Sequences and Loops

Motivation: How to create these pattern?

- How to achieve repetition?
- How to store color values so as to repeat them in the same order?

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

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

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

Reminder from last lecture: `isVowel`

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

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

Slides 6-4 to 6-12 explain what we need to know/learn to solve these problems.
Indices: accessing characters in a string

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 [6]: word[4]
Out[6]: 'o'
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 \( \text{len}(\text{word})-1 \).

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
print counter
```

Does our solution work for all words?

- Do you think the right solution from 6-6 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
```

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.

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

A list is a new Python type. It stores a sequence of any values, delimited by square brackets, and separated by commas. More on slide 6-10.
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]
```

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

```
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']
```

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
```

```
In [1]: word = 'boston'
In [2]: range(len(word))
Out[2]: [0, 1, 2, 3, 4, 5]
In [3]: word = 'wellesley'
In [4]: range(len(word))
Out[4]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
In [5]: word = 'lynn'
In [6]: range(len(word))
Out[6]: [0, 1, 2, 3]
```

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
```

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

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

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

### 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!
autumn!
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 + '!
```

---

**Define countAllVowels and countVowel**

- In the notebook, write definitions for **countAllVowels** and **countVowel** using for-loops
  
  - **countAllVowels**("america") should return 4
  - **countVowel**("america", "a") should return 2
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(str(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.

For loop model example #2

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

Sequences/Loops 6-17

Fill in the BLANK

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

Sequences/Loops 6-18

Getting Started 6-19

Sequences/Loops 6-20
## Operations in Sequences

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>x in seq</td>
<td>True if an item of seq is equal to x</td>
</tr>
<tr>
<td>x not in seq</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: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>

## The Slicing operation

```
In [1]: word = 'boston'
In [2]: word[2]
Out[2]: 's'
In [3]: word[2:4]
Out[3]: 'st'
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]: 'notsob'
Out[8]: 'notsob'
```

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

## How do indices work?

```
word = "boston"
digits = range(5, 10)
```

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.

## IMPORTANT: Nested loops

```
verse = "Two roads diverged in a yellow wood"
for word in verse.split():
    counter = 0
    for letter in word:
        if isVowel(letter):
            counter += 1
    print 'Vowels in', word, '->', counter
```

Vowels in Two -> 1
Vowels in roads -> 2
Vowels in diverged -> 3
Vowels in in -> 1
Vowels in a -> 1
Vowels in yellow -> 2
Vowels in wood -> 2
Let’s fill in the missing connector lines in this diagram for the problem in the previous slide.

Graphics Examples with **for** Loops

We can use **for** loops in conjunction with the **range** function and the **cslgraphics** module to create complex pictures with repeated subpatterns that are transformed by scaling, rotation, etc.

```python
from cslgraphics import *
paper = Canvas(800, 700, 'skyBlue', 'Rotation Designs')
for i in range(12):
    petal = Ellipse(150, 30)
    petal.setFillColor('yellow')
    # 75 is one half 150; try -95 instead
    petal.adjustReference(-75, 0)
    petal.rotate(i*30)  # 30 is 360/12
    petal.moveTo(200, 200)
paper.add(petal)
```

Each of these pictures is created by using a loop to create multiple copies of a simple shape (ellipse, circle, square) that differ in their rotation, size, and/or color.
Abstracting over our flower with `makeFlower`

Define a function `makeFlower` that takes as arguments (1) the number of petals (2) the color of each petal (3) the width of each petal and (4) the height of each petal and returns a `Layer` object with an appropriately constructed flower object.

```
makeFlower(12, 'yellow', 30, 150)
makeFlower(10, 'brown', 90, 150)
makeFlower(30, 'magenta', 20, 150)
```

A simple nautilus shell

```
for i in range(50):
    ring = Circle(100)
    ring.setFillColor('white')
    # adjust by radius size
    ring.adjustReference(-100, 0)
    # 10 is just a small amount
    ring.rotate(i*10)
    # 0.95 just makes it smaller by a tad
    ring.scale(0.95**i)
    ring.moveTo(500, 150)
    paper.add(ring)
```

Parameterize it: `makeNautilus`

```
shell = makeNautilus(50, 100, 10, .95, 'pink')
shell.moveTo(200,200)
paper.add(shell)
```

```
def makeNautilus(num, size, angle, shrink, color):
    nautilus = Layer()
    for i in range(num):
        ring = Circle(size)
        ring.setFillColor(color)
        ring.adjustReference(-size, 0)
        ring.rotate(i*angle)
        ring.scale(shrink**i)
        nautilus.add(ring)
    return nautilus
```

Make it fancy: `makeColorfulNautilus`

```
colorfulShell = makeColorfulNautilus(50,100,
10, .95, ['pink','blue','green','magenta'])
colorfulShell.moveTo(400,400)
paper.add(colorfulShell)
```

```
def makeColorfulNautilus(num, size, angle, shrink, colorList):
    nautilus = Layer()
    for i in range(num):
        ring = Circle(size)
        ring.setFillColor(colorList[i % len(colorList)])
        ring.adjustReference(-size, 0)
        ring.rotate(i*angle)
        ring.scale(shrink**i)
        nautilus.add(ring)
    return nautilus
```

The % operator makes sure that despite the value of num (and as a result, of i), the indices always are only between 0 and len of colorList.
Rotated squares

The class `Color` is defined in cs1graphics. The function `randomColor` will generate new colors, thus, when running this code you’ll see a differently colored graphics.

```python
for i in range(16):
    s = Square(200, Point(200, 500))
    s.rotate(6*i)
    s.scale(0.9**i)
    s.setFillColor(Color.randomColor())
    paper.add(s)
```

Rose-colored squares

```python
# Helper function
def makeColor(redFraction, greenFraction, blueFraction):
    return Color((255.0*redFraction, 255.0*greenFraction, 255.0*blueFraction))

for i in range(25):
    s = Square(10 + 10*i, Point(600, 500))
    s.rotate(15*i)
    s.setDepth(i)
    s.setFillColor(makeColor(i/24.0, 0.5, 0.5))
    paper.add(s)
```

If we don’t use `setDepth`, the bigger square (the last one) will be displayed on top. Comment out that line to see for yourself.