**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

https://docs.google.com/a/berkeley.edu/drawings/d/s5djIBCzNnGglwA8Bul1XjQ/image?w=290&h=34&rev=26&ac=1&parent=14wVXEVlYTxTkKBznkBmTgHBZhttYk2EahdA_eJhYFyU

**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)

https://docs.google.com/a/berkeley.edu/drawings/d/s2m6cgV5JZ_EtRr8bc3kZUw/image?w=290&h=34&rev=1&ac=1&parent=14wVXEVlYTxTkKBznkBmTgHBZhttYk2EahdA_eJhYFyU

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)

**Trees**

A **tree** is just some data structure that holds a **node** (which can be a number, string or any other value) and a collection of **branches** (which is either empty or consists of other trees). They turn out to be extremely useful in computer science, so 61A is not trying to torture you without good reason.

There are a few tricky things about trees:

* You have to understand data abstraction to use them correctly -- don’t directly index into a tree! Only use its selectors and constructors (this is easier when you use objects)
* Trees are inherently recursive data structures because they satisfy the **closure property** (this is in your textbook, but you won’t need to know it for this class). Why is this difficult? A tree could be many layers down, so if you’re trying to iterate through it normally you won’t know how to get to every leaf. To solve this problem, we use **recursion**.

Here’s a typical template for a tree processing function (note that there are other formats, but this is a common one):

def f(t):

[BASE CASE]

[DO SOMETHING WITH THE FIRST LAYER]

for b in branches(t):

[RECURSIVE CALL]

[POSSIBLY RETURN SOMETHING]

A few things to note here:

* Often times the base case is implicit. For example, if a tree is a leaf then it will skip over the recursive calls anyway because there are no branches to iterate over.
* Sometimes you need to return something, and sometimes you don’t (especially if you’re just mutating the original tree). Check the problem specifications to be sure.

As a final note, these types of problems really require recursive thinking. If you’re having a lot of trouble, brush up on tree recursion and that should help. Like everything else in 61A, it takes effort and practice to understand trees -- pretty much no one just “gets it” from day one, but by the end of the semester you’ll look back and realize that you’ve mastered the concept.

**Data Abstraction**

*Objectives: Understand the motivation for abstracting away functions and other information, apply the concept of data abstraction towards the solving of large problems, design modular programs. We will approach data abstraction from two directions: from low-level to high-level and then high-level to low-level.*

Data abstraction is really, really powerful. It is the transformation of data into concepts. Consider a printed piece of paper with a few paragraphs of text on it. At a low level of abstraction the paper is a piece of compressed and bleached wood fiber and the ink has some complex chemical structure. There was a very complicated process to make the paper and the ink as well. The ink outlines characters that our brains process as letters of the alphabet. Take a step back, viewing the text at a higher level of abstraction, and the letters compose words. Another step and the words compose sentences. The sentences convey ideas and the paragraphs form arguments and perspectives.

The take-away from this is that we have a choice between treating things as concepts or as ink on wood fiber. ‘Ink on wood fiber’ could be so many things--it could be an essay, a book, [SICP](http://web.mit.edu/alexmv/6.037/sicp.pdf), or even this guide. When we concentrate on the conceptual ideas of our functions we can refer and manipulate them as units, which lets us solve more complex problems.

**Functions**

At this point, we’ve written a variety of different functions. We’ve written functions that print output, calculate fibonacci, and play simple games with the user--to name a few. In this guide one of our goals is to understand what certain functions are doing “at a higher level”, which means without getting into all the technical things, talk about what the function does in words we can understand outside of the context of computer science. Let’s try out an example with the following function.

def is\_odd(x):

if x % 2 == 1:

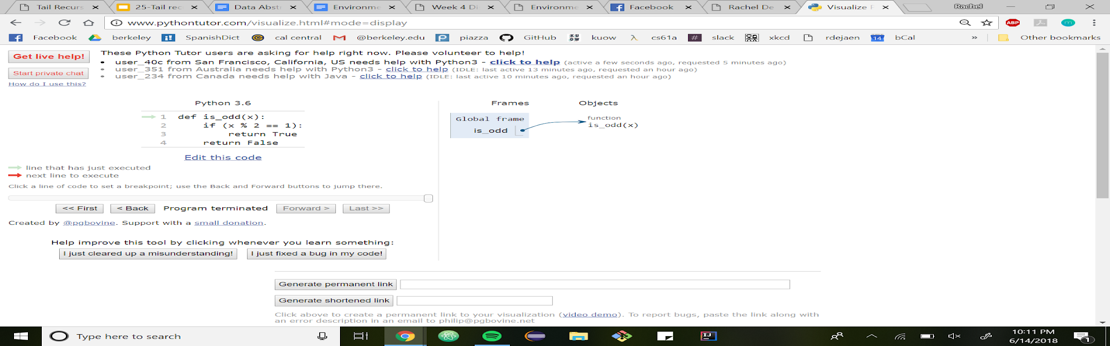
return True

return False

Now at a higher level, what is the function doing?

So for ourselves, we can say that our is\_odd function will tell us whether or not the argument is odd. This is an effective higher level definition because it concisely summarizes the behavior of the function in words we can understand. An example of an ineffective higher level definition would be “the function returns true if x % 2 is 1, and false otherwise”. This doesn’t work as well because first off it gives away the implementation, and it also doesn’t explain the purpose of the function, which is to determine if the argument is odd.

Let’s see how the is\_odd function is shown in an environment diagram.



How much information do we have from the diagram alone? Not a whole lot is happening, we just have the name “is\_odd” assigned to some function that takes in one argument, and a parent frame.

Now looking at the environment diagram and our higher level definition for what our function does, where are we actually doing the action of checking if we have an odd number? Not in either one, the actual check is happening inside the body of our function. But does that stop us from calling our function? Of course not! When we’re only concerned with the functionality and applicability of a function, there’s no need to concern ourselves with the inner workings as long as we trust that it works as it’s supposed to.

In our environment diagram, we can’t see any of the code that’s actually doing the work. Even if we could, it would make our diagram pretty messy and difficult to read, especially if we had an even longer or more complicated function. The work is all kept secret, behind the scenes, and this allows us to treat our functions as things instead of lines of code.

This avoidance of messiness translates over to the idea of abstraction as well, because it’s possible that our programs end up being relatively complicated, so much so that it’s difficult for another person to understand it. We’ll discuss how to avoid this later on, but it’s important to understand that dealing with only the higher level definitions of functions means we don’t have to worry about the ground level work our function is doing as long as we trust that it works properly and use it as it was defined. This means, we should be able to call is\_odd(5)  and trust that we get “True” no matter the implementation.

**What We Can Do**

Abstraction gives us a lot of flexibility. If we only consider our functions as their high level definitions, it lets us do some pretty powerful things. Let’s consider a problem: we want to write a function that prints the following output:

one

two

two

three

three

three

four

four

four

four

This function will print the English spelling of the numbers 1 through 4 inclusive, and repeat each one an increasing number of times. There are many ways to approach this function, the first one that may come to mind is a function with 10 print statements, let’s call it numbers().

def numbers():

print(“one”)

print(“two”)

print(“two”)

print(“three”)

print(“three”)

print(“three”)

print(“four”)

print(“four”)

print(“four”)

print(“four”)

This is kind of messy. It may be easy to understand but there are a few things to note about this function, the first being repetitiveness. Look at how many times the word print appears, and how many times we had to type out our numbers! Let’s look for any patterns in our desired output to help us condense this a bit.

