

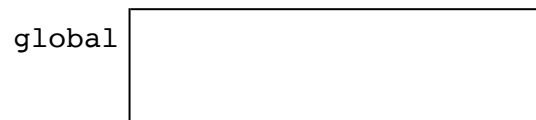
Intro

Today I have two cool things to share with you: environment diagrams, and functions. Let's get right to it.

Environment Diagrams

Environment diagrams are the single best tool I know for learning CS. (Yes, better than computers. Actually.) They are already worth a significant fraction of your grade, thanks to What Would Python Do and similar questions which appear consistently on exams. But even more importantly, environment diagrams are how you work your way through confusing bugs or edge cases you don't understand. So, how do they work? Basically, we pretend we're the computer, and we work through a piece of code one line at a time to see what it does.

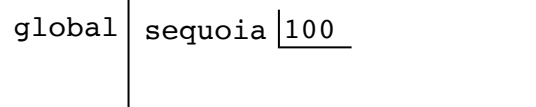
To the right, you'll see an empty environment diagram. The box is called a *frame*, which just means it's a place where code gets executed. The word "global" tells you this particular frame is the *global frame*. It's always the first thing you draw in an environment diagram.



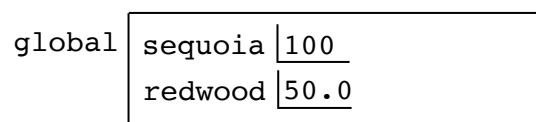
Whenever you make an environment diagram, you'll have a piece of code that you're analyzing. Let's use this code as an example:

```
sequoia = 99 + 1
redwood = 50.
sequoia * redwood
stick = int(sequoia // redwood)
stick *= "Tree "
```

In the first line we assign the variable `sequoia` to the integer 100. Here's how that gets expressed in our diagram.



Similarly, here's the diagram after we assign `redwood`. I added a 0 after the decimal point because Python actually evaluates `50.` to be `50.0`, to make it more obvious that it's a float and not an integer. This is a very minor detail.



Make sure you understand the diagram so far, before you move on to the next page.

Here's the code again, for reference:


```
sequoia = 99 + 1
redwood = 50.
sequoia * redwood
stick = int(sequoia // redwood)
stick *= "Tree "
```

global	sequoia	100
	redwood	50.0

The next line is `sequoia * redwood`. Since `redwood` is a float, our answer will also be a float, namely `5000.0`. You might notice this value doesn't get assigned to a variable. Python still calculates the number `5000.0`, but since we don't tell it to keep track of that value in a variable, Python will forget about it immediately. Our environment diagram looks the same.

Now let's move on to the fourth line. It's a bit too complicated to do mentally, so let's use the 3-step process for variable assignment from the previous chapter.

```
stick = int(sequoia // redwood)
```

1. Cover the left	2. Evaluate the right	3. Assign the variable
 = <code>int(sequoia // redwood)</code>	<code>sequoia // redwood</code> evaluates to <code>2.0</code> , and <code>int(2.0)</code> equals <code>2</code> .	<code>stick : 2</code>

Now our environment diagram looks like this:

global	sequoia	100
	redwood	50.0
	stick	2

We just have one more line to go. If you don't know how to approach it, recall that these two expressions are exactly the same:

```
stick *= "Tree "
```

```
stick = stick * "Tree "
```

Then you can apply the same 3-step process from before, to find that `stick` should be reassigned to the string `'Tree Tree '`. If you want to review why, refer to the previous chapter. Now we have finished the environment diagram, and it looks like this:

```
sequoia = 100
redwood = 50.
sequoia * redwood
stick = int(sequoia // redwood)
stick *= "Tree "
```

global	sequoia	100
	redwood	50.0
	stick	'Tree Tree '

Check you understand it fully, before moving on to the next page. We'll be using environment diagrams a lot in this class, so it's important that you get comfortable with them now.

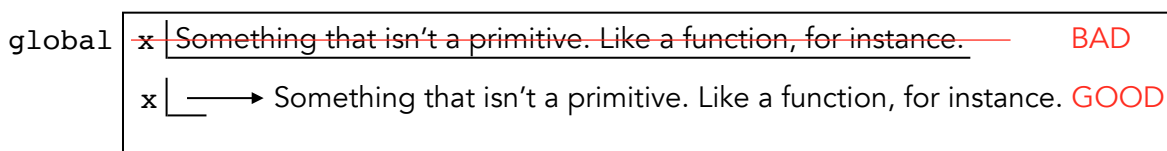
Defining Functions

Now let's get to the second topic of the day: functions. Functions are a way to wrap up a bunch of code into one word. There are 2 main advantages. Don't worry about memorizing these, just read them and understand why they make functions cool.

1. Functions let you reuse code, without having to rewrite it all over again.
2. After you know your function works, you never again have to worry about *how* it works. This lets you break up problems into smaller, easier problems.

Recall in the previous chapter, I said variables can only be assigned to one of a few different primitives: integers, floats, strings, booleans, or None. Well, there's a problem with that. What if I want a variable bound to a list, or a function, or a dog, or whatever else you might imagine? We can't do that. It's literally impossible. But we can do something very close.

