Defining functions

Functions are a way to wrap up a bunch of code into one word. They make it possible to reuse the same code over and over again, without having to rewrite it. Pretty useful, especially for operations you want to do often. Functions also make it easier to think about your code, because once you know your function works, you never again have to worry about *how* it works. This enables you to break up problems into smaller, easier problems.

Syntax for defining functions

So how do we define a new function? Use the `def` keyword, short for "define". Then we'll have to specify a few things:

- The name of the function. This comes right after `def`.
- The inputs to the function. These come inside parentheses, right after the name of the function. For now we'll just talk about functions that don't have any inputs, so they do the same thing every time you use them.
- How the function works. This can get really complicated, so it would be unwise to try and cram it all onto the same line that has the `def` keyword, the function's name, and the inputs to the function. Instead we go to a new line and indent a little bit. There we write all the code that should happen when we use the function.
- The output of the function. This comes at the end. We specify the output using the keyword `return`, because it's what the function returns to us when we use it.

Let's see an example:

```
def fifth_fibonacci():
    fib_0 = 0
    fib_1 = 1
    fib_2 = fib_1 + fib_0
    fib_3 = fib_2 + fib_1
    fib_4 = fib_3 + fib_2
    return fib_4
```

number in the Fibonacci sequence, which we learned about at the end of the previous chapter. Then we have a pair of empty parentheses. If this function took an input then we would put the input inside the parentheses, but it doesn't have an input so the parentheses are empty. On the next few lines we see how the function works. It computes the fifth Fibonacci number, using basically the same calculation we did at the end of the previous chapter. Finally we see the `return` keyword, which specifies the output of the function. In particular this function's output is `fib_4`, which has been assigned to the value of the fifth number in the Fibonacci sequence.

Here's an important point. I'll mention it again later because it's so gosh darn essential, but pay attention anyways. After running the code above, we have *not* computed the fifth Fibonacci number. We have made a function, called `fifth_fibonacci`. If we use this function, then it will compute the fifth Fibonacci number. But if we never use this function, then the code inside it will never get executed and we will never compute the fifth Fibonacci number.

Also keep in mind that every function has an output. If you don't specify an output with the keyword `return`, then Python will make your function output `None` by default. For instance, these next three functions are all completely the same and they all output `None`:

```
def example_1():
    variable = 10
    return None

def example_2():
    variable = 10
    return

def example_3():
    variable = 10
```

Python also stops reading as soon as it sees the keyword `return`. The next two functions are also completely identical, as far as Python is concerned:

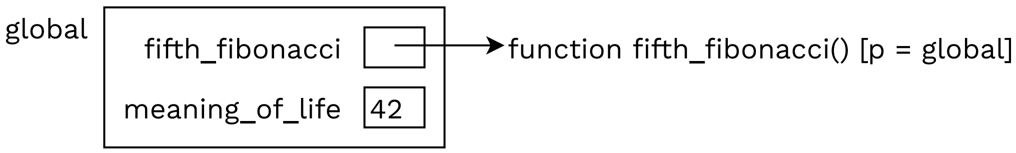
```
def example_1():
    return 10
    return 20
```

That's because it doesn't really make sense to keep reading `example_1`, after you already know it's output is the number 10.

Pointers

Recall from the previous chapter, I said variables can only be assigned to one of a few different primitive types: integers, floats, strings, booleans, or `None`. Well there's a problem with that. A function is not any of those primitive types! So how's it even possible for a variable to be assigned to a function? Short answer: It isn't. That's literally impossible. But we can do something very close.

Instead of binding the variable to a function, you can bind it to an arrow (a.k.a. pointer) that *points* at a function. This is a subtle difference but it will become very important later on. Take a look at this pyagram, for the `fifth_fibonacci` function we defined in the previous section. For comparison I've also included a variable called `meaning_of_life`, bound to 42.



