CS134: Lists & Loops
Announcements & Logistics

- **Homework 3** is due Monday @ 10 pm
- **Lab 1** graded feedback was released on Wed
  - Any problems? Email cs134staff@williams.edu
- **Lab 3** starter code will be pushed today
  - Try to spend 30-60 minutes on it before your scheduled lab
  - A collection of word puzzles: can use your newly acquired knowledge of strings, lists (today), functions and loops to solve them

Do You Have Any Questions?
Last Time

- Started discussing sequences in Python
  - Focused on **strings** (sequences of characters)
  - Discussed **slicing** [: :], **indexing** [], **in** operators on strings
    - Note: We also already know about the + operator on strings
    - Note: There is a **not in** operator addition to **in**
  - Also learned about string **methods**.lower() and .upper()
    - Note: There are also string methods .islower() and .isupper() that return True if string is in lowercase/uppercase, else return False
Today’s Plan

• Learn about **for loops** for iterating over sequences

• Introduce a new sequence: **Lists**
  • Apply indexing [], slicing [:], **in**, + operators to lists

• Start building a collection of functions that iterate over sequences (lists and strings)
For Loops
Iterating with **for** Loops

- One of the most common ways to manipulate a sequence is to perform some action **for each element** in the sequence.
- This is called **looping** or **iterating** over the elements of a sequence.
- Syntax of a for loop:

```python
for var in seq:
    # body of loop
    (do something)
```

- `var` is called the loop variable.
- `seq` is a sequence (for example, a string).
Iterating with **for** Loops

- As the loop executes, the loop variable (\texttt{char} in this example) takes on the value of each of the elements of the sequence one by one.

```python
>>> # simple example of for loop
>>> word = "Williams"

>>> for char in word:
...     print(char)

W
i
l
l
i
a
m
s

Williams
```

This is a special kind of **for..loop** called a **for-each loop**.

Why might we call it that?
Counting Vowels

• We can use a for loop to improve our `countVowels()` function.
• Notice how `count` “accumulates” values in the loop.
• We call `count` an accumulation variable.
• Works for any string!

```python
def countVowels(word):
    ''' Takes a string as input and returns the number of vowels in it'''

    count = 0 # initialize the counter

    # iterate over the word one character at a time
    for char in word:
        if isVowel(char): # call helper function
            count += 1

    return count
```
Counting Vowels: Tracing the Loop

- How are the local variables updated as the loop runs?

```python
def countVowels(word):
    '''Returns number of vowels in the word'''
    count = 0
    for char in word:
        if isVowel(char):
            count += 1
    return count

countVowels('Boston')
```

Loop variable

<table>
<thead>
<tr>
<th>word</th>
<th>'Boston'</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>0</td>
</tr>
<tr>
<td>char</td>
<td>'B' 'o' 's' 't' 'o' 'n'</td>
</tr>
</tbody>
</table>
Counting Vowels: Tracing the Loop

- How are the local variables updated as the loop runs?

```python
def countVowels(word):
    '''Returns number of vowels in the word'''
    count = 0
    for char in word:
        if isVowel(char):
            count += 1
    return count
```

```python
countVowels('Boston')
```

<table>
<thead>
<tr>
<th>char</th>
<th>'B'</th>
<th>'o'</th>
<th>'s'</th>
<th>'t'</th>
<th>'o'</th>
<th>'n'</th>
</tr>
</thead>
<tbody>
<tr>
<td>word</td>
<td>'Boston'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>count</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Loop variable
Counting Vowels: Tracing the Loop

• How are the local variables updated as the loop runs?

def countVowels(word):
    '''Returns number of vowels in the word'''
    count = 0
    for char in word:
        if isVowel(char):
            count += 1
    return count

countVowels('Boston')

Loop variable

word  'Boston'
count 1
char  'B'  'o'  's'  't'  'o'  'n'
Counting Vowels: Tracing the Loop

- How are the local variables updated as the loop runs?

```python
def countVowels(word):
    '''Returns number of vowels in the word'''
    count = 0
    for char in word:
        if isVowel(char):
            count += 1
    return count
```

```python
countVowels('Boston')
```

- Loop variable

- `word` variable

- `count` variable

- `char` variable
Counting Vowels: Tracing the Loop

- How are the local variables updated as the loop runs?

```python
def countVowels(word):
    '''Returns number of vowels in the word'''
    count = 0
    for char in word:
        if isVowel(char):
            count += 1
    return count
```

```python
word = 'Boston'
countVowels(word)
```

```
word: 'Boston'

char: 'B'  'o'  's'  't'  'o'  'n'

count: 2
```
Counting Vowels: Tracing the Loop

