Lecture 6: Lists, Nested Loops and Files[*]

Today we'll talk about how to read from a file using python, and how to use nested loops to build on what we learnt about **sequences** such as `str` and `list` in last lecture.

Recall: Helper Functions

Last lecture together we constructed helper functions such as `isVowel`,

```
In [1]: def isVowel(char):
            """Predicate that returns true only when a letter is a vowel."""
            return char.lower() in 'aeiou'

In [2]: def startsWithVowel(word):
            """Predicate that returns true only when a word starts with a vowel."""
            if len(word):  # any length > 0 evaluates to true
                return isVowel(word[0])
            return False

In [3]: def countAllVowels(word):
            '''Returns number of vowels in the word'''
            count = 0 # initialize the counter
            for char in word: # iterate over the word one character at a time
                if isVowel(char):
                    count += 1
            return count

In [4]: # our updates function definition for countChar

        def countChar(char, word):
            '''Counts the number of times a character appears in a word'''
            count = 0
            for letter in word:
                if char.lower() == letter.lower():
                    count += 1
            return count

In [5]: def wordStartEnd(wordList):
            '''Takes a list of words and counts the
            number of words in it that start and end
            with the same letter'''
```

[*]**Acknowledgement.** This notebook has been adapted from the Wellesley CS111 Spring 2019 course materials (http://cs111.wellesley.edu/spring19).

```
        count = 0 #initilize counter
        for word in wordList:
            if len(word):  #why do we need this?
                if word[0].lower() == word[-1].lower():  # will this work?
                    # debugging print here perhaps
                    # print(word)
                    count += 1
        return count
```

Summary. We built various helper functions that iterate over sequences and returns a statistic (in our case counts) of certain things.

List Accumulation

You can imagine wanting to iterate over a sequence and compute a collection of items from it with a cetain property. For example, instead of counting the number of words that start and end with the same letter in a list, we may want to return the words themselves.

To accumulate a collection of items, we can use the lists, almost the same as we accumulate in a counter.

- First, similar to how we initialize a counter, we must initialize our list
- Second, similar to how we update our counter, we must update our list

List updates. Lists a *mutable* structure which means we can update them (delete things from them, add things to them, etc.) . Today we will only look at how we can accumulate using a list: that is how to add things to a given list. Two ways in particular:

- List concatenation
- Appending to a list

In later lectures, we will subtle ways in which these two are different.

```
In [6]: firstList = [1, 2, 3, 4, 5]
        secondList = [8, 9, 10, 11]
```

```
In [7]: newList = firstList + secondList # list concatenation
```

```
In [8]: newList
```

```
Out[8]: [1, 2, 3, 4, 5, 8, 9, 10, 11]
```

List concatenation creates a **new** list and does not modify the original lists.

```
In [9]: firstList
```

```
Out[9]: [1, 2, 3, 4, 5]
```

```
In [10]: secondList
```

```
Out[10]: [8, 9, 10, 11]
```

We can also append a given item to an existing list by using the `append` method. This modifies the list it is called on by adding the item to it.

```
In [11]: firstList.append(6)

In [12]: firstList  #it has changed!

Out[12]: [1, 2, 3, 4, 5, 6]
```

Summary. There are two ways to accumulate in a list: list concatenation or appending to a list. List concatenation uses the + operator and returns a new list which is a concatenation of the two lists. Using the append method on a list modifies it by *appending* the item to it. We will discuss the subtleties between the two next week.

List Accumulation Exercise

It is often the case that we use loops to iterate over a sequence to "accumulate" certain items from it. Suppose someone gave us a list of words and we want to collect all words in that list that start with a vowel. We can use our `isVowel` helper function, but should we approach this problem?