First look at the variable called `meaning_of_life`. It's bound to 42, so the number 42 appears in the little box next to it. Now compare this against the variable named `fifth_fibonacci`. Notice how the little box does *not* contain a function! It contains the beginning of an arrow. So just like `meaning_of_life` is bound to 42, we can see `fifth_fibonacci` is bound to the beginning of an arrow. That arrow can point wherever it wants. In this example it points at a function, but it could just as feasibly point at a list, or a dog, or whatever else you like. Whenever you want to bind a variable to something that isn't a primitive type, you can just bind the variable to an arrow that points at the thing you want.

Once this makes sense, look at how we represent the function in the pyagram above. First we write "function", to specify that it is a function. Then we write the name of the function, along with the parentheses. If `fifth_fibonacci` had any inputs, then those would go inside the parentheses. We also write "[p = global]", which tells us the global frame is the *parent* of the function named `fifth_fibonacci`. Every function has a parent frame, and the parent frame is always the frame where the function was defined. In other words, it's the frame you were in when you saw the `def` statement. For the next few chapters, every function we work with will be

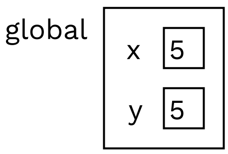
In summary, you can't bind a variable to a function. Instead you bind it to an arrow, also called a pointer, that points at the function. We'll use this same trick later on, to refer to lists and other things that aren't primitives. It may seem kind of nit-picky and irrelevant now, but this is going to be a big topic later on. Please check that you really understand this section, before you keep reading.

Working with pointers

Really, working with pointers is no different from working with any of the primitive types we learned about in the previous chapter. You just have to keep in mind that the variable is not bound to the function. It's bound to a pointer at the function. Let's see some examples.

Consider the code below, and the corresponding pyagram:

```
x = 5
y = x
```

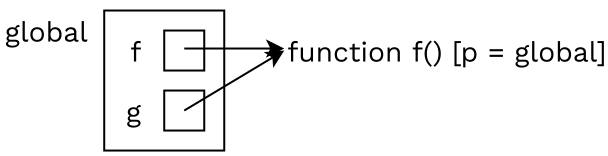


First we bind `x` to 5. Then we bind `y` to the value of `x`, which is 5. So we copy 5 from `x` into `y`, and both variables end up with the value 5. This should just be review, if you read the previous chapter.

Now consider this next example, and the pyagram that goes along with it:

```
def f():
    return 5


g = f
```



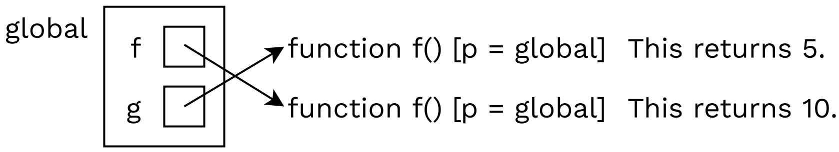
First we bind `f` to a pointer at a function. Then we bind `g` to the value of `f`, which is a pointer. So we copy the pointer from `f` into `g`, and both variables end up bound to

Be careful though! It can get tricky. Take a look at this example:

```
def f():
    return 5

g = f

def f():
    return 10
```



First we define `f`, and it gets bound to a pointer at a function whose output is 5. Then we see `g = f` and `g` becomes a pointer at the same function. Last, we reassign `f` to pointer at a function whose output is 10. `g` still points at the original function, but `f` points to the new one.

If you then evaluated the variable `g` in the Python interpreter, you'd see something like this:

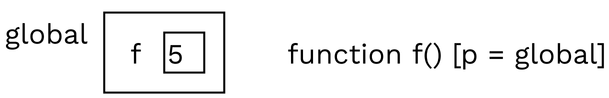
```
>>> g
<function f at 0x101e736a8>
```

This is just Python's way of telling you the variable `g` evaluates to a pointer at a function named `f`. The big scary hexadecimal number is telling you where to find that pointer in your computer, but you don't have to worry about that until you take a computer architecture course.

