**Recursion**

*Objectives: Become comfortable with treating functions as their high level definitions, understand the purpose and function of the base case(s) in a recursive function, be able to reduce large problems into more manageable subproblems.*

In this guide we’ll use an example to understand the three main steps to any recursive problem:

* Choose the base case(s)
* Move closer towards said base case(s)
* Put it all together to solve the larger problem

**Our Problem**

We want to write sum\_range(lst, i, j) which is a function that returns the sum of the values between indices i and j (inclusive) of lst using recursion. We’ll assume that lst contains only integers. For example, the following call to sum\_range will sum up the values 4, 5, and 6 and return 15 as the final result.

>>> lst = [3, 4, 5, 6, 7, 8]

>>> sum\_vals = sum\_range(lst, 1, 3)

>>> sum\_vals

15 # 4 + 5 + 6



**Forming a High Level Definition**

At a high level, what should sum\_range do? Assuming it works correctly, given a python list and two valid indices, our function should return the sum of the values within that range. We can consider sum\_range as an item in our “toolkit”, meaning it is at our disposal to use however we like to solve our problem. As long as we use our function as it was defined at a higher level (and maintain the integrity of the inputs/outputs), we can abstract away the implementation and use it to our advantage.

**Breaking Down the Problem**

From the doctest, our goal is to return the sum of 4, 5, and 6 (the values from our list). How can we reduce this problem a bit further? Well, it’s a bit easier if we only have to deal with two numbers rather than three. So we could reduce our problem to something a bit more manageable, such as summing the numbers in a smaller range.

sum\_range(lst, 2, 3)



But wait, aren’t we changing our whole problem? We just lost the value lst[1], and following our high level definition, this different call to sum\_range will only sum up 5 and 6 from our original list. We did solve part of our problem, all we’re missing now is that one value at lst[1]. We can generalize this subproblem as being the sum of the values from indices i + 1 to j, or sum\_range(lst, i + 1, j).

**Putting it All Together**

Since we know that sum\_range(lst, 2, 3) will do the work of summing 5 and 6 for us, all we’re missing is the value 4, found at lst[1]. So to solve our entire problem of summing all the values from i to j, we can simply add the element from lst[i] to our subproblem.

def sum\_range(lst, i, j):

return lst[i] + sum\_range(lst, i + 1, j)

We’re still not done yet. Although we did break it up into one value and a subproblem, we’ll still throw an infinite recursion error since our function will never stop. We still need some sort of stopping condition so that our function doesn’t run forever.

**Choosing the Base Case(s)**

By definition, a function must call itself in order to be recursive. Though like with our initial implementation shown above, this could potentially go on forever. Our base case provides a condition for when we want to stop the recursion, or when we can’t reduce the problem any further. It can also handle invalid input, or anything that would cause the rest of our function to break or malfunction.

By looking at our subproblem, we can see that we’re reducing our problem by incrementing our i index value, which makes our range smaller. How do we know when to stop decreasing the size of our range? Well, we know that a range of length 1 only contains one element. Since we defined our range to be inclusive, meaning we want to include that values at our original lst[i] and lst[j] in our final sum, we know that once our i index increases enough to become the same as our j index, we can stop the recursion because we only have one final element to add--the value at lst[i].

def sum\_range(lst, i, j):

if i == j:

return lst[i] # or lst[j], as the indices are the same

return lst[i] + sum\_range(lst, i + 1, j)

**Trusting the Abstraction**

How can we be sure that this solution works? According to our high level definition, we abstracted away the sum\_range function as something that takes in a list and two indices and returns the sum of the elements within that range. We solved our larger problem by using this, and breaking up the entire sum into the sum of the first value and the sum of the smaller range. We are able to assume that our call sum\_range(lst, i + 1, j) works because we maintained the integrity of our initial high level definition and assumed it would give us the sum of a specified range.

We can also add an extra condition in case our user tries to plug in some invalid values. We want to make sure that i <= j so by adding the following check we make sure that we only sum across valid ranges, and return 0 for all invalid ranges.

def sum\_range(lst, i, j):

if i > j:

return 0

elif i == j:

return lst[i] # or lst[j]

return lst[i] + sum\_range(lst, i + 1, j)

Something fun to try out: justify whether or not the elif clause is necessary for the correctness of our function! In other words, does the following version work the same or differently from what we had before?

def sum\_range(lst, i, j):

if i > j:

return 0

return lst[i] + sum\_range(lst, i + 1, j)