Recursion and exceptions

11.1. Tuples and mutability

So far, you have seen two compound types: strings, which are made up of characters; and lists, which are made up of elements of any type. One of the differences we noted is that the elements of a list can be modified, but the characters in a string cannot. In other words, strings are **immutable** and lists are **mutable.**

A **tuple**, like a list, is a sequence of items of any type. Unlike lists, however, tuples are immutable. Syntactically, a tuple is a comma-separated sequence of values:

```
>>> tup = 2, 4, 6, 8, 10
```

Although it is not necessary, it is conventional to enclose tuples in parentheses:

```
>>> tup = (2, 4, 6, 8, 10)
```

To create a tuple with a single element, we have to include the final comma:

```
>>> tup = (5,)
>>> type(tup)
<type 'tuple'>
```

Without the comma, Python treats (5) as an integer in parentheses:

```
>>> tup = (5)
>>> type(tup)
<type 'int'>
```

Syntax issues aside, tuples support the same sequence operations as strings and lists. The index operator selects an element from a tuple.

```
>>> tup = ('a', 'b', 'c', 'd', 'e')
>>> tup[0]
'a'
```

And the slice operator selects a range of elements.

```
>>> tup[1:3]
```

```
('b', 'c')
```

But if we try to use item assignment to modify one of the elements of the tuple, we get an error:

```
>>> tup[0] = 'X'
TypeError: 'tuple' object does not support item assignment
```

Of course, even if we can’t modify the elements of a tuple, we can replace it with a different tuple:

```
>>> tup = ('X',) + tup[1:]
>>> tup
('X', 'b', 'c', 'd', 'e')
```

Alternatively, we could first convert it to a list, modify it, and convert it back into a tuple:

```
>>> tup = ('X', 'b', 'c', 'd', 'e')
>>> tup = list(tup)
>>> tup
['X', 'b', 'c', 'd', 'e']
>>> tup[0] = 'a'
>>> tup = tuple(tup)
>>> tup
('a', 'b', 'c', 'd', 'e')
```

11.2. Tuple assignment

Once in a while, it is useful to swap the values of two variables. With conventional assignment statements, we have to use a temporary variable. For example, to swap a andb:

```
temp = a
a = b
b = temp
```

If we have to do this often, this approach becomes cumbersome. Python provides a form of **tuple assignment** that solves this problem neatly:

```
a, b = b, a
```

The left side is a tuple of variables; the right side is a tuple of values. Each value is assigned to its respective variable. All the expressions on the right side are evaluated before any of the assignments. This feature makes tuple assignment quite versatile.

Naturally, the number of variables on the left and the number of values on the right have to be the same:

```
>>> a, b, c, d = 1, 2, 3
ValueError: need more than 3 values to unpack
```

11.3. Tuples as return values

Functions can return tuples as return values. For example, we could write a function that swaps two parameters:

```
def swap(x, y):
   return y, x
```

Then we can assign the return value to a tuple with two variables:

```
a, b = swap(a, b)
```

In this case, there is no great advantage in making swap a function. In fact, there is a danger in trying to encapsulate swap, which is the following tempting mistake:

```
def swap(x, y):      # incorrect version
    x, y = y, x
```

If we call this function like this:

```
swap(a, b)
```

then a and x are aliases for the same value. Changing x inside swap makes x refer to a different value, but it has no effect on a in __main__. Similarly, changing y has no effect on b.

This function runs without producing an error message, but it doesn't do what we intended. This is an example of a semantic error.

11.4. Pure functions and modifiers revisited

In *Pure functions and modifiers* we discussed *pure functions* and *modifiers* as related to lists. Since tuples are immutable we can not write modifiers on them.