Here's another thing you should be aware of. A function can exist, even if there's no pointer to it. For instance:

```
def f():
    return 5

f = 5
```



previous value of `f` — the pointer — disappeared. We never got rid of the function we created. We can't use it anymore because there are no other variables pointing to it, but it's still there.

Calling functions

Now that we have talked about how to define functions, let's talk about actually using them. We refer to this as calling the function.

Syntax for calling functions

Consider our example `fifth_fibonacci` from earlier:

```
def fifth_fibonacci():
    fib_0 = 0
    fib_1 = 1
    fib_2 = fib_1 + fib_0
    fib_3 = fib_2 + fib_1
    fib_4 = fib_3 + fib_2
    return fib_4
```

`fifth_fibonacci` is just a function whose output is the fifth Fibonacci number. Writing `fifth_fibonacci` doesn't actually execute any of the code above. In order to do that, you need to write `fifth_fibonacci()` with the parentheses at the end. This tells you to go actually go through the code above and compute the fifth Fibonacci number. The difference is that `fifth_fibonacci` is a function, whereas `fifth_fibonacci()` is a call to that function. The function call returns a value, which you can use or assign to a variable. For instance, this code will evaluate the function call `fifth_fibonacci()` and bind the resulting integer to a variable called `fifth_fibonacci_number`:

```
fifth_fibonacci_number = fifth_fibonacci()
```

Now let's examine a slightly more nuanced example. Read this code, and keep in mind that dividing by 0 will cause an error to occur in your program:

```
def divide_by_zero():
    return 1 / 0
```

Remember, the code inside `divide_by_zero` doesn't happen until we call it. So we actually don't get an error until the very last line in the code above. The `def` is okay, because that just makes a function and binds the variable `divide_by_zero` to a pointer at that function. We haven't yet used the function, which means we haven't yet tried to divide by zero. It's also okay to say `function = divide_by_zero`, because that's just binding the variable `function` to a pointer. We only encounter an issue when we see `function_call = divide_by_zero()`, because calling the function causes us to go and execute the code inside it. At last, we attempt to divide by zero and this causes an error to occur in our program.

It's very important to recognize the difference between a function and a function call. When you're just talking about a function, you don't execute the code inside it. That only happens when you're calling the function. It can be easy to confuse the two, especially if you're just starting out, so be extra vigilant about this.

Functions with parameters

So far we've only seen functions that return the same thing every time you call them. Most of the time, though, you'll write functions that return something different depending on a few input variables called parameters. When you want your function to take input parameters, you write them inside the parentheses after the function's name. Then, within the function, you can refer to these parameters just like you would refer to any normal variable. For example, here's a function that takes one parameter called `number` and outputs its square:

```
def square(number):
    return number ** 2
```

Whenever you call a function that requires parameters, you need to provide a concrete value for each parameter. These values are called arguments.

```
>>> square(number)
Error: 'number' not defined
>>> square(5)
25
```

Technically the parameter is the variable itself, whereas the argument is the value given to the variable. So in the above example, `number` is a parameter and 5 is the folks refer to the signature of a function. That's the name of the function, including all its parameters — for instance `square(number)`.

Here's another example of calling a function that has parameters, this time taking the average of 3 numbers:

```
>>> def average(x, y, z):
...     return (x + y + z) / 3
...
>>> z = 8
>>> average(4, z, z - 2)
6.0
```

The order of evaluation

Before we go any further, we should take the time to understand how function calls actually work. Now is a good time to review the pyagrams we learned about in the previous chapter, because we're about to see them get a little bit more complicated.