• How are the local variables updated as the loop runs?

def countVowels(word):
    '''Returns number of vowels in the word'''
    count = 0
    for char in word:
        if isVowel(char):
            count += 1
    return count

countVowels('Boston')
Exercise: Count Characters
Exercise: Count Characters

- Define a function `countChar()` that takes two arguments, a character and a word (both strings), and returns the number of times (int) that character appears in the word (ignoring case).

```python
def countChar(char, word):
    '''Counts # of times char appears in word'''
    pass

>>> countChar('m', "ammonia")
2

>>> countChar('a', "Alabama")
4

>>> countChar('a', "rhythm")
0
```
Exercise: Count Characters

- Define a function `countChar()` that takes two arguments, a character and a word (both strings), and returns the number of times (int) that character appears in the word (ignoring case).

```python
def countChar(char, word):
    '''Counts # of times char appears in word'''
    count = 0  # initialize accumulation var
    for letter in word:  # letter is the loop variable
        if char.lower() == letter.lower():
            count += 1  # increment count (accumulate)
    return count
```
Exercise:
Vowel Sequences
Exercise: Vowel Sequences

- Define a function `vowelSeq()` that takes a string `word` as input and returns a string containing all the vowels in `word` in the same order as they appear.

```python
def vowelSeq(word):
    '''Returns the vowel subsequence in word'''
    pass

>>> vowelSeq("Chicago")
'iao'

>>> vowelSeq("protein")
'oei'

>>> vowelSeq("rhythm")
''
```

What might be other good values to test edge cases?
Exercise: Vowel Sequences

- Define a function `vowelSeq()` that takes a string `word` as input and returns a string containing all the vowels in word in the same order as they appear.
- Accumulation variables don’t have to be counters! Can accumulate strings as well

```python
def vowelSeq(word):
    '''returns the vowel subsequence in word'''
    vowels = ""  # accumulation variable
    for char in word:  # char is loop variable
        if isVowel(char):  # if char is a vowel
            vowels += char  # accumulate characters
    return vowels
```
Code from today can be found in sequenceTools.py
Lists
Moving on: Lists

- Lists are another type of sequence in Python
- Definition: A list is a comma separated, ordered sequence of values
- Unlike strings, which can only contain characters, lists can be collections of heterogenous objects (strings, ints, floats, etc)
- Today we’ll focus on iterating over lists (i.e., looking at the elements sequentially) using for loops
- In upcoming lectures we’ll focus on manipulating and using lists to store dynamic sequences of objects
Lists

- Lists are:
  - Comma separated, ordered sequences of values
  - Heterogenous collections of objects
  - Mutable (or “changeable”) objects in Python. In contrast, strings are immutable (they cannot be changed).
  - We will discuss mutability in more detail soon!

```python
In [1]: # Examples of various lists:

wordList = ['What', 'a', 'beautiful', 'day']
numList = [1, 5, 8, 9, 15, 27]
charList = ['a', 'e', 'i', 'o', 'u']
mixedList = [3.145, 'hello', 13, True] # lists can be heterogenous

In [2]: type(numList)
```

Out[2]: list
Operations on Sequences

• We already saw several sequence operators and functions last time
  • We looked at strings last time
  • These apply to lists as well!
• We can do the following operations on lists:
  • Indexing elements of lists using [] operator
  • Slicing lists using [:] operator
  • Testing membership using in/not in operators
  • Concatenation using + operators
  • Using len() function to find length of list
## Basic Operations on Sequences

- **Indexing lists using `[ ]**
  ```python
  In [1]: wordList = ['What', 'a', 'beautiful', 'day']
         
         wordList[3]
  Out[1]: 'day'
  ```

- **Finding length of list using `len()`**
  ```python
  In [2]: len(wordList)
  Out[2]: 4
  ```

- **Slicing lists using `[:]` (can also use optional step)**
  ```python
  In [4]: nameList = ['Aamir', 'Beth', 'Chris', 'Daxi', 'Emory']
  
  In [5]: nameList[2:4]
  Out[5]: ['Chris', 'Daxi']
  ```
Membership in Sequences

