CS 134:
Strings, Lists, & Ranges
Announcements & Logistics

- **Lab 1** feedback returned last week
- **Lab 2** feedback coming soon (by Wed)
- **Lab 3** is tonight/tomorrow, due Wed/Thur at 10pm
  - Covers lists, strings, and loops!
- **HW 3** due tonight on Glow

Do You Have Any Questions?
Last Time

• Reviewed iterating over sequences with for loops
  • Used accumulation variables to collect "items" from sequences
• Introduced new sequence: lists
  • Learned how to index, slice, concatenate, iterate over lists just like we did with strings
Recap: Iterating with for Loops

• Suppose we want to perform an action for each element in a sequence
• This is called looping or iterating over the elements of a sequence

• Syntax of a for loop:

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

var is called the loop variable
seq is a sequence (for example, a string)
Recap: 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?

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')

word  'Boston'
count  0
char  'B'  'o'  's'  't'  'o'  'n'
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')

word  'Boston'

count  1

char  'B'  'o'  's'  't'  'o'  'n'  '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
countVowels('Boston')
```

Counting Vowels:  Tracing the Loop

<table>
<thead>
<tr>
<th>word</th>
<th>'Boston'</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>1</td>
</tr>
<tr>
<td>char</td>
<td>'B' 'o'</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

countVowels('Boston')
```

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

Loop 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

countVowels('Boston')
```

<table>
<thead>
<tr>
<th>Loop variable</th>
<th>count</th>
<th>word</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>count</td>
<td>'Boston'</td>
</tr>
<tr>
<td>'B'</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>'o'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>'s'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>'t'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>'o'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>'n'</td>
<td></td>
<td></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

countVowels('Boston')
```

Trace:
- `word` = 'Boston'
- `count` = 0
- `char` = 'B'
  - `count` = 0
- `char` = 'o'
  - `count` = 1
- `char` = 's'
  - `count` = 1
- `char` = 't'
  - `count` = 1
- `char` = 'o'
  - `count` = 2
- `char` = 'n'
  - `count` = 2

Result: `count = 2`
Recap: `wordStartEnd`

- Write a function that iterates over a given list of words `wordList`, and returns a (new) list containing all the words in `wordList` that start and end with the same letter (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'''
    # initialize accumulation variable (of type list)
    result = []
    for word in wordList:  # iterate over list

        # 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
```
Recap: wordStartEnd

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

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

# Accumulating in a list. Always initialize our accumulation variable before we enter loop.
# List concatenation
```
Today’s Plan

• Learn about **nested for loops**
• Review **sequence** operations
• Review **list** and **string** operations (so far!)
  • Discuss convenient method and functions for working with strings and lists (we’ll continue to expand on this in upcoming lectures)
  • Investigate list **mutability** versus string **immutability**
• (Maybe) Learn about one more sequence: **ranges**
Nested Loops
Nested Loops

• A **for loop** body can contain one (or more!) additional **for loops**:
  • Called **nested for loops**
  • Conceptually similar to nested conditionals
• Example: What do you think is printed by the following Python code?

```python
# What does this do?

def mysteryPrint(word1, word2):
    """Prints something""
    for char1 in word1:
        for char2 in word2:
            print(char1, char2)

mysteryPrint('123', 'abc')
```
In [9]: # What does this do?

def mysteryPrint(word1, word2):
    """Prints something""
    for char1 in word1:
        for char2 in word2:
            print(char1, char2)

In [11]: mysteryPrint(‘123’, ‘abc’)

1 a  char1 = 1  char2 = a
1 b  char2 = b
1 c  char2 = c
2 a  char1 = 2  char2 = a
2 b  char2 = b
2 c  char2 = c
3 a  char1 = 3  char2 = a
3 b  char2 = b
3 c  char2 = c

Inner loop (w/ char2 and word2) runs to completion on each iteration of the outer loop
Let’s look at another example involving lists of strings

What is printed by the nested loop below?

