Functions as arguments

Higher-order functions are functions that either take in other functions for their input, or return other functions for their output. Let's take a closer look at when they're useful, and how to write them.

Operations on functions

So far we've written a lot of functions that take in numbers for their input. `square(x)` and `nth_prime(n)` are two good examples, which respectively take in the numbers `x` and `n` for their inputs.

There are also functions that are meant to take in functions for their input, rather than numbers. One example is the derivative function, from calculus. The derivative of a function, denoted mathematically like $\frac{d}{dx}f(x)$, takes the function $f(x)$ for its input and returns a different function that represents the rate of change in $f(x)$. You could also hypothesize a function like `is_quadratic`, which could take in a function $f(x)$ for its input and return `True` or `False` depending on whether or not $f(x)$ can be expressed in the form $ax^2 + bx + c$ for some constants a, b, and c.

When you're writing a function that takes in another function for its input, you have to be mindful that the way we represent functions in math is different from the way we represent functions in computer science. In math, f is more or less interchangeable with $f(x)$, and they both refer to a function. In computer science `f(x)` is a function call on the argument `x`. This function call evaluates to something like a number or a boolean or whatever. If you want to refer to the function itself, you have to write `f`. Consider `square(4)` for instance. It's a function call on the argument 4, and it evaluates to the number 16. The function itself is just `square`.

So if we wanted to program the derivative function, which requires a function for its input, then we would have to use something like `f` for the input rather than `f(x)`. Here's an example:

```
def derivative(f):
    """Return a function representing the derivative of the function f
    ...
```

Practice: the average difference between two functions

Suppose we have two functions, `f1` and `f2`, and we want to find out roughly how different they are from one another. To do that, we'll compare them on three inputs: `x`, `y`, and `z`. Our goal is to find the average difference between `f1` and `f2` on those three inputs. Define the function `average_difference(f1, f2, x, y, z)` to carry out this procedure. You may assume the function `average` has already been defined for you.

```
def average_difference(f1, f2, x, y, z):
    ...
```

First let's find out how different `f1` is from `f2`, on the input `x`. If you plotted `f1` and `f2` on a graph, then this would be the vertical distance between them at the number `x` on the x-axis. We might consider the calculation `f1(x) - f2(x)`. That's not quite right, though, because it could be negative if `f2(x)` is bigger than `f1(x)`, and we always want a nonnegative number when we're looking for the difference between two things. To solve this problem, we'll just use the built-in `abs` function to take the absolute value of `f1(x) - f2(x)` and make it positive.

```
def average_difference(f1, f2, x, y, z):
    x_diff = abs(f1(x) - f2(x))
    ...
```

Then we can use similar calculations to find out how different `f1` and `f2` are from one another, on the inputs `y` and `z`.

```
def average_difference(f1, f2, x, y, z):
    x_diff = abs(f1(x) - f2(x))
    y_diff = abs(f1(y) - f2(y))
    z_diff = abs(f1(z) - f2(z))
    ...
```

Now we know how different `f1` is from `f2` on the inputs `x`, `y`, and `z`. But we set out to calculate how different `f1` was from `f2` *on average*, so we should take the average of these three differences.

```
    y_diff = abs(f1(y) - f2(y))
    z_diff = abs(f1(z) - f2(z))
    return average(x_diff, y_diff, z_diff)
```

If `average_difference(f1, f2, x, y, z)` returns a very large number then `f1` and `f2` are very different, on average, when we compare them on the inputs `x`, `y`, and `z`. Meanwhile if `average_difference(f1, f2, x, y, z)` returns 0, then we know `f1(x) == f2(x)`, `f1(y) == f2(y)` and `f1(z) == f2(z)`.

Writing versatile programs

Now we've seen how it could be useful to have a function as an argument, when we're writing a procedure that's meant to operate on a function (or even multiple functions) rather than a number. You might also want to have a function as an argument when you're writing a versatile, general-use program. Let's talk about that some more.

