Lab 7: Lists

CSE/IT 107

Introduction

The purpose of this lab is to dig a little deeper into lists.

Reading

Chapter 10 in Think Python: How to Think Like a Computer Scientist (Think)

Coding Conventions

Follow PEP-8 recommendations for your code.

Problems

Make sure your source code files are appropriately named. Make sure your code has a main function; use `boilerplate.py` you created in Lab 1.

Think Python is available from `http://www.greenteapress.com/thinkpython/`. Page numbers and exercise numbers refer to those found in the PDF file available at the website.

1. Exercise 10.1 *Think Python*, page 91. Name your source code `think_10-1.py`. As you walk through a list you can test an element's type in the following way:

   ```
   lst = [1, [1, 2], [1, 2, 3]]
   for i in lst:
       if type(i) is list:
           print i
   ```

   Your function just needs to work with one level of nesting. In other words, lists that contain lists. Not lists that contain lists that contain lists, etc.

2. Exercise 10.2 *Think Python*, page 91. Name your source code `think_10-2.py`. Your function just needs to work with one level of nesting. In other words, lists that contain lists. Not lists that contain lists that contain lists, etc.
3. Exercise 10.3 *Think Python*, page 92. Name your source code `think_10-3.py`. Find the cumulative sum of the first 100 positive integers.
4. Exercise 10.4 *Think Python*, page 92. Name your source code `think_10-4.py`
5. Exercise 10.5 *Think Python*, page 92. Name your source code `think_10-5.py`
6. Exercise 10.6 *Think Python*, page 98. Name your source code `think_10-6.py`

List Comprehensions

Python has a nifty little feature for generating lists known as list comprehensions. List comprehensions create a list and have two forms:

[*expression* **for** *value* **in** *iterable*]

This is shorthand for the following code:

```
lst = []
for value in iterable:
    lst.append(expression)
```

or you can add a conditional statement:

[*expression* **for** *value* **in** *iterable* **if** *condition*]

This is shorthand for the following code:

```
lst = []
for value in iterable:
    if condition:
        lst.append(expression)
```

For example, to generate a list of squares of the numbers from 1 to n, [1, 4, 9, 16, 25, ..., n^2], you could do

```
squares = []
for k in range(1, n + 1):
    squares.append(k * k)
```

Using a list comprehension, you could do this as one line:

```
squares = [k * k for k in range(1, n + 1)]
```

Or if you wanted to create a list of factors of a number n. Rather than doing this:

```
factors = []
for k in range(1, n + 1):
    if n % k == 0:
        factors.append(k)
```

Using a list comprehension, you could do this:

```
factors = [k for k in range(1, n + 1) if n % k == 0]
```

Place the following 7 problems in a file named `list_comp.py`. Unless otherwise directed, just place the list comprehension in `main()` followed by a print statement.

7. Use a list comprehension to create the list [1, 2, 4, 8, 16, 32, 64, 128, 256].
8. Use a list comprehension to create the list [0, 2, 6, 12, 20, 30, 42, 56, 72, 90].
9. Use a list comprehension to create the list ['a', 'b', 'c', ..., 'z'] without typing all 26 characters.
10. Use a list comprehension to create the list ['Z', 'Y', 'X', ..., 'A'] without typing all 26 characters.

    Since list comprehensions just return a list, you can use them whenever you need a list. For instance, you can find the sum of the squares of the first 100 integers by doing:

```
sum([k * k for k in range(1, 101)])
```

11. Write a function named `sum_cubes` that uses a list comprehension to find the sum of the first n cubes. That is return the sum of [1, 8, 27, ..., n^3]
12. Write a function named `sum_even` that uses a list comprehension to find the sum of the even integers in a list.
13. Write a function named `sum_factors` that uses a list comprehension to find the sum of all the factors of a number n. That is, if $n = 6$, the function should return 12 as the factors of 6 are 1, 2, 3, and 6 ($1 + 2 + 3 + 6 = 12$).

Miscellaneous

14. Given a list of items, write a program that generates a list of lists of the following form:

    [a, b, c, . . ., z] → [[z], [y, z], [x, y, z], . . ., [a, b, c, . . . , y, z]]

    Name your source code `list_list.py`

15. Given a list of numbers, create a new list of numbers such that the first and last numbers are added together and stored as the first number, the second and second-to-last numbers are stored as the second number, and so on. You need to check for even and odd lists. In case the length is odd, the middle value of an odd length list is its final value. Name your source code file `new_list.py`

16. Write a function that takes a string as an argument, converts the string to a list of characters (technically, strings of length one), sorts the list, converts the list back to a string, and returns the new string. Name your source code `string.py`.

17. The Birthday problem. Exercise 10.8 *Think Python*, page 98. Name your source code `birthday.py`. You are doing a simulation, so you are going to run this for $n = 5, 10, 15, 20, ..., 100$ where n is the number of people in the room. For each n the number of simulations is 1000. The probability two people share the same birthday is the number times the list of birthdays contains a duplicate divided by the total number of simulations (1000). Since you are only concerned that a pair of people share the same birthday, you only need to find one duplicate in the list not all duplicates. Use Julian dates for birthdays. That is, a birthday is an integer between [1, 365] (ignoring leap years).

Stacks

One nice feature of lists in Python is that you have built-in functions for stack operations: push and pop. Pushing (adding an element) is implemented via `append()` and popping (remove the top element of the stack) an element is accomplished via `pop()`. Besides the call stack, stacks are very useful in programming. One such use is to evaluate postfix and prefix expressions.

18. Evaluate a Prefix Notation. `http://en.wikipedia.org/wiki/Polish_notation`. Convert the algorithm to evaluate prefix expressions into Python. The algorithm starts at "Scan the given prefix expression from right to left". Read in the prefix expression from the command line. Name your source code file `prefix.py`

Submission

Create a tarball of your *.py files.

```
tar czvf cse107_firstname_lastname_lab7.tar.gz *.py
```

To check the contents of your tarball, run the following command:

```
tar tf cse107_firstname_lastname_lab7.tar.gz *.py
```

You should see a list of your Python source code files.

Upload your tarball in Moodle before the start of you next lab.