Intro

Let's talk some more about the best thing ever: variable assignment. By now we have had lots of practice with this, binding variables to numbers and strings and so on.

```
x = 5
```
← Binding a variable to some data. In this case, that data is a number.

```
x = 'robot ninja'
```
← Binding a variable to some data. In this case, that data is a string.

We have also used `def` statements to bind variables to functions (or more accurately, pointers at functions).

```
def f():
    return 5 or 'robot ninja'
```
← Binding a variable to some data. In this case, that data is a function.

The important takeaway is that functions are just another kind of data, like numbers or strings. It's not too strange to think about assigning variables to functions, in the same way we would assign them to primitive values. What if, instead of writing "`def f()`", we could just do something like "`f = <pointer at a function>`"? Well, we can! This is where `lambda` comes in.

Syntax

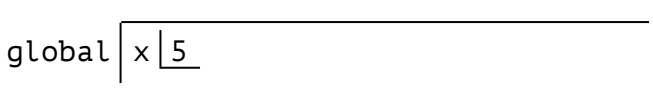
Recall from chapter 1 the 3-step procedure we use for variable assignment:
1. Do not look at the left side of the equals sign.
2. Evaluate the right side of the equals sign. Write the result somewhere with enough space.
3. Now look at the left side of the equals sign, and assign that value to what you just wrote.

Let's take a look at the first example above, as a quick refresher.

`x = 5`

1. Cover the left	2. Evaluate the right	3. Assign the variable
`[redacted] = 5`	5 evaluates to 5.	`x : 5`

This corresponds to the following environment diagram:

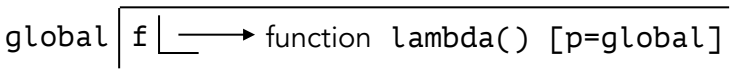


Now let's consider this next line. It's incomplete, but the important thing is that we're assigning `f` to a `lambda` statement. Just like we did with `x = 5`, we can apply the 3-step procedure from chapter 1.

`f = lambda ...`

1. Cover the left	2. Evaluate the right	3. Assign the variable
`[redacted] = lambda ...`	A `lambda` evaluates to `<pointer at a function>`.	`f : <pointer at a function>`

This corresponds to the following environment diagram:



The takeaway? `lambda` *literally* evaluates to a pointer at a function. So, when we set a variable equal to a `lambda`, that binds the variable to a pointer at a function. Make sure you understand everything so far, before moving on. Especially that underlined part. It's very important.

Now let's talk about how to specify what the `lambda` function does. It always follows this format:

`lambda` | `input_1, input_2, ... input_n` | : | Your `return` value, in terms of the inputs.

For example, consider `lambda x: 5`. This is a pointer at a function that takes one input, which is called `x`, and returns the integer `5`.

```
>>> f = lambda x: 5
>>> f
<function>
>>> f()
ERROR: missing one argument 'x'
>>> f(10)
5
>>> f(314159)
5
```

We can also define a `lambda` function that takes in multiple arguments.

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

And, taking this a bit further, we're even allowed to define a `lambda` function that doesn't take in any arguments.

```
>>> answer_to_life = lambda: 42
>>> answer_to_life
<function>
>>> answer_to_life()
42
```

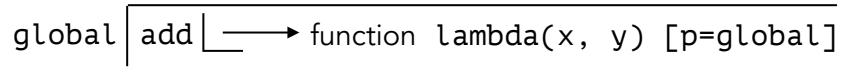
`def` and `lambda`

So ... what's the difference between a `def` statement and a `lambda`? A `def` statement actually creates a function and binds it to a variable, whereas a `lambda` creates a function without binding it to a variable. It is 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 between `def` and `lambda` is what you can actually do with them. Within a `def` statement, you can assign local variables, call all the other functions you want, and do iteration within `while` loops. A `lambda` is much more limited. You have your parameters before the colon, and you have your `return` value after the colon. There's no room to do anything else, like variable assignment or function calls or iteration. Literally, a `lambda` lets you return a value, and do nothing else.

That means you can also rewrite rewrite `lambda` functions as one-line `def` statements, if you so desire. This might help you get used to them, when you're just starting out. For example, the two snippets of code below are basically the same. (Technically, there is a very tiny difference though. The one on the left binds a variable called `add` to an addition function that's named `lambda(x, y)`. The one on the right binds a variable called `add` to an addition function that's named `add(x, y)`. This is just a quirk of the `def` statement — Python names the function the same thing as the variable it's bound to, just because the people who made Python thought that would be nice. This feature of the language will rarely be important for problems you see in this book.)

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

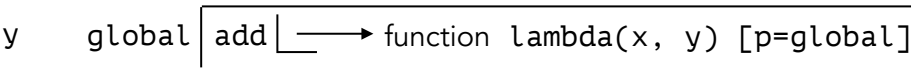



```
def add(x, y):
    return x + y
```

`lambda`s in Environment Diagrams

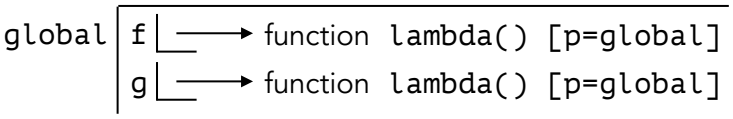
Now we're ready to have a look at how this all comes together in an environment diagram. Remember, `lambda` literally evaluates to a pointer at a function. The function is named `"lambda"`, and as with any function in an environment diagram, we write its inputs in parentheses after its name.