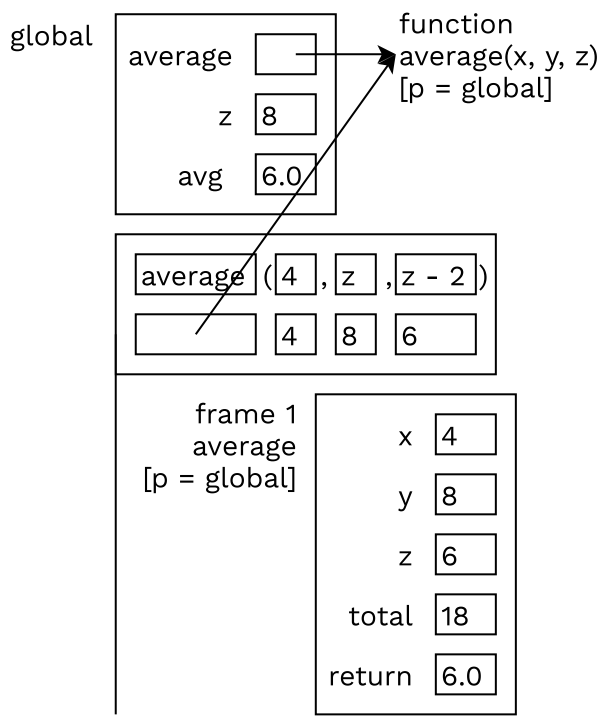
Imagine I asked you to compute $f(x)$. You would probably have a few questions. First of all, what does f do? And second, what's x? It's kind of impossible to tell me what $f(x)$ is, when you don't know either of these things. It's similar when Python sees a function call, like `average(4, z, z - 2)`. First of all Python needs to know what `average` does, and second it needs to process `4`, `z`, and `z - 2`. It's only possible for Python to compute `average(4, z, z - 2)` after it knows all those things. So when you see a function call, like `average(4, z, z - 2)`, two things happen:

1. Python evaluates everything in your expression. It evaluates `average` to find out what it does. Then it evaluates `4`, `z`, and `z - 2` because it's going to need to know what they are too.
2. Now that it knows all the constituent parts of your function call, Python has to actually perform the computation. Line by line, it executes the code inside `average` until it reaches the `return` statement.

Let's see how to do that whole procedure in a pyagram. We'll use this code as an example:

```
def average(x, y, z):
    total = x + y + z
    return total / 3
```

```
avg = average(4, z, z - 2)
```



At this point, we're done evaluating `average(4, z, z - 2)`, marking the end of both frame 1 and the flag. We can go back to the global frame, having discovered the function call evaluates to 6.0. Now we are able to finish the last line in the code above, by binding `avg` to its proper value.

This is hella important, so double-check that you thoroughly understand everything above. For future reference, here's the procedure. You should apply it whenever you come across a function call.

1. Make a flag underneath the current frame. Write the entire function call on the flag banner.
2. Evaluate the function and all its inputs.
    - First look up the function's name in the frame above the flag. If you don't find it there, search in the parent frame. Then the parent frame's parent frame, and so on, until you find it. If you don't even find it after searching the global frame, then Python will throw an error. Whatever you find it bound to, copy that value down into the banner, just beneath the function's name.

variables, then look up those variables using the same procedure as in the bullet point above. If any arguments involve function calls, then pause, start a new flag inside this one, and apply this entire procedure to evaluate that function call before you proceed.

3. Perform the computation.
    - Make a new frame inside the flag and give it a frame number. Then write the name and parent frame of the function being called. To get this information, you should simply have to follow the arrow that you copied down to the flag banner under the function's name.
    - Inside the frame, bind all the parameters to their respective arguments. For this step, you should only have to refer to the values you wrote in the flag banner.
    - Then walk through the code that gets executed, line by line. If you come across another function call, then apply this entire procedure to evaluate it.
    - Once you reach the `return` statement, make a corresponding `return` box at the end of the frame. In that box, write the output from the function call. This marks the end of both the frame and the flag.
    - Now that you know the output of the function call, return to where you were before and continue from there. This will be the frame above the flag that you just finished.

Use this procedure as a guide while you're just learning these concepts, but don't become reliant on it.