Here is a modifier that inserts a new value into the middle of a list:

```
#
# seqtools.py
#

def insert_in_middle(val, lst):
    middle = len(lst)/2
    lst[middle:middle] = [val]
```

We can run it to see that it works:

```
>>> from seqtools import *
>>> my_list = ['a', 'b', 'd', 'e']
>>> insert_in_middle('c', my_list)
>>> my_list
['a', 'b', 'c', 'd', 'e']
```

If we try to use it with a tuple, however, we get an error:

```
>>> my_tuple = ('a', 'b', 'd', 'e')
>>> insert_in_middle('c', my_tuple)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "seqtools.py", line 7, in insert_in_middle
    lst[middle:middle] = [val]
TypeError: 'tuple' object does not support item assignment
>>>
```

The problem is that tuples are immutable, and don’t support slice assignment. A simple solution to this problem is to make insert_in_middle a pure function:

```
def insert_in_middle(val, tup):
    middle = len(tup)/2
    return tup[:middle] + (val,) + tup[middle:]
```

This version now works for tuples, but not for lists or strings. If we want a version that works for all sequence types, we need a way to encapsulate our value into the correct sequence type. A small helper function does the trick:

```
def encapsulate(val, seq):
    if type(seq) == type(""):
        return str(val)
    if type(seq) == type([]):
        return [val]
    return (val,)
```

Now we can write insert_in_middle to work with each of the built-in sequence types:

```
def insert_in_middle(val, seq):
    middle = len(seq)/2
    return seq[:middle] + encapsulate(val, seq) + seq[middle:]
```

The last two versions of insert_in_middle are pure functions. They don’t have any side effects. Adding encapsulate and the last version of insert_in_middle to theseqtools.py module, we can test it:

```
>>> from seqtools import *
>>> my_string = 'abde'
>>> my_list = ['a', 'b', 'd', 'e']
>>> my_tuple = ('a', 'b', 'd', 'e')
>>> insert_in_middle('c', my_string)
'abcde'
>>> insert_in_middle('c', my_list)
['a', 'b', 'c', 'd', 'e']
>>> insert_in_middle('c', my_tuple)
('a', 'b', 'c', 'd', 'e')
>>> my_string
'abde'
```

The values of my_string, my_list, and my_tuple are not changed. If we want to use insert_in_middle to change them, we have to assign the value returned by the function call back to the variable:

```
>>> my_string = insert_in_middle('c', my_string)
>>> my_string
'abcde'
```

11.5. Recursive data structures

All of the Python data types we have seen can be grouped inside lists and tuples in a variety of ways. Lists and tuples can also be nested, providing myriad possibilities for organizing data. The organization of data for the purpose of making it easier to use is called a **data structure.**

It’s election time and we are helping to compute the votes as they come in. Votes arriving from individual wards, precincts, municipalities, counties, and states are sometimes reported as a sum total of votes and sometimes as a list of subtotals of votes. After considering how best to store the tallies, we decide to use a *nested number list*, which we define as follows:

A *nested number list* is a list whose elements are either:

a. numbers
b. nested number lists

Notice that the term, nested number list is used in its own definition. **Recursive definitions** like this are quite common in mathematics and computer science. They provide a concise and powerful way to describe **recursive data structures** that are partially composed of smaller and simpler instances of themselves. The definition is not circular, since at some point we will reach a list that does not have any lists as elements.

Now suppose our job is to write a function that will sum all of the values in a nested number list. Python has a built-in function which finds the sum of a sequence of numbers:

```
>>> sum([1, 2, 8])
11
>>> sum((3, 5, 8.5))
16.5
>>>
```

For our *nested number list*, however, sum will not work:

```
>>> sum([1, 2, [11, 13], 8])
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'list'
>>>
```

The problem is that the third element of this list, [11, 13], is itself a list, which can not be added to 1, 2, and 8.

11.6. Recursion

To sum all the numbers in our recursive nested number list we need to traverse the list, visiting each of the elements within its nested structure, adding any numeric elements to our sum, and *repeating this process* with any elements which are lists.