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

# What does this print?
In [12]: # What does this print?

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

```
bad  bib  bump
dad  dib  dump
rad  rib  rump
sad  sib  sump
```

Inner loop (w/ suffixes) runs to completion on each iteration of the outer loop (w/ prefixes)
Lists and Strings Revisited
Sequence Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq[i]</td>
<td>The i'th item of seq, when starting with index 0</td>
</tr>
<tr>
<td>seq[si:ee]</td>
<td>slice (subsequence) of seq from si to ee</td>
</tr>
<tr>
<td>seq[si:ee:s]</td>
<td>slice of seq from si to ee with step s</td>
</tr>
<tr>
<td>len(seq)</td>
<td>length of seq</td>
</tr>
<tr>
<td>seq1 + seq2</td>
<td>The concatenation of seq1 and seq2</td>
</tr>
<tr>
<td>x in seq</td>
<td>True if x is contained within seq</td>
</tr>
<tr>
<td>x not in seq</td>
<td>False if x is contained within seq</td>
</tr>
<tr>
<td>seq<em>n, n</em>seq</td>
<td>n copies of seq concatenated</td>
</tr>
<tr>
<td>min(seq)</td>
<td>smallest item of seq</td>
</tr>
<tr>
<td>max(seq)</td>
<td>largest item of seq</td>
</tr>
</tbody>
</table>

All of these operators work on both **strings** and **lists**!
Sequence Operations with Strings

"a" in "aeiou"  # in operator
True

"b" not in "aeiou"  # not in operator
True

"CS" + "134"  # concatenation with +
'CS134'

"abc" * 3  # * operator
'abcabcabc'

myString = "abc"
myString[1]  # indexing with []
'b'

myString[1:2]  # slicing with [:]
'b'

# using negative step in slicing
myString[::-1]
'cba'

len(myString)  # length function
3

# min function (finds smallest character)
min(myString)
'a'

# max function (finds largest character)
max(myString)
'c'
Sequence Operations with Lists

1 in [1, 2, 3]  # in operator
True

1 not in [1, 2, 3]  # not in operator
False

[1] + [2]  # concatenation with +
[1, 2]

[1, 2] * 3  # * operator
[1, 2, 1, 2, 1, 2]

myList = [1, 2, 3]
myList[1]  # indexing with []
2

myList[1:2]  # slicing with [:]
[2]

# slicing with negative step
myList[:::-1]
[3, 2, 1]

len(myList)  # len function
3

min(myList)  # min function
1

max(myList)  # max function
3
String Methods

• What about the `.upper()` & `.lower()` methods?

• Return new string containing lowercase/uppercase characters from original string

```python
>>> astring = "Pixel"
>>> astring.upper()
'PIXEL'
>>> astring.lower()
'pixel'
```
String Methods

• What about the `.upper()` & `.lower()` methods?

• Will they work on lists?

```python
>>> alist = ['P', 'i', 'Xe', 'L']
>>> alist.upper()
Traceback (most recent call last):
  File "<stdin>" , line 1, in <module>
AttributeError: 'list' object has no attribute 'upper'
```

```python
>>> alist.islower()
Traceback (most recent call last):
  File "<stdin>" , line 1, in <module>
AttributeError: 'list' object has no attribute 'islower'
```
String Operations, Methods, and Functions
str() function

- str() function allows us to convert other data types to strings

```python
>>> classList = ['C', 's', 'CI', 1, 3, 4]
>>> str(classList)
"['C', 's', 'CI', 1, 3, 4]"

>>> str(134)
'134'

>>> str(6.02)
'6.02'
```

Converting a list to a string in this way is somewhat limiting.
List to Strings: `join()`

- Given a list of strings, the `.join()` string method, when applied to a string separator, concatenates the strings together with the string separator between them.

- `.join()` requires a list to be passed as a parameter, and elements of the list must be strings.

```python
>>> songList = ["Mary", "had", "a", "bicycle"]
>>> '*'.join(songList)
'Mary*had*a*bicycle'
>>> '_'.join(songList)
'Mary_had_a_bicycle'
>>> ' '.join(songList)
'Mary had a bicycle'
```
String to Lists: `split()`

- `.split()` is a string **method** that splits strings at “spaces” (the default separator) and returns a list of (sub)strings

- Can optionally specify other **delimiters** (or separators) as well

```python
>>> phrase = "What a lovely day"
>>> phrase.split()
['What', 'a', 'lovely', 'day']

>>> newPhrase = "What a *lovely*      day!"
>>> newPhrase.split()
['What', 'a', '*lovely*', 'day!']

>>> lovelyDays = "Friday, Saturday,Sunday"
>>> lovelyDays.split(',
['Friday', ' Saturday', 'Sunday']
```

Blank space is default separator (ignores punctuation)
Remove whitespace w/ `strip()`

- The `.strip()` string method strips away whitespace and (sometimes hidden) new line (`\n`) characters from the beginning and end of strings and **returns a new string**.