Sometimes you will want a bunch of functions that are all variations on the same main idea. For example, you might want a function that computes the sum of the first n positive numbers, another that computes the sum of the first n squares, and a third that computes the sum of the first n cubes. These three functions are very similar, because they share the main idea of summing the first n terms in a sequence.

Though it may be tempting, it would be a very bad idea to write all three of these functions separately. Why? Let's see what it would look like:

```
def sum_numbers(n):
    total = 0
    while n > 0:
        total += n
        n -= 1
    return total

def sum_squares(n):
    total = 0
    while n > 0:
        total += square(n)
```

```
def sum_cubes(n):
    total = 0
    while n > 0:
        total += cube(n)
        n -= 1
    return total
```

In the end we have a lot more functions than we really need, and they all repeat basically identical logic. This makes our code super hard to maintain, for two reasons. First of all, it means that if we ever find an error we'll have to change a whole bunch of different functions. Second, it means that if we want to add another similar function, like `sum_fourth_powers` or `sum_fifth_powers`, then we would have to write even more functions and before you know it we have an entire library full of code that's all pretty much the same.

So to be clear, you should *never* write multiple functions that all share the same main idea. What's the alternative? Write one function that handles whatever your problems have in common. Then handle the rest with a function that you get as input. We might come up with something like this:

```
def sum_sequence(n):
    total = 0
    while n > 0:
        total += ____
        n -= 1
    return total
```

Notice how this template fits all three of the functions we wrote above. Depending on how we fill in the blank, it could become `sum_numbers`, `sum_squares`, or `sum_cubes`. We'll use a parameter to specify how we want to fill in the blank:

```
def sum_sequence(n, func):
    total = 0
    while n > 0:
        total += func(n)
        n -= 1
    return total
```

parameter `func`, then `sum_sequence` becomes identical to `sum_cubes`. In fact, now we can sum any sequence at all — not just the the first n positive numbers, squares, or cubes! Here's an example, assuming `square(n)`, `cube(n)`, `fourth_power(n)`, and `factorial(n)` are all defined:

```
>>> sum_sequence(3, square)       # 3**2 + 2**2 + 1**2
14
>>> sum_sequence(3, cube)         # 3**3 + 2**3 + 1**3
36
>>> sum_sequence(3, fourth_power) # 3**4 + 2**4 + 1**4
98
>>> sum_sequence(3, factorial)    # 3! + 2! + 1!
9
```

Practice: the order-2 composition of a function

Suppose we want a function that finds $(x^2)^2$, another function that finds $\exp(\exp(x))$, and a third function that finds $\frac{d}{dx}\frac{d}{dx}f(x)$. These three functions share the main idea of applying an operator twice to their input. This is called the order-2 composition of a function. Here's how we might go about solving these problems naively, assuming `square`, `exp`, and `derivative` are already defined:

```
def compose_square(x):
    return square(square(x))

def compose_exp(x):
    return exp(exp(x))

def compose_derivative(x):
    return derivative(derivative(x))
```

But this is quickly getting out of hand. Every time we want to get the order-2 composition of another function, we'll have to write a new version of `compose` for it. Like we saw earlier with the `sum_numbers`, `sum_squares`, and `sum_cubes` functions, we're just making more work for ourselves in the future. Let's try consolidating these three functions into just one.

