new list with *fewerWords* itself. So now *fewerWords* labels *['double', 'toil', 'finger', 'double', 'toil', 'finger']*.

7. Similar to a list is a tuple. After the >>> type *myTuple = ('one', 'two', 'three')*, then press *enter*. Check what the location named *myTuple* contains. Try out *myTuple[0]* and *myTuple[0:2]*. You might be tempted to think that tuples and lists are the same until you try *myTuple.append('four')*. Compare the effect of *fewerWords[0] = 'elbow'*, then press *enter*, to *myTuple[0] = 'elbow'*, then press *enter*.

SOLUTION: Okay, the location labelled *myTuple* contains the list-like *('one', 'two', 'three')*. And (just like lists) *myTuple[0]* gives the first element of *myTuple*: 'one', and *myTuple[0:2]* gives a tuple consisting of consecutive elements of *myTuple* that are at positions 0 up to (but not including) 2: *('one', 'two')*. So far, identical to lists except for using parentheses instead of brackets. But, no — *myTuple.append('four')* doesn't work (there's a message saying there's no attribute 'append'). So, how do you add things to a tuple? More bad luck: *fewerWords[0] = 'elbow'* works as I expected — it replaces the first (i.e. zeroeth) element of the list with the string 'elbow', but I can't do the analogous thing with tuples. Typing *myTuple[0] = 'elbow'* generates an error about not supporting "item assignment" (who knew item assignment could be so controversial?). So far as I know, there seems to be no way of changing an already-existing tuple.

After the >>> type *tuple1 = ('one', 'two', 'three')* and press *enter*. Now type *tuple2 = ('four',)* and press *enter*. Finally type *tuple1[1:] + tuple2*. Experiment with their weird tuple arithmetic.

SOLUTION: Well, *tuple1* contains *('one', 'two', 'three')* as I expected, and `tuple2` contains `('four',)` (no surprise there). Now `tuple[1:] + tuple2` contains `('two', 'three', 'four')` — the elements at position 1 and 2 from `tuple1` with `tuple2`'s contents appended to the end. If I experiment a bit, `(0,1,2,3,4)[1:]` gives me a tuple consisting of the consecutive elements from those at position 1 up to the last position in `(0,1,2,3,4)`, so that explains what that `(:)` expression with only one position indicated does. And I can combine two tuples using ``+''.

8. With both lists and tuples you could get an element if you knew its position, for example, in the last exercise *myTuple[0]* held the element 'one'. Sometimes it is more convenient to access elements without knowing their position, but some other key. Try typing *province = {'toronto' : 'ontario', 'vancouver' : 'british columbia', 'montreal' : 'quebec'}*, then press *enter*. Now type *province['toronto']*, then press *enter*. Try some other cities.

   SOLUTION: When I type *province['toronto']* I get *'ontario'*, and (similarly), when I type *province['vancouver']* I get *'british columbia'*. It seems like I'll be able to match provinces to cities (although the all lower-case is annoying, but I can fix that). I can retrieve things on the left side of the colon in *province* by providing the things on the right side of the colon.

   In the example above, *province* is a look-up table: we look up the character string for a province by providing the character string for a city. Look-up tables are even more flexible than that. After the >>> type *bookTable = {('Alice', 'was', 'becoming') : ['very', 'to']}* and then press *enter*. In the look-up table *bookTable* you can retrieve the list of words ['very', 'to'] using the key ('Alice', 'was', 'becoming'). Try typing *bookTable[('Alice', 'was', 'becoming')]* and then pressing *enter*.

   SOLUTION: As expected, I get back *['very', 'to']* when I type *bookTable[('Alice', 'was', 'becoming')]*. I'm not sure why I'd want this, but there it is. I also checked out what's in *bookTable*, and it contains `{('Alice', 'was', 'becoming') : ['very', 'to']}` — the thing to the left of the colon is a tuple, 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 tuple 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").

9. Time for some lazy 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 ...):

```
>>> 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*, type *nextWord* 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 ...):

```
>>> for nextWord in wordList:
...   print 'nextWord is', nextWord
...
```

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.

10. Exit Python (by holding down the *Ctrl* key and pressing *D*). Make sure that *A2* is the current working directory (type *pwd* and then press *enter* to verify this). Now type *cp ~heap/pub/alice30.txt .* and press *enter*. This copies the text of "Alice in Wonderland" to your directory *A2*. You can examine it with the text editor *Scite*. 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).

