Chapter 5

- Loop Patterns
- Two-Dimensional Lists
Within loop vs after loop

Exam problem: check if 1\textsuperscript{st} list is the reverse of the 2\textsuperscript{nd}

```python
def isReverse(list1, list2):
    for i in range(len(list1)):
        if list1[i] != list2[-(i+1)]:
            return false
    else:
        return true
```

← wrong place

← wrong place
Within loop vs after loop

Exam problem: check if 1st list is the reverse of the 2nd

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

← right place (after loop)
For loop patterns

1. **Iteration loops** for use with strings of lists
2. **Counter loops** sometimes used with loops/strings, other times not
For loop patterns

Iteration loops

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

<iitems> is a list or a string

<iitem> is assigned to be an item in the list, or a character of the string, each time through the loop
For loop patterns

**Iteration loops** are typically used with strings or lists, when position does **not** matter

```python
def qWords(words):
    answer = []
    for word in words:
        if word[0] == 'q':
            answer.append(word)
```
For loop patterns

Counter loops

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

<variable> takes on an integer value each time through the loop (0,1,2,...)
For loop patterns

**Counter loops** for use with strings of lists, position does matter

def isReversed(list1, list2):
    for i in range(len(list1)):
        if list[i] != list2[-(i+1)]:
            return false
    return true
For loop patterns

**Counter loops** for use with strings of lists, position does matter

```python
def isPalindrome(word1, word2):
    if len(word1) != len(word2):
        return False
    for i in range(len(word1)):
        if word[i] != word[-i]:
            return False
    return True
```
For loop patterns

**Counter loops** are also used to execute a loop body several times, irregardless of the use of strings/lists

```python
def factorial(n):
    answer = 1
    for i in range(2,n+1):
        answer *= i
    return answer
```
**Exercise**

Write function `evens()` that:
- takes as input a list of numbers
- returns the even numbers in the list

```python
>>> evens([3, 6, 9, 12, 15])
[6, 12]
```

Which loop pattern: iteration or counter?
Exercise

Write function `arithmetic()` that:

* takes as input a list of numbers
* returns True if the numbers in the list form an arithmetic sequence, False otherwise

```python
>>> arithmeticSequence([3, 6, 9, 12, 15])
True
>>> arithmeticSequence([3, 6, 9, 11, 14])
False
>>> arithmeticSequence([3])
True
```

Which loop pattern: iteration or counter?
Accumulator loop pattern

Accumulating something in every loop iteration

For example: the sum of numbers in a list

```
>>> lst = [3, 2, 7, 1, 9]
>>> res = 0
>>> for num in lst:
    res += num
>>> res
22
```
Accumulator loop pattern

Accumulating something in every loop iteration

What if we wanted to obtain the product instead? What should res be initialized to?

```python
>>> lst = [3, 2, 7, 1, 9]
>>> res = 1
>>> for num in lst:
...    res *= num

res = 1
res *= num (= 3)
res *= num (= 6)
res *= num (= 42)
res *= num (= 42)
res *= num (= 378)
```
Exercise

Write function acronym() that:

• takes a phrase (i.e., a string) as input
• returns the acronym for the phrase

```python
>>> acronym('Random access memory')
'RAM'
```
Exercise

Write function `sumOfSquares(n)` that:
- takes an integer `n` as input
- returns $1^2 + 2^2 + 3^2 + \ldots + n^2$

```python
>>> sumOfSquares(4)
30
```
Exercise

Write function `polynomial(coefficients, n)` which computes the value for a polynomial \( p(x) \) for \( x = n \) and returns it

Example: \([2, 3, 1]\) represents \( 2n^{2} + 3n + 1 \)

```python
>>> polynomial([2, 3, 1], 2)
15
```

Counter or iterator loop?
Accumulator or no?
Nested loop pattern

Nesting a loop inside another

```
def nested(n):
    for j in range(n):
        for i in range(n):
            print(i, end=' ')
        print()
```

```
def nested2(n):
    for j in range(n):
        for i in range(j+1):
            print(i, end=' ')
        print()
```

```
>>> n = 5
>>> nested(n)
0 1 2 3 4
0 1 2 3 4
0 1 2 3 4
0 1 2 3 4
0 1 2 3 4
```

When $j = 0$ inner for loop should print 0
When $j = 1$ inner for loop should print 0 1
When $j = 2$ inner for loop should print 0 1 2
When $j = 3$ inner for loop should print 0 1 2 3
When $j = 4$ inner for loop should print 0 1 2 3 4

```
>>> n = 5
>>> nested2(n)
0
0 1
0 1 2
0 1 2 3
0 1 2 3 4
```

When $j = 0$ inner for loop should print 0
When $j = 1$ inner for loop should print 0 1
When $j = 2$ inner for loop should print 0 1 2
When $j = 3$ inner for loop should print 0 1 2 3
When $j = 4$ inner for loop should print 0 1 2 3 4
Exercise: sorting

def mySort(numbers):
    for i in range(numbers):
        # minIndex = the index of the smallest of the remaining numbers
        # will require a nested loop
        # swap numbers[i] and numbers[minIndex]
        pass
    return numbers
Exercise: sorting

Algorithm example

mySort([7, 6, 8, 5, 9, 3])

[7, 6, 8, 5, 9, 3]
[3, 6, 8, 5, 9, 7]
[3, 5, 8, 6, 9, 7]
[3, 5, 6, 8, 9, 7]
[3, 5, 6, 7, 9, 8]
[3, 5, 6, 7, 8, 9]
Two-dimensional lists