For our first step, let's fill in `compose` with all the stuff in common between `compose_square`, `compose_exp`, and `compose_derivative`.

```
def compose(x):
    return ____(____(x))
```

Now we have a template that can fit `compose_square`, `compose_exp`, and `compose_derivative` depending on how we fill in the blank. To determine what needs to go in the blank, let's add another parameter. We'll call it `func`.

```
def compose(func, x):
    return func(func(x))
```

If we pass in the function `square` for `func`, then we get the same behaviour as `compose_square`. If we pass in the function `exp` for `func`, then we get the same behaviour as `compose_exp`. And if we pass in the function `derivative` for `func`, then we get the same behaviour as `compose_derivative`. In fact, we can find the order-2 composition of any function at all! We just have to call `compose`, passing in whatever function we want for the parameter `func`. This makes it a very versatile and useful function.

```
>>> compose(square, 2) # (2**2) ** 2
16
>>> compose(add_2, 7)  # (7 + 2) + 2
9
```

Practice: the Taylor series for e^x and $\sin(x)$

Warning: This section includes the answers to some of the practice problems from the chapter on iteration. If you intend to do those practice problems, and you haven't yet, then you may want to skip this section.

There are a lot of really weird and complicated functions in mathematics, like e^x and $\sin(x)$. Sometimes, rather than working with these functions directly, math folk prefer to approximate them with very big polynomials. (A polynomial is a function that can be written using powers of the same variable, multiplied by constants. Every polynomial can be written like $a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0 x^0$, for some choice of

In particular, this is the Taylor series for e^x: $\sum_{i=0}^{\infty} \frac{x^i}{i!} = \frac{x^0}{0!} + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots$

And this is the Taylor series for $\sin(x)$:
$\sum_{i=0}^{\infty}(-1)^i \frac{x^{2i+1}}{(2i+1)!} = \frac{x^1}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \cdots$

Since both of these Taylor series are infinitely long polynomials, you can't compute the whole thing for either of them. Instead, most math folk just take the first n terms in a Taylor series and that usually works good enough for whatever calculation they're doing. So pretend for a moment that we're cool math folk, and we've written the two functions below:

```
def taylor_approximate_exp(x, n):
    """Return our best approximation for exp(x), using the first n ter
    total, i = 0, 0
    while i < n:
        total += (x ** i) / factorial(i)
        i += 1
    return total

def taylor_approximate_sin(x, n):
    """Return our best approximation for sin(x), using the first n ter
    total, i = 0, 0
    while i < n:
        is_neg = i % 2 == 1
        ith_odd = 2 * i + 1
        total += is_neg * (x ** ith_odd) / factorial(ith_odd)
        i += 1
    return total
```

Let's try consolidating them into just one function.

```
def taylor_approximate(x, n):
    ...
```

We'll start, as always, by filling it in with the stuff in common between `taylor_approximate_exp` and `taylor_approximate_sin`.

```
    while i < n:
        total += ____
        i += 1
    return total
```

Now we have a template that can fit `taylor_approximate_exp` as well as `taylor_approximate_sin`, depending on how we fill in the blank. As with our previous examples, we'll add a parameter to determine what needs to go there. Since this parameter is a function that computes the ith term in the Taylor polynomial, we'll call it `ith_term`. Looking at our code above, we can see that the stuff in this blank depends on both `i` and `x`, so we should provide them both to `ith_term` function.

```
def taylor_approximate(ith_term, x, n):
    total, i = 0, 0
    while i < n:
        total += ith_term(i, x)
        i += 1
    return total
```

Depending on what `ith_term` does with `i` and `x`, this code could behave like `taylor_approximate_exp` or `taylor_approximate_sin`. Now we just need some functions to pass in for `ith_term`.

We'll start with the one that computes the ith term in the Taylor polynomial for e^x. Referring back to `taylor_approximate_exp` and `taylor_approximate_sin`, we should arrive at something like this:

```
def ith_exp_term(i, x):
    """Return the ith term in the Taylor polynomial for exp(x)."""
    return (x ** i) / factorial(i)

def ith_sin_term(i, x):
    """Return the ith term in the Taylor polynomial for sin(x)."""
    is_neg = i % 2 == 1
    ith_odd = 2 * i + 1
    return is_neg * (x ** ith_odd) / factorial(ith_odd)
```

Similarly, `taylor_approximate(ith_exp_term, 3, 10)` will compute our best approximation for $\exp(3)$ using the first 10 terms of the Taylor polynomial for $\exp(x)$.

```
>>> taylor_approximate(ith_sin_term, 3, 10) # Approximately sin(3).
4.938377459671376
>>> taylor_approximate(ith_exp_term, 3, 10) # Approximately exp(3).
20.063392857142855
```

And if we ever want to find the Taylor approximation for a new function, like $\cos(x)$ or $\ln(1+x)$, then we can still use our general-purpose `taylor_approximate` function. All we'd have to do is define a real short and simple function (like `ith_cos_term` or whatever) to get the ith term in the Taylor polynomial that we desire.

