Loop patterns

- Text, Section 5.2

Loop patterns

- Loops often operate on **iterative** objects
  - Strings, lists, sets, dictionaries, even files

- Also used with non-iterative data, such as numbers

For loop patterns

1. **Iteration loops** for use with strings of lists
2. **Counter loops** sometimes used with loops/strings, other times not
3. **Accumulators** are often used with iteration or counter loops to build the answer to a function

For loop patterns

4. **Nested loops** allow iteration through 2 container objects or ranges at once
   - A loop that occurs in the body of another loop is **"nested"**

```
# outer loop
for … :
    # inner (nested loop)
    for … :
        <body>
```

For loop patterns

Iteration loops

for <item> in <items>:
    <do-stuff>

<items> is a an **iterative** object such as a list or a string (or other Python container)

Each time through the loop , <item> is assigned to be one of the items in the container, until all of the items have been used

Iteration loops are typically used with strings or lists, when position within the “in” variable does **not** matter. Example using a list:

```
# print words that start with ‘q’
def qWords(words):
    for word in words:
        if word[0] == ‘q’:
            print(word)
```

Iteration loops are typically used with strings or lists, when position within the “in” variable does **not** matter. Example using a string:

```
# print the vowels in a word
def vowels(word):
        for ch in word:
                if ch in 'aeiou':
                        print(ch, end='')
```

Another list example:

```
# see if x is in a list
def contains(x, items):
        for item in items:
                if x == item:
                        return True
        return  False
```

Example using a file

```
def qLines(file):
    f = open(file, 'r')
    for line in f:
        if 'q' in line:
            print(line)
```

Counter loops

for <variable> in range(…):
    <do-stuff>

<variable> takes on an integer value each time through the loop

for i in range(n):
    …

i = 0,1,2,…n-1

Counter loops are often used with strings or lists, when position need to explicitly be referenced

```
def isReversed(list1, list2):
    for i in range(len(list1)):
        if list[i] != list2[-(i+1)]:
            return False
    return True
```

Example using a string

```
def isPalindrome(word):
    for i in range(len(word)):
        if word[i] != word[-(i+1)]:
            return False
    return True
```

Counter loops are also used with other types of data

```
def inAscendingOrder(numbers):
    for i in range(len(numbers)-1):
        if numbers[i] > numbers[i+1]:
            return False
    return True
```

Accumulator loop pattern

Often, we need to initialize a variable **(accumulator)** prior to the beginning of a loop

Body of loop then modifies the accumulator with each iteration

Accumulator loop pattern

```
def qList(words):
    answer = [ ]    # initialize accumulator
    for word in words:
        if word[0] == 'q':
            answer.append(word)
    return answer
```

Accumulator loop pattern

```
def sumList(numbers):
    answer = 0      # initialize accumulator
    for x in numbers:
        answer += x
    return answer
```

Accumulator initialization

Very important to think about how an accumulator should be initialized

```
def factorial(n):
    answer = 1      # initialize accumulator
    for i in range(2,n+1):
        answer *= i     return answer
```

Within loop vs after loop

Important to think about when an answer should be returned

```
# is n a power of 2?
def powerOf2(n):
    if n in [0,1]:
        return True
    power = 1
    while power < n:
        power *= 2
        if power == n:
            return True
        else:
                return False ← wrong place
```

Within loop vs after loop

Important to think about when an answer should be returned

```
# is n a power of 2?
def powerOf2(n):
    if n in [0,1]:
        return True
    power = 1
    while power < n:
        power *= 2
        if power == n:
            return True
    return False ← right place:  after loop
```

Nested loops

When the body of one loop contains another loop, we say that they are **nested**

```
for <whatever>:
    for <whatever>:
        <do-stuff>
```

Nested loops

Allows iteration between 2 container objects, using all possible pairs of items

```
# print all pairs of items in x and y
def printcombos():
    for x in [1, 2, 3]:
        for y in ['a', 'b']:
            print('{} {}'.format(x,y))

>>> printxy()
1 a
1 b
2 a
2 b
3 a
3 b
```

Nested loops

Each loop in a nested loop construct may be an iteration or a counter loop

```
# print all pairs of numbers x and y, each between 1 and n
def printxy(n):
    for x in range(1,n+1):
        for y in range(1,n+1):
            print('{} {}'.format(x,y))

>>> printxy(2)
1 1
1 2
2 1
2 2
```

Nested loops

We might mix iteration and counter loops

There may or may not be accumulators

```
# this could be done much more easily
def toLowerCase(word):
    answer = str()          # accumulator variable for the outer loop
    for letter in word:
        found = False     # accumulator variable of sorts for the inner loop
        for i in range(26):
            if letter == 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'[i]:
                answer += 'abcdefghijklmnopqrstuvwxyz'[i]
                found = True
                break
        if not found:
            answer += letter
    return answer
```

Controlling execution of a loop

break and **continue** are control statements that modify a loop's behavior

Break stops a loop

Continue stop execution of a particular iteration of a loop

Controlling execution of a loop

```
# return the first even number in a list
def firstEven(numbers):
    answer = 0
    for number in numbers:
        if number%2 == 0:
            answer = number
            break
    return answer
```

Can also be done with a return statement within the loop

Controlling execution of a loop

```
# this can be done in other ways
def returnOdds(numbers):
    answer = [ ]
    for number in numbers:
        if number%2 == 0:
            continue
        answer.append(number)
    return answer
```