Modern programming languages generally support **recursion**, which means that functions can *call themselves* within their definitions. Thanks to recursion, the Python code needed to sum the values of a nested number list is surprisingly short:

```
def recursive_sum(nested_num_list):
    sum = 0
    for element in nested_num_list:
        if type(element) == type([]):
            sum = sum + recursive_sum(element)
        else:
            sum = sum + element
    return sum
```

The body of recursive_sum consists mainly of a for loop that traverses nested_num_list. If element is a numerical value (the else branch), it is simply added to sum. If elementis a list, then recursive_sum is called again, with the element as an argument. The statement inside the function definition in which the function calls itself is known as the**recursive call.**

Recursion is truly one of the most beautiful and elegant tools in computer science.

A slightly more complicated problem is finding the largest value in our nested number list:

```
def recursive_max(nested_num_list):
     """
```

```
    >>> recursive_max([2, 9, [1, 13], 8, 6])
    13
    >>> recursive_max([2, [[100, 7], 90], [1, 13], 8, 6])
    100
    >>> recursive_max([2, [[13, 7], 90], [1, 100], 8, 6])
    100
    >>> recursive_max([[[13, 7], 90], 2, [1, 100], 8, 6])
    100
    """
    largest = nested_num_list[0]
    while type(largest) == type([]):
        largest = largest[0]

    for element in nested_num_list:
        if type(element) == type([]):
            max_of_elem = recursive_max(element)
            if largest < max_of_elem:
                largest = max_of_elem
        else:                           # element is not a list
            if largest < element:
                largest = element

    return largest
```

Doctests are included to provide examples of recursive_max at work.

The added twist to this problem is finding a numerical value for initializing largest. We can't just use nested_num_list[0], since that my be either a number or a list. To solve this problem we use a while loop that assigns largest to the first numerical value no matter how deeply it is nested.

The two examples above each have a **base case** which does not lead to a recursive call: the case where the element is a number and not a list. Without a base case, you have **infinite recursion**, and your program will not work. Python stops after reaching a maximum recursion depth and returns a runtime error.

Write the following in a file named infinite_recursion.py:

```
#
```

```
# infinite_recursion.py
#
def recursion_depth(number):
    print "Recursion depth number %d." % number
    recursion_depth(number + 1)

recursion_depth(0)
```

At the unix command prompt in the same directory in which you saved your program, type the following:

```
python infinite_recursion.py
```

After watching the messages flash by, you will be presented with the end of a long traceback that ends in with the following:

```
  ...
  File "infinite_recursion.py", line 3, in recursion_depth
    recursion_depth(number + 1)
RuntimeError: maximum recursion depth exceeded
```

We would certainly never want something like this to happen to a user of one of our programs, so before finishing the recursion discussion, let’s see how errors like this are handled in Python.

11.7. Exceptions

Whenever a runtime error occurs, it creates an **exception**. The program stops running at this point and Python prints out the traceback, which ends with the exception that occured.

For example, dividing by zero creates an exception:

```
>>> print 55/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
>>>
```

So does accessing a nonexistent list item:

```
>>> a = []
>>> print a[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>>
```

Or trying to make an item assignment on a tuple:

```
>>> tup = ('a', 'b', 'd', 'd')
>>> tup[2] = 'c'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>>
```

In each case, the error message on the last line has two parts: the type of error before the colon, and specifics about the error after the colon.

Sometimes we want to execute an operation that might cause an exception, but we don't want the program to stop. We can **handle the exception** using the try andexcept statements.

For example, we might prompt the user for the name of a file and then try to open it. If the file doesn't exist, we don't want the program to crash; we want to handle the exception:

```
filename = raw_input('Enter a file name: ')
try:
    f = open (filename, "r")
except:
    print 'There is no file named', filename
```

The try statement executes the statements in the first block. If no exceptions occur, it ignores the except statement. If any exception occurs, it executes the statements in the except branch and then continues.

We can encapsulate this capability in a function: exists takes a filename and returns true if the file exists, false if it doesn't:

```
def exists(filename):
    try:
        f = open(filename)
        f.close()
        return True
    except:
        return False
```

You can use multiple except blocks to handle different kinds of exceptions (see the Errors and Exceptions lesson from Python creator Guido van Rossum's Python Tutorialfor a more complete discussion of exceptions).