The main takeaway from all this? Don't write a bunch of functions that do basically the same stuff. Instead just make one, and delegate the details to a function you get in your input. Often times, if you want to make your code more versatile, that's the way to do it. Make sure you understand everything we've covered so far, before you keep reading.

Functions that make functions

One day you might need to write a program whose output has to be a function. Once again, consider writing the derivative function. Since the derivative of a function is a function, you'd need `derivative(f)` to return a function.

You may someday also want a function that's really hard or tedious to write, like something that computes the order-n composition of a function. Then, rather than writing the function you want, you might find it easier to write a function that produces the function you want.

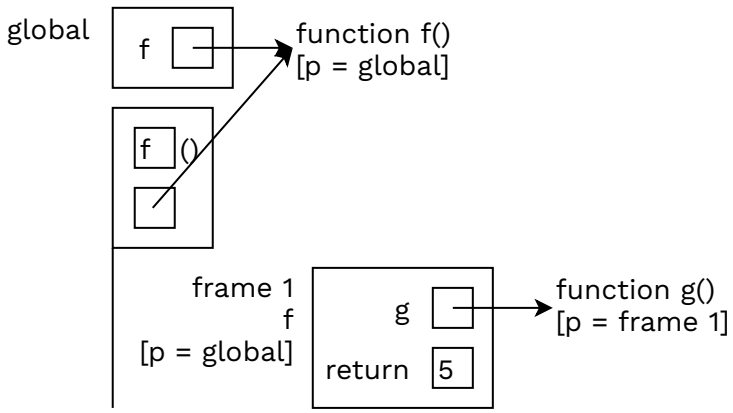
These are just two examples of times when you'll need to write a function whose job is first of all to create a new function, and second to return that function. We'll start by learning how these functions work.

Defining functions in functions

As you might guess, we can define a function within another function by using a `def` statement. For example, here's some code that defines a function `f` and then calls `f`.

try looking for `g` in the global frame, since `g` is only defined in frame 1.

```
>>> def f():
...     def g():
...         return 5
...     return 5
...
>>> f()
5
>>> g
Error: 'g' is not defined
```



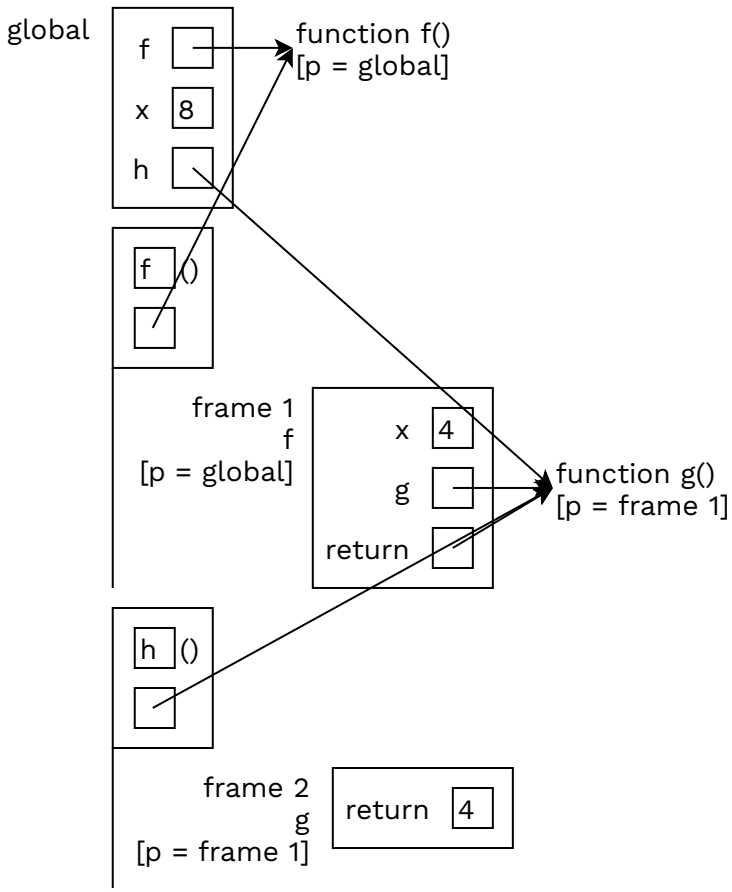
Recall from the chapter on functions, the parent frame of a function is the frame where it's defined. In the example above, for instance, notice how `f` is defined in the global frame so it's parent is global, but `g` is defined within frame 1 so it's parent is frame 1.

