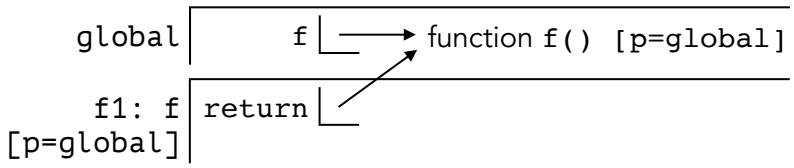
Intro

Iteration and variable reassignment can be handy, but they are the dark arts of computer science. They open the flood gates to what we call data mutation, they're often inefficient, and they make it very hard to parallelize programs. To a trained programmer, they're ugly and needlessly complicated. What is the alternative? Recursion. This is how the functional programmer writes good code.

Self-Access

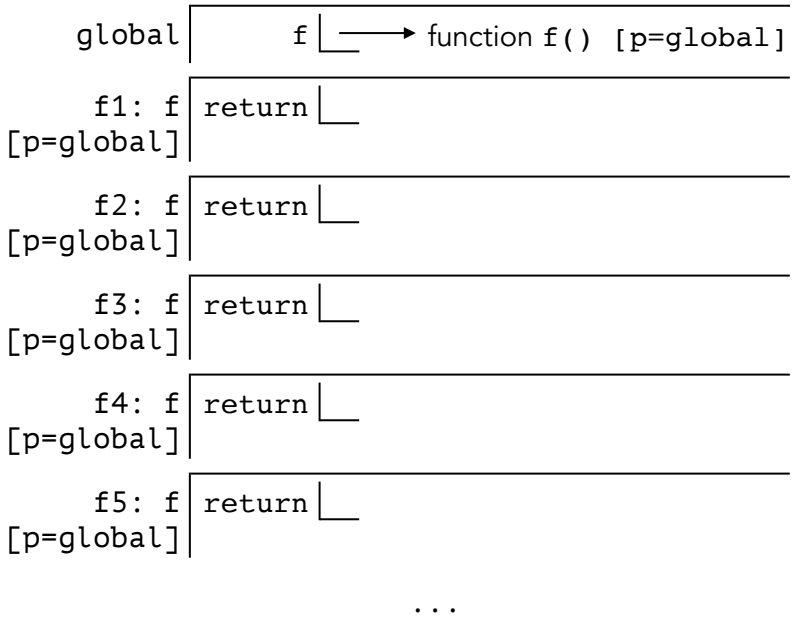
Take a look at the next piece of code. Notice that since `f`'s parent is `global`, it is able to access the name `f` as defined in the global frame. In other words, it can refer to itself during execution.

```
def f():
    return f
f()
```



This is important. It means that while we're evaluating `f`, we could potentially make another call to `f` before the first one even finishes. Why would we do that? We'll see soon enough. In the above example, `f` doesn't call itself. It only refers to itself. Now lets write a function that actually calls itself.

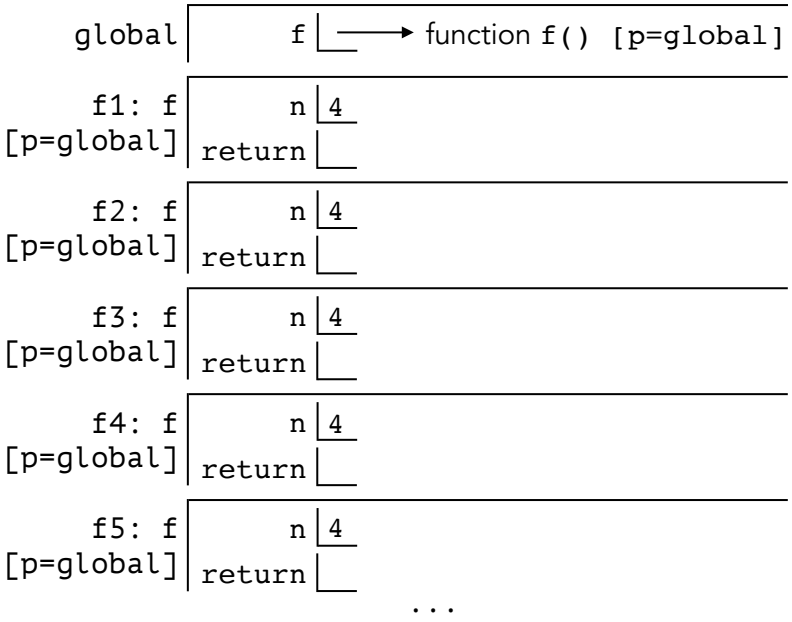
```
def f():
    return f()
f()
```



Here, we open frame 1 to evaluate the function `f`. It returns another call to the function `f`, so we open frame 2 to evaluate the function `f`. It returns another call to the function `f`, so we open frame 3 to evaluate the function `f`. This keeps going ... *forever!* Oops.