Instead of binding the variable to a list, or function, or dog, or whatever — you can bind it to an arrow that *points* to a list, or function, or dog, or whatever. That's called a *pointer*. This is a subtle difference but it will become extremely important later in the class. It's good to make sure you have a solid grasp of it now.



So, how do we bind a variable to a *pointer* at a function? That's where the `def` keyword comes in, short for "define". We'll start off with a very small function, which I'll name `five`. Here's how it looks in code, and how we represent it in an environment diagram.

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

We'll talk about `return` in more detail soon.

```
global | five | → function five() [p=global]
```

This tells us the variable `five` is a pointer to a function.

This tells us the name of the function we're pointing to.

This tells us `five`'s parent frame is the global frame. We'll see what that means soon.

This is *super* important! `five` is just the variable assigned to the function we just made. If you actually want to *use* the function we made, you need to call it by putting parentheses afterwards.

```
>>> five
<function>
```

```
>>> five()
5
```

Make sure you understand everything so far, before you continue to the next page.

Returning From Functions

As you might have guessed from the previous page, the `return` keyword means that's what you get when you call the function.

```
>>> def example():
...     x = 2
...     return x * 'Tree '
>>> example()
'Tree Tree '

>>> def example():
...     'this does nothing'
...     return 70 % 3
>>> example()
1
```

If you don't write `return`, or if you write nothing after it, then Python assumes the function will return `None`. These three functions are the same:

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

def example():
    variable = 10
    return

def example():
    variable = 10
```

When Python sees the word `return`, it also stops reading your function. The next two functions are completely identical, as far as Python is concerned:

```
def example():
    return 10
    return 20
    LOL
    return 'happy'
    42

def example():
    return 10
```

Functions And Environment Diagrams

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.

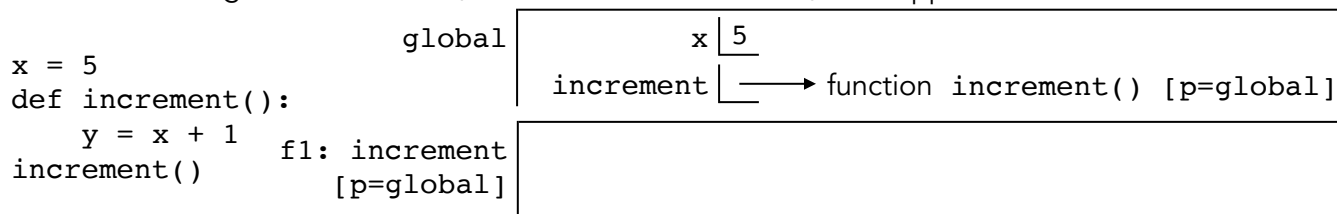
```
>>> x = 5
>>> def increment():
...     y = x + 1
>>> increment()
>>> y
Error: 'y' not defined
```

Looks like it doesn't work. To find out why, we need to draw an environment diagram. Here's what we have after just assigning `x` and `increment`. We haven't yet called `increment`.

