ICS 32 Winter 2014
Code Example: Recursion

The problem

Summing the numbers in lists of integers

You've no doubt seen, in previous coursework, how to iterate over the elements of a list in Python using a **for** loop. For example, given a list of integers, you could use the following function to calculate the sum of all of the integers in the list. (You could also use the built-in **sum** function, but let's look at the detailed pattern; the nice thing about understanding the underlying pattern is that you can use the pattern to solve similar problems for which there aren't built-in solutions like **sum**.)

sum_numbers1.py

```
def sum_numbers(numlist: [int]) -> int:
    '''Adds up the integers in a list of integers'''
    sum = 0

    for num in numlist:
        sum += num

    return sum
```

Now suppose we change the problem just slightly, so that we instead are given a list containing *lists* of integers (always two levels of depth), but maintain the same goal of summing all of the integers. Our function changes somewhat, because we need to handle the additional level of depth; we need to loop over the sublists in the list, then loop over the integers in the sublists.

sum_numbers2.py

```
def sum_numbers(numlist: [[int]]) -> int:
    '''Adds up the integers in a list of lists of integers'''
    sum = 0

    for sublist in numlist:
        for num in sublist:
            sum += num

    return sum
```

So far so good. We can still solve the problem using tools we already have, though we did have to nest the **for** loops, so things have gotten a bit more complicated.

To add one final twist to the problem, now let's assume we want to sum the numbers in a list whose elements are *either* integers *or* lists of integers; so, in total, *no more* than two levels of depth. We can add the appropriate condition to our function to handle this case, which requires only that we know that the function **type(x)** returns the type of the object **x**, and that we can compare types using == to see if they're the same; that's a tool we could use to differentiate between the elements of the list that are integers and the elements that are lists.

sum_numbers3.py

```
def sum_numbers(numlist: [int or [int]]) -> int:
    '''
    Adds up the integers in a list whose elements are either integers or
    lists of integers
    '''
    sum = 0

    for element in numlist:
        if type(element) == list:
            for num in element:
                sum += num
        else:
            sum += element

    return sum
```

Unconstraining the problem

Imagine now what you might need to do to our function in order to be able to sum the numbers in a list like this one: **[[1, [2, 3], 4], [[5], 6], 7]**. What's different about this list is that it is three levels deep — there is a list containing a list that contains another list. Even the most complex of the three functions above is constrained to the problem of summing lists that might be up to two levels deep.

Using the strategy above, supporting a third level of depth would require another level of nesting in our function; the innermost **for** loop would need to contain another **if** statement to check the type of the elements in each sublist, and yet another **for** loop inside that to loop over the elements of the third-level sublist.

You could indeed solve the problem this way, but it wouldn't support four levels without a fourth level of nesting in the code. And then a fifth level of nesting in the list would require a fifth level of nesting in the code. And no matter how much patience you have in writing a function that's more deeply nested, there is a list that is potentially deeper than that.

But what if we unconstrain the problem altogether? Consider the following definition of a data structure:

- A *nested list of integers* is a list in which every element is either:
    - an integer
    - a nested list of integers

This data structure is *recursive*; it is included in its own definition. Inside of nested lists of integers can be smaller nested lists of integers, inside of which can be smaller ones still, and so on. Why this nesting does not continue forever is because we have a *base case*; our definition allows us to have a nested list of integers that contains only integers, but no more nested lists inside.

A function that can sum the numbers in a nested list of integers is the generalization of the three functions we wrote above. It would handle all of the cases those three functions handle, plus any other combination of nesting that is possible. In order to achieve that generalization, however, we need to embrace a new concept: a *recursive function*.

The code

The final code example from lecture, in which we summed the integers in a nested list of integers, is below.

- The **sum_recursive** module

The moral

Some programming languages encourage recursion as a primary form of repetition (i.e., if you need to do something repeatedly, you tend to want to use recursion above all else). Python is not one of them, however; in Python, your first inclination should be to use loops to solve problems of repetition.

However, what you soon discover is that not all problems lead to a well-formulated solution this way. Particularly when you start wanting to iterate through recursive data structures like our nested list of integers here, you find that no combination of nested loops will ever solve your entire problem. No matter how deeply you nest your loops, you can find an input that nests more deeply than your loops do. When you find yourself in that situation, recursion offers an approach that will lead to a general solution.

Why this is of particular interest in the context of Project #1 is because a file system *is* a recursive data structure. Directories contain collections of files and other directories, which, in turn, contain collections of files and other directories, and so on. You'll find the technique of using recursion very useful in traversing the file system and searching for files in directory structures that could be deeply nested.

On the other hand, you don't want to take this new knowledge too far, using recursion in situations for which it's not an appropriate solution in Python. For example, in the user interface you're building in Project #1, you'll sometimes need to repeatedly ask a user to answer a question until a valid answer is given (e.g., specifying the path to the directory where the search should begin). This kind of repetition is much better implemented in Python using a loop. In general, your inclination should be to use loops whenever you can, and recursion when you must. (And you should also bear in mind that this is not the same decision you would make if you were programming in a language other than Python.)