CMSC 201 Fall 2015

Lab 07 – Lists

Assignment: Lab 07 – Lists
Due Date: During discussion, October 12th through October 15th
Value: 1% of final grade

Lab 7 focuses on using lists and loops, but is also a review of much of the material we have covered so far. Although understanding individual concepts is very important, being able to put them together in new and interesting ways is what will allow you to create engaging programs.

To complete this lab you will need to use lists, indexing, for loops, user input, interactive while loops, decision structures (`if/elif/else`), and `print()` statements. (Because much of this material has been covered in previous labs, the pre-lab review will be shorter than normal, and may contain material from previous lab descriptions.)

In Lab 7 you will be creating a simple "store" where a user can purchase different articles of clothing. The clothing items and their corresponding prices will be stored in two separate lists. You will need to access these lists to print the store's inventory and to "charge" the user for each purchase.

Part 1: Lists and Indexing

Lists are an easy way to hold lots of individual pieces of data without needing to make lots of variables. They are a type of *data structure*, which are specialized ways of organizing and storing data.

In order to get a specific variable, or *element*, from a list, we need to access that *index* of the list. NOTE: Lists don't starting counting from 1 – the first element in the list is at index 0.

For example, the following line of code creates a list called **`names`**:

```
names = ["Aya", "Brad", "Carlos", "David", "Emma"]
```

Which creates the list (called **`names`**) below:

Part 2: For Loops

We can use **`for`** loops to perform two different actions: *iterating* over a list, or performing an action a certain number of times. We will focus on *iterating* over a list – this means moving through a list, one element at a time.

For example:

```
list_of_fruits = ["kiwi", "banana", "peach"]
for fruit in list_of_fruits:
    print("I ate a", fruit)

Output:
I ate a kiwi
I ate a banana
I ate a peach
```

Sometime we may want to display a list and number its contents. The easiest way to do this is with a **`for`** loop over the length of the list, and using the index to access each element.

```
subjects = ["dragonology", "chem", "english", "bio"]

# we want to loop over the length of the list
for i in range( len(subjects) ):
    # numbering will start at 0
    print(i, ":", subjects[i])
```

When run, the code above will print the following:

```
0 : dragonology
1 : chem
2 : english
3 : bio
```

Part 3: While Loops

A **`while`** loop statement in the Python programming language repeatedly executes a target statement as long as a given Boolean condition is **`True`**.

The syntax of a **`while`** loop in the Python programming language is:

```
while CONDITION:
    STATEMENTS(S)
```

Here, **`STATEMENT(S)`** may be a single statement or a *block* of statements. The **`while`** loop continues to run *while* the **`CONDITION`** is still **`True`**.

Part 4: Interactive While Loops

Another way to use a **`while`** loop is as an *interactive* or *sentinel* loop. An interactive (sentinel) loop continues to process data until reaching a special value that signals the end. The special value is called the *sentinel*.

The pseudocode for an interactive (sentinel) loop in the Python programming language is as follows:

```
Get the first data item from the user
While data item is not the sentinel
    Process the data item
    Get the next data item from the user
```

One of the scenarios in which we can implement this type of loop is a version of our grocery list program that allows us to enter as many items as we like. Although it is similar to previous versions, the interactive (sentinel) while loop of the grocery list program allows us to enter as many items as we like until the sentinel value of **`"exit"`** is entered.

```
def main():
    grocery_list = []   # initialize the list to be empty
    # get an initial user value
    userVal = input("Enter an item, or 'exit' to end: ")

    # run the while loop until the user enters "exit"
    while userVal != "exit":
        grocery_list.append(userVal)
        # get the next value from the user
        userVal = input("Enter an item, or 'exit': ")

    # once the user is done with the list, print it out
    for g in grocery_list:
        print("Remember to buy", g)

main()
```

Part 5A: Writing Your Program

After logging into GL, navigate to the **Labs** folder inside your **201** folder. Create a folder there called **lab7**, and go inside the newly created **lab7** directory.

```
linux2[1]% cd 201
linux2[2]% cd Labs
linux2[3]% pwd
/afs/umbc.edu/users/k/k/k38/home/201/Labs
linux2[4]% mkdir lab7
linux2[5]% cd lab7
linux2[6]% pwd
/afs/umbc.edu/users/k/k/k38/home/201/Labs/lab7
linux2[7]%
```