The list [3, 5, 7, 9] can be viewed as a 1-D table

\[ [3, 5, 7, 9] = \begin{array}{cccc}
3 & 5 & 7 & 9 \\
\end{array} \]

How to represent a 2-D table?

\[
\begin{array}{cccc}
[ [3,5,7,9] ] &=& 0 & 3 & 5 & 7 & 9 \\
[ [0,2,1,6] ] &=& 1 & 0 & 2 & 1 & 6 \\
[ [3,8,3,1] ] &=& 2 & 3 & 8 & 3 & 1 \\
\end{array}
\]

A 2-D table is just a list of rows (i.e., 1-D tables)
Nested loop pattern and 2-D lists

A nested loop is often needed to access all objects in a 2-D list

```python
def print2D(t):
    'prints values in 2D list t as a 2D table'
    for row in t:
        for item in row:
            print(item, end=' ')
    print()

>>> table = [[3, 5, 7, 9],
            [0, 2, 1, 6],
            [3, 8, 3, 1]]

>>> print2D(table)
3 5 7 9
0 2 1 6
3 8 3 1

>>> incr2D(t)

>>> print2D(t)
4 6 8 10
1 3 2 7
4 9 4 2
```

(Using the iteration loop pattern)

```python
def incr2D(t):
    'increments each number in 2D list t'
    # for every row index i
    # for every column index j
    for i in range(len(t)):
        for j in range(len(t[0])):
            t[i][j] += 1
```

(Using the counter loop pattern)
Exercise

Write a function `readcsv` that reads a .csv file (comma separate values). Excel files can be converted to .csv notation.

**Parameter:** .csv file name

**Return value:** A 2-dimensional list containing the values in the .csv file

```
.csv:
10, 8, 9, 8
7, 8, 6, 10
9, 9, 7, 7
```

Return value:
```
[[10, 8, 9, 8], [7, 8, 6, 10], [9, 9, 7, 7]]
```
Exercise

Write a function `averages` that is passed a table (from last exercise) and a list of weightings. It returns a list of weighted sums.

Table
[[10,8,9,8], [7,8,6,10], [9,9,7,7]]

Weightings
[.3, .3, .2, .2]

Return value
[8.8, 7.7, 8.2]

8.8 = 10*.3 + 8*.3 + 9*.2 + 8*.2
while loop pattern

Syntax:

<setup>
while <condition>:
    <do stuff>
<after stuff>

Execution:

1. Do setup
2. Evaluate condition
3. If it’s True, then do stuff; otherwise do after stuff
4. Repeat steps 2-3
while loop example

def factorial(n):
    answer = 1                  # a
    i = 2                              # b
    while i <= n:                # c
        answer *= i            # d
        i += 1               # e
    return answer            # f

Execution:
1. Execute statements a-b
2. Evaluate i <= n
3. If it’s True, then execute statements d-e; otherwise
   Execute statement f
4. Repeat steps 2-3
for vs. while

• for loops are generally easier and more intuitive.

• If you can’t apply one of the for loop patterns, try a while loop.
  • Example: range() doesn’t apply
Exercise

Write a function `readTillDone` that reads words from the console until the user types “done”. The function returns a list of the words.

```python
>>> readTillDone()
Type words. End with “done”
computer
science
241
section
405
Done
```
Exercise

e is the base such that \log(x) = x and is approx 2.718281828

E can be approximated as \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \ldots

def e(decimals):
    precision = 10 ** -(decimals + 1)
    e = 0
    # ...
    return round(e, decimals)
An infinite loop provides a continuous service

def hello2():
    """a greeting service; it repeatedly requests the name of the user and then greets the user""

    while True:
        name = input('What is your name? ')
        print('Hello {}'.format(name))

>>> hello2()
What is your name? Sam
Hello Sam
What is your name? Tim
Hello Tim
What is your name? Alex
Hello Alex
What is your name?
Loop-and-a-half pattern

Cutting the last loop iteration “in half”

Example: a function that creates a list of cities entered by the user and returns it

The empty string is a “flag” that indicates the end of the input

```python
def cities():
    lst = []
    city = input('Enter city: ')
    while city != '':
        lst.append(city)
        city = input('Enter city: ')
    return lst

>>> cities()
Enter city: Lisbon
Enter city: San Francisco
Enter city: Hong Kong
Enter city:
['Lisbon', 'San Francisco', 'Hong Kong']
```

```python
def cities2():
    lst = []
    while True:
        city = input('Enter city: ')
        if city == '':
            return lst
        lst.append(city)

>>> cities2()
Enter city: Lisbon
Enter city: San Francisco
Enter city: Hong Kong
Enter city:
['Lisbon', 'San Francisco', 'Hong Kong']
```
The break statement

The break statement:

- is used inside the body of a loop
- when executed, it interrupts the current iteration of the loop
- execution continues with the statement that follows the loop body.

```python
def cities2():
    lst = []
    while True:
        city = input('Enter city: ')
        if city == "":
            return lst
        lst.append(city)
return lst
```
The continue statement:

- is used inside the body of a loop
- when executed, it interrupts the current iteration of the loop
- execution continues with beginning of the loop

```python
def qWords(words):
    lst = []
    for word in words:
        if word[0] != 'q' and word[0] != 'Q':
            continue
        lst.append(word)
    return lst
```

```python
>>> qWords(['high', 'quality', 'large', 'quantity'])
['quality', 'quantity']
```