Parent frames are important because they tell you where to look up variables. So for example, if you need to know what `x` is but it's not in your current frame, then you look for it in the parent frame. If you still don't see it there, then look for `x` in your parent frame's parent frame. And if you don't see it there either, then you go to that one's parent frame, and so on until you reach the global frame. If you don't find `x` even after searching the global frame, then Python throws an error. To review the basics of pyagrams, parent frames, and variable lookup, refer back to the chapter on functions.

Now I could stop talking about parent frames right here, but I want to make sure you really understand how they work. In my experience a lot of students get the misconception that the parent of a function is the frame where it's called. **To be**

```
>>> def f():
...     x = 4
...     def g():
...         return x
...     return g
...
>>> x = 8
>>> h = f()
>>> h()
4
```

Let's draw the corresponding pyagram. Again, it may help to review the procedure for drawing pyagrams, from the chapter on functions.



According to the defintion of `g` in the code above, we need to return `x`. But there's no `x` in frame 2. We search for it in frame 2's parent, which is frame 1.

would've been wrong!

The takeaway is that the parent of a function is where it's defined, not where it's called.

How to write functions that make functions

Now that we know how Python handles functions that define and return other functions, let's discuss how to approach writing them. I've seen a lot of students try to write higher-order functions line by line, in order from start to end. These students usually end up kind of frustrated and confused. And frankly, that makes sense. You're not supposed to write code like that. Instead, you should approach it layer by layer.

What do I mean by that? Pretend you're writing a function that's meant to have another function defined inside it. For now, just focus on the outer function. While you write it, ignore the function that's supposed to be defined inside. (Feel free to write the `def` statement, but no more.) Just pretend the inner function is already done, and you're only being asked to write the outer function. Then, *after* you're done with that and you're confident it works, you can write the inner function.

The reasoning here is that you don't want to get caught up writing two functions at once. It can get very confusing, if you're constantly switching between writing the outer function and the inner function. Things are a lot easier if you just focus on one function at a time.

It might also help to consider how you want your function call to behave, when you're done. Use these questions to help guide you:

1. What should you get from calling your function? If it's supposed to return a function, then what does that one return when you call it?
2. What information does your outer function need? Where's it get that information from? Is it the parameters, or somewhere else?
3. What information does your inner function need? Where's it get that information from? Is it the parameters, or does it look for information in the outer function's frame? (Recall from earlier, how parent frames and variable lookup work.) Or does it perhaps get the information it needs from somewhere else?

