Introduction to Computer Science I
Spring 2014
MID-TERM EXAM — SOLUTIONS

1. QUESTION: Consider the following module of Python code...

```python
def thing_one (x):
    y = 0
    if x == 1:
        y = x
        x = 2
    if x == 2:
        y = -x
        x = 3
    elif x == 3:
        y = 2 * x
        x = 2 * y
    else:
        y = x
    print('x = ' + str(x))
    print('y = ' + str(y))

def thing_two (l, v):
    for i in range(len(l)):
        l[i] += v
        v = v - 1

def main ():
    thing_one(7)
    thing_one(3)
    thing_one(1)
    l = [20, 40, 30]
    v = 6
    thing_two(l, v)
    print('l = ' + str(l))
    print('v = ' + str(v))

if __name__ == '__main__':
    main()
```

What output is printed when this module is run?

ANSWER: The output produced is...

```
x = 7
y = 7
x = 12
y = 6
x = 3
y = -2
l = [26, 45, 34]
v = 6
```

DISCUSSION: The great majority of people got the first four lines of the output—that is, the first two pairs of lines that show the output for the function calls, `thing_one(7)` and `thing_one(3)`.

It was downhill from there. The call `thing_one(1)` required a careful reading of the conditional statements in that function. Specifically, notice that there are two distinct conditional statements in that function: first, an *if-then* statement; second, an immediately subsequence *if-then-elif-then-else* multi-part statement. The key here was to observe that each of these statements will be applied by Python in sequence.

That is, with a paramater of $x = 1$, the condition on the first statement will be `True`. As a consequence, the *then*-branch will assign $y = 1$ and then $x = 2$. Many people stopped there in determining the ultimate value of x and y, but doing so was incorrect. The next conditional statement would be executed next. Since $x = 2$ at that point, the leading *if*-condition would be `True`, triggering that statement's *then*-branch. Consequently, $y = -2$, followed by $x = 3$. It is these values that are then printed.

Sadly, the evaluation of `thing_two` wasn't much better for most people. Let's begin by tracking the values of l and v and their changes as the code progresses. To do so, we have to remember the effect of *scope*. That is, the spaces named l and v in `main` are different from the parameters named l and v in `thing_two`. Let's call the former l_m and v_m (m for `main`), and the latter l_t and v_t (t for `thing_two`).

After all of the calls to `thing_one`, we see that `main` will assign $l_m = [20, 40, 30]$, and $v_m = 6$. When `main` performs the the function call to `thing_two`, the arguments l_m and v_m are passed to `thing_two` such that the called function's parameters are assigned $l_t = l_m = [20, 40, 30]$ and $v_t = v_m = 6$.

The function `thing_two` will loop through the indices of l_t—0, 1, and 2. The steps that occur within the body of this loop are performed—indeed, they could only be performed—on l_t and v_t. Thus, as the loop iterates, the values will change in the following sequence:

i	l_t	v_t
before loop	[20, 40, 30]	6
0	[20, 40, 36]	5
1	[20, 45, 36]	4
2	[24, 45, 36]	3

When the loop is complete and `thing_two` is about to return, its local variables have the values on the last row of this table. The program then returns from this function call, resuming within `main` where it had left off, leaving the final two `print` calls to be performed.

To determine what these calls will print, we must keep in mind the relationships between the local variables within `main` that were used as arguments to the call to `thing_two`, and the local variables within `thing_two` that were defined as parameters. First, notice that v_m and v_t are two different spaces. When v_m was passed as an argument to provide an initial valuable for the parameter v_t, **any changes to that parameter by `thing_two` change only the local space**. That is, updates to v_t do not change v_m at all. The call to `print` at the end of `main` is printing a copy of v_m, which remains unchanged with the value of 6.

However, l_m and l_t are another story. Remember that the spaces named by these variables **do not contain a list**; rather, each space **points to a list**. The difference is a critical one here. When `thing_two` is called, the pointer contained in l_m is passed as an argument, thus initializing l_t with a copy of that pointer. In other words, l_m and l_t **point to the same list**. Since lists are *mutable*, then the changes in the values of the list that occur as `thing_two` executes are modifications to that one list. When `thing_two` completes its work and returns, l_m still points to that very same list **and its modified contents**. When `main` calls `print` to show `l`, it is printing the list to which l_m refers, which is the same list that `thing_two` modified through its pointer l_t.

This question was challenging in that you had to understand Python's rules about variables, scope, arguments, parameters, and simple values vs. pointers to lists/strings. However, as always, if you could **draw the data**, the outcome was clear.

2. QUESTIONS: Provide short answers (no more than a few sentences) to each of the following questions:

(a) In Python, strings are *immutable* while lists are *mutable*. What is the difference?