Our high level definition said that we should repeat “one” once, “two” twice, “three” three times and so on. Let’s modify our function to replicate this pattern, and introduce some loops to help us out:

def numbers():

for i in range(0, 1): #repeats once

print(“one”)

for i in range(0, 2): #repeats twice

print(“two”)

for i in range(0, 3): #repeats three times

print(“three”)

for i in range(0, 4): #repeats four times

print(“four”)

This function is shorter, and we solved the problem only having to type out our words one time each, so we’ve made some progress. But we’re still repeating a series of actions. Notice how we have 4 different for loops, what are they each doing individually? They each repeat a word a certain number of times, and we increment the number of time we repeat as we move along. So the only things that are different between each of the for loops are:

1. The number of iterations
2. The word that’s printed

def numbers():

for i in range(0, **1**):

print(“**one**”)

for i in range(0, **2**):

print(“**two**”)

for i in range(0, **3**):

print(“**three**”)

for i in range(0, **4**):

print(“**four**”)

We can generalize our function’s behavior even further. Currently, our function prints out a word (let’s call it w) a certain number of times (let’s call it n). We’re repeating this action four different times, and the only things that are changing are w and n. Our entire problem would be easier if we had a function that let us plug in w and n, and run the same action four different times. Let’s define repeat\_print(w, n) that repeats our word, w, n number of times. The implementation isn’t important for this problem as long as we know the functionality, though feel free to write it yourself.

We generalized our entire problem as a repetition of smaller actions. To print our entire output, we had to print “one” a certain number of times, “two” a certain number of times, and so on. How does that change our numbers() function?

def numbers():

repeat\_print(“one”, 1)

repeat\_print(“two”, 2)

repeat\_print(“three”, 3)

repeat\_print(“four”, 4)

We structured our program around the task we were repeating, which was printing, and delegated the work to repeat\_print. We were able to change the words we printed, and the number of times they were printed, by specifying the arguments we pass in. This allowed us to break down our problem into a series of smaller, easier to handle ones.

We’re still calling repeat\_print a bit excessively, and we notice as well that our n value is increasing by a constant amount. To simply this a bit further, let’s introduce a loop and stick all of our strings in a python list to make them easier to handle.

def numbers():

words = [“one”, “two”, “three”, “four”]

for i in range(0, 4):

repeat\_print(words[i], i + 1)

**Modularity**

We can design our functions keeping in mind that we want to do as little work as possible. We’re lazy programmers, but still want beautiful, readable code, so any time that we can reuse functions is really nice. This has a lot of applications as well, because in the real world other people are going to be using our programs, or programs that depend on our programs, and in order for them all to work nicely we should make as few assumptions about our data as possible. This gives our functions as many applications as possible. For example, we designed repeat\_print to not assume anything about the w variable--it could be a number, a string, a list...anything--all we cared about was that we printed it.

Currently, numbers correctly displays the desired output without the repetitiveness of our previous attempts. But say instead we wanted to print the same thing but with numbers five through ten? Or with the Spanish spelling of our numbers? Then we’d have to re-write our entire function. Let’s modify numbers to take into account the fact that we may want to have these kinds of variations of output.

First, let’s consider the behavior of the function we’re trying to write. We want to maintain the general idea of numbers so far, but want the flexibility of having it work on different sets of words and different ranges of numbers. We’ll let the user specify the list of words to be printed and the number of times the first word should be repeated as arguments. Let’s assume each element in words represents a word to be repeated.

def numbers(words, start):

Considering our initial problem, what would we have to pass in as words and start to get the same output?

words = [“one”, “two”, “three”, “four”]

start = 1

So, trusting our high level definition, calling numbers([“one”, “two”, “three”, “four”], 1) should get us the desired output. Let’s rewrite numbers to allow for a variety of output. Here’s our old implementation for reference:

def numbers():

words = [“one”, “two”, “three”, “four”]

for i in range(0, 4):

repeat\_print(words[i], i + 1)

def numbers(words, start):

How many times should we be calling repeat\_print? Well, in our previous implementation we called it four times because we had four different words to be printed. We won’t always have four words, we could have five or six or however many the user specifies. Let’s have it loop through however many words the user provides in the words list since that’s what it’s dependent on.

def numbers(words, start):

for i in range(0, len(words)):

repeat\_print(words[i], \_\_\_)

What determines our n value when we call repeat\_print? In the previous interpretation it was simply i since our range was from 1 to 4. We have to account for if the desired range of times to be repeated is anything else. For instance, the user may want to repeat “one” three times, “two” four times, and so on.

def repeat\_print(w, n):

#prints w n times

def numbers(words, start):

for i in range(0, len(words)):

repeat\_print(words[i], start + i + 1)

We added modularity to our program because at first, numbers had only one purpose: print the numbers 1 through 4 an increasing number of times. But through generalizations and slight modifications we greatly expanded the capabilities of our function to accept any list of words and any start value.