We need to be able to stop the recursion under some condition, eventually. How about we add an `if` statement? It will basically say "if we're done, then we can stop". But now we also need to define what it means to be "done", so let's add a variable `n` and say we're done when `n` reaches `0`. Now our code looks like this:

```
def f(n):
    if n == 0:
        return 1
    return f(n)
f(4)
```

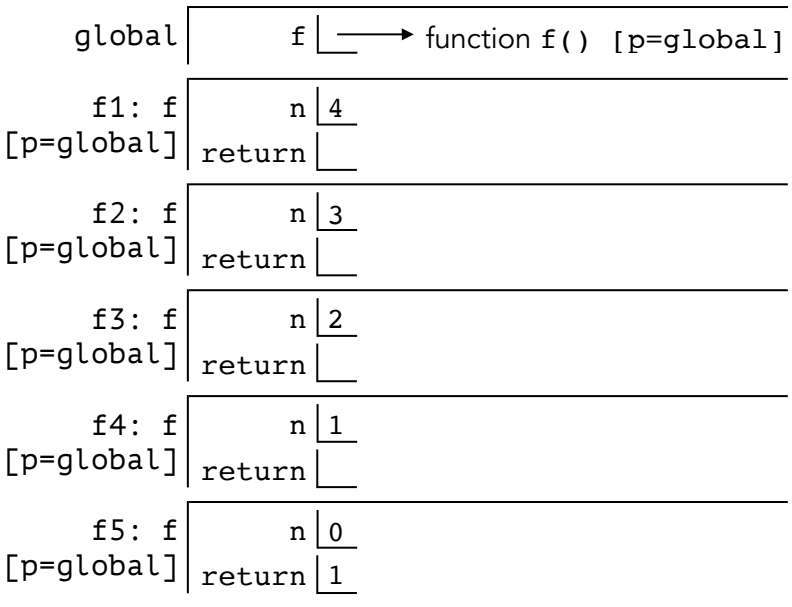


It's pretty much the same as before. There's now a case that tests whether we've finished, but right now we never actually move closer to finishing. In other words, we set up a finish line but we're not getting any closer to it. In order for recursion to work, we need to do two things:

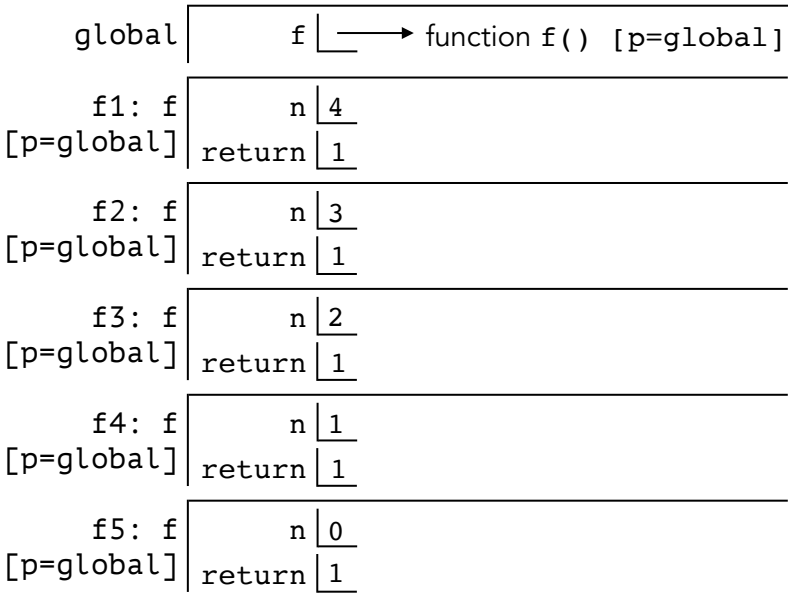
- Test whether we're done.
- Move closer to being done.

