```
# sum_recursive.py
#
# ICS 32 Winter 2014
# Code Example
#
# This module contains a function capable of summing the numbers in a
# "nested list of integers".  A nested list of integers, by our definition
# from lecture, is a list in which each element is either:
#
#    * an integer
#    * a nested list of integers
#
# The key technique in use here is called "recursion".  More precisely,
# the function is a "recursive function", which is a function that uses
# itself in its own solution; part of what nested_sum does is to call
# nested_sum again.
#
# We talked at some length in lecture about the thinking behind writing
# a function like this, centering on a concept I call the "leap of faith",
# which means this:
#
#    When you're considering making a recursive call to a function,
#    make the assumption that it will do what it's supposed to do.
#    If, under that assumption, the function works -- if you work
#    out the details on paper and, based on that assumption, the
#    entire function will do what it's supposed to do -- then, as a
#    general rule, the function works.
#
# This may seem like an overly optimistic approach, but it has rarely
# led me astray (and when it has, it's generally been because I've made
# a mistake in assessing whether the function is correct).


def nested_sum(nested_list: 'nested list of integers') -> int:
    '''Adds up the integers in a nested list of integers'''
    sum = 0

    for element in nested_list:
        if type(element) == list:
            sum += nested_sum(element)
        else:
            sum += element

    return sum



# It's also possible to reason about a recursive function in a different
# way, by considering all of the steps.  This requires realizing that
# recursive function calls are no different than any other:
#
# * When a function is called, the calling function pauses until the
#   called function is complete.
# * When the called function is complete, the calling function picks up
#   where it left off, using the result returned from the called function.
#
# If the calling function and the called function are the same, that just
# means that there are two copies of the function running simultaneously,
# each with their own local variables and their own parameters.
#
# For example, given the input [[1, 2], 3], our function will do this:
```

```
#
# * nested_sum([[1, 2], 3]) will create a local variable "sum" and assign
#   0 to it, then will begin looping.  The first element is the list [1, 2],
#   so it will make a recursive call to nested_sum([1, 2]).
#
# * nested_sum([1, 2]) will create a local variable "sum" (separate from the
#   one in the caller) and assign 0 to it, then will begin looping.  The first
#   element in the list is 1, which is an integer, so it's added to sum.
#   Ditto the second element.  sum, in this iteration, has the value 3 after
#   the "for" loop completes.  3 is returned.
#
# * nested_sum([[1, 2], 3]) picks up where it left off, by adding the result
#   of the recursive call (3) to its copy of sum (0), so its copy of sum now
#   has the value 3.  Continuing in its loop, the next element is 3, which is
#   added to the sum (3) giving 6.  There are no more elements in the list,
#   so the final result is 6.



assert(nested_sum([3, 6, 4]) == 13)
assert(nested_sum([[[1, 2], 3], 4]) == 10)
assert(nested_sum([[2, 7], [3, 8], [4, 9]]) == 33)
assert(nested_sum([1, [2, [3, [4, [5], 6], 7], 8], 9]) == 45)
assert(nested_sum([]) == 0)
```