To open the file for editing, type

emacs store.py &

and hit enter. (The ampersand at the end of the line is important – without it, your terminal will "freeze" until you close the emacs window. **Do not include the ampersand if you are not on a lab computer.**)

The first thing you should do in your new file is create and fill out the comment header block at the top of your file. Here is a template:

```
# File:          store.py
# Author:        YOUR NAME
# Date:          TODAY'S DATE
# Section:       YOUR SECTION NUMBER
# E-mail:        USERNAME@umbc.edu
# Description:   YOUR DESCRIPTION GOES HERE AND HERE
#                YOUR DESCRIPTION CONTINUED SOME MORE
```

Now you can start writing your code for the lab, following the instructions in Parts 5B and 5C.

Part 5B: Printing the Store Inventory

You will be coding Lab 7 in an incremental manner – in other words, you will code up one piece of the lab and test that it works **before** moving on to the next piece. Incremental development is a very effective way of tackling a problem, in part because it makes it easier to pinpoint where an error occurs when you are only working on a small part of the code at a time.

The first piece of code we will write is printing the store inventory. Copy the two lists below, **`items`** and **`prices`**, into your program. The lists are of the same length, and the elements in each list directly correspond (*e.g.*, the price of a pair of shoes is $54.99, socks are $7.11, etc.).

```
items  = ["shoes", "socks", "hat", "belt", "blouse", "dress", "tie"]
prices = [  54.99,    7.11,  6.49,  22.58,    21.73,   38.99, 14.83]
```

To print the inventory, you should write code that will print three different things on each line:

- **The number of the item** (start counting from 1)
- **The item for sale** (*e.g.*, belt, blouse, tie, etc.)
- **The price of the item** (including a dollar sign)

When you have completed this, the output should look something like this:

```
bash-4.1$ python store.py
1 - shoes       $ 54.99
2 - socks       $ 7.11
3 - hat         $ 6.49
4 - belt        $ 22.58
5 - blouse      $ 21.73
6 - dress       $ 38.99
7 - tie         $ 14.83
```

(There are hints on the next page if you need them.)

Try to solve Part 5B on your own before you turn to these hints!

Are you stuck on how to print elements from two lists at the same time?
Because we want to print two lists at once, we cannot use a loop that iterates over a single list's contents. Instead, we can look at the same index in both lists to print out the item and its corresponding price.

Still stuck on how to print two lists at once?
You will want to use a loop similar to the one at the top of page 3. It should iterate over a range that is the length of one of the lists.

Having trouble making the numbering start at 1 instead of 0?
Remember that the index of a list begins counting at 0. Also recall that the **`range()`** function starts at 0 by default. If you are printing the current index as the number, you will start at 0 – in order to start counting at 1, you will need to print something like **`index + 1`**.

Wondering how to make the costs in the sample output neatly line up?
Remember that the tab character in Python (**`\t`**) works similar to a tab in a word processing program. It doesn't indent a specific number of spaces, but rather indents to a specific point. Putting a tab in front of the dollar sign should line the dollar signs up at the same place on the line.

Part 5C: Shopping in the Store

Do not move on to this part until your program can print the store's inventory!

Now that you have one piece of your program working, we can focus on the next task. A store is nothing without customers to shop in it! To complete your program, you will **implement code that lets a user "shop" in your store, choosing items to purchase and being "charged" for those items**. They can continue to shop until they choose to quit – even if they can't afford to buy anything else, you won't kick them out.

A user starts with $100 in their wallet, and chooses what to purchase by the number of the inventory item. If they can afford to make the purchase, you should print out a confirmation message and deduct the price from their wallet. If not, you should print out a message letting them know the purchase was not made (and should not charge them for the item). Your store will not run out of stock, so they can purchase multiples of any item.

For example, "shoes" are the first item in the inventory list, at spot #1; if a user chooses item 1, they are choosing to purchase shoes. If they have at least $54.99 in their wallet, then the purchase should go through, and they should have 54.99 removed from their total funds.
(You can assume that the user's input will be valid: between 0 and 7.)

