Sequences and Loops

Motivation: How to count the number of vowels in a word?
- You’re given words like 'boston', 'wellesley', 'needham', 'lynn', etc.
- Tasks:
  - count the number of vowels in a word.
  - count the number of times a certain vowel appears in a word

```
def countAllVowels(word):
    # body here

def countVowel(word, vowel):
    # body here
```

Slides 8-3 to 8-12 explain what we need to know/learn to solve these problems.

Old friend: `isVowel`

```
def isVowel(char):
    c = char.lower()
    return (c == 'a' or c == 'e' or c == 'i' or c == 'o' or c == 'u')
```

```
def isVowel(char):
    c = char.lower()
    return c in 'aeiou'
```

To think: How will the function `isVowel` be useful for solving our “counting vowels” problem?

Indices: accessing characters in a string

```
In [1]: word = 'boston'
Out[1]: 'boston'
In [2]: word[0]
Out[2]: 'b'
In [3]: word[1]
Out[3]: 'o'
In [4]: word[2]
Out[4]: 's'
In [5]: word[3]
Out[5]: 't'
In [6]: word[4]
Out[6]: 't'
In [7]: word[5]
Out[7]: 'n'
```

Notice
- 0, 1, 2, etc. are the indices (plural of index).
- Indices start at 0.
- Indices go from 0 to `len(word)-1`.
- We read `word[0]` as word sub 0.
- [] is known as the indexing operator.

To think: How will indices be useful for solving our “counting vowels” problem?
A possible solution: which is correct?

```python
word = 'boston'
counter = 0
if isVowel(word[0]):
    counter += 1
if isVowel(word[1]):
    counter += 1
if isVowel(word[2]):
    counter += 1
if isVowel(word[3]):
    counter += 1
if isVowel(word[4]):
    counter += 1
if isVowel(word[5]):
    counter += 1
print counter
```

Does our solution work for all words?

- Do you think the right solution from 8-5 will work for all words: 'wellesley', 'needham', 'lynn', etc.?
- What happens if we use an index that's greater than or equal to the length of the word?

```python
In [1]: word = 'lynn'
In [2]: word[4]
IndexError: string index out of range
```

How to generate the correct indices of the string?

Creating a list of indices with `range`

When the `range` function is given two integer arguments, it returns a list of all integers starting at the first and up to, but not including, the second.

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]: range(0, 10)</td>
<td>[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]</td>
</tr>
<tr>
<td>[2]: range(3, 7)</td>
<td>[3, 4, 5, 6]</td>
</tr>
<tr>
<td>[3]: range(3, 2)</td>
<td>[]</td>
</tr>
<tr>
<td>[4]: range(3, 3)</td>
<td>[]</td>
</tr>
<tr>
<td>[5]: range(3)</td>
<td># missing first argument defaults to 0</td>
</tr>
</tbody>
</table>

Properties of the `range` function

An optional third argument to `range` controls the step size between elements (which defaults to 1).

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]: range(1, 10, 2)</td>
<td>[1, 3, 5, 7, 9]</td>
</tr>
<tr>
<td>[2]: range(3, 70, 10)</td>
<td>[3, 13, 23, 33, 43, 53, 63]</td>
</tr>
<tr>
<td>[3]: range(9, 0, -1)</td>
<td>[9, 8, 7, 6, 5, 4, 3, 2, 1]</td>
</tr>
<tr>
<td>[4]: range(9, 0, -2)</td>
<td>[9, 7, 5, 3, 1]</td>
</tr>
<tr>
<td>[5]: range(63, 0, -10)</td>
<td>[63, 53, 43, 33, 23, 13, 3]</td>
</tr>
</tbody>
</table>

To notice:
- With the help of the third argument of `range`, we can create different lists of integers.
- Step can be positive or negative.
- When step is negative, start value has to be larger than end value.
Introducing a new value type: 
lists

range() returns values of type list

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

list() converts a string into lists of characters

In [2]: list("Wendy Wellesley")
Out[2]: ['W', 'e', 'n', 'd', 'y', ' ', 'W', 'e', 'l', 'l', 'e', 's', 'l', 'e', 'y']

We can also specify a list directly as a comma separated list of values

In [3]: phrase = ["a", "lovely", "autumn", "day"]
In [4]: phrase
Out[4]: ['a', 'lovely', 'autumn', 'day']

Sequences/Loops

Loops to the rescue!

word = 'boston'
counter = 0
if isVowel(word[0]):
    counter += 1
if isVowel(word[1]):
    counter += 1
if isVowel(word[2]):
    counter += 1
if isVowel(word[3]):
    counter += 1
if isVowel(word[4]):
    counter += 1
if isVowel(word[5]):
    counter += 1
print counter

Back to our vowel counting problem

word = 'boston'
counter = 0
if isVowel(word[0]):
    counter += 1
if isVowel(word[1]):
    counter += 1
if isVowel(word[2]):
    counter += 1
if isVowel(word[3]):
    counter += 1
if isVowel(word[4]):
    counter += 1
if isVowel(word[5]):
    counter += 1
print counter

range() solves our indexing problem, by generating the correct list of indices.

Loops again in a few lectures. Lists are an example of “mutable” objects in Python, different from the “immutable” strings.

