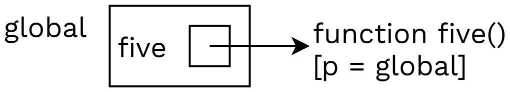
Defining `lambda` functions

Consider the code below. It does two things.

```
def five():
    return 5
```

First of all, it creates a new function whose output is 5. Second, it makes the variable `five` point at that function.



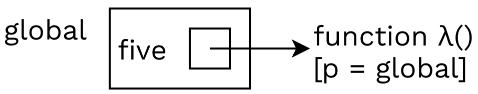
To be honest, though, that syntax is a little weird. If I want to bind `five` to the number 5, I write `five = 5`. If I want to bind it to the string `'five'`, I write `five = 'five'`. If I want to make `five` point to a function that returns 5, why can't I do something similar? Why not something like `five = <pointer at a function that returns 5>`? That's where the `lambda` keyword comes in.

Syntax for defining `lambda` functions

A `lambda` expression **directly evaluates to a pointer at a function**, in the same way that an arithmetic expression directly evaluates to a number. `lambda` expressions follow this format:

```
lambda PARAMETERS: RETURN VALUE
```

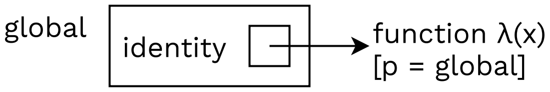
For example `lambda: 5` is a pointer at a function named "λ", which has no parameters and returns 5. You could bind the variable `five` to that pointer, by writing something like `five = lambda: 5`.



Now we have a variable that points at a function. We can call it in the usual way.

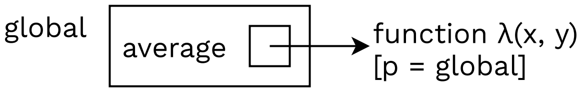
```
>>> five = lambda: 5
>>> five()
5
```

like `identity = lambda x: x`.




```
>>> identity = lambda x: x
>>> identity(4)
4
```

It's also possible to use the `lambda` keyword to make a function that takes in multiple arguments. For instance, consider `average = lambda x, y: (x + y) / 2`.




```
>>> average = lambda x, y: (x + y) / 2
>>> average(4, 8)
6.0
```

The takeaway? A `lambda` expression evaluates to a pointer at a function. So when we bind a variable to a `lambda` expression, that binds the variable to a pointer at a function. Then we can use the variable to call the function like we're used to.

`def` statements versus `lambda` expressions

So … what's the difference between a `def` statement and a `lambda` expression?

A `def` statement creates a function and binds it to a variable, whereas a `lambda` expression creates a function without binding it to a variable. It's up to you what to do with that `lambda` function — bind it to a variable, call it, whatever. In this way, `def` is just a shorthand way of making a function and binding it to a variable at the same time, without having to explicitly assign the variable using the `=` operator.

The most important difference is what you can actually do with them. Within a `def` statement you can assign local variables, evaluate chains of `if` / `elif` / `else` statements, and do iteration with `while` loops. A `lambda` expression is much more limited. You have your parameters before the colon, and your output after the colon. There's no room to do anything else, like variable assignment or boolean logic or iteration. Literally, a `lambda` expression lets you return a value, and do nothing else.

For example, these two snippets of code are basically the same:

```
add = lambda x, y: x + y
```

```
def add(x, y):
    return x + y
```

Make sure you understand everything we've covered so far in this chapter, before reading on.

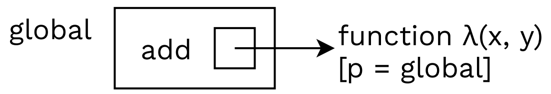
Pyagrams with `lambda` functions

In the previous section we saw a few examples of how to draw `lambda` functions in pyagrams. Now it's time to go into more detail.