Also notice in this procedure that we evaluate all the arguments before looking at the code within the function. This is still the case, even if the function never uses its parameters! So, if the provided arguments cause an error then the function never gets executed at all.

```
>>> def five(x, y):
...     return 5
...
>>> five(3, 1 / 0)
Error
```

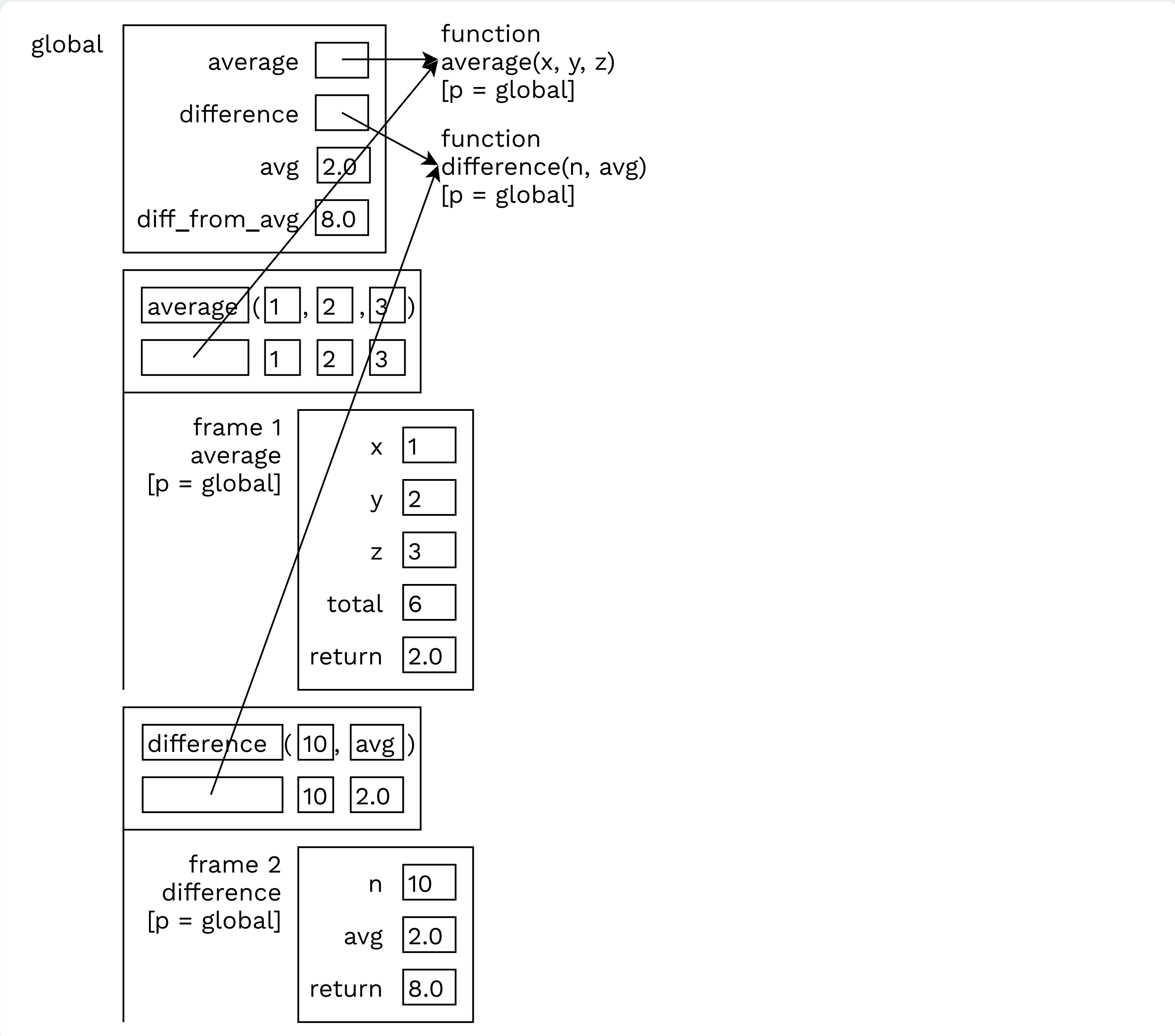
Again, this whole section is super duper important so make sure you really get it. We'll see some more complicated examples in the next section.