Iterating Over Sequences 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, and in Python is accomplished with a for loop.

for var in sequence:
    # Body of the loop
    statements using var

Generic form of a for loop

Concepts in this slide:
The combination of range and len to generate indices for a sequence.

Concepts in this slide:
New execution kind: iteration done through loops.

Concepts in this slide:
An example of a for loop with a conditional.
**for loop model example**

```python
word = 'boston'
counter = 0

for i in range(len(word)):
    if isVowel(word[i]):
        counter += 1

print(counter)
```

**Concepts in this slide:**
Modeling how code in a loop is executed.

---

**for loops without range**

- The `range` function provides a list of indices

**Mode 1**

```python
phrase = ['an', 'autumn', 'day']
for i in range(len(phrase)):
    print(phrase[i] + '!')
```

```
an!
adown!
day!
```

- We can also loop directly over any list. The code below produces the same output.

**Mode 2**

```python
phrase = ['an', 'autumn', 'day']
for word in phrase:
    print(word + '!
```

**When is it better to use range instead of directly looping?**

- Let's modify the previous example to print both the index and the item for each item in the list.

  ```python
  for i in range(len(phrase)):
      print(i, phrase[i], '!
  ```

  0 an!
  1 autumn!
  2 day!

- Notice this would NOT be possible if we directly looped over the list.

---

**Strings and lists are both sequences**

```python
In [1]: word = 'boston'
In [2]: word[2]
Out[2]: 's'
In [3]: len(word)
Out[3]: 6
In [4]: word + ' globe'
Out[4]: 'boston globe'
```

```
In [5]: 'o' in word
Out[5]: True
```

```python
In [1]: digits = range(5,10)
In [2]: digits[2]
Out[2]: 7
In [3]: len(digits)
Out[3]: 5
In [4]: digits + range(5)
Out[4]: [5, 6, 7, 8, 9, 0, 1, 2, 3, 4]
In [5]: 10 in digits
Out[5]: False
```

A sequence is an “abstract” type, which serves as template for “concrete” types such as string or list.
Looping over a string

Can we avoid `range` in this code as we did in 8-15? It turns out yes, in the same way.

```python
word = 'boston'
counter = 0
for i in range(len(word)):
    if isVowel(word[i]):
        counter += 1
print counter
```

Getting Started  8-17

Operations in Sequences

<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>seq1 + seq2</td>
<td>The concatenation of seq1 and seq2</td>
</tr>
<tr>
<td>seq<em>n, n</em>seq</td>
<td>n copies of seq concatenated</td>
</tr>
<tr>
<td>seq[i]</td>
<td>i'th item of seq, where origin is 0</td>
</tr>
<tr>
<td>seq[i:j]</td>
<td>slice of seq from i to j</td>
</tr>
<tr>
<td>seq[i:j:k]</td>
<td>slice of seq from i to j with step k</td>
</tr>
<tr>
<td>len(seq)</td>
<td>length of seq</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>

Sequences/Loops  8-19

The Slicing operator `[ : ]`

```python
In [1]: word = 'boston'
In [2]: word[2]
Out[2]: 's'
```

```python
In [3]: word[2:4]
Out[3]: 'st'
```

```python
In [4]: word[:3]
Out[4]: 'bos'
```

```python
In [5]: word[3:10]
Out[5]: 'ton'
```

```python
In [6]: word[10:]
Out[6]: 'ton'
```

```python
In [7]: word[0:6:2]
Out[7]: 'bso'
```

```python
In [8]: word[::1]
Out[8]: 'boston'
```

```python
In [1]: digits = range(5,10)
In [2]: digits[2]
Out[2]: 7
```

```python
In [3]: digits[2:4]
Out[3]: [7, 8]
```

```python
In [4]: digits[:3]
Out[4]: [5, 6, 7]
```

```python
In [5]: digits[3:10]
Out[5]: [8, 9]
```

```python
In [6]: digits[3:]
Out[6]: [8, 9]
```

```python
In [7]: digits[0:5:2]
Out[7]: [5, 7, 9]
```

```python
In [8]: digits[::1]
Out[8]: [9, 8, 7, 6, 5]
```

Sequences/Loops  8-20
How do indices work?

Indices in Python are both positive and negative. Everything outside these values will cause an IndexError.

In [7]: word[::-1]
Out[7]: 'notsob'

This means: start at 0 until the end of sequence with step -1. And it works because of the negative indices.

Summary

1. Strings and lists are examples of sequences, ordered items that are stored together. Because they are ordered, we can use indices to access each of them individually and sequentially.
2. The indexing operator [] takes index values from 0 to len(sequence)-1. However, negative indices are possible too in Python, see 8-21.
3. If we can access each element of a sequence (string or list) one by one, then we can perform particular operations with them.
4. To access each element we need a loop, an execution mechanism that repeats a set of statements until a stopping condition is fulfilled.
5. When we loop over a sequence, the stopping mechanism is the arrival at the last element and not having where to go further.
6. We use the built-in function range to generate indices for sequences.
7. Python makes it easy for us to iterate over a sequence’ elements even without the use of indices. In fact we can write: for item in sequence: and that will access each item of the sequence.
8. In addition to accessing one element at a time with [], one can use [:] (slicing) to get a substring or a sublist.