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```

```python
>>> word = "   ** Cozy Autumns **   
>>> word.strip()
'** Cozy Autumns **'
```
More Useful String Methods!

• **word.find(s)**
  - Return the first (or last) position (index) of string s in word. Returns -1 if not found.

• **char.isspace()**
  - Returns `True` if char is not empty (""") and char is composed of white space (or lowercase, uppercase, alphabetic letters, digits, or either letters or digits).
  - Can also do: `islower()`, `isupper()`, `isalpha()`, `isdigit()`, `isalnum()`.

• **word.count(s)**
  - Returns the number of (non-overlapping) occurrences of s in word

• **word.index(s)**
  - Return the lowest index in word where substring s is found. Returns ValueError if not found.

• **word.replace(old, new)**
  - Returns a new string with all occurrences of substring old in word replaced by new.

• Many, many more: see `pydoc3 str`
String Methods in Action

word = 'Williams College'
word.split()  # ['Williams', 'College']
word.upper()  # 'WILLIAMS COLLEGE'
word.lower()  # 'williams college'
word.replace('iams', 'herst')  # 'Willherst College'
word.replace('mith', 'herst')  # 'Williams College'

newWord = '   Spacey College   '  # 'Spacey College'
newWord.strip()

myList = ['Williams', 'College']
' '.join(myList)  # 'Williams College'

Important note: Strings are immutable. None of these operations change/affect the original string. They all return a new string!
List Operations, Methods, and Functions
list() Function

- **list() function**, when given another sequence (like a string), returns a list of elements in the sequence.

```python
word = "Computer Science!"

list(word)  # can turn a string into a list of its characters
```

```
['C', 'o', 'm', 'p', 'u', 't', 'e', 'r', 'S', 'c', 'i', 'e', 'n', 'c', 'e', '!' ]
```

```python
list(str(3.14159265))
```

```
['3', '.', '1', '4', '1', '5', '9', '2', '6', '5']
```
Modifying Lists

- Lists are **mutable** data structures
  - This means we can update them (delete things from them, add things to them, etc.)
- List **concatenation** (using +) creates a new list and does not modify any existing list
  - Important point: Concatenating to a list returns a new list!

- We can also append to or extend a list, which modifies the existing list
  - The list method `myList.append(item)` modifies the list `myList` by adding `item` to it at the end
  - The list method `myList.extend(otherList)` modifies the list `myList` by adding all elements from `otherList` to `myList` at the end
  - Often more efficient to append/extend rather than concatenate
  - But we have to be very careful when modifying the list
  - **Important point:** Appending to or extending a list modifies the existing list!
Adding elements to a List

- Here are a few examples that show how to use the list `.append()` method vs + operator to add items to the end of an existing list.

```python
numList = [1, 2, 3, 4, 5]

numList + [6]  # list concatenation
[1, 2, 3, 4, 5, 6]  # this is a new list!

numList  # numList has not changed
[1, 2, 3, 4, 5]

numList.append(6)  # list append, notice dot notation
numList  # numList has been updated to include 6
[1, 2, 3, 4, 5, 6]
```
More Useful List Methods

- `myList.extend(itemList)`: *appends all items* in `itemList` to the end of `myList` (modifying `myList`)
- `myList.count(item)`: counts and returns the number (int) of times `item` appears in `myList`
- `myList.index(item)`: returns the first index (int) of `item` in `myList` if it is present, else throws an error

```python
myList = [1, 7, 3, 4, 5]
myList.extend([6, 4])
myList
[1, 7, 3, 4, 5, 6, 4]
myList.count(4)
2
myList.index(3)
2
myList.index(10)
ValueError
```
}<ipython-input-38-14d2e386c720: ----> 1 myList.index(10)

`ValueError`: 10 is not in list
Summarizing Mutability in Strings vs Lists

Strings are **immutable**

- Once you create a string, it cannot be changed!
- All operations that we have seen on strings return a new string and do not modify the original string

Lists are **mutable**

- Lists are mutable (or changeable) sequences
- You can concatenate items to a list using +, but this does not change the list
- You can append items using append() method, and this does change the list
Ranges
Moving on: Ranges (another sequence!)