This stuff can get pretty tricky, so let's practice by drawing a few pyagrams. In particular, we'll look at three equivalent ways of calculating the difference between a number `n` and the average of three numbers `x`, `y`, and `z`. Even though all three methods produce the same result, they work differently and so they correspond to different pyagrams.

Here's our first method for calculating the difference between `n` and the average of `x`, `y`, and `z`. At the end, the result is bound to the variable `diff_from_avg`.

```
def difference(n, avg):
    return n - avg

avg = average(1, 2, 3)
diff_from_avg = difference(10, avg)
```

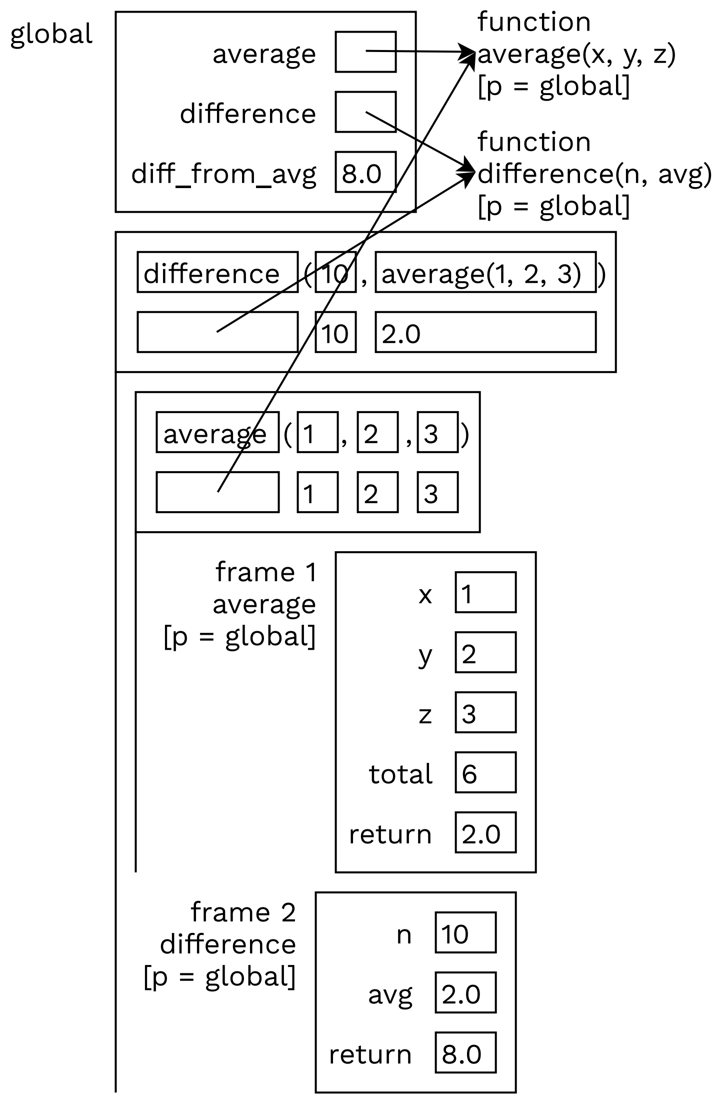


Having determined the value of the function call, we can go back to the global frame. There we bind the result to `diff_from_avg`. This completes the pyagram.

Here's another method for computing the difference between `n` and the average of `x`, `y`, and `z`. Instead of two distinct function calls, it uses a nested function call.

```
def difference(n, avg):
    return n - avg

diff_from_avg = difference(10, average(1, 2, 3))
```

