CSci 1133
Lab Exercise 5

Linear Data Structures

Python provides several simple and powerful mechanisms to structure data. The *list* is a vital one that enables us to manipulate and reason with *ordered collections*. We can use an ordered collection in Python (list) to represent and store numerical information (test scores, daily temperature measurements, time-ordered experimental results, and the like). We can also represent more abstract things like decks of cards and traffic patterns. In this lab, you will explore several problems that utilize lists.

Like Alice Through the Looking Glass

Mutability refers to the ability to change something. So far, most of the things we've seen are *immutable*… they can't be changed. "But what do you mean?" you ask, "we've been changing variables all the time!". Yes you have. The *variables* you're using are changing (that's why we call them 'variable'… duh) But the *objects* your variables have been referring to are not! Numbers and strings in Python are *immutable*. They can't be changed, only created or destroyed. So then, what exactly does a variable contain? A variable contains a *reference* to an object. Let's explore this a bit. First, set up a couple of simple variables:

```
>>> var1 = 0 <ENTER>
>>> var2 = 0 <ENTER>
```

This looks like `var1` contains the value zero as does `var2`. In a way, they do, but they really don't contain any values at all. To see this, let's use the Python `id()` function:

```
>>> id(var1) <ENTER>
>>> id(var2) <ENTER>
```

They both have the *same* identifier… they are both names for the same thing! namely the "*integer object identified as nnnn with the value 0*". So what happens if we change the 'value' of `var2`?

```
>>> var2 = 6 <ENTER>
>>> id(var1) <ENTER>
>>> id(var2) <ENTER>
```

`var2` has changed. It now points to (references) a different object… an integer object with the value 6.

This is a very subtle, but important thing to understand. Python maintains values in *objects* (recall that every object has a unique identifier, a type and a value). Variables do *not* contain values, they contain *references*… the identifiers of the objects they point to. When you change the 'value' of a variable that is bound to an *immutable* object, a new object is created (if not already in existence) and the variable binding is updated to reference the new object.

So far, all of this detail hasn't mattered much. This is because numbers (integers, floats, complex) and strings are *immutable*, so worrying about where the value is stored doesn't really trouble us.

But Python `list` objects are *mutable*. They can be changed! This means that the object itself is updated. Let's look at one way in which this may cause issues. Start by creating a simple list:

```
>>> mylist = [1,2,3,4] <ENTER>
>>> list1 = mylist*3 <ENTER>
>>> list1  <ENTER>
```

That looks like we might expect… a list consisting of 3 copies of `mylist`. Now let's change one of the `list1` values:

```
>>> list1[2] = 55 <ENTER>
>>> list1 <ENTER>
```

All is well. The third `list1` element has been changed as we expected (actually, the reference to the object in the list has been changed) Now let's create another list in similar fashion:

```
>>> list2 = [mylist]*3 <ENTER>     # be sure to put brackets around [mylist]
>>> list2 <ENTER>
```

hmm… this looks a little different. That's because it *is* different. It's actually a list of 3 lists. Now try to change the second element as before:

```
>>> list2[2] = 55 <ENTER>
```

oops… the element at `list2[2]` was another list. We just replaced it with a single value (55). But now for the weird part:

```
>>> mylist[2] = 99 <ENTER>
>>> list2
```

Oh my! We changed an element in `mylist` and suddenly `list2` was altered as well! We didn't do anything to `list2` (explicitly anyway). We'd better check to see if `list1` suffered a similar fate:

```
>>> list1 <ENTER>
```

Why didn't `list1` change as well??! The answer to all of this lies in the fact that lists 1) do not contain values, they contain *references* that point to things and 2) lists are *mutable*. In the first example, `list1` actually contains the identifiers of the individual *immutable* numeric objects (1, 2 , 3 and 4) whereas the second list (because we surrounded `mylist` with square brackets when we created `list2`) contains three references (pointers) to `mylist`. When `mylist` is subsequently changed, the references still point to the original object (`mylist`) which is no longer the same… its been changed!

