CSCI 134 Fall 2021:
(More) Lists and Loops

September 27, 2021

Shikha Singh, 9AM
Jeannie Albrecht, 10AM
Announcements & Logistics

• **Homework 3** is due tonight 10 pm

• **Lab 3** is today and tomorrow, due Wed 10 pm/Thurs 10 pm
  • Lab 3 is a collection of word puzzles: can use our knowledge of strings, functions and loops to solve them

• If you are having problems with anything, please come see us during office hours!

Do You Have Any Questions?
Lab Grading Guidelines

- **A+**: An absolutely perfect submission (both in terms of correctness and style) that goes above and beyond our expectations.

- **A**: A submission that meets every requirement and has no mistakes (even style is perfect!)

- **A-**: A submission where everything works with 1-2 minor mistakes/stylistic concerns.

- **B+**: A submission that has several minor problems that add up.

- **B**: A submission that has problems serious enough to fall short of the requirements for the assignment.

- **C**: A submission that has extremely serious problems, but nonetheless shows some effort and/or understanding.

- **D**: A submission that shows little effort and does not represent passing work.
Last Time and Lab 3 Prelab Video

• Reviewed iterating over sequences with for loops
  • Used accumulation variables to collect "items" from sequences, e.g., vowel sequences, counters, etc

• Introduced new sequence: lists
  • Learned how to index, slice, iterate over lists just like we did with strings
  • Example: wordStartEnd

• Learned about doctests in Python and importing modules (prelab video):
  • Another way to test functions: embed interactive python test cases into docstrings of our functions
  • __all__ special variable
Today’s Plan

- Gain more experience with **iterating** over lists
- Learn how to **accumulate** in and return a new list containing items with interesting properties from our original list
- Introduce **nested for loops**
- Discuss **range** data types and ways to iterate over numerical sequences
- Summarize important string, list, and sequence operations
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 """
    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
Exercise: palindromes

- Write a function that iterates over a given list of strings `sList`, and returns a (new) list containing all the strings in `sList` that are palindromes (i.e., read the same backward and forward).

```python
def palindromes(sList):
    '''Takes a list of words and returns a new list of words comprised of words from the original list that are palindromes'''
    pass

>>> palindromes(['Anna', 'banana', 'kayak', 'rigor', 'tacit', 'hope'])
['Anna', 'kayak']
>>> palindromes(['1313', '1110111', '0101'])
['1110111']
>>> wordStartEnd(['Level', 'Stick', 'Gag'])
['Level', 'Gag']
```
Exercise: palindromes

- **Step by step approach (organize your work):**
  - Go through every word in wordList
  - Check if word is same forward and backwards
  - 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)**
  - How do we test if word is same forward and backwards:
    - Can use slicing with optional step [:::-1]
    - Think about **corner cases**: what if string is empty? what about case?
Exercise: palindromes

• Write a function that iterates over a given list of strings `sList`, and returns a (new) list containing all the strings in `sList` that are palindromes (i.e., read the same backward and forward).

```python
def palindromes(sList):
    '''Takes a list of words and returns a new list of words comprised of words from the original list that are palindromes'''
    # initialize accumulation variable (of type list)
    result = []
    for word in sList:  # iterate over list
        wLower = word.lower()  #ignore case
        if wLower[:::-1] == wLower:  # [:::-1] returns wLower in reverse
            result += [word]  # concatenate to resulting list, notice []
    return result
```
Nested Loops

• A **for loop** body can contain one (or more!) additional **for loops**:  
  • Called **nesting for loops**

• 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
Nested Loops

• **Exercise**: What is printed by the nested loop below:

```python
# What does this print?

for letter in ['b', 'd', 'r', 's']:
    for suffix in ['ad', 'ib', 'ump']:
        print(letter + suffix)
```
```python
In [12]: # What does this print?

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
```
A New Type of Sequence: Range

• 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 New Type of Sequence: Range

• 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)
```

What are the values of i and j???
# what does this print?

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

# what does this print?

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

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

$  
$$  
$$$  
$$$$  

*  
**  
***  
****

# what does this print?

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

$  
*  
$$  
**  
**  
$$$  
***  
***  
***  
$$$$  
*****  
*****  
*****
Loops and Ranges to Print Patterns

- When loop variable is not needed in the body of the loop, we can use _ as the loop variable:

```python
for _ in range(10):
    print('Hello World!')
```

Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Summary:
List Operations (so far)
Modifying Lists

• Lists are **mutable** structures which means we can update them (delete things from them, add things to them, etc.)

• We have looked at list **concatenation** (using +) which **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 a list**, which adds items by modifying the existing list
  • Important point: Appending to a list modifies the existing list!
  • We can use the list method `myList.append(item)` that **modifies** the list `myList` by adding `item` to it at the end
  • Often more efficient to append rather than concatenate!
Appending to a List

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

```python
In [8]: numList = [1, 2, 3, 4, 5]

In [9]: numList + [6]
Out[9]: [1, 2, 3, 4, 5, 6]

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

In [12]: numList.append(6)

In [14]: numList # numList has been updated to include 6
Out[14]: [1, 2, 3, 4, 5, 6]
```
list() Function

- **list() function**, when given another sequence (range or string), returns a list of elements in the sequence
- Let’s review how it works with **strings** and **ranges**

```python
In [1]: spell = "Avada Kedavra!"

In [2]: list(spell) # can turn a string into a list of its characters
Out[2]: ['A', 'v', 'a', 'd', 'a', ' ', 'K', 'e', 'd', 'a', 'v', 'r', 'a', '!

In [16]: list(range(-10, 10))
Out[16]: [-10, -9, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [21]: list(range(3))
Out[21]: [0, 1, 2]
```
Summary:
String Operations and Methods
Review: Strings to Lists w/ `split()`

- `split()` is used to convert strings to lists
- The `split()` string method splits strings at “spaces” (the default separator) and returns a list of (sub)strings
- Can optionally specify other **delimiters** 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']
```
Review: List to Strings w/ `join()`

- `join()` is a string method that converts lists to strings.
- Given a list of strings, the `join()` string method, when applied to a string char, concatenates the strings together with the string char between them.

```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'
```
Remove whitespace w/ `strip()`

- The `strip()` string method strips away whitespace and 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'
```
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)
```

<table>
<thead>
<tr>
<th>Returned value</th>
</tr>
</thead>
<tbody>
<tr>
<td>['Williams', 'College']</td>
</tr>
<tr>
<td>'WILLIAMS COLLEGE'</td>
</tr>
<tr>
<td>'williams college'</td>
</tr>
<tr>
<td>'Willeslley College'</td>
</tr>
<tr>
<td>'Williams College'</td>
</tr>
<tr>
<td>'Spacey College'</td>
</tr>
<tr>
<td>'Williams College'</td>
</tr>
</tbody>
</table>

Remember: None of these operations change/affect the original string. They all return a new string!
Even More String Functions!

- **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).

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

- Many more: see `pydoc3 str`
Summarizing Mutability in Strings vs Lists

Strings are **immutable**

- Once you create a string, it cannot be changed!
- All functions 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
Summary: Sequence Operations (Strings, Lists, Ranges)

<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>

* Concatenation is not supported on range objects