```
>>> def readSomeLines(numLines):
...   bookFile = file('alice30.txt', 'r')
...   for nextLine in bookFile:
...     numLines = numLines - 1
...     print nextLine
...     print nextLine.split(), '\n' * 2
...     if numLines <= 0: break
...   bookFile.close()
```

After the >>> type *readSomeLines(3)*. Of course, you don't have to stop after 3 lines. This creates a function (program) called *readSomeLines* 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?

SOLUTION: 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 "Alice in Wonderland," 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 *readSomeLines(...)*. It looks as though *nextLine* holds a string, but *nextLine.split()* 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 *bookFile* ends up accessing the file *alice30.txt*, so I suppose the command *file* does that. The TA confirmed my conjecture, and pointed out that the *'r'* meant that `bookFile` was allowed to Read things from the file.

Whatever number I type between the parentheses gets stored in *numLines* and this is decreased by 1 each time the loop repeats, until finally it will be no more than 0, when the loop *break*s. I guess I *close* the file out of tidiness at the end.

That's probably plenty of exploring for now. The next section explains the motivation for the *lazyLit* module, and the section following that explains how to complete the file *lazyLit.py* so that you can run the module.

Lazy literature

Throughout your life you are occasionally asked to write documents. Sometimes this task provides an outlet for creative expression, but sometimes it is simply a chore that must be accomplished. This assignment provides a modest solution to the latter situation.

We'd all love to write beautifully, in the style of Shakespeare, Brecht, Brönte, (fill in your favourite author here), but beautiful prose takes hard work. On the other hand, there are severe penalties and social sanctions for trying to pass off other authors' work as our own. The solution (clearly) is a computer program that produces prose in the style of (favourite author here), but without copying an actual work of (same favourite author here).

Here's how the program will work. You give it an example of your favourite author's work, plus a parameter that we'll call *prefLen* (for prefix-length), which is a positive whole number. For the sake of concreteness, let's suppose that *prefLen* is 3. The program that you will help build scans your author's work, looking for groups of 3 adjacent words (prefixes), and for each prefix it keeps a list of which words follow it. For example, in Lewis Carroll's "Alice in Wonderland," the three-word prefix ('Alice', 'was', 'beginning') is followed by one of the words in the list: ['to', 'very'].

Once your program has recorded all the possible completions that Carroll would use for three-word prefixes, we begin our literary adventure by starting out with the first three words of "Alice in Wonderland," which are ('Alice', 'was', 'beginning'), and then randomly choose one of the words from the list [to, very]. Suppose the next word selected is 'very'. We then check our list to see which words follow the three-word prefix ('was', 'beginning', 'very'), and continue building our chain of words, until we have a brand-new piece of literature in the style of Lewis Carroll.

In the previous section you had an exercise to copy "Alice in Wonderland" from my 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.

Your contribution

I have written two short functions (or programs) in the file *lazyLit.py*. Emulate the finger exercise for copying *alice30.txt* to your *A2* directory to copy *lazyLit.py* to your *A2* directory.

Change your current working directory to *A2*. In your terminal window type *idle lazyLit.py* and then press enter. Two windows will pop up: one where you can edit the definitions of functions *buildTable* and *useTable*, 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 *buildTable* and *useTable* 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 *buildTable*, and a tuple 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 lazyLit functions. Switch to the other window and try:

```
>>> import lazyLit
>>> bookTable = {}
>>> lazyLit.buildTable('alice30.txt', 3, bookTable)
>>> lazyLit.useTable(100, 3, bookTable)
```

If your functions are working, you'll get some random prose in the style of Lewis Carroll. Otherwise, you can return to the function definitions, fix things up, return to the window for experimentation, type F6, 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, your TAs and I will consider some judicious hints.

Solution: Copying *lazyLit.py* was identical to copying *alice30.txt*, except replacing *alice30.txt* with *lazyLit.py*. Same, only different. The two windows popped after I typed *idle lazyLit.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 *wordCount = wordCount + 1* was pretty similar to *sum = sum + nextNumber*. There were a couple of weird ones, though. The comment *## append nextWord to bookTable[prefix]* was followed by the python command: *bookTable[prefix].append(nextWord)*. 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).

Finally I got it working (see below). When I typed *lazyLit.useTable(100,3,bookTable)*, I got 100 words of *Alice*-like prose. My image is that during *buildTable*, a prefix of length 3 is sliding along the text of "Alice in Wonderland," and storing the following word in the list of possible completions for that prefix. During *useTable*, a prefix of length 3 starts out at the beginning of the text, but then rolls a dice (shuffle) at each step to determine the next word.

If I increase *prefLen* beyond 3, I start to get exactly the text of "Alice in Wonderland" back. Prefixes of length 4 or greater give me too much "context", so when I roll the dice (shuffle) there's only one possible completion. If I use *prefLen* shorter than 3, I get thoroughly random prose back, and it doesn't resemble "Alice in Wonderland" very much.

SOLUTION: Here's a version of the code that worked:

```
import random # some tools to make things random

# buildTable names a function.  You provide values for the parameters
# filename (a string of characters naming a file)
# prefLen (a positive integer)
# bookTable (a lookup table)
def buildTable(filename, prefLen, bookTable):
    ## store a tuple of prefLen '\n' characters at prefix
    prefix = ('\n',) * prefLen
    bookFile = file(filename, 'r')
    for nextLine in bookFile:
        for nextWord in nextLine.split():
            if not bookTable.has_key(prefix):
                ## store an empty list at bookTable[prefix]
                bookTable[prefix] = [] # empty list to start
            ## append nextWord to bookTable[prefix]
            bookTable[prefix].append(nextWord)
            prefix = prefix[1:] + (nextWord,) # next prefix
    bookTable[prefix] = ['\n'] # special signal value for end-of-book
    bookFile.close()
    print 'All done building table.'

# useTable names another function.  The names created in buildTable are
# not visible inside useTable.  You provide values for the parameters:
# numWords (a positive integer)
# prefLen (a positive integer)
# bookTable (a lookup table)
def useTable(numWords, prefLen, bookTable):
    ## store a tuple of prefLen '\n' characters at prefix
    prefix = ('\n', ) * prefLen
    ## store 0 at wordCount
    wordCount = 0
    while 1:
        ## increase value in wordCount by 1
        wordCount += 1
        random.shuffle(bookTable[prefix])
        ## store the 0th string from bookTable[prefix] in nextWord
        nextWord = bookTable[prefix][0]
        ## print nextWord, followed by a space
        print nextWord, " ",
        if nextWord == '\n' or wordCount > numWords: break
        prefix = prefix[1:] + (nextWord, )
```