Here is a sample run of the store program, with user input in blue. Output other than the store inventory and input prompts is highlighted in orange.
(If you need some help, the hints are available after this sample output.)

```
bash-4.1$ python store.py

You have $ 100.0 in funds available
1 - shoes       $ 54.99
2 - socks       $ 7.11
3 - hat         $ 6.49
4 - belt        $ 22.58
5 - blouse      $ 21.73
6 - dress       $ 38.99
7 - tie         $ 14.83
Please choose an item # to purchase, or '0' to quit: 6
Thank you for purchasing: dress
```

```
You have $ 61.01 in funds available
1 - shoes        $ 54.99
2 - socks        $ 7.11
3 - hat          $ 6.49
4 - belt         $ 22.58
5 - blouse       $ 21.73
6 - dress        $ 38.99
7 - tie          $ 14.83
Please choose an item # to purchase, or '0' to quit: 4
Thank you for purchasing: belt

You have $ 38.43 in funds available
1 - shoes        $ 54.99
2 - socks        $ 7.11
3 - hat          $ 6.49
4 - belt         $ 22.58
5 - blouse       $ 21.73
6 - dress        $ 38.99
7 - tie          $ 14.83
Please choose an item # to purchase, or '0' to quit: 1
Sorry, but you are unable to afford that item.

You have $ 38.43 in funds available
1 - shoes        $ 54.99
2 - socks        $ 7.11
3 - hat          $ 6.49
4 - belt         $ 22.58
5 - blouse       $ 21.73
6 - dress        $ 38.99
7 - tie          $ 14.83
Please choose an item # to purchase, or '0' to quit: 0

Thank you for shopping at Python Mart!
You have $ 38.43 left in your wallet.
Have a great day!
```

Try to solve Part 5C on your own before you turn to these hints!

Is the user's wallet showing way too many decimals? Something like this?

```
You have $ 23.689999999999998 in funds available
```

There is nothing wrong with your program if this happens!
This is called a floating point error, and was discussed briefly in class. Essentially, because Python represents numbers in binary, it is impossible for it to accurately represent some decimal numbers. (Just like how in decimal 2/3 is 0.666666666667.) This leads to numbers like the one above.

Are you stuck on how to interact with the user?
Take a look at the example on page 4 of an interactive while loop. You should use the same basic code setup to allow the user to keep shopping until they choose to quit by entering "0".

Having trouble seeing the "big picture" of how your program should work?
Try drawing a quick flowchart or planning out what needs to happen on paper in pseudocode. Don't worry about the specific details, just try to visualize what needs to happen overall – how do you stop once the user wants to quit? When do you need to "charge" them for a purchase?

Is your program charging the user for the wrong item?
Remember, the user's numbering starts at 1, but the indexing in a list starts at 0. If a user chooses to purchase item #3, the name of that item is stored at `items[2]`, and its price is stored at `prices[2]`.

Still stuck on what to do for your program?
Here is some basic pseudocode for what your program should be doing:

- Print the user's available and the inventory of the store
- Prompt the user for an item to purchase (or "0" to stop shopping)
- "Charge" the user for the item by removing the cost of the item from the funds in their wallet
    - If they can't afford the item, don't charge them for it
- Thank them for purchasing that specific item
    - If they couldn't afford the item, print out a message that says that
- Repeat… until they choose "0" to quit shopping

Part 6: Completing Your Lab

To test your program, first enable Python 3, then run **`store.py`**. Try a few different inputs to see how well your program works.

```
linux2[7]% /usr/bin/scl enable python33 bash
bash-4.1$ python store.py
You have $ 100.0 in funds available
1 - shoes       $ 54.99
2 - socks       $ 7.11
3 - hat         $ 6.49
4 - belt        $ 22.58
[etc]
```

Since this is an in-person lab, you do not need to use the **`submit`** command to complete your lab. Instead, raise your hand to let your TA know that you are finished.

They will come over and check your work – they may ask you to run your program for them, and they may also want to see your code. Once they've checked your work, they'll give you a score for the lab, and you are free to leave.

IMPORTANT: If you leave the lab without the TA checking your work, you will receive a **zero** for this week's lab. Make sure you have been given a grade before you leave!