In our example, we said that being done means `n` reaches `0`. For that to happen, we have to make `n` smaller. How about we decrement it by one, each time we call the function recursively?

```
def f(n):
    if n == 0:
        return 1
    return f(n - 1)
f(4)
```



In frame `f5`, we finally arrive at `n == 0`. Now the `if` case passes, and we return `1`. That value climbs back up the stack of recursive frames until we see that `f1`'s final return value is also `1`.



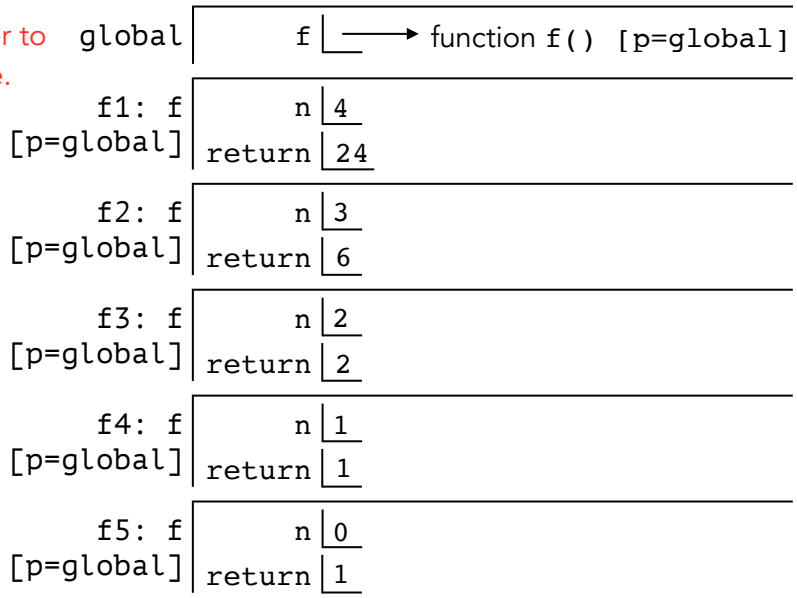
That's the basic idea behind recursion. But how can we use it to make something useful? Well, we need to involve the recursive call with another calculation. Here's a `factorial` function that's very similar to the example we had before. Notice how we multiply the result of the recursive call by a factor of `n`, to do something meaningful.

Test whether we're done.

Move closer to being done.

```
def f(n):
    if n == 0:
        return 1
    return n * f(n-1)
factorial(4)
```

Use the recursive call in a useful calculation.



Recursion is actually a lot like eating pie. It's very hard to eat an entire pie in one bite, so what do we do? Make the problem smaller, by cutting out one bite-sized scoop from the pie. Then repeat with the rest of the pie. This is what recursion is all about. Make the problem smaller by cutting off one piece, then recursively address whatever is left of the problem.

Before we get into code writing, there are two important ideas which you should know. Refer to these two points while you're starting out.

- Recursion is all about making your parameters smaller.
  This should manifest in two ways. First of all, every recursive call you make should take in smaller parameters than the call you're currently in. And second, your base case should reflect the smallest value you ever want your parameters to have. (Don't address your second-smallest or third-smallest parameter in the base case. Only the very smallest one. Usually, if you need to use an `if` statement to break your base case down even further, then you're doing something wrong.) We'll revisit this point in the next chapter.

- Stay in the same frame.
  You shouldn't get caught up worrying whether your recursive call works. You just need to trust that it will do what you want. If you're testing out `factorial(5)`, you should only worry about what happens inside that frame. Don't worry how `factorial(4)` gets handled, just assume it works and it will return 24. If you try to think about what goes on inside the frame for `factorial(4)`, and then `factorial(3)`, and so on, then you've fallen down a rabbit hole you'll never climb out of. Don't think about how the recursive call works. Just think, "What should the recursive call evaluate to?" and assume it will evaluate to that. Focus only on the current frame, assuming every recursive call outputs what it should.