(b) How does the `//` operator differ from the `/` operator? Give an example of where `//` would be needed.

(c) United Airlines will only allow carry-on bags that are no more than 22 inches long, 14 inches wide, and 9 inches deep. Assuming that variables named `length`, `width`, and `depth` have already been assigned values, **write an expression** combining the three that evaluates to `True` if bag fits within those limits, and `False` otherwise.

ANSWERS:

(a) One a string is created, it cannot be changed. Characters cannot be added, removed, or altered; any seeming modification of a string (e.g., *slices* or *append* operations) cause a modification to be carried out on a duplicate. In contrast, lists can be extended, shortened, or its elements altered.

(b) Both operators perform division, but `/` produces a real-valued result, while `//` produces only an integer result. Specifically, any fractional result from integer division is discarded, truncating the result. We often use integer division when computing list indices, which cannot be fractional.

(c) `length <= 22 and width <= 14 and depth <= 9`

DISCUSSION:

(a) While most stated something resembling the correct intuition behind this difference—one is changeable, the other is not—the devil is sometimes in the details. Specifically, it was important to remember that each string and list are not directly stored in a space named by a variable; rather, the variable spaces *point* to a string or list. The difference is critical when describing just what is changeable (or not): the variable's space; or the string/list itself. It is the latter to which these terms refer. For the former, the pointer in each variable's space is always changeable, for each can be made to point to a different string or list.

(b) Most of the answers here were again mostly on the mark. There was the occassional inversion of the two operators, as well as a few who confused `//` with the *modulus/remainder operator*, `%`, but otherwise the description of the two were correct. Some people did not read the question carefully regarding the requested *example*. Specifically, the example was either forgotten completed, or more often, the example was just a demonstion of how the two would calculate differing results given the same input values (e.g., `5 / 2 = 2.5` vs. `5 // 2 = 2`. Of course, the question clearly asks

for an example situation in which integer division would be desirable or necessary.

(c) Far and away the most common mistake was the failure to provide **only an expression**, and the question specified. Some wrote conditional statements, others complete functions, adding superfluous material beyond the desired expression itself. A less common mistake was to take the product of the length, width, and depth, and thus test the total volume of the bag, rather than the length of each dimension.

3. QUESTION:
Write a function named `print_big_v (size)`, where `size` indicates the size of the pattern that the function will print. Specifically, this function is called with 6 as the argument for `size`, then it should print the following:

```
\           /
 \         /
  \       /
   \     /
    \   /
     \ /
```
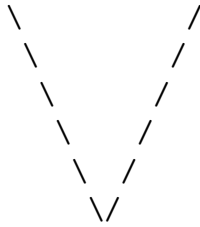
That is, the "big V", made up of forward and backward slash characters, should be 6 rows tall.

ANSWER:

```python
def print_big_V (size):
    for i in range(size):
        front_spaces = ' ' * i
        middle_spaces = ' ' * (2 * (size - i - 1))
        print(front_spaces + '\\' + middle_spaces + '/')
```

DISCUSSION: Answers for this problem were largely good, and a number of valid approaches were taken (of which the code above is just one). There were two common errors:

(a) **No leading spaces:** Some forgot to have their function print **any** spaces before the \ character on each line. Whoops.

(b) **Insufficient middle spaces:** Many printed some spaces between the left and right "arms" of the big V, but not enough of them. Usually, there was an expression to print a number of spaces tied to the row number—a valid approach—but off by a factor of two.

There were a number of less common but more profound errors, including a surprising number of people who used an extra inner loop, often using the same variable as the outer loop as a counter. None of these made much sense, and I can only assujme that they are the result of some rote repetition of a pattern seen on a sample problem. If you made this mistake, be certain that you know why it was wrong, and how to approach this type of problem correctly.

4. QUESTION: **Write a function** named `extract_lesser (l, v)` that, from a list of numbers `l` finds all of the values less than `v`, puts them into a new list, and returns that new list.

ANSWER:

```
def extract_lesser (l, v):
    less_list = []
    for num in l:
        if num < v:
            less_list.append(num)
    return less_list
```

DISCUSSION: Again there were a number of valid approaches to this problem. Most commonly, people looped over the indices of l (with a loop something like: `for i in range(len(l)):`, but then compared $i < v$ in the next step, confusing the use of i as a position, not a value itself.

Others tried to keep track of the next available position in the new list (above, `less_list`) and assign that position directly. Such approaches are not valid ways to append a value to an existing list, and would crash the program; either the `append()` method or the use of concatenation is necessary too append an item to the end of the existing list.

A few others printed the final result rather than returning it. Be sure that you know that difference between returning and printing!