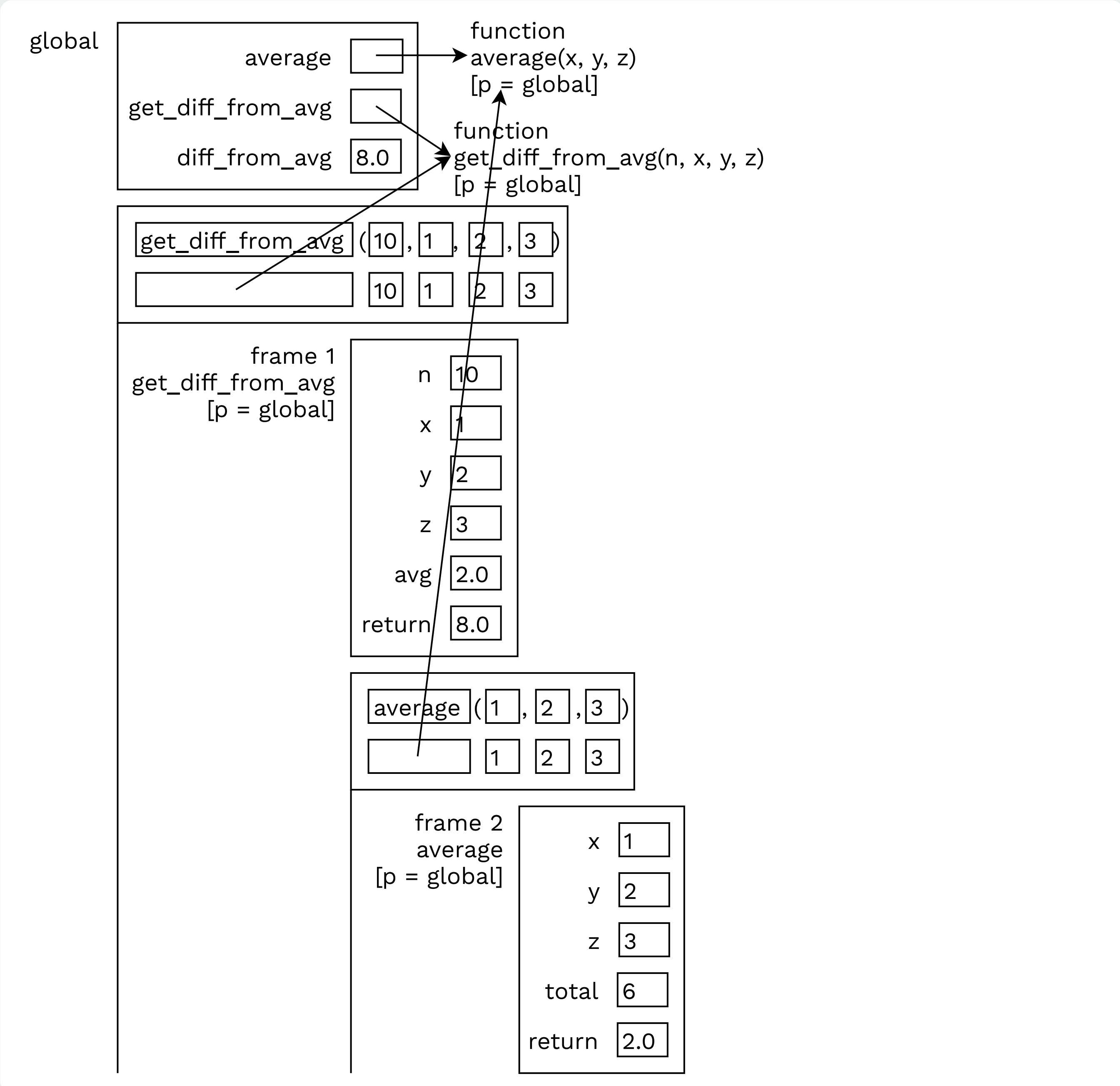


its appropriate value.

Here's our third and last method of computing the difference between `n` and the average of `x`, `y`, and `z`. It has a function call inside `get_difference_from_avg`.

```
def get_diff_from_avg(n, x, y, z):
    avg = average(x, y, z)
    return n - avg

diff_from_avg = get_diff_from_avg(10, 1, 2, 3)
```



Then we can go back to where we left off in the global frame. Having found out `get_diff_from_avg(10, 1, 2, 3)` is 8.0, we are able to finish binding `diff_from_avg`.