Try this:

```
>>> id(mylist)  <ENTER>
>>> id(list2[0])  <ENTER>
>>> id(list2[2])  <ENTER>
```

What do you notice? Explain to one of your TAs why this is the case.

Warm-up

1). **Histograms**

Write a program that will simulate the tossing of two dice and keep track of the frequency of each roll's outcome (2..12). You should generate 10,000 *pairs* of random integers in the range 1 through 6 and, using a list containing individual integer counters (one for each possible outcome), determine the number of times each outcome occurs. After 10,000 simulated "rolls" of the dice, output the totals for each possible outcome. [Hint: consider simply using the dice total as an index into the list of counts]

Theoretically, 7 is the most likely outcome. Does your histogram support this?

Stretch

1). **List Construction** (adapted from Dierbach, p164)

Write a program (script) that prompts the user to enter a list of words and stores in a list only those words whose first letter occurs again elsewhere in the word (e.g., "Baboon"). Continue entering words until the user enters a null string, at which point your program should print the elements stored in the list, one word per line.

Write your program code as the body of a non-pure function named `main()` and then be sure to invoke main as the last line of the module:

```
def main():
      .
      .
      .

main()
```

Workout

1). **Selection Sort**

Sorting a list of objects "in place" (without the use of an additional list structure) is an important Computer Science problem that has been extensively studied. One of the simplest methods is known as *selection-sort*. The *selection-sort* algorithm is relatively easy to understand and implement, however it is not very efficient because it takes roughly n^2 operations on a list with n elements. The *selection-sort* algorithm is as follows: Given a list of numbers whose values are in some arbitrary order e.g.:

3 5 2 8 9 1

determine the *smallest* element in the list and swap it with the first element, then repeat the process for the *remaining* elements in the list. When you've completed the entire list, it will be sorted in ascending order of value:

3 5 2 8 9 1	…swap minimum value of [3, 5, 2, 8, 9, 1] with 1st element
1 5 2 8 9 3	…swap minimum value of [5, 2, 8, 9, 3] with 2nd element
1 2 5 8 9 3	…swap minimum value of [5, 8, 9, 3] with 3rd element
1 2 3 8 9 5	…swap minimum value of [8, 9, 5] with 4th element
1 2 3 5 9 8	…swap minimum value of [9, 8] with 5th element
1 2 3 5 8 9	…sorted!

Write a Python program that will implement the *selection-sort* algorithm to sort an integer array in ascending order. Your program must do the following:

- First, implement a non-pure function named `ssort(somelist)` that will take a single list argument and sort it using the *selection-sort* algorithm as described above. Your function should not return a "value" per se, the list should be sorted "in-place", i.e., by modifying the caller's list.
- Input a value *n* from the console and construct an integer list consisting of the *n* integer values: 1..*n* (the so-called "counting numbers") using list comprehension.
- Randomize the list using the `.shuffle` method from the `random` module. `random.shuffle` will take any list as an argument and "shuffle" it by performing a random permutation of the elements. Read the `random.shuffle` help page to learn more.
- Print the contents of the list
- Sort the list in ascending order using a call to your `ssort` function.
- Print the contents of the newly sorted list

Constraints:

- Do not use any Python `list` *methods*. Use only basic sequence operations.

2). **Blackjack Scoring**

BlackJack is a casino card game in which the objective is to continue adding cards to your hand without its value exceeding 21. The 'value' of the hand is the sum of the individual card values as follows: The 2-9 cards count at "face value" (2-9 points), The 10, Jack, Queen and King count as 10 points each and the Ace counts as *either* 1 or 11 depending on the most advantageous count.