Evaluating `lambda` expressions

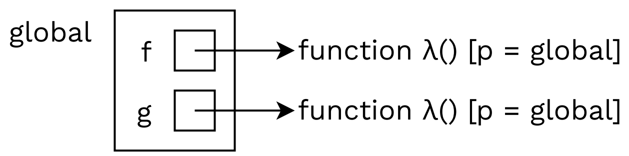
Remember, a `lambda` expression literally evaluates to a pointer at a function. As we saw earlier the function is named "λ", and as with any function in a pyagram, we write its inputs in parentheses after its name.

```
add = lambda x, y: x + y
```



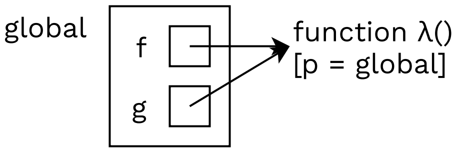
Each time you read the word `lambda`, it evaluates to a new pointer at a new `lambda` function, even if that results in having two identical `lamdba` functions. That's because each `lambda` expression is evaluated individually. This code, for instance, creates two separate `lambda` functions that both return 5.

```
f = lambda: 5
g = lambda: 5
```



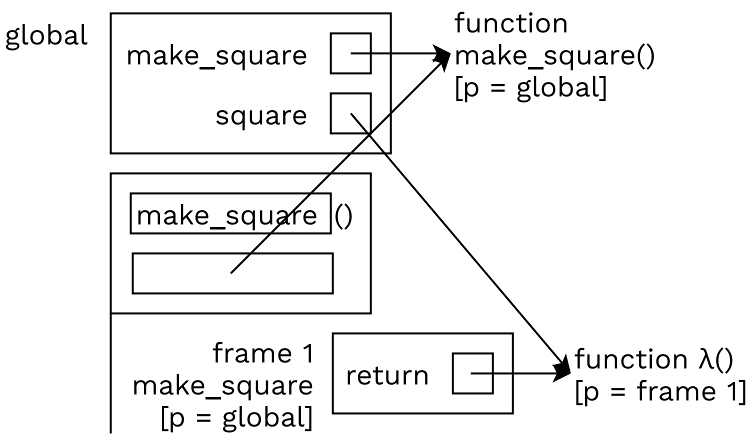
Meanwhile this code creates only one `lambda` function. Then it copies the pointer down from `f` into `g` like we learned how to do when we first learned about functions

```
f = lambda: 5
g = f
```



Also, a `lambda` function's parent frame is the frame you're in when you actually read the word `lambda`. (This is similar to how the parent of a normal function is the frame you're in when you read the word `def`.) In the code below, we evaluate the `lambda` expression in frame 1 so its parent is frame 1.

```
def make_square():
    return lambda x: x ** 2

square = make_square()
```



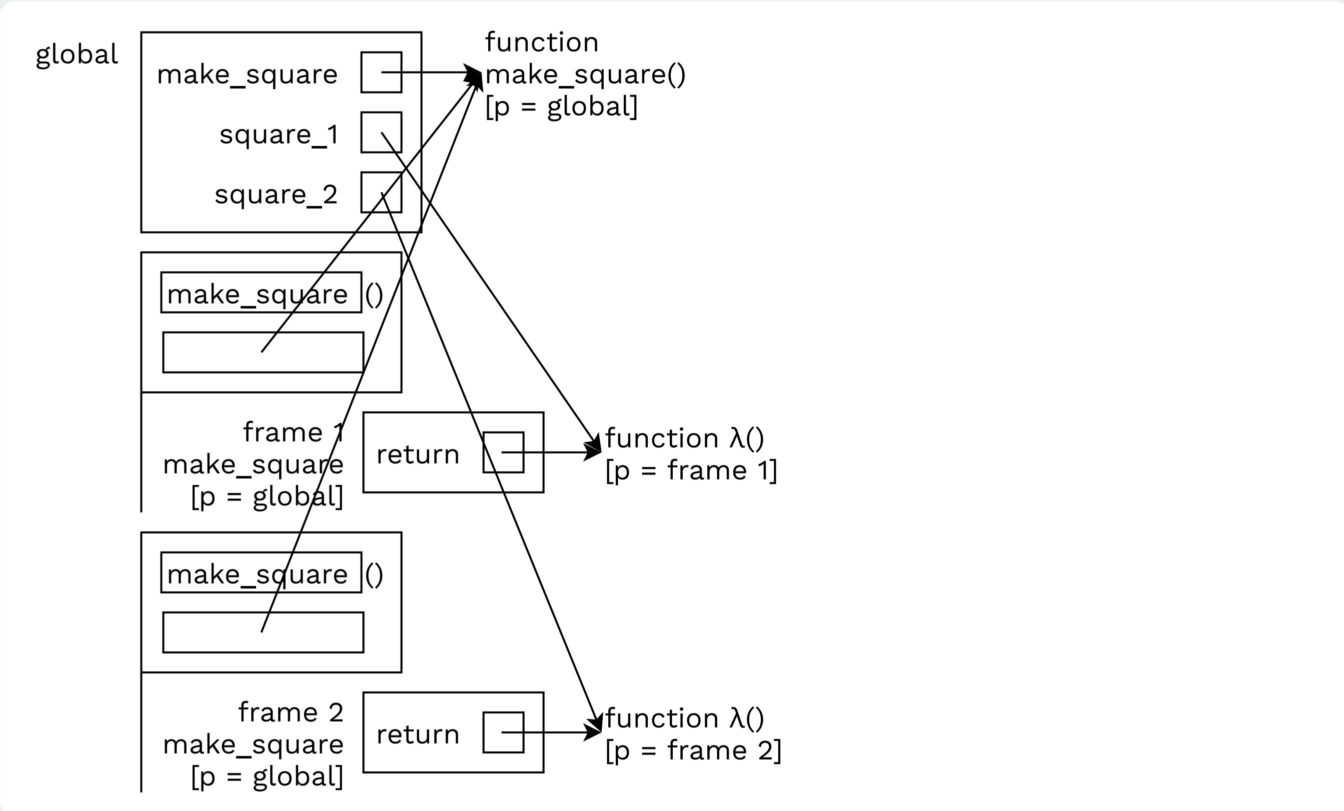
Last of all we go back to the global frame, where we can finish binding `square` to the value of `make_square()`.