- Python provides an easy way to iterate over numerical sequences using *ranges*, another sequence data type
- When the `range()` function is given two integer arguments, it returns a *range object* of all integers starting at the first and up to, *but not including*, the second; if the first integer is 0, it may be omitted.
- To see the values included in the range, we can pass our range to the `list()` function which returns a list of them

```
In [1]: range(0,10)
Out[1]: range(0, 10)

In [2]: type(range(0, 10))
Out[2]: range

In [3]: list(range(0, 10))
Out[3]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [4]: list(range(10))
Out[4]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```
Moving on: Ranges (another sequence!)

- Python provides an easy way to iterate over numerical sequences using **ranges**, another sequence data type.
- When the `range()` function is given two integer arguments, it returns a **range object** of all integers starting at the first and up to, but not including, the second; if the first integer is 0, it may be omitted.
- To see the values included in the range, we can pass our range to the `list()` function which returns a **list** of them.

```
In [1]: range(0,10)
Out[1]: range(0, 10)

In [2]: type(range(0, 10))
Out[2]: range

In [3]: list(range(0, 10))
Out[3]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [4]: list(range(10))
Out[4]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

A range is a type of sequence in Python (like string and list)

To see elements in range, pass range to list() function

First argument omitted, defaults to 0
Loops and Ranges to Print Patterns

- Sometimes we might use a for loop, not to iterate over a sequence, but just to repeat a task over and over. The following loops print a pattern to the screen. (Look closely at the indentation!)

```python
# what does this print?  # what does this print?
for i in range(5):
    print('\$' * i)
for j in range(5):
    print('*' * j)
```

```python
for i in range(5):
    print('$' * i)
for j in range(5):
    print('*' * j)
```

What are the values of i and j???
Iterating Over Ranges

# what does this print?
for i in range(5):
    print('$' * i)
for j in range(5):
    print('*' * j)

# what does this print?
for i in range(5):
    print('$$' * i)
    for j in range(i):
        print('*' * i)
Iterating Over Ranges

# what does this print?

```python
for i in range(5):
    print('$' * i)
for j in range(5):
    print('*' * j)
```

```
i = 0
$  
i = 1
$$ 
i = 2
$$$ 
i = 3
$$$$ 
i = 4

j = 0
*  
j = 1
** 
j = 2
*** 
j = 3
**** 
j = 4
```

# what does this print?

```python
for i in range(5):
    print('$' * i)
    for j in range(i):
        print('*' * i)
```

```
i = 0
$  
i = 1
  * 
i = 2
    * 
i = 3
      * 
i = 4
        *
```

```
i = 0
$  
i = 1
  j = 0

j = 0
*  
j = 1
  ** 
j = 2
    $$ 
j = 3
      $$$ 
j = 4
        $$$$
```

```
i = 0
$  
i = 1
  j = 0

j = 0
  * 
j = 1
    ** 
j = 2
      *** 
j = 3
        **** 
j = 4
```
Lab 3
Lab 3: Goals

- In this lab, you will accomplish two tasks:
  - Construct a module of tools for manipulating strings and lists of strings (in wordTools.py)
  - Use your toolbox to answer some (fun?) trivia questions (in puzzles.py)
- You will gain experience with the following:
  - Sequences (lists and strings), and associated operators/methods
  - Writing simple and nested for loops
  - Writing doctests to test your functions
Testing Functions: Doctests

• We have already seen two ways to test a function
  • You can run your code 1) interactively or 2) as a script
• Python’s doctest module allows you to embed test cases and expected output directly into a function’s docstring
• To use the doctest module, we must import it using:
  from doctest import testmod
• To make sure the test cases are run when the program is run as a script from the terminal, we then need to call testmod().
• To ensure that the tests are not run in interactive Python, we place this command within a “guarded” if block:
  if __name__ == '__main__':
Testing Functions: Doctests

```python
def isVowel(char):
    """""""Takes a letter as input and returns true if and only if it is a vowel.
    >>> isVowel('e')
    True
    >>> isVowel('U')
    True
    >>> isVowel('t')
    False
    >>> isVowel('Z')
    False
    """""""
    return char.lower() in 'aeiou'

if __name__ == '__main__':
    # the following code tests the tests in the docstrings ('doctests').
    # as you add tests, re-run this as a script to test your work
    from doctest import testmod  # this import is necessary when testing
    testmod()  # test this module, according to the doctests
```

Run the doctests only when file is executed as a script