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



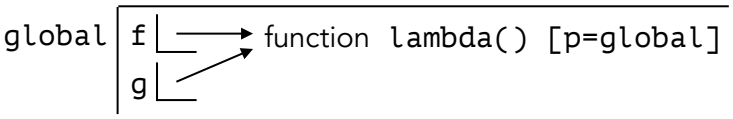
Each time you read the word `"lambda"`, that makes a new pointer at a function. If you want to refer to the same `lambda` multiple times, you have to refer to a variable that it's assigned to. That's because Python evaluates a `lambda` just like any other value — so even if it reads two identical `lambda`s, it will evaluate them individually. Check these next examples make sense:

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



Two different, but identical, `lambda`s.

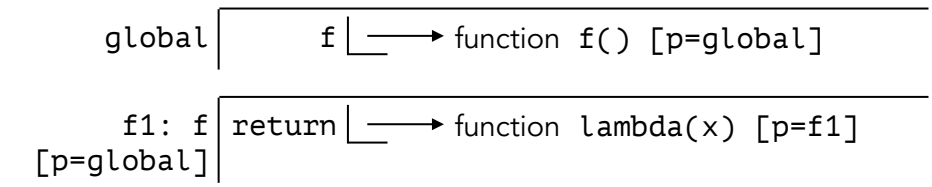
```
f = lambda: 5
g = f
```



One `lambda`.

The parent of a `lambda` is the frame you're in when you actually read the word `"lambda"`. In the example above, we read the word `"lambda"` in the `global` frame so its parent is `global`. In the example below, we read the word `"lambda"` in the frame `f1`, so its parent is `f1`. Recall from the chapter on Functions, this is because we don't execute the code in `f` until we call it.

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



If we call `f` for a second time in this example, then we run into a sort of weird situation. We end up reading the same `lambda` in both `f1` and `f2`, since both `f1` and `f2` correspond to the function `f`. No problem, we just have to apply the same rules as before:

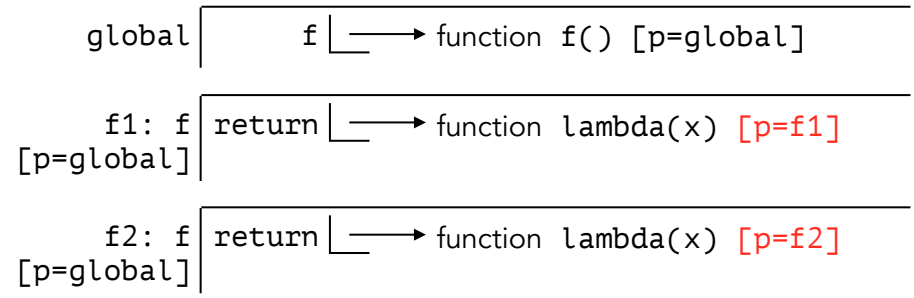
- When you read the word `"lambda"`, it's evaluated to be a new `lambda` function, even if that results in having two identical `lambda` functions.
- The parent of the new `lambda` is the frame where you read the word `"lambda"`.

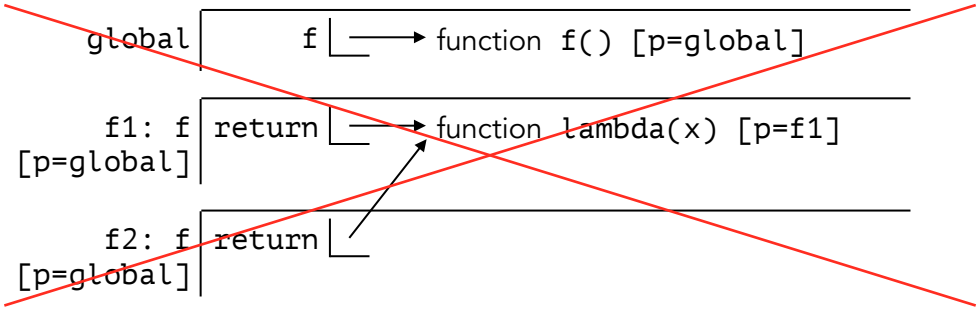
Continue to the next page to see how it would look in an environment diagram.

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

The function call for frame `f1`.

The function call for frame `f2`.

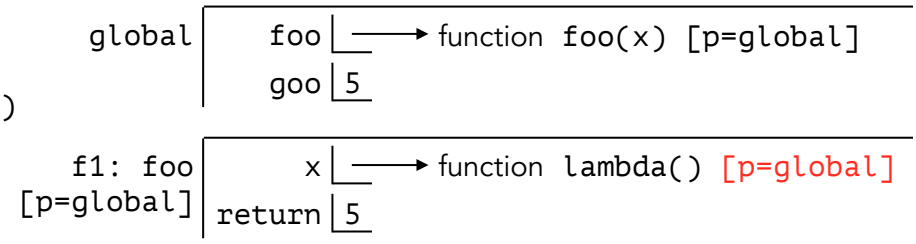




Only keep reading once you feel good with this material.

It's important to recognize the difference between the frame where the `lambda` is used, and the frame where the `lambda` is evaluated. Take a look at this next snippet of code.

```
def foo(x):
    return x()
goo = foo(lambda: 5)
```



The `lambda` is used inside frame `f1`, but it's actually written in the `global` frame. Since the `lambda` is not in `foo`, its parent is not `f1`. Instead its parent is `global`, since that is where it is written and evaluated. (Remember from earlier, where it's written is where it's evaluated.)