So in summary:

- Whenever you read the word `lambda`, it evaluates to a new pointer at a new `lambda` function, even if that results in having two identical `lambda` functions.
- The parent of the `lambda` function is the frame where you read the word `lambda`.

Consider this modification of the example from the previous section:

```
def make_square():
    return lambda x: x ** 2

square_1 = make_square()
square_2 = make_square()
```



Finally that pointer is bound to `square_2` in the global frame. Notice how `square_1` and `square_2` end up pointing to different functions. That's because `square_1` got bound to the pointer returned from the function call `make_square()`, and `square_2` got bound to the pointer returned from a *different* function call `make_square()`. Each function call produced a new pointer and a new `lambda` function.

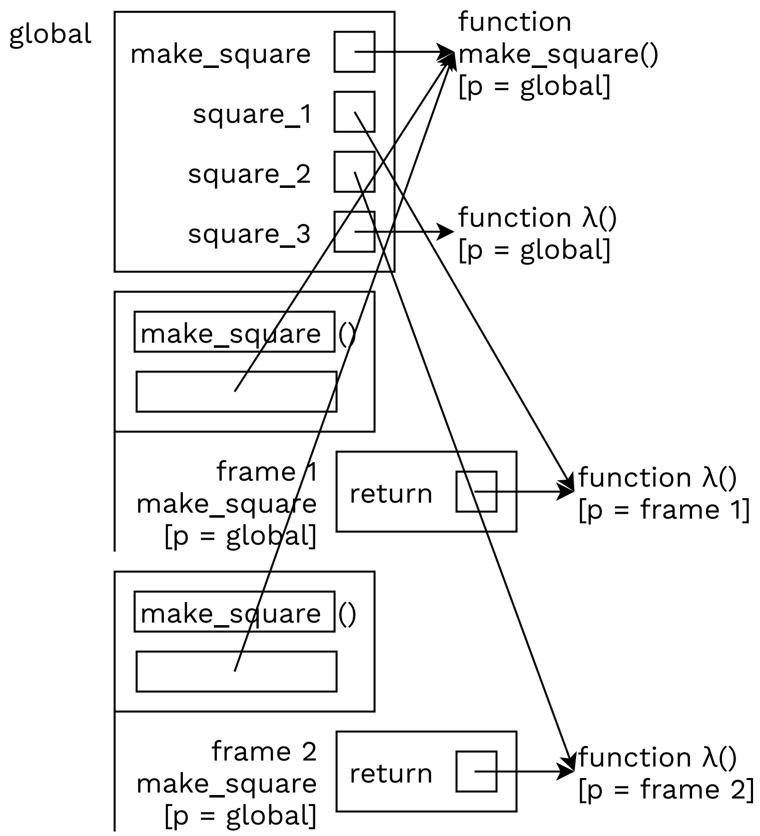
Check that you agree with this example, before you continue.