• Recall: The `in` operator in Python is used to test if a given sequence is a subsequence of another sequence; returns True or False

```python
In [20]: nameList = ["Anna", "Beth", "Chris", "Daxi", "Emory", "Fatima"]

In [28]: "Anna" in nameList # test membership
Out[28]: True

In [30]: "Jeannie" in nameList
Out[30]: False
```
not in sequence operator

- The `not in` operator in Python returns True if and only if the given element is **not** in the sequence.

```python
In [20]: nameList = ["Anna", "Beth", "Chris", "Daxi", "Emory", "Fatima"]

In [28]: "Anna" in nameList  # test membership
Out[28]: True

In [30]: "Jeannie" in nameList
Out[30]: False

In [31]: "Jeannie" not in nameList  # not in returns true if el not in seq
Out[31]: True

In [33]: "a" not in "Chris"
Out[33]: True
```

Note that `not in` also works for strings.
List Concatenation

- We can use the `+` operator to **concatenate** lists together.
- Creates a **new list** with the combined elements of the sublists.
- Does not modify original lists!

```python
aList = ['the', 'quick', 'brown', 'fox']
bList = ['jumped', 'over', 'the', 'dogs']
aList + bList  # concatenate lists
['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'dogs']

aList  # aList is unchanged
['the', 'quick', 'brown', 'fox']

bList = bList + ['back']  # add 'back' to bList

bList  # since we reassign result to bList, bList has changed
['jumped', 'over', 'the', 'dogs', 'back']
```

To change bList, we have to reassign bList to the new list.
Looping over Lists

- We can **loop** over **lists** the same way we looped over **strings**
- As before, the **loop variable** iteratively takes on the values of each item in the list, starting with the 0th item, then 1st, until the last item
- The following loop iterates over the list of ints, printing each item in it

```
In [15]: numList = [0, 2, 4, 6, 8, 10]
```

```
In [16]: for num in numList:
    print(num)
```

0
2
4
6
8
10
List Exercises
Exercise: `countItem`

- Let’s write a function `countItem()` that takes as input a sequence `seq` (can be a string or a list), and an element `el`, and returns the number of times `el` appears in the sequence `seq`.

```python
def countItem(seq, el):
    """Takes seq as input, and returns the number of times el appears in seq""
    pass
```
Exercise: countItem

• Let’s write a function `countItem()` that takes as input a sequence `seq` (can be a string or a list), and an element `el`, and returns the number of times `el` appears in the sequence `seq`.

```python
def countItem(seq, el):
    """Takes seq as input, and returns the number of times el appears in seq""
    count = 0  # initialize counter

    for item in seq:
        if item == el:  # if this item matches el
            count += 1  # increment counter
    # else do nothing, go to next item
    return count
```

Another accumulation variable!
Exercise: `wordStartEnd`

- Write a function that iterates over a given list of strings `wordList`, returns a (new) list containing all the strings in `wordList` that start and end with the same character (ignoring case).

```python
def wordStartEnd(wordList):
    ''' Takes a list of words wordList and returns a list of all words in wordList that start and end with the same letter''
    pass

>>> wordStartEnd(['Anna', 'banana', 'salad', 'Rigor', 'tacit', 'hope'])
['Anna', 'Rigor', 'tacit']

>>> wordStartEnd(['New York', 'Tokyo', 'Paris'])
[]

>>> wordStartEnd(['*Hello*', '', 'nope'])
['*Hello*']
```
Step by step approach (organize your work):

- Go through every word in wordList
- Check if word starts and ends at same letter*
- If true, we need to “collect” this word (remember it for later!)
  - Else, just go on to next word
- Takeaway: need a new list to accumulate desirable words

*Break down bigger steps (decomposition!)

- If word starts and ends at same letter:
  - Can do this using string indexing
  - Think about corner cases: what if string is empty? what about case?
Exercise: `wordStartEnd`

- Write a function that iterates over a given list of strings `wordList`, returns a (new) list containing all the strings in `wordList` that start and end with the same character (ignoring case).

```python
def wordStartEnd(wordList):
    ''' Takes a list of words and returns a list of words in it that start and end with the same letter'''
    result = []
    for word in wordList:
        # check for empty strings before indexing
        if len(word) != 0:
            if word[0].lower() == word[-1].lower():
                result += [word]  # concatenate to resulting list
    return result  # notice the indentation of return
```

Notice this syntax! We are adding `word` (a string) to `result` (a list).