CSC 104 Lab 3, 3 February 2012

This lab contains a component which comprises 2% of your grade in this course. The graded component must be completed by *half an hour after* the end of the lab time you have signed up for. **You must sign up for a lab time on the course web page**; you are not enrolled automatically. Please sign up well in advance.

This lab can be done individually or in pairs. If working in pairs, please make sure that the student who isn't typing is following everything which is going on, and please switch places around halfway through.

Part A: Introducing the Python programming language

1. In this course we will dabble in a programming language named "Python". In this lab and in assignment two you will see how the computer follows a sequence of instructions which the programmer provides; that's a "program".

Recall our lecture discussion of the fact that software is also a kind of data. In particular, a python program is a text file.

Here is a complete python program:

```
print 3
```

What do you think it does?

2. Where will the number 3 appear when you run this program?

3. How about this?

```
print 'hello'
```

4. What about this?

```
print 3+4
```

5. Let's run these programs. Create a text file containing one of the above python "statements", and execute it by typing "python filename". Try each of the above small programs, one at a time, and see if they did what you thought they would do.

6. What will this do?

```
print '3+4'
```

Guess what it will do; then try it.

So, what do those single-quote marks actually mean?

7. We see that a Python program can consist of the word "print" followed by either a string (in single-quotes) or a numeric expression. Obviously, more complicated python programs exist as well. But these tiny ones are valid python programs.

A Python program consists of a *series* of statements. You can have just one statement, or you can have more than one.

What will the following two-line program do? Think about it, then try it.

```
print 'hello'
print 'goodbye'
```

What will the following one-line program do? Think about it, then try it.

```
print 'hello', 'goodbye'
```

(continued)

Part B: Slightly more complex Python programs

1. You will find some sample python programs in the directory /u/ajr/104/lab3. We normally name python program files with names ending in ‘‘.py’’. You can execute these programs directly from this directory (e.g. you could type ‘‘python /u/ajr/104/lab3/x.py’’; or if you do ‘‘cd /u/ajr/104/lab3’’ then you could type ‘‘python x.py’’; or if you do ‘‘cd /u/ajr/104’’ then you could type ‘‘python lab3/x.py’’; etc).

Or you can copy these programs to your own directory so that you can edit them and experiment with them.

2. First examine the file x.py, which looks like this:

```
x = 3
print x
```

Run it. ‘‘x=3’’ is an *assignment statement*; it gives meaning to ‘‘x’’. Thus, the subsequent ‘‘print x’’ becomes the same as ‘‘print 3’’.

The name ‘x’ (or whatever you put on the left side of an equals sign) is called a *variable*.

3. Assignment statements are more interesting when we can’t see the value assigned so easily. In ‘‘chatty.py’’ (which you should look at to follow the following), we use the function ‘‘raw_input’’ to get input from the user of the program. We use that in an assignment statement, so that this value is henceforth known as ‘‘name’’. And then we can use ‘‘name’’ in a later print statement, *outside* the quotes.

Try running chatty.py (type your name when it asks). Then examine the chatty.py file in detail, noting particularly how we go in and out of quotes in the last ‘print’ statement. ‘‘Hello’’ and a comma is inside the quotes; then we have a comma (outside the quotes), and a reference to the variable ‘‘name’’; then (after the comma separating ‘‘name’’ from the next item) we are back in the quotes for the semicolon and the rest of the sentence.

The output of ‘‘chatty.py’’ is flawed. What’s wrong with the output? (We won’t fix this problem today.)

4. ‘‘numberinput.py’’ shows a simple calculation with numbers. We use ‘‘input’’ instead of ‘‘raw_input’’ because we want to *interpret* the input—if the user types ‘‘38’’, we want that to be the *number* 38, not the string `'38'`. Examine and try numberinput.py.

5. ‘‘numbers.py’’ illustrates some calculations with numbers, and more-exciting assignment statements with use of variables. Examine numbers.py. Predict its output, then run it and see if you were right. Be sure you understand why it does what it does.

Make a change to numbers.py (e.g. using nedit) and try it again.

Part C: For marks

Recall the shell script ‘‘add’’ from lab 2. A session with ‘‘add’’ looked like this:

```
% sh add 12 23
Ah, I see you want to add 12 and 23
The sum is
35
%
```

(There is a sample solution on the course web page.)

Let us now do this in Python. Call your file ‘‘add.py’’. It will call input() twice to get the two numbers, then output the little story. The sum can be on the same line as the introductory text; this is easier to do in

(continued)

Python than it is in shell script.

So, a session will look like this. The ‘%’ is the shell prompt, and then the user typed 12, return, 23, return; and then your Python program output the rest.

```
% python add.py
12
23
Ah, I see you want to add 12 and 23
The sum is 35
%
```

Submit your file for credit using the command

```
submit −c csc104h −a lab3 add.py
```

If you are working with a partner, also submit a file named “partner” which says who the other student is, so that you both get credit for this work. The “partner” file should contain the CDF account name of the other student only (and no other text).

Part D: Formatting (optional)

Consider the following program:

```
print 4*28+12
print 1*2+3
```

(“*” is multiplication.)

The output will look like this:

```
124
5
```

But columns of numbers are much easier to deal with if they line up so that the hundreds columns are lined up, the tens columns are lined up, etc. So we’d like that “5” to be under the 4, not under the 1.

The way we fix this in Python is to say “display the numbers but always take up 4 columns while doing so”.

In other words, in the above output example, the 124 takes up three columns of output and the 5 takes up only one column. That’s because 124 has three digits but 5 has only one. But the varying number of columns taken up is the problem. We want to be able to *specify* the number of columns taken up. The Python processor will use spaces on the left to make them line up.

Here’s how we do this:

```
print ’%4d’ % (4*28+12)
print ’%4d’ % (1*2+3)
```

Experiment with this, and note exactly where the output characters go on the screen.