Practice: the average of two functions

numbers `x`. You may assume the function `average` has already been defined for you.

```
def get_avg_func(f, g):
    ...
```

First of all, we know we want `get_avg_func(f, g)` to return a function. It sounds like a good idea for us to define a function that we can return! Let's do that. We'll call it `avg_func(x)`, and it'll return a value half way in between `f(x)` and `g(x)` for all numbers `x`.

```
def get_avg_func(f, g):
    def avg_func(x):
        ...
    ...
```

But remember, for now we're going to focus on writing `get_avg_func`. We'll only move on to `avg_func` after `get_avg_func` is complete. So, what else does `get_avg_func` need to do? According to the problem description, it's supposed to return the function whose output is half way in between `f(x)` and `g(x)` for all numbers `x`. In other words, it has to return `avg_func`. Let's add that to our code.

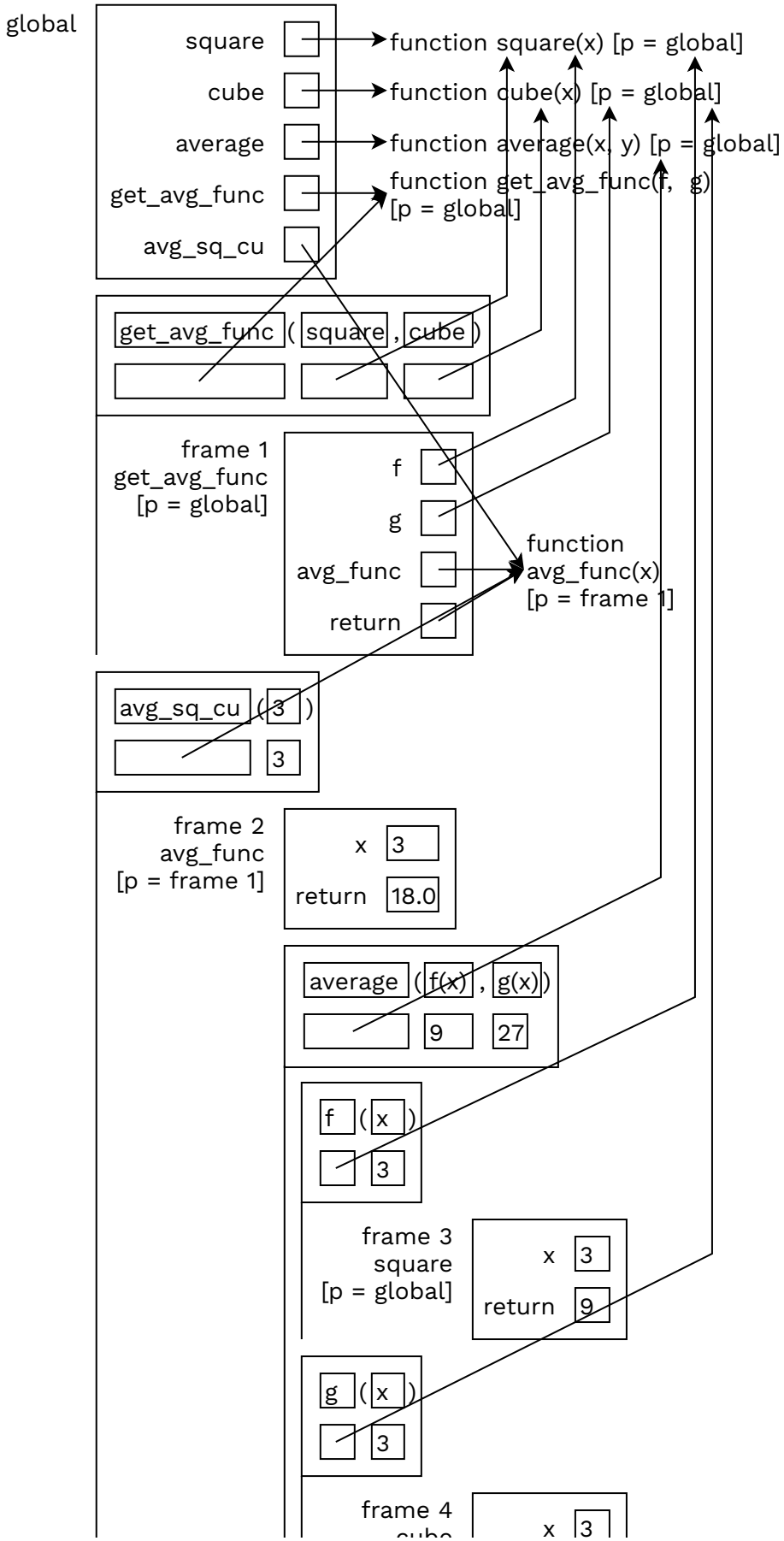
```
def get_avg_func(f, g):
    def avg_func(x):
        ...
    return avg_func
```

