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

- **Lab 3** is due tonight/tomorrow at 11 pm
- **HW 4** will be posted later today
- If you are having problems with anything, please come see us during office hours
  - Always refer to course calendar for updated hours!

Do You Have Any Questions?
Last Time

• Reviewed iterating over **sequences** with **for loops**
  • Used **accumulation variables** to collect "items" from sequences, e.g., vowel sequences, counters, etc
  • Looked at **nested for loops**
• Introduced new sequence: **lists**
  • Learned how to index, slice, concatenate, iterate over lists just like we did with strings
  • Example: wordStartEnd
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""
    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 return
```

Accumulating in a list. Always initialize our accumulation variable before we enter loop.

List concatenation
Today’s Plan

• 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**

• Introduce **range** data types and ways to iterate over numerical sequences
### Review: Sequence Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x in seq</code></td>
<td>True if an item of seq is equal to x</td>
</tr>
<tr>
<td><code>x not in seq</code></td>
<td>False if an item of seq is equal to x</td>
</tr>
<tr>
<td><code>seq1 + seq2</code></td>
<td>The concatenation of seq1 and seq2</td>
</tr>
<tr>
<td><code>seq*n, n*seq</code></td>
<td>n copies of seq concatenated</td>
</tr>
<tr>
<td><code>seq[i]</code></td>
<td>i’th item of seq, where origin is 0</td>
</tr>
<tr>
<td><code>seq[i:j]</code></td>
<td>slice of seq from i to j</td>
</tr>
<tr>
<td><code>seq[i:j:k]</code></td>
<td>slice of seq from i to j with step k</td>
</tr>
<tr>
<td><code>len(seq)</code></td>
<td>length of seq</td>
</tr>
<tr>
<td><code>min(seq)</code></td>
<td>smallest item of seq</td>
</tr>
<tr>
<td><code>max(seq)</code></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

```python
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
```
List Operations, Methods, and Functions
list() Function

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

In [32]: word = "Computer Science!"

In [33]: list(word) # can turn a string into a list of its characters

Out[33]: ['C', 'o', 'm', 'p', 'u', 't', 'e', 'r', 'S', 'c', 'i', 'e', 'n', 'c', 'e', '!

In [30]: list(str(3.14159265))

Out[30]: [ '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
In [8]: numList = [1, 2, 3, 4, 5]

In [9]: numList + [6]  # list concatenation
Out[9]: [1, 2, 3, 4, 5, 6]  # this is a new list!

In [10]: numList  # numList has not changed
Out[10]: [1, 2, 3, 4, 5]

In [12]: numList.append(6)  # list append, notice dot notation

In [14]: numList  # numList has been updated to include 6
Out[14]: [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
In [39]: myList = [1, 7, 3, 4, 5]

In [40]: myList.extend([6, 4])

In [41]: myList
Out[41]: [1, 7, 3, 4, 5, 6, 4]

In [42]: myList.count(4)
Out[42]: 2

In [43]: myList.index(3)
Out[43]: 2

In [38]: myList.index(10)

ValueError: 10 is not in list
```
String Operations, Methods, and Functions
The `str()` function allows us to convert other data types to strings. For example:

- **In [1]:** `myList = [2, 3, 4]`
- **In [2]:** `str(myList)`
  - **Out[2]:** `'[2, 3, 4]'`
- **In [3]:** `str(1)`
  - **Out[3]:** `'1'`
- **In [4]:** `str(2.3)`
  - **Out[4]:** `'2.3'`

Converting a list to a string in this way is somewhat limiting.
List to Strings: \texttt{join()}

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

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

```python
In [11]: wordList = ['Everybody', 'is', 'looking', 'forward', 'to', 'the', 'weekend']

In [12]: '*'.join(wordList)
Out[12]: 'Everybody*is*looking*forward*to*the*weekend'

In [13]: '_'.join(wordList)
Out[13]: 'Everybody_is_looking_forward_to_the_weekend'

In [14]: ' '.join(wordList)
Out[14]: 'Everybody is looking forward to the weekend'
```
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
In [5]: phrase = "What a lovely day"

In [6]: phrase.split()
Out[6]: ['What', 'a', 'lovely', 'day']

In [7]: newPhrase = "What a *lovely* day!"  # multiple spaces or punctuations dont matter

In [8]: newPhrase.split()
Out[8]: ['What', 'a', '*lovely*', 'day!']

In [9]: commaSepSpells = "Impervius, Portus, Lumos, Reducio, Protego"  #comma separated strings

In [10]: commaSepSpells.split(',')
Out[10]: ['Impervius', 'Portus', 'Lumos', 'Reducio', 'Protego']
```
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
In [1]: word = " ** Snowy Winters ** 

In [2]: word.strip()

Out[2]: ' ** Snowy Winters ** '

In [8]: "\nHello World\n".strip()

Out[8]: 'Hello World'
```
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.

- `replace(old, new)`
  - Return a string with all occurrences of substring old replaced by new.

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

```python
word = 'Williams College'
word.split()
word.upper()
word.lower()
word.replace('iams', 'eslley')
word.replace('tent', 'eselley')
newWord = ' Spacey College '  
newWord.strip()
myList = ['Williams', 'College']
' '.join(myList)
```

**Returned value**

```python
['Williams','College']
'WILLIAMS COLLEGE'
'williams college'
'Willeslley College'
'Williams College'
'Spacey College'
'Williams College'
```

Important note: Strings are **immutable**. None of these operations change/affect the original string. They all **return a new string**!
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
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]: \textbf{range}(0,10)  
Out[1]: range(0, 10)  

In [2]: \textbf{type}(\textbf{range}(0, 10))  
Out[2]: range  

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

In [4]: \textbf{list}(\textbf{range}(10))  
Out[4]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Python provides an easy way to iterate over numerical sequences using ranges, another sequence data type.

To see the values included in the range, we can pass our range to the list() function which returns a list of them.

```python
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.
• 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(i):
    print('*' * i)
```

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('*' * j)
```

i = 0

$ i = 1

$ i = 2

* i = 3

* i = 4

j = 0

j = 1

j = 2

j = 3

j = 4

i, not j!