If your program detects an error condition, you can make it **raise** an exception. Here is an example that gets input from the user and checks that the number is non-negative.

```
#
# learn_exceptions.py
#
def get_age():
    age = input('Please enter your age: ')
    if age < 0:
        raise ValueError, '%s is not a valid age' % age
    return age
```

The raise statement takes two arguments: the exception type, and specific information about the error. ValueError is the built-in exception which most closely matches the kind of error we want to raise. The complete listing of built-in exceptions is found in the Built-in Exceptions section of the Python Library Reference, again by Python's creator, Guido van Rossum.

If the function that called get_age handles the error, then the program can continue; otherwise, Python prints the traceback and exits:

```
>>> get_age()
Please enter your age: 42
42
>>> get_age()
```

```
Please enter your age: -2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "learn_exceptions.py", line 4, in get_age
    raise ValueError, '%s is not a valid age' % age
ValueError: -2 is not a valid age
>>>
```

The error message includes the exception type and the additional information you provided.

Using exception handling, we can now modify infinite_recursion.py so that it stops when it reaches the maximum recursion depth allowed:

```
#
# infinite_recursion.py
#
def recursion_depth(number):
    print "Recursion depth number %d." % number
    try:
        recursion_depth(number + 1)
    except:
        print "Maximum recursion depth exceeded."

recursion_depth(0)
```

Run this version and observe the results.

11.8. Tail recursion

When the only thing returned from a function is a recursive call, it is refered to as **tail recursion.**

Here is a version of the countdown function from chapter 6 written using tail recursion:

```
def countdown(n):
    if n == 0:
        print "Blastoff!"
    else:
        print n
        countdown(n-1)
```

Any computation that can be made using iteration can also be made using recursion. Here is a version of find_max written using tail recursion:

```
def find_max(seq, max_so_far):
    if not seq:
        return max_so_far
    if max_so_far < seq[0]:
        return find_max(seq[1:], seq[0])
    else:
        return find_max(seq[1:], max_so_far)
```

Tail recursion is considered a bad practice in Python, since the Python compiler does not handle optimization for tail recursive calls. The recursive solution in cases like this use more system resources than the equivalent iterative solution.

11.9. Recursive mathematical functions

Several well known mathematical functions are defined recursively. Factorial, for example, is given the special operator, !, and is defined by:

```
0! = 1
n! = n(n-1)
```

We can easily code this into Python:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Another well know recursive relation in mathematics is the fibonacci sequence, which is defined by:

```
fibonacci(0) = 1
fibonacci(1) = 1
fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
```

This can also be written easily in Python:

```
def fibonacci (n):
    if n == 0 or n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

Calling factorial(1000) will exceed the maximum recursion depth. And try running fibonacci(35) and see how long it takes to complete (be patient, it will complete).

You will be asked to write an iterative version of factorial as an exercise, and we will see a better way to handle fibonacci in the next chapter.

11.10. List comprehensions

A **list comprehension** is a syntactic construct that enables lists to be created from other lists using a compact, mathematical syntax:

```
>>> numbers = [1, 2, 3, 4]
>>> [x**2 for x in numbers]
[1, 4, 9, 16]
>>> [x**2 for x in numbers if x**2 > 8]
[9, 16]
>>> [(x, x**2, x**3) for x in numbers]
[(1, 1, 1), (2, 4, 8), (3, 9, 27), (4, 16, 64)]
>>> files = ['bin', 'Data', 'Desktop', '.bashrc', '.ssh', '.vimrc']
>>> [name for name in files if name[0] != '.']
['bin', 'Data', 'Desktop']
>>> letters = ['a', 'b', 'c']
>>> [n*letter for n in numbers for letter in letters]
['a', 'b', 'c', 'aa', 'bb', 'cc', 'aaa', 'bbb', 'ccc', 'aaaa', 'bbbb', 'cccc']
>>>
```

The general syntax for a list comprehension expression is:

```
[expr for  item1 in  seq1 for item2 in seq2 ... for itemx in seqx if condition]
```

This list expression has the same effect as:

```
output_sequence = []
for item1 in seq1:
    for item2 in seq2:
        ...
            for itemx in seqx:
                if condition:
                    output_sequence.append(expr)
```

As you can see, the list comprehension is much more compact.

11.11. Mini case study: tree

The following program implements a subset of the behavior of the Unix tree program.