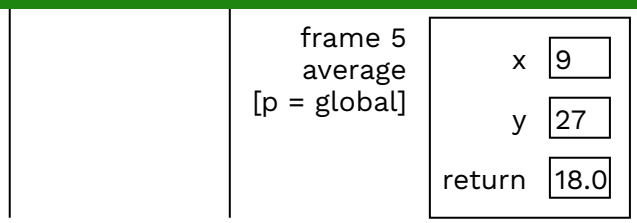
And now we're all done writing `get_avg_func`! All that's left is to write `avg_func`, so that it actually works like we want it to. For it to return a value half way in between `f(x)` and `g(x)`, we just need to find the average of those two values. Luckily we are allowed to use the `average` function, so let's just use that.

```
def get_avg_func(f, g):
    def avg_func(x):
        return average(f(x), g(x))
    return avg_func
```

Now try drawing a pyagram for the code below. Assume `average(x, y)` directly returns `(x + y) / 2`.

```
18.0
```


global
square
cube
average
get_avg_func
avg_sq_cu
function square(x) [p = global]
function cube(x) [p = global]
function average(x, y) [p = global]
function get_avg_func(f, g)
[p = global]
get_avg_func (square , cube)
frame 1
get_avg_func
[p = global]
f
g
avg_func
return
function
avg_func(x)
[p = frame 1]
avg_sq_cu (3)
3
frame 2
avg_func
[p = frame 1]
x 3
return 18.0
average (f(x) , g(x))
9
27
f (x)
3
frame 3
square
[p = global]
x 3
return 9
g (x)
3
frame 4
x 3



At long last we know `average(f(x), g(x))` is 18.0. We can fill in the `return` box for frame 2.

Notice how `avg_func` can still use `f` and `g`, since they're in its parent frame. In general, inner functions are able to use the variables and functions that are defined in the frame for their outer function. (In this case, the outer function is `get_avg_func`.) Just know that while it's possible to use the variables defined in your parent function, it's not allowed to change them. Python would throw an error if you tried.

If this pyagram was at all confusing, go back to the chapter on functions and review the step-by-step procedure that we came up with.

Practice: the derivative of a function

At long last! We've done a lot of talk about the derivative function, and now it's finally time to write it. Define `derivative(f)`, which should return a function representative of the rate of change in the function `f`. You may assume `f` has a single number for its input, and it's a smooth, continuous function whose domain is the real numbers. (In other words, you could draw it on a graph without lifting your pencil, and it has no sharp corners or weird asymptotes. Its input can be anything from $-\infty$ to ∞.)

```
def derivative(f):
    ...
```

First and foremost, we know that the derivative of a function is a function. So let's define a function for us to return, and then return that function. We'll call it `f_tag`, since mathematically the derivative of $f(x)$ is often written like $f'(x)$. Its input should be a single number (which we'll call `x`) since the derivative of a smooth, continuous function has the same domain as the function itself.

```
        ...
    return f_tag
```

As in the previous example, that's all there is to the outer function. Now that we're done writing `derivative`, we can move on to `f_tag`. This should be a function that takes in `x`, and returns the slope of the function `f` at the point `x` on the x-axis.

