Sequences and Loops

Motivation: How to count the number of vowels in a word?
- You’re given words like 'Boston', 'Wellesley', 'abracadabra', 'bureaucracies', etc.

- Tasks:
  - count the number of vowels in a word.
  - count the number of times a certain character appears in a word

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

def countChar(char, word):
    # body here
```

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

Old friend: `isVowel`

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

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

Indices: accessing characters in a string

```python
In [1]: word = '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 [5]: word[4]
Out[5]: 'o'
In [5]: word[5]
Out[5]: '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)
```

Sequence/Loops

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.

```
In [1]: range(0, 10)
Out[1]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
In [2]: range(3, 7)
Out[2]: [3, 4, 5, 6]
In [3]: range(3, 2)
Out[3]: []
In [4]: range(3, 3)
Out[4]: []
In [5]: range(3)   # missing first argument defaults to 0
Out[5]: [0, 1, 2]
```

Properties of the `range` function

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

```
In [1]: range(1, 10, 2)
Out[1]: [1, 3, 5, 7, 9]
In [2]: range(3, 70, 10)
Out[2]: [3, 13, 23, 33, 43, 53, 63]
In [3]: range(9, 0, -1)
Out[3]: [9, 8, 7, 6, 5, 4, 3, 2, 1]
In [4]: range(9, 0, -2)
Out[4]: [9, 7, 5, 3, 1]
In [5]: range(63, 0, -10)
Out[5]: [63, 53, 43, 33, 23, 13, 3]
```

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

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

We'll return to lists again in a few lectures. Lists are an example of "mutable" objects in Python, different from the "immutable" strings.

```python
list() converts a string into a list of characters
```

```python
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

```python
In [3]: phrase = ['A', 'lovely', 'autumn', 'day']
In [4]: phrase
Out[4]: ['A', 'lovely', 'autumn', 'day']
```

Back to our vowel counting problem

```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
```

Try it out in the notebook to check that we get the same result.

Loops to the rescue!

```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
```

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

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.

Try it out in the notebook to check that we get the same result.
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:
Looping over any list. The code below produces the same output.

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

Sequences/Loops  8-13

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

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
```

```python
word = 'Boston'
counter = 0
for char in word:
    if isVowel(char):
        counter += 1
print counter
```

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

Sequences/Loops 8-19

A simpler version of `isVowel` using `in`

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

# Simpler version
def isVowel(char):
    return char.lower() in 'aeiou'
```

Sequences/Loops 8-20
The Slicing operator [:]

In [1]: word = 'Boston'
In [2]: word[2]
Out[2]: 's'
In [3]: word[2:5]
Out[3]: 'sto'
In [4]: word[:3]
Out[4]: 'bos'
In [5]: word[3:10]
Out[5]: 'ton'
In [6]: word[3::]
Out[6]: 'ton'
In [7]: word[0:6:2]
Out[7]: 'bso'
In [8]: word[::-1]
Out[8]: 'notsob'

How do indices work?

In [1]: digits = range(7, 12)
In [2]: digits[2]
Out[2]: 9
In [3]: digits[2:5]
Out[3]: [9, 10, 11]
In [4]: digits[:3]
Out[4]: [7, 8, 9]
In [5]: digits[3:10]
Out[5]: [10, 11]
In [6]: digits[3::]
Out[6]: [10, 11]
In [7]: digits[0:6:2]
Out[7]: [7, 9, 11]
In [8]: digits[::-1]
Out[8]: [11, 10, 9, 8, 7]

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’s 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.