```
#!/usr/bin/env python

import os
import sys



def getroot():
    if len(sys.argv) == 1:
        path = ''
    else:
        path = sys.argv[1]

    if os.path.isabs(path):
        tree_root = path
    else:
        tree_root = os.path.join(os.getcwd(), path)

    return tree_root



def getdirlist(path):
    dirlist = os.listdir(path)
    dirlist = [name for name in dirlist if name[0] != '.']
    dirlist.sort()
    return dirlist


def traverse(path, prefix='|--', s='.\n', f=0, d=0):
    dirlist = getdirlist(path)

    for num, file in enumerate(dirlist):
        lastprefix = prefix[:-3] + '`--'
        dirsize = len(dirlist)

        if num < dirsize - 1:
            s += '%s %s\n' % (prefix, file)
        else:
            s += '%s %s\n' % (lastprefix, file)
        path2file = os.path.join(path, file)

        if os.path.isdir(path2file):
            d += 1
            if getdirlist(path2file):
                s, f, d = traverse(path2file, '|   ' + prefix, s, f, d)
        else:
            f += 1

    return s, f, d


if __name__ == '__main__':
    root = getroot()
    tree_str, files, dirs = traverse(root)

    if dirs == 1:
        dirstring = 'directory'
    else:
        dirstring = 'directories'
```

```
if files == 1:
    filestring = 'file'
else:
    filestring = 'files'

print tree_str
print '%d %s, %d %s' % (dirs, dirstring, files, filestring)
```

You will be asked to explore this program in several of the exercises below.

11.12. Glossary

base case
: A branch of the conditional statement in a recursive function that does not result in a recursive call.

data structure
: An organization of data for the purpose of making it easier to use.

exception
: An error that occurs at runtime.

handle an exception
: To prevent an exception from terminating a program using the try and except statements.

immutable data type
: A data type which cannot be modified. Assignments to elements or slices of immutable types cause a runtime error.

infinite recursion
: A function that calls itself recursively without ever reaching the base case. Eventually, an infinite recursion causes a runtime error.

list comprehension
A syntactic construct which enables lists to be generated from other lists using a syntax analogous to the mathematical set-builder notation.

mutable data type

A data type which can be modified. All mutable types are compound types. Lists and dictionaries (see next chapter) are mutable data types; strings and tuples are not.

raise
To signal an exception using the raise statement.

recursion
The process of calling the function that is currently executing.

recursive call
The statement in a recursive function with is a call to itself.

recursive definition
A definition which defines something in terms of itself. To be useful it must include *base cases* which are not recursive. In this way it differs from a *circular definition*. Recursive definitions often provide an elegant way to express complex data structures.

tail recursion
A recursive call that occurs as the last statement (at the tail) of a function definition. Tail recursion is considered bad practice in Python programs since a logically equivalent function can be written using *iteration* which is more efficient (see the Wikipedia article on tail recursion for more information).

tuple
A data type that contains a sequence of elements of any type, like a list, but is immutable. Tuples can be used wherever an immutable type is required, such as a key in a dictionary (see next chapter).

tuple assignment
An assignment to all of the elements in a tuple using a single assignment statement. Tuple assignment occurs in parallel rather than in sequence, making it useful for swapping values.

11.13. Exercises