In order to get the slope of `f` at the point `x`, we'll pull a neat little trick that math folk sometimes use. See, in your high school or college calculus class you had to solve for the derivative of a function by hand. Well guess what? We can use computers to avoid all that grueling work. To get the slope of the `f` at the point `x`, just imagine drawing a very short line from the point $(x - \varepsilon, f(x - \varepsilon))$ to the point $(x + \varepsilon, f(x + \varepsilon))$, where ε is a really small number like 0.0001. Now our code looks something like this:

```
def derivative(f):
    def f_tag(x):
        x1, x2 = x - 0.0001, x + 0.0001
        y1, y2 = f(x1), f(x2)
        return slope(x1, x2, y1, y2)
    return f_tag
```

We just have to define a `slope` function and then we're done! But before we keep going, check that you understand everything so far. Once you're ready we'll move on to writing `slope(x1, x2, y1, y2)`, which should calculate the slope between a coordinate (x_1, y_1) and a coordinate (x_2, y_2).

Luckily, there's this formula from algebra that pretty much tells us the answer: $\frac{y_2 - y_1}{x_2 - x_1}$.

Here it is, in code:

```
def slope(x1, x2, y1, y2):
    y_diff = y2 - y1
    x_diff = x2 - x1
    return y_diff / x_diff
```

on some of the math from this section. We'll see plenty more practice problems that don't involve any calculus.

Practice: the order-n composition of a function

The order-n composition of a function is when we apply a function n times to its argument. For example, `f(f(f(f(x))))` is the order-4 composition of a function `f` on the argument `x`. Our goal is to define `compose(n, f)`. Its input is a number `n`, and a function `f` that takes in a number `x` for input. The output of `compose` is a function that takes in `x`, and returns the order-`n` composition of `f` on `x`.

```
def compose(n, f):
    ...
```

Like with the previous practice problems, we know `compose` should return a function. Let's define one for it to return, and then return it. We'll call it `f_composition` since it represents the composition of the function `f`. We know from the problem description above that it should take in `x` for input, so that will be its only parameter.

```
def compose(n, f):
    def f_composition(x):
        ...
    return f_composition
```

That's all `compose` has to do. We can move on to writing `f_composition(x)` now. Its goal is to apply `f` to the argument `x`, `n` times over, and then output the final result. The fact that we're repeating something `n` times should be a big hint to use a `while` loop. With that in mind, let's write a loop that happens `n` times. There are many ways to do this right, but we'll do it using a variable `i` to count up from 0 to `n`.

```
def compose(n, f):
    def f_composition(x):
        i = 0
        while i < n:
            ...
            i += 1
        ...
    return f_composition
```

and then call `f` on it one more time in each iteration of the loop. We'll keep track of our result in a variable called `result`, as we build it up. `result` will also be the final output from `f_composition`.

```
def compose(n, f):
    def f_composition(x):
        i, result = 0, x
        while i < n:
            ...
            i += 1
        return result
    return f_composition
```

Make sure you understand the code so far. All that's left is to is apply `f` to `result` one more time in each iteration of the loop. So in the first iteration of the loop, `result` should change from `x` to `f(x)`. In the second iteration of the loop, it should change from `f(x)` to `f(f(x))`. And so on, until in the last iteration it changes from the order-`n - 1` composition of `f` on `x`, to the order-`n` composition of `f` on `x`. We can achieve this behaviour by reassigning `result` to `f(result)` each time we go through the `while` loop.

```
def compose(n, f):
    def f_composition(x):
        i, result = 0, x
        while i < n:
            result = f(result)
            i += 1
        return result
    return f_composition
```

Now we can give it a spin. Here's an example, assuming `square` and `add_2` have both been defined:

```
>>> square_3x = compose(3, square)
>>> square_3x(4) # ((4**2)**2)**2
65536
>>> add_10 = compose(5, add_2)
>>> add_10(6) # ((((6+2)+2)+2)+2)+2
16
```