```
"""
Toward higher-order functions

Dan Armendariz
danallan@cs.harvard.edu
"""

# create an arbitrary list of numbers
nums = [1, 5, 10, 8, -4, 432]

print "Initial:", nums

# now let's add 1 to each number
result = []
for num in nums:
    result.append(num + 1)

print "Add 1:  ", result

# perhaps we now want to multiply each by some value
result = []
for num in nums:
    result.append(num * 2)

print "Double: ", result
```

```
"""
Toward higher-order functions

Dan Armendariz
danallan@cs.harvard.edu
"""

# create an arbitrary list of numbers
nums = [1, 5, 10, 8, -4, 432]

print "Initial:", nums

def incrementor(num):
    """ Increment a number by 1 """
    return num + 1

# iterate over every item in the list, applying our function to it
result = []
for num in nums:
    result.append(incrementor(num))

print "Add 1:  ", result


def doubler(num):
    """ Double a number """
    return num * 2

# iterate over every item in the list, applying our function to it
result = []
for num in nums:
    result.append(doubler(num))

print "Double: ", result
```

```
"""
Toward higher-order functions

Dan Armendariz
danallan@cs.harvard.edu
"""

# create an arbitrary list of numbers
nums = [1, 5, 10, 8, -4, 432]

print "Initial:", nums

def apply_to_all(f, items):
    """ Apply a function f to every item in a list """
    result = []
    for item in items:
        new_item = f(item)
        result.append(new_item)
    return result

def incrementor(num):
    """ Increment a number by 1 """
    return num + 1

def doubler(num):
    """ Double a number """
    return num * 2

print "Add 1:  ", apply_to_all(incrementor, nums)
print "Double: ", apply_to_all(doubler, nums)
```

```
"""
Using Python's built-in Higher-Order Functions (HOFs): map

Dan Armendariz
danallan@cs.harvard.edu
"""

# create an arbitrary list of numbers
nums = [1, 5, 10, 8, -4, 432]

print "Initial:", nums

def incrementor(num):
    """ Increment a number by 1 """
    return num + 1

def doubler(num):
    """ Double a number """
    return num * 2

# the `map` function is a HOF that applies a function to an entire list
print "Add 1:  ", map(incrementor, nums)
print "Double: ", map(doubler, nums)
```

```
"""
Using Python's built-in Higher-Order Functions (HOFs): reduce

Dan Armendariz
danallan@cs.harvard.edu
"""

# create an arbitrary list of numbers
nums = [1, 5, 10, 8, -4, 432]

print "Initial:", nums

def add(x, y):
    """ Sum two numbers """
    return x + y

def multiply(x, y):
    return x * y

# the `reduce` function is a HOF that collapses a list to a single value
print "Sum of all values: ", reduce(add, nums, 0)
print "Multiply all values: ", reduce(multiply, nums, 1)

"""
Reduce operates in this way:
We want to sum all of the objects, so we provide our function `sum` to reduce()
It then applies sum on the first element and the initial value, and the
output of that function is then used as input to the next iteration.

Let's say we have a list [1, 5, 8]
The first operation would be:
sum(0, 1) -- "0" is our initial value. The output is 1

1 is used as one of the next inputs; the other is the next item in the list
sum(1, 5) -- outputs "6"

sum(6, 8) -- outputs our final value, 14!

Note: the order is not guaranteed!
"""
```

```
"""
Use MapReduce (Hadoop) style programming to count the most common letter
unigrams (single characters) and bigrams (character pairs) in a text.
See: http://en.wikipedia.org/wiki/Bigram

Hadoop assumes that the input data is a text file. This is convenient,
because text files can be easily broken into chunks at individual lines
and distributed to many computers. It then roughly operates in this fashion:

MAP STEP: runs a developer-provided function on one line at a time of the input
files. The Map step is therefore generally used to convert the data represented
in the line into one or more key/value pairs. Normally, we could think of
"key/value" pairs as a dictionary, but Hadoop lets us have duplicate keys.
So, in essence, if we're writing code for Hadoop (which is normally done in the
Java programming language, but we're using Python for familiarity), we want
to write a map function that will accept a line of text and output a dictionary.

SHUFFLE STEP: As key/value pairs come in from the map step, hadoop sorts them
and redirects all values that have the same key to the same computer to run
the Reduce step. The map step must finish completely before moving on to
the Reduce step.

REDUCE STEP: Another developer-provided function, the reducer is given a
key and a list of all of the values for that key by Hadoop. It then
performs some computation and outputs a *single* value for that key.