Solving the smaller problem of repeating a single word a certain number of times (repeat\_print)  helped us solve our larger problem, since we were able to generalize it as a series of subproblems.

**Deconstructing the Problem**

Solving multiple small problems is a lot easier than trying to solve one big one. We’ll go over how to break down a high level concept (like a square) into its fundamentals to make it easier to implement in code.

Our eventual goal is to have a variable called square that represents a square with 4 coordinate points. We can start by forming some ideas about squares to get a generalized definition, and once we have something easier to handle, we’ll start implementing it and work from there.

Squares have 4 points, and a point is just a pair of numbers representing the x and y coordinates. To solve our entire problem, we need a way to construct a square from 4 coordinate points, as well as a way to make said points. We’re going to assume we have all of those, trust that they are implemented correctly, and create our square.

We’re assuming we have:

* A function that takes in 4 coordinate points and returns a square
* A function that takes in an x and a y value and returns a coordinate point

square = make\_square(make\_point(0, 0),

make\_point(1, 0),

make\_point(1, 1),

make\_point(0, 1))

That wasn’t too bad! Of course we haven’t implemented our helper functions yet, but as long as our subdivisions were small enough (the assumptions we made) the actual implementation should not be too complicated. In general, if your helper functions are so complicated that they’re difficult to understand, or there is clear repetition, that probably means there’s more abstraction to be done.

We’ll start by writing the first assumption we made. Squares have 4 points, so our function should return a representation of a square with all the information from the 4 coordinate points included. For this example we’ll use an underlying list implementation, though other data structures also could’ve worked.

def make\_square(p1, p2, p3, p4):

return [p1, p2, p3, p4]

Our second assumption said that given two numbers, we should get a representation of a coordinate point with those values as x and y. We’ll use a list structure for this one as well to represent the pair.

def make\_point(x, y):

   return [x, y]

Notice how the only times we see the list brackets are within the make\_square and make\_point functions, and not when we actually create our square. This is because the underlying structure of our square is behind the abstraction barrier. This gives us the flexibility if in the future we want to represent our square with some other data structure (like tuples), as long as we preserved the invariants (the integral parts, should never change) of our functions, our code will work exactly the same.

make\_square is great at making squares, but let’s extend it one step further. What if we wanted to make a triangle, or a hexagon? We’d have to write a unique function for each one, each with a different number of points. We can generalize all polygons by writing a make\_polygon function instead to encapsulate the characteristics of all potential polygons we might want to represent.

For make\_square, we had 4 different parameters since we needed 4 points to make a square. Since polygons can have a varying number of points, we’ll introduce the [\*args](https://stackoverflow.com/questions/3394835/args-and-kwargs) parameter, which means our make\_polygon function can take in any number of arguments. args is a list that contains all of our arguments, so our make\_polygon function can simply return args so that we still get a list of all our arguments in one, just as we did in make\_square.

def make\_polygon(\*args):

   return args

Now that we have our new polygon constructor, let’s rewrite square.

square = make\_polygon(make\_point(0, 0),

make\_point(1, 0),

make\_point(1, 1),

make\_point(0, 1))

This may look nearly identical to what we had before, but it has that extra level of modularity that our previous implementation didn’t have. This way, with the single function make\_polygon, we can make as many different shapes as we want--before we were just limited to squares.

We can compare this to another implementation without abstraction, which lacks the multifunctionality, modularity, and organization of our previous implementation.

square = [[0, 0], [1, 0], [1, 1], [0, 1]]