Practice: sum_lst
Write a function `sum_lst` that takes in a list and returns the sum of every element in the list.

```
def sum_lst(lst):
    ...
```

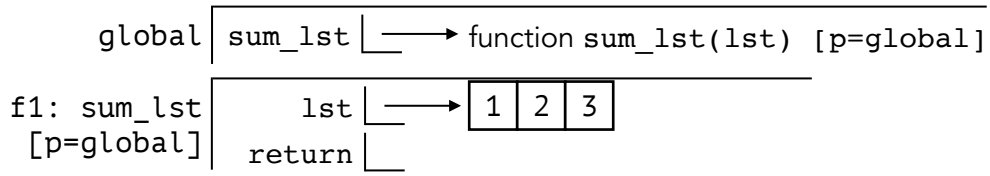
First things first, we need a base case. Our parameter is a list, so the smallest it'll ever be is an empty list. We choose this as our base case. Since an empty list has nothing to sum, we'll return `0`.

```
def sum_lst(lst):
    if lst == []:
        return 0
    ...
```

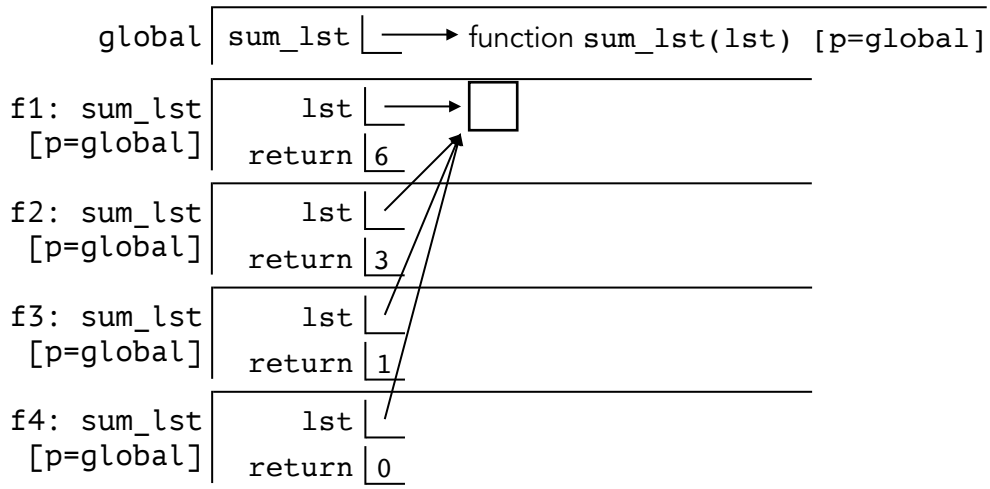
Now we need to add in the recursive call. Remember, we have to give it a parameter smaller than the one in the current frame. Let's try popping the last element off, and using the shortened list as the argument to the recursive call. When you write this recursive call, think about what it should evaluate to. Since we're giving it all but the last element in the list, it will return the sum of every element except the last one. In order for our function to work, we have to add the element we popped off, together with result of the recursive call.

```
def sum_lst(lst):
    if lst == []:
        return 0
    return lst.pop() + sum_lst(lst)
```

Here's what the environment diagram looks like, at the beginning of the initial call `sum_lst([1,2,3])`.



At the end, it looks like this.



`sum` is actually a built-in function, and does the exact same thing as the function `sum_lst` that we defined here. We'll see it come back a few chapters from now, when we're talking about trees.

Practice: comprehension

Recall list comprehensions from the previous chapter. We will implement the same thing recursively. It should take in a list `lst`, a function `f`, and a predicate `pred`. For example:

```
>>> comprehension([0,1,2,3,4,5], lambda x: x * x, lambda x: x % 2)
[1, 9, 25]
```

```
def comprehension(lst, f, pred):
    ...
```

Notice `f` and `pred` shouldn't change in our recursive calls, so we will recurse on `lst`. That means our base case should be the smallest possible input for `lst`, which is the empty list. If that is our input, we'll return the empty list, since there are no elements in the comprehension.

```
def comprehension(lst, f, pred):
    if lst == []:
        return []
    ...
```

Now for the recursive call. It's input should be smaller than the input to the current frame, so we will pass in `lst[1:]` as our argument. That recursive call would return the comprehension on every element except the first one. We'll have to add in `[f(lst[0])]` at the beginning, if the first element is to be included in our final answer.

```
def comprehension(lst, f, pred):
    if lst == []:
        return []
    first = [f(lst[0])] if pred(lst[0]) else []
    return first + comprehension(lst[1:], f, pred)
```

Notice that for `sum_lst` we made a smaller list by popping off elements, but in `comprehension` we used list slicing instead. You could accomplish both problems either way. The only difference is that popping mutates the original list, whereas slicing does not.