Write a Python program to determine and output the score of a BlackJack hand. The Program will input a single string representing a hand of cards and return an integer value equivalent to the *highest possible score* that is less than 21 (or the lowest score over 21). Individual cards are represented using character values as follows:

- For the cards 2-9, use characters '2', '3', '4', … ,'9' respectively
- For the 10, use 'T'
- For the Jack, use 'J'
- For the Queen, use 'Q'
- For the King, use 'K'
- For the Ace, use 'A'

Note again that the Ace can count as either 1 or 11. Your procedure should initially value any Ace as 11 unless the total of the hand is greater than 21. If the value of the hand is greater than 21, you should re-compute it assuming the Ace(s) is(are) valued as 1.

Challenge

Try these challenge problems if you have extra time or would like additional practice outside of lab.

1). **Casino Rules**
Write a simulation program that determines how frequently the dealer will go "bust" when playing the casino game "Blackjack". The simulation involves dealing a large number of Blackjack hands according to the dealer-rule "stand on soft-17" and determining the frequencies of various scoring outcomes.

In the game of Blackjack, a player initially receives 2 cards and optionally draws 1, 2 or 3 more in an attempt to bring the total value of the cards as close as possible to 21 without going over. To determine the value of the hand, the cards labeled 2 through 10 are scored respectively 2 through 10 points each, so-called "Face cards" (jack, queen, and king) are scored as 10 points, and the ace can count as either 1 or 11, whichever is better for the player. If the score is over 21 the player loses (the player is "busted"), regardless what the dealer does. When played in a casino, the dealer plays according to fixed rules. For this simulation, we'll assume the casino employs the "stand on soft-17" rule that is more advantageous for the player.

Regardless the player's score, the dealer is required to continue drawing cards until his/her hand achieves a value of 17 or more, or goes bust. Since an ace counts as either 1 or 11 points, each ace results in two possible scores for a hand. A "soft score" is a score that includes one or more aces. For example, ace-ace-3-2 could be one of three possible scores: 27 (busted), 17 (soft-17), or 7. The actual "score" of a soft hand is the largest score that is less than or equal to 21; in this case 17. Therefore, ace-ace-3-2 is an example of a "soft-17" and would require the dealer to stop taking cards.

Construct a Python program that will "deal" 10,000 Blackjack hands with the "stand on soft-17" rule (i.e. for each hand, continue to deal cards until the score reaches 17 or more). You should count the number of occurrences of the following:

- dealer scores 17
- dealer scores 18
- dealer scores 19
- dealer scores 20
- dealer scores 21
- dealer "busted"
- "natural" blackjack (score equal to 21 with first two cards dealt)

Display each of these statistics on the terminal as both frequency (count) and the percentage of total hands played.

Your program should do the following:

a). Represent the card 'deck' as a list of integers ranging from 0..51. This list must be 'shuffled' using the `random.shuffle()` method. You simulate the dealing of cards by selecting each integer from this list in order until all 52 (shuffled) 'cards' have been used. At that point, you must "reshuffle" the list and begin dealing anew from the first card in the deck.

b). Keep each 'hand' as a second list of integers in the range 0-51.

c). Continue re-shuffling and dealing cards until 10,000 hands have been scored.

d). Display the resulting statistics and determine what percentage of the time the dealer will "bust" given the "stand on soft 17" rule.

2). **English to Pig Latin Translator**

Write a Python program that will translate an English word-phrase into Pig Latin. Pig Latin is a fun word game in which words are translated according to the following "rules":

1. For any word that begins with one or more consonants (y is considered a consonant): move the consonants to the end of the word and append the string 'ay'.
2. For all other words, append the string 'way' to the end.

For example, the phrase "Can you speak pig latin?" would be translated:

"Ancay ouyay eakspay igpay atinlay?"

Or, "An apple a day keeps the doctor away" would be translated:

"Anway appleway away ayday eepskay ethay octorday awayway"

Your translator should preserve any punctuation and capitalization of words (see examples). For this exercise, you can assume that words will be separated by at least one space.