Telling apart different `lambda` functions

example from before:

```
>>> def make_square():
...     return lambda x: x * x
...
>>> square_1 = make_square()
>>> square_2 = make_square()
>>> square_3 = lambda x: 1 / 0
>>> square_2(4)
16
```

Here's the pyagram, right before the last line gets executed:

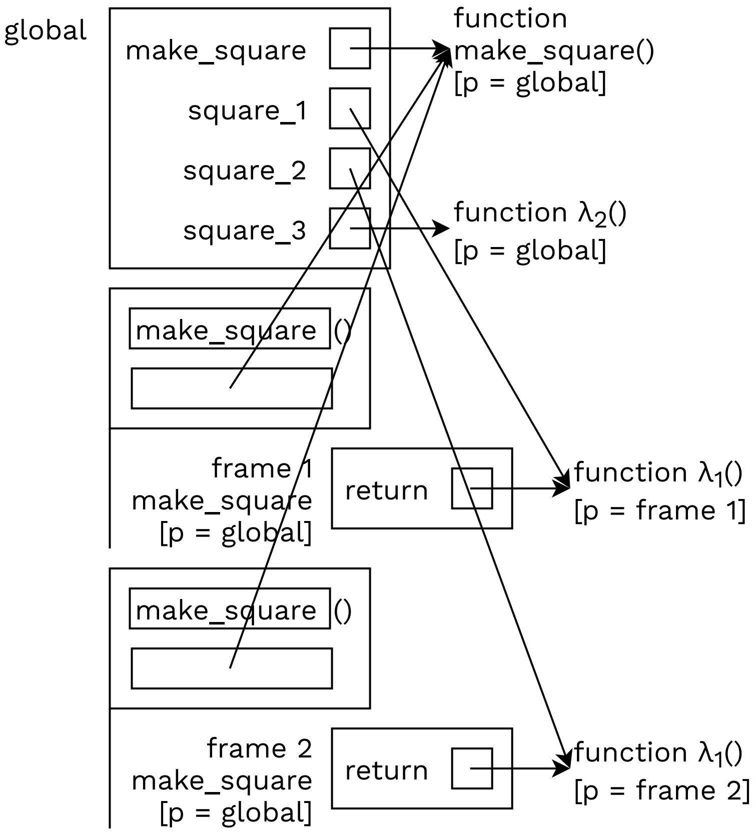


Now we're at the function call `square_2(4)`. But there's a problem. Which `lambda` function does `square_2` correspond to? Is it the one that squares its input, or the one that tries to divide by 0? The pyagram shows us `square_2` is bound to a function named "λ", but it doesn't show us *which* `lambda` expression to look at in the code.

To deal with this, we're going to write a little number next to every `lambda` expression in our code. This will help us tell them apart. Then, when we draw the

```
>>> def make_square():
...     return lambda₁ x: x * x
...
>>> square_1 = make_square()
>>> square_2 = make_square()
>>> square_3 = lambda₂ x: 1 / 0
>>> square_2(4)
16
```

Now when we draw `lambda`$_1$ in our pyagram we'll name the function "λ_1" rather than "λ", and when we draw `lambda`$_2$ in our pyagram we'll name the function "λ_2".



With this modification to the pyagram, it shows that `square_2` in the global frame corresponds to `lambda`$_1$ in our code. So when it's time to do the function call `square_2(4)`, it's easy to see we should do `4 * 4` rather than `1 / 0`.