Variables as Parameters

Often times, you need to keep track of some information and change it across multiple recursive calls. If you were solving things iteratively, you could just define a variable outside your `while` loop, and then modify it from within. We can't do that with recursion, since we're not able to change a variable that's in a different frame. The solution? Instead of storing that information as an external variable, we can store it as a parameter to the recursive call.

Look at the next example. In the iterative solution, we keep track of `i` as a variable outside the `while` loop. In the recursive solution, we keep track of `i` as a parameter whose default value is the starting value of `i`.

```
def count(total):
    i = 0
    while i < total:
        i += 1
        print(i)
```

```
def count(total, i=0):
    i += 1
    print(i)
    if i < total:
        return count(total, i+1)
```

Practice: balanced

Let's try writing a function `balanced`, which takes in a string and returns whether the parentheses are balanced within that string. For example:

```
>>> balanced(':)  :(')
False

>>> balanced('((this many parens)')
False

>>> balanced('(yeeeee :) (haww(!))')
True

def balanced(s):
    ...
```

Our parameter is a string, so its smallest possible value is an empty string. We will use this as our base case, and return `True` because there are no mismatched parentheses in an empty string.

```
def balanced(s):
    if not s:
        return True
    ...
```

The argument to the recursive call has to be smaller than the current input. Let's do the same thing we did in the previous examples, and slice off the first character in the string `s`. Now we can make the recursive call `balanced(s[1:])`, which will return whether the rest of `s` is balanced. The issue is, that doesn't give us enough information to determine whether `s` is balanced or not.

In the last two practice problems, we only needed to know the first element, and the result of the recursive call. That was enough to determine our output. But notice that for both examples below, `s[0]` is `'('` and `balanced(s[1:])` is `False`. Yet, their outputs are different. It seems we need more information than just those two things, to determine what our output should be.

```
>>> balanced('(hi)')
True
```

```
>>> balanced('(hi(')
False
```

This is where you would define an external variable, if we were solving this iteratively. Since we're using recursion, we'll add another parameter instead. It will represent what depth of parenthetical the string starts at. If it ends at depth `0`, then it's balanced. Otherwise, it's not. In the next line, each number represents the parenthetical depth at that point in the expression.

```
0 ( 1 ( 2 ) 1 ) 0 ( 1 ( 2 ( 3 ) 2 ) 1 ) 0
```

Adding in our brand new parameter, this is what we get:

```
def balanced(s, depth=0):
    if not s:
        return depth == 0
    ...
```

We have already determined the base case for `s` is the empty string. The base case for `depth` is `0`, since it's a number. But we could receive an input that's less than `0`, or bigger than `0`. Let's address each of these cases individually.

If `depth` is negative at any point, that means we've encountered more end-parens than begin-parens. In that case, `s` would be unbalanced so we can return `False`.

```
def balanced(s, depth=0):
    if not s:
        return depth == 0
    if depth < 0:
        return False
    ...
```

Now we have addressed the cases where `s` is empty, or `depth` is less than or equal to `0`. If the function still hasn't terminated, that means `s` is nonempty and `depth` is positive. It is time to think about making a recursive call. Remember, we want to pass in a smaller string than we took as input, so we'll slice `s` like we did before. But now we need to also pass in the appropriate value for `depth`. If the letter we took off was a `'('`, then the rest of the string is one level deeper so we should increment `depth`. If the letter we took off was a `')'`, then the rest of the string is one level shallower so we should decrement `depth`. If the element we took off was not a paren, then the depth would stay the same so we don't modify `depth`. In the end, this is what we get:

```
def balanced(s, depth=0):
    if not s:
        return depth == 0
    if depth < 0:
        return False
    if s[0] == '(':
        return balanced(s[1:], depth + 1)
    if s[0] == ')':
        return balanced(s[1:], depth - 1)
    else:
        return balanced(s[1:], depth)
```

For extra practice, try writing this function iteratively. Analyze how the two implementations are similar and different.

Remember, you can always add more params if you want more info. Here it works out well because we need to keep track of what parenthetical depth we're at, across multiple recursive calls. We'll talk more about designing base cases and planning recursion in the next chapter.