- First we need to be able to iterate over all words in the master list.
- Then, for each word we must check if it starts with a vowel
- If the word starts with a vowel, we want to store it somewhere
- Since we want to store a sequence of such words, we can use a list type

Such processes where we are accumulating something in a list are called *list accumulation*. You can accumulate items in a list using concatenation, similar to strings. For example:

```
In [13]: myList = ['apple', 'orange', 'banana']

In [14]: myList += ['papaya']  # concatenate myList and ['papaya']
         myList

Out[14]: ['apple', 'orange', 'banana', 'papaya']
```

Back to the exercise. Let us now define a function `vowelList` that iterates over a given list of words `wordList` and collects all the words in the list that begin with a vowel (in a new list) and returns that list.

```
In [15]: def vowelWordList(wordList):
             '''Returns a list of words that start with a vowel from the input list'''
             result = [] # initialize list accumulation variable
             for word in wordList:
                 if startsWithVowel(word):
                     result.append(word)
             return result

In [16]: phrase = ['The', 'sun', 'rises', 'in', 'the',\
                                 'east', 'and', 'sets', 'in', 'the', 'west']

In [19]: vowelWordList(phrase)

Out[19]: ['in', 'east', 'and', 'in']
```

Testing Functions

Suppose we want to test a function we have written. There are several ways to do so. You can test using interactively python by importing the function from checking to see if it returns the correct output when called on a bunch of different values.

Testing using doctests. Python's doctest module allows you to embed your test cases and expected output directly into a functions docstring. To use the doctest module we must import it. To make sure the test cases are run when the program is run as a script from the terminal, we need to insert a doctest.testmod(). To make sure that the tests are not run in an interactive shell or when the functions from the module are imported, we should place the command within a guarded if `__name__ == "__main__":` block. See slides for more explanation.

```
In [ ]: def isVowel(char):
            """Predicate that returns true only when a letter is a vowel.
            >>> isVowel('d')
            False
            >>> isVowel('e')
            True
            """
            return char.lower() in 'aeiou'
```

```
In [ ]: import doctest
        doctest.testmod(verbose = True)

        # Task:  try this out as a script and
        # run from the terminal us try this out
```

Nested Loops

Similar to nested if statements, we can nest loops as well. Similar to nested ifs, in a nested loop, the inner loop's body is determined by the indentation. Let us take an example of two nested `for` loops to generate the multiplication table.

```
In [20]: for i in range(5): # This is called the "outer loop"
             for j in range(5): # This is called the "inner loop"
                 print(i, j)
```

```
0 0
0 1
0 2
0 3
0 4
1 0
1 1
1 2
1 3
1 4
2 0
2 1
```

```
2 2
2 3
2 4
3 0
3 1
3 2
3 3
3 4
4 0
4 1
4 2
4 3
4 4
```

```
In [21]: for i in range(2, 6): # This is called the "outer loop"
             for j in range(2, 6): # This is called the "inner loop"
                 print(i, 'x', j, '=', i*j)
```

```
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
3 x 2 = 6
3 x 3 = 9
3 x 4 = 12
3 x 5 = 15
4 x 2 = 8
4 x 3 = 12
4 x 4 = 16
4 x 5 = 20
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
```

What happens if we add conditionals in the outer and/or inner loop? Predict the output of the following examples:

```
In [ ]: for i in range(2, 6):
            for j in range(2, 6):
                if i <= j:
                    print(i, 'x', j, '=', i*j)
```

```
In [ ]: for i in range(2, 6):
            if i % 2 == 0:
                for j in range(2, 6):
                    if i <= j:
                        print(i, 'x', j, '=', i*j)
```

As another simple example of nested loops, this one to print words. Predict the words that will be printed, in order.

```
In [ ]: for letter in ['b','d','r','s']:
            for suffix in ['ad', 'ib', 'ump']:
                print(letter + suffix)
```

File Reading

You can read and write to a file using python commands. Today we will only focus on reading from a file. Next week, we will look at how to write to a file.