SORT STEP: Hadoop will generally sort the final output data.

It is not unusual to have to run several Map/Reduce steps on a big data set
before receiving the output you want. In our example below, we'll only need a
single Map and single Reduce step to complete our goal.

See more information on MapReduce applications, of which Hadoop is a part, here:
http://en.wikipedia.org/wiki/MapReduce

For a good exercise, consider how you could modify the code below to count
unigram WORDS and bigram WORDS. In other words, instead of counting individual
characters or character pairs, counting whole words and counting word pairs.

Dan Armendariz, Dan Bradley
danallan@cs.harvard.edu, dbradley@cs50.harvard.edu
"""

# itemgetter will assist in sorting
from operator import itemgetter

def unigram_mapper(line):
    """Given a line of text, remove extraneous characters and output a key/value
```

```
    pair for each character. The character will be the key and the value will
    be 1.

    For example, given the string "Hello, world" we will output:
    [("h",1), ("e",1), ("l",1), ("l",1), ("o",1), ("w",1), ("o",1), ("r",1),
     ("l",1), ("d",1)]
    """
     output = []

     # perform the step for every character in the string
     for char in line:
         # skip the character if it is not a letter
         if not char.isalpha():
             continue

         # create our keyvalue pair, converting the letter to lower case
         keyvalue_pair = (char.lower(), 1)
         output.append(keyvalue_pair)

     return output


def bigram_mapper(line):
    """Given a line of text, remove extraneous characters and output a key/value
    pair for each bigram. The bigram will be the key and the value will be 1.

    For example, given the string "Hello, world" we will output:
    [("he",1), ("el",1), ("ll",1), ("lo",1), ("wo",1), ("or",1), ("rl",1),
     ("ld", 1)]
    """
     output = []

     # iterate over each character in the line
     for i in range(len(line)-1):
         # fetch a bigram by splicing the line at i to grab 2 neighboring chars
         bigram = line[i:i+2]

         # skip the bigram if either character isn't a letter
         if not bigram.isalpha():
             continue

         # create the key/value pair, and ensure the bigram is lower case
         keyvalue_pair = (bigram.lower(), 1)
         output.append(keyvalue_pair)

     return output
```

```
 97. def count_reducer(key, values):
 98.     """Given a list of values for a key, we'll return the sum of all values."""
 99.     total = 0
100.     for value in values:
101.         total += value
102.     return (key, total)
103.
104.
105. def hadoop(file, mapper, reducer):
106.     """A higher-order function that accepts a mapper and a reducer and
107.    performs Hadoop/MapReduce-style processing on a given file.
108.
109.    Essentially, this code (in a very simple way) performs on a single computer
110.    what Hadoop and other MapReduce implementations do across a wide number
111.    of computers.
112.    """
113.     # a structure to store the final output
114.     data = []
115.
116.     # MAP STEP
117.     pairs = []
118.     with open(file) as f:
119.         # iterate over every line in the opened file
120.         for line in f:
121.             # ask our map step to process the line
122.             output = mapper(line)
123.
124.             # store the newly computed key/value pairs in our `pairs` structure
125.             pairs.extend(output)
126.
127.     # SHUFFLE STEP: aggregate all values for a given key
128.     collapsed = {}
129.     for pair in pairs:
130.         # split the pair into its key and value
131.         key, value = pair
132.
133.         # if the key doesn't exist yet in our dictionary, add it
134.         if collapsed.get(key, None) is None:
135.             collapsed[key] = []
136.
137.         # append the value to the list of values for a given key
138.         collapsed[key].append(value)
139.
140.
141.     # REDUCE STEP: provide all of the values for a key to the reducer
142.     for key, values in collapsed.iteritems():
143.         # the key is different; run the reduce step on our accumulated vals
144.         output = reducer(key, values)
```

```
145.         data.append(output)
146.
147.     # SORT STEP: sort the output by values in descending order
148.     data.sort(key=itemgetter(1), reverse=True)
149.
150.     # all done!
151.     return data
152.
153. # Using "Hadoop" and our reducer and unigram mapper, count the number of
154. # instances each character appears in Carroll's "Alice and Wonderland"
155. print "Unigram character counts:"
156. result = hadoop("aliceandwonderland.txt", unigram_mapper, count_reducer)
157. for count in result:
158.     print count
159.
160. print
161.
162. # On to bigrams. Our reducer is so simple we can use the same one for both!
163. print "Bigram character counts:"
164. result = hadoop("aliceandwonderland.txt", bigram_mapper, count_reducer)
165.
166. # there are many bigrams, so we'll only print the first 20
167. for count in result[0:10]:
168.     print count
```