```
def swap(x, y):      # incorrect version
```

```
    print  "before swap statement: id(x):", id(x), "id(y):", id(y)
    x, y = y, x
    print  "after swap statement: id(x):", id(x), "id(y):", id(y)

a, b = 0, 1
print  "before swap function call: id(a):", id(a), "id(b):", id(b)
swap(a, b)
print  "after swap function call: id(a):", id(a), "id(b):", id(b)
```

Run this program and describe the results. Use the results to explain why this version of swap does not work as intended. What will be the values of a and b after the call to swap?

Create a module named seqtools.py. Add the functions encapsulate and insert_in_middle from the chapter. Add doctests which test that these two functions work as intended with all three sequence types.

Add each of the following functions to seqtools.py:

```
def make_empty(seq):
    """
     >>> make_empty([1, 2, 3, 4])
     []
     >>> make_empty(('a', 'b', 'c'))
     ()
     >>> make_empty("No, not me!")
     ''
    """

def insert_at_end(val, seq):
    """
     >>> insert_at_end(5, [1, 3, 4, 6])
     [1, 3, 4, 6, 5]
     >>> insert_at_end('x', 'abc')
     'abcx'
     >>> insert_at_end(5, (1, 3, 4, 6))
     (1, 3, 4, 6, 5)
    """
```

```
def insert_in_front(val, seq):
    """
      >>> insert_in_front(5, [1, 3, 4, 6])
      [5, 1, 3, 4, 6]
      >>> insert_in_front(5, (1, 3, 4, 6))
      (5, 1, 3, 4, 6)
      >>> insert_in_front('x', 'abc')
      'xabc'
    """

def index_of(val, seq, start=0):
    """
      >>> index_of(9, [1, 7, 11, 9, 10])
      3
      >>> index_of(5, (1, 2, 4, 5, 6, 10, 5, 5))
      3
      >>> index_of(5, (1, 2, 4, 5, 6, 10, 5, 5), 4)
      6
      >>> index_of('y', 'happy birthday')
      4
      >>> index_of('banana', ['apple', 'banana', 'cherry', 'date'])
      1
      >>> index_of(5, [2, 3, 4])
      -1
      >>> index_of('b', ['apple', 'banana', 'cherry', 'date'])
      -1
    """

def remove_at(index, seq):
    """
      >>> remove_at(3, [1, 7, 11, 9, 10])
      [1, 7, 11, 10]
      >>> remove_at(5, (1, 4, 6, 7, 0, 9, 3, 5))
      (1, 4, 6, 7, 0, 3, 5)
      >>> remove_at(2, "Yomrktown")
      'Yorktown'
    """

def remove_val(val, seq):
    """
     >>> remove_val(11, [1, 7, 11, 9, 10])
     [1, 7, 9, 10]
     >>> remove_val(15, (1, 15, 11, 4, 9))
     (1, 11, 4, 9)
     >>> remove_val('what', ('who', 'what', 'when', 'where', 'why', 'how'))
     ('who', 'when', 'where', 'why', 'how')
    """

def remove_all(val, seq):
    """
     >>> remove_all(11, [1, 7, 11, 9, 11, 10, 2, 11])
     [1, 7, 9, 10, 2]
     >>> remove_all('i', 'Mississippi')
     'Msssspp'
    """

def count(val, seq):
    """
     >>> count(5, (1, 5, 3, 7, 5, 8, 5))
     3
     >>> count('s', 'Mississippi')
     4
     >>> count((1, 2), [1, 5, (1, 2), 7, (1, 2), 8, 5])
     2
    """

def reverse(seq):
    """
     >>> reverse([1, 2, 3, 4, 5])
     [5, 4, 3, 2, 1]
     >>> reverse(('shoe', 'my', 'buckle', 2, 1))
     (1, 2, 'buckle', 'my', 'shoe')
```

```
      >>> reverse('Python')
      'nohtyP'
    """

def sort_sequence(seq):
    """
      >>> sort_sequence([3, 4, 6, 7, 8, 2])
      [2, 3, 4, 6, 7, 8]
      >>> sort_sequence((3, 4, 6, 7, 8, 2))
      (2, 3, 4, 6, 7, 8)
      >>> sort_sequence("nothappy")
      'ahnoppty'
    """

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

As usual, work on each of these one at a time until they pass all of the doctests.