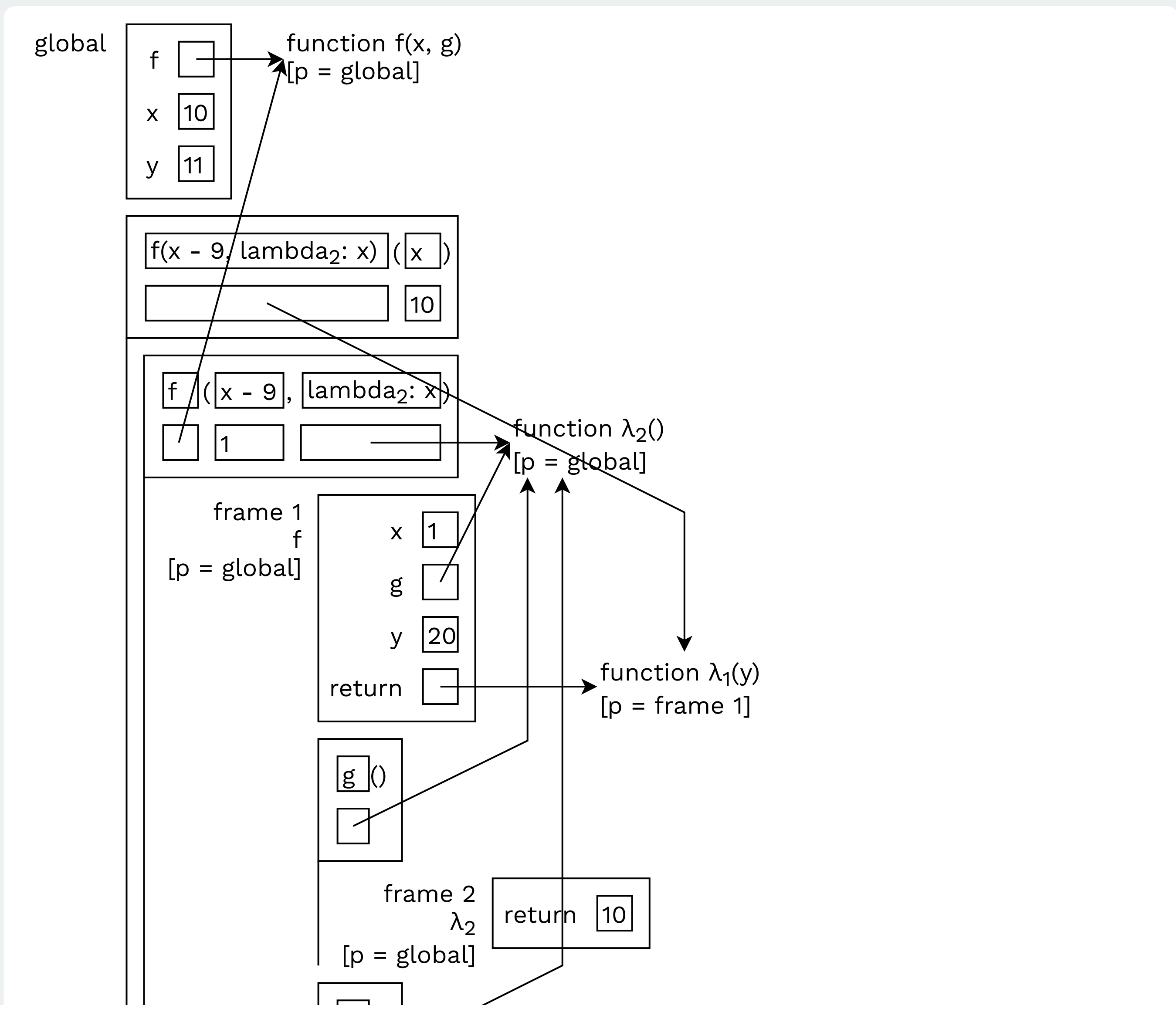
Practice: a `lambda` function as an argument