Opening a file A file is an object that is created by the built-in function `open`.

```
In [22]: myFile = open('textfiles/prideandprejudice.txt', 'r') # 'r' means open the file for rea
         print(myFile)

<_io.TextIOWrapper name='textfiles/prideandprejudice.txt' mode='r' encoding='UTF-8'>
```

The object returned from `open()` is a type of file object called `io.TextIOWrapper`. In short, it is a type of file object that interprets the file as a stream of text.

```
In [23]: type(myFile)

Out[23]: _io.TextIOWrapper
```

The mode `'r'` for reading the file is optional, because the most basic way for opening a file is for reading:

```
In [24]: myFile2 = open('textfiles/prideandprejudice.txt')
         myFile2

Out[24]: <_io.TextIOWrapper name='textfiles/prideandprejudice.txt' mode='r' encoding='UTF-8'>
```

Aside: Note that by default Python assumes that your text file (in this case, thesis.txt) is encoded in ASCII. Recall, the ASCII table: http://www.asciitable.com/. There are other types of encodings for text, in particular Unicode encoding which encompasses most of the world's writing characters. To learn more about how the computer interprets text encoding, take CS240!

With... as block and Iterating over files

With block to open and close files. Technically when you open a file, you must also close it. To avoid writing code to explicitly open and close, we will use the with... as block which keeps the file open within it.

Within a with...as block, we can iterate over the lines of a file the same way we would iterate over any sequence.

```
In [25]: with open('textfiles/classNames.txt') as roster:  #  roster: name of file object
             for line in roster:
                 print(line)
         # file is implicitly closed here
```

```
Omar Ahmad

Zoe Bennett

Kary Chen

Bernal Cortés

Sarah Dean

Jacob Eckerle

María Fernanda Estrada

Keith Grossman

Ching-Hsien Ho

Sebastian Job

Onder Kilinc

Long Le

William Litton

Lauren Lynch

Marika Massey-Bierman

Lauren McCarey

Victoria Michalska

Avery Mohan

Mohammad Mehdi Mojarradi

Abigail Murray-Stark

Islam Osman

Jonathan Paul
```

```
Maximilian Peters

Salvador Robayo

Evan Ruschil

Sarah Shi

Shikha Singh

Benjamin Siu

April Su

Ryan Watson

Olivia White

Samuel Wolf
```

String functions helpful for file reading. Notice the newline after every line of the file. This is cause by the special newline character \n. We can remove that using the strip() string method. Suppose we wanted to split the name into a list containing the first name and the last name, we can do that with the split method. Lets try out these methods.

```
In [26]: myLine = '   Trying out the strip function to see what it does.  '

In [27]: myLine.strip()  # notice the whitespace removed

Out[27]: 'Trying out the strip function to see what it does.'

In [28]: message = '\n \n Trying out the strip function to see what it does. \n \n'

In [29]: print(message)
```

```


 Trying out the strip function to see what it does.


```

```
In [30]: message.strip()  # notice the whitespace removed

Out[30]: 'Trying out the strip function to see what it does.'
```

```
In [31]: print(message.strip())

Trying out the strip function to see what it does.


In [32]: listOfWords = message.split() # splits line around space and turns it into a list

In [33]: listOfWords

Out[33]: ['Trying',
          'out',
          'the',
          'strip',
          'function',
          'to',
          'see',
          'what',
          'it',
          'does.']

In [1]: # lets try the same example again with .strip()
        with open('textfiles/classNames.txt') as roster:  #  roster: name of file object
            for line in roster:
                print(line.strip())
        # file is implicitly closed here

Omar Ahmad
Zoe Bennett
Kary Chen
Bernal Cortés
Sarah Dean
Jacob Eckerle
María Fernanda Estrada
Keith Grossman
Ching-Hsien Ho
Sebastian Job
Onder Kilinc
Long Le
William Litton
Lauren Lynch
Marika Massey-Bierman
Lauren McCarey
Victoria Michalska
Avery Mohan
Mohammad Mehdi Mojarradi
Abigail Murray-Stark
Islam Osman
Jonathan Paul
Maximilian Peters
```

```
Salvador Robayo
Evan Ruschil
Sarah Shi
Shikha Singh
Benjamin Siu
April Su
Ryan Watson
Olivia White
Samuel Wolf
```

Exercises with Files and Nested Loops

Now that we know how to write nested loops, accumulate in lists and read from files, let us do some fun exercises with these concepts. We will build these examples live in class.