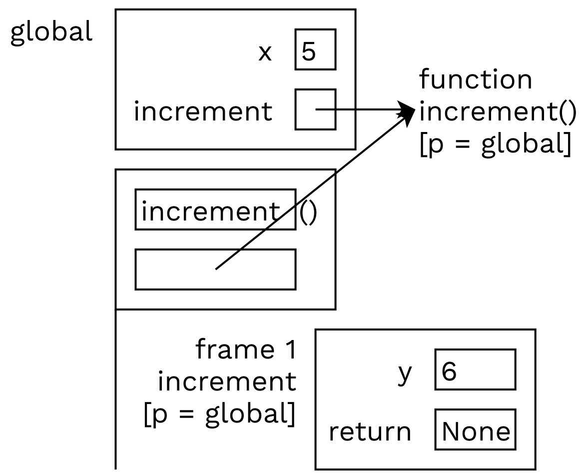
Make sure you're comfortable with these examples, before you keep reading.

The scope of a function

Let's open the Python interpreter and try out a short function that we'll name `increment`. If it works, it will assign a variable `y` to a value one more than `x`, whenever we call it.

```
>>> x = 5
>>> def increment():
...     y = x + 1
>>> increment()
>>> y
Error: 'y' is not defined
```

Looks like it doesn't work. To find out why, we should draw a pyagram. Refer back to the past few sections if necessary.



Since no `return` value is specified, we return `None` since that's the default.

Notice that `y` exists in frame 1, but not in the global frame. It makes sense, then, why Python gets confused when we ask for it in the global frame. This illustrates that

- When you're looking up a variable: First look up the variable in the current frame. If you don't find it there, search in the parent frame. Then the parent frame's parent frame, and so on, until you find it. If you don't even find it after searching the global frame, then Python will throw an error.
- When you're assiging a variable: You can only assign the variable in your current frame. You can't change or assign any variables outside your current frame.

Impure functions

We saw earlier that every function call has an output — a `return` value. Well some functions, called impure functions, also have side-effects. Side-effects are events that occur, not values to be evaluated. Whenever you call an impure function, it will perform its side-effect and then evaluate to its `return` value. For example, imagine a function called `launch_rocket` as described below:

- Side-effect: A rocket gets launched into space.
- `return` value: The string `'Launch successful.'`.

This is an example of an impure function because it has a side-effect. Namely, a rocket will launch into space whenever you call it. Then, after this is done, the function will output the string `'Launch successful.'`.

The `print` function

There's one particularly important impure function that the creators of Python have already implemented for you. It's the `print` function, and here's how it works:

- Side-effect: All its arguments are displayed on the screen.
- `return` value: `None`.

For instance, consider the code below. The initial call `print(4)` displays the number 4 on the screen. This function call also returns `None`, so `x` gets bound to `None`. Recall from the previous chapter, this means nothing gets displayed when we evaluate `x` in the Python interpreter. Then we have `print(4, 5, 6)` which displays all three of its arguments on the screen. Like the first expression, it evaluates to `None`. The returned `None` doesn't show up for the same reason that `x` didn't get displayed on the screen.

```
>>> x = print(4)
4
```

```
4 5 6
```

You could also do something wacky like this next piece of code. First of all, recall from earlier that we're going to evaluate all the arguments before actually performing the function call. (If you don't remember why, go back to the previous section and review the procedure for making pyagrams. Notice how we evaluate all the arguments in the flag banner, before we open the frame for the function call.) That means we'll evaluate 9 and the inner `print` statement before doing the outer `print` statement. The inner call to `print` causes the string `'To infinity and beyond!'` to get displayed on the screen. (It appears without quotes, because `print` always displays strings like that to make them easier to read.) Then it returns `None`, which means the outer call is basically saying `print(9, None)`. That's why `9 None` gets displayed on the screen. The outer `print` statement also returns `None`, but as in the previous example with `print(4, 5, 6)`, this doesn't get displayed.

```
>>> print(9, print('To infinity and beyond!'))
To infinity and beyond!
9 None
```