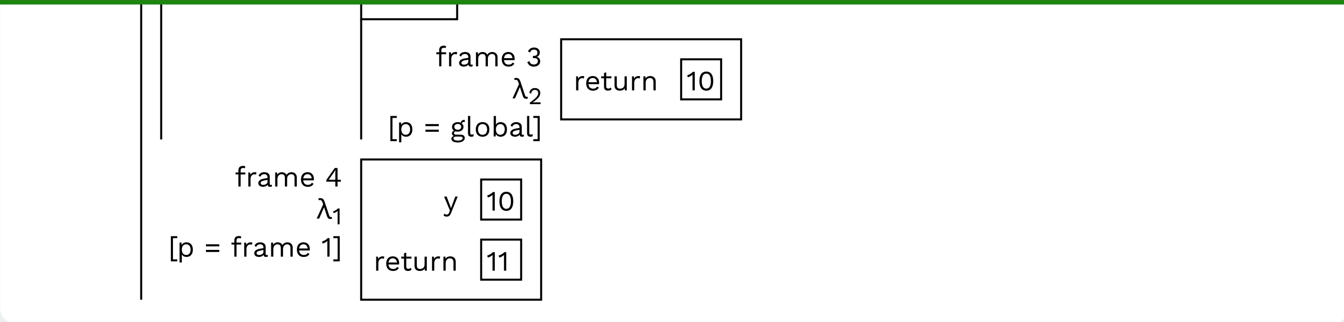
This may get confusing. If you find yourself lost, refer back to the procedure for drawing pyagrams. Draw the pyagram for this code:

```
    return lambda y: x + y

x = 10
y = f(x - 9, lambda: x)(x)
```

First let's number the `lambda` expressions so we can tell them apart later. Here's the code after we annotate it, like we just learned in the previous section:

```
def f(x, g):
    y = g() + g()
    return lambda_1 y: x + y

x = 10
y = f(x - 9, lambda_2: x)(x)
```





Now that the function call it complete, we can go back to the global frame where we left off on the line `y = f(x - 9, lambda`$_2$`: x)(x)`. Since we just learned that `f(x - 9, lambda`$_2$`: x)(x)` is 11, it looks `y` is getting bound to 11. This completes the pyagram.

Again, if this exercise was confusing you may want to review the procedure for drawing pyagrams. In fact, you may even want to skim or re-read some of the stuff in the sections above, just to check that everything jives with you. `lambda` functions can be a tricky topic, but they're something you should get familiar with.

Calling `lambda` functions

Now that we're familiar with defining `lambda` functions, let's talk about how to use them. Calling a `lambda` function is pretty much the same as calling any regular function, but there are a few subtle points that we should talk about explicitly.

Calling `lambda` functions on-the-spot

So far we've only been able to call a `lambda` function by first binding a variable to it. In the code below, for instance, we make `square` point to the function created by the `lambda` expression. Then we are able to call the function in the usual way by writing `square(4)`.

```
>>> square = lambda x: x * x
>>> square(4)
16
```

But we could also just plug in the value of `square` directly:

The only difference is that now we're creating the squaring function on-the-spot when we evaluate `(lambda x: x * x)`, rather than referring to it with the variable `square`. The important thing is that `(lambda x: x * x)(4)` is still a function call, since it has both a function and an argument in parentheses. (The function is `(lambda x: x * x)` and 4 is the argument in parentheses.)

Brief aside here, also notice how we had to use parentheses around the `lambda` expression. That's important. Without the parentheses you get `lambda x: x * x(4)`, which is a function that seems to take a parameter `x` and then multiply `x` with the result of a function call `x(4)`. Whenever you want to call a `lambda` function on-the-spot like this, you should use parentheses to denote exactly what is part of the `lambda` expression, and what isn't.

Here's another example, where we use a `lambda` expression to get the average of 4 and 8. In this case the function is `(lambda x, y: (x + y) / 2)`, and the arguments in parentheses are 4 and 8.

```
>>> (lambda x, y: (x + y) / 2)(4, 8)
6.0
```

Review: functions and function calls

Sometimes this stuff can get tricky. To avoid getting confused, you should get good at telling the difference betwen a function and a function call. Functions can either be bound to variables, or created on-the-spot by a `lambda` expression. When either of these things are followed by parentheses, that's a function call. For example:

```
>>> identity          # Function.
>>> identity(4)       # Function call.
>>> lambda x: x       # Function.
>>> (lambda x: x)(4) # Function call.
```

Also, be careful to evaluate things only when you should. Remember how you only do the stuff inside a `def` statement, once you call the function? The same goes for `lambda` expressions. You only do the stuff after the colon once you call the function. This is a pretty common mistake so be vigilant. Consider this code for example:

```
both_prime = lambda x, y: is_prime(x) and is_prime(y)
```

More Chapters