Write a function, recursive_min, that returns the smallest value in a nested number list:

```
def recursive_min(nested_num_list):
    """
      >>> recursive_min([2, 9, [1, 13], 8, 6])
      1
      >>> recursive_min([2, [[100, 1], 90], [10, 13], 8, 6])
      1
      >>> recursive_min([2, [[13, -7], 90], [1, 100], 8, 6])
      -7
      >>> recursive_min([[[-13, 7], 90], 2, [1, 100], 8, 6])
      -13
    """
```

Your function should pass the doctests.

Write a function recursive_count that returns the number of occurences of target in nested_number_list:

```
def recursive_count(target, nested_num_list):
    """
     >>> recursive_count(2, [2, 9, [2, 1, 13, 2], 8, [2, 6]])
     4
     >>> recursive_count(7, [[9, [7, 1, 13, 2], 8], [7, 6]])
     2
     >>> recursive_count(15, [[9, [7, 1, 13, 2], 8], [2, 6]])
     0
     >>> recursive_count(5, [[5, [5, [1, 5], 5], 5], [5, 6]])
     6
    """
```

As usual, your function should pass the doctests.

Write a function flatten that returns a simple list of numbers containing all the values in a nested_number_list:

```
def flatten(nested_num_list):
    """
     >>> flatten([2, 9, [2, 1, 13, 2], 8, [2, 6]])
     [2, 9, 2, 1, 13, 2, 8, 2, 6]
     >>> flatten([[9, [7, 1, 13, 2], 8], [7, 6]])
     [9, 7, 1, 13, 2, 8, 7, 6]
     >>> flatten([[9, [7, 1, 13, 2], 8], [2, 6]])
     [9, 7, 1, 13, 2, 8, 2, 6]
     >>> flatten([[5, [5, [1, 5], 5], 5], [5, 6]])
     [5, 5, 1, 5, 5, 5, 5, 6]
    """
```

Run your function to confirm that the doctests pass.

Write a function named readposint that prompts the user for a positive integer and then checks the input to confirm that it meets the requirements. A sample session might look like this:

```
>>> num = readposint()
Please enter a positive integer: yes
yes is not a positive integer.  Try again.
Please enter a positive integer: 3.14
3.14 is not a positive integer.  Try again.
Please enter a positive integer: -6
-6 is not a positive integer.  Try again.
Please enter a positive integer: 42
>>> num
42
>>> num2 = readposint("Now enter another one: ")
Now enter another one: 31
>>> num2
31
>>>
```

Use Python's exception handling mechanisms in confirming that the user's input is valid.

Give the Python interpreter's response to each of the following:

```
>>> nums = [1, 2, 3, 4]
>>> [x**3 for x in nums]
```

```
>>> nums = [1, 2, 3, 4]
>>> [x**2 for x in nums if x**2 != 4]
```

```
>>> nums = [1, 2, 3, 4]
>>> [(x, y) for x in nums for y in nums]
>>> nums = [1, 2, 3, 4]
>>> [(x, y) for x in nums for y in nums if x != y]
```

You should anticipate the results *before* you try them in the interpreter.

Use either pydoc or the on-line documentation at http://pydoc.org to find out what sys.getrecursionlimit() and sys.setrecursionlimit(n) do. Create several *experiments* like what was done in infinite_recursion.py to test your understanding of how these module functions work.

Rewrite the factorial function using iteration instead of recursion. Call your new function with 1000 as an argument and make note of how fast it returns a value.

Write a program named litter.py that creates an empty file named trash.txt in each subdirectory of a directory tree given the root of the tree as an argument (or the current directory as a default). Now write a program named cleanup.py that removes all these files. *Hint:* Use the tree program from the mini case study as a basis for these two recursive programs.

Source: http://openbookproject.net/thinkcs/python/english2e/ch11.html