```
x = 5
def increment():
    y = x + 1
```

```
global | x | 5 |
       | increment | → function increment() [p=global]
```

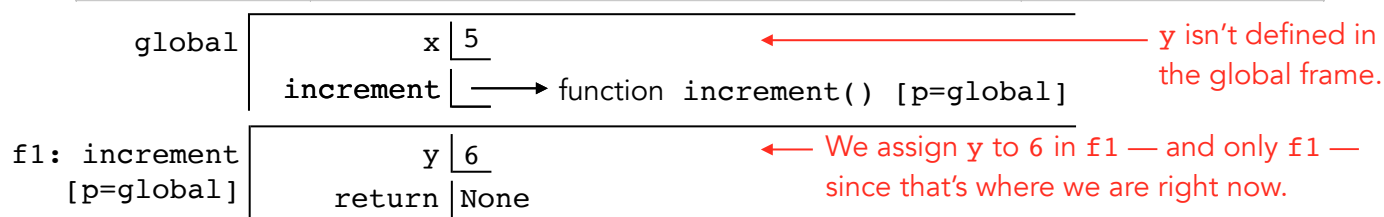
This is important: As soon as we call a function, we have to start a new frame in the environment diagram. That means, when we call `increment`, this happens:



`f1` indicates this is frame 1 — the first frame in our environment diagram other than the global frame. This frame will be used explicitly for the `increment` function. As soon as `increment` is done, we will automatically go back up to the global frame. Notice `f1`'s parent frame is the global frame, since `increment`'s parent frame is the global frame.

Inside the frame `f1`, we do everything that happens when we call `increment`. That's just the line `y = x + 1`. Let's apply the 3-step process for variable assignment from the previous chapter.

1. Cover the left	2. Evaluate the right	3. Assign the variable
<code>y = x + 1</code>	To evaluate <code>x+1</code> , we need to know <code>x</code> . First we look for it in <code>f1</code> , since that's where we are right now. Then , when we don't find it in <code>f1</code> , we look at <code>f1</code> 's parent frame, which is the global frame. There we find <code>x</code> equals 5, so <code>x+1</code> equals 6.	<code>y : 6</code> Note this doesn't affect the global frame, only <code>f1</code> since that's where we are.



Notice the box labelled `return` at the end of `f1`. You will always include this box at the end of every frame except the global frame. Since `increment` doesn't have a `return` statement, recall that it defaults to `None`. Now that `increment` is done, we move back to the global frame, where `y` still isn't defined. When we ask Python about `y`, it doesn't know what we mean.

Here are the three key takeaways from this example:

1. When you need to know the value of a variable, search your current frame. If you don't find it there, go to the parent frame. Then the parent's parent, and so on, until you find it. If you never find the variable, then Python will throw an error.
2. Always include a box labelled `return` at the end of any frame that isn't the global frame.
3. You can't change or assign variables outside your current frame.

This should make sense before you continue to the next page.

Reassignment

Now that we have talked about assigning variables to functions, let's talk about reassigning those variables. It works basically the same as reassigning any normal variable.

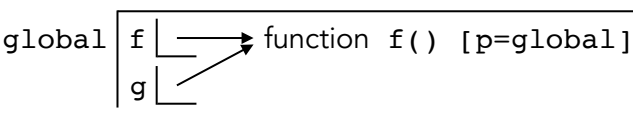
Look at the example below. When you change `y` to equal `x`, you just copy over the value of `x` into the box for `y`.

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

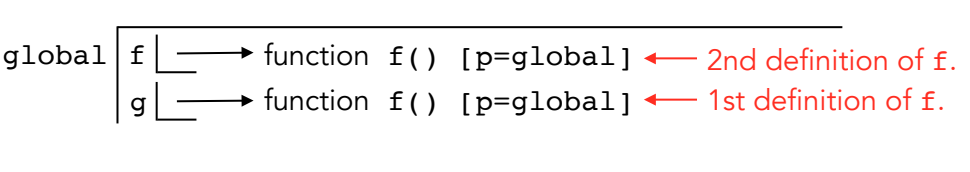


Now look at the next example. When you change `g` to equal `f`, you just copy over the value of `f` into the box for `g`. Since `f` is a pointer at a function, `g` becomes a pointer at the same function.

```
>>> def f():
...     return
>>> g = f
```



```
>>> def f():
...     return 5
>>> g = f
>>> def f():
...     return 10
```



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 variables you provide. Those variables are called parameters, or arguments, and they go inside the parentheses after the function's name. You can use these variables inside the function.

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

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

To call a function that has parameters, you need to give it specific values to use.

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

>>> z = 8
>>> average(4, z - 2, z)
6.0
```


Continue to the next page, and we'll see how to handle parameters in environment diagrams.

There are only two things to know about adding parameters to an environment diagram.

1. The function name includes its parameters.

Look at the code below, and the corresponding environment diagram. Notice, in the environment diagram, the function name includes its parameters. This is the same as what we've already seen, but now we have things to put inside the parentheses.

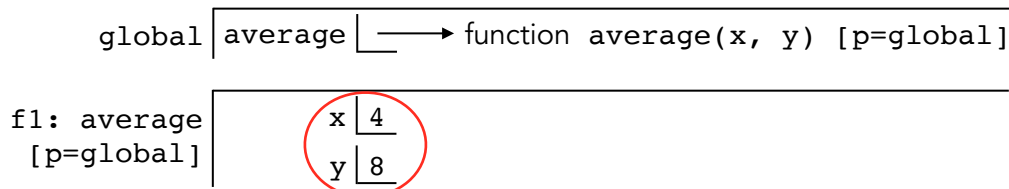
```
def average(x, y):  
    return (x+y)/2
```



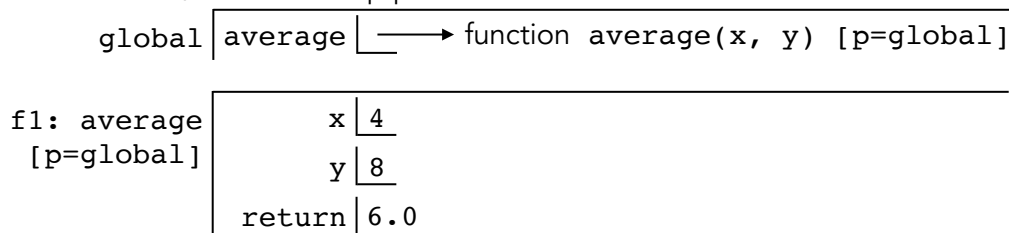
2. When you open the new frame, assign the parameters passed into the function call.

```
def average(x, y):  
    return (x+y)/2  
z = average(4, 8)
```

Try applying the 3-step process for variable assignment to the line with `z`. We need to know what `average(4, 8)` evaluates to. Like before, we open up a new frame for the call to `average`, but now there's an important difference! The new frame isn't empty. When the function was called, we were given `x=4` and `y=8`, so we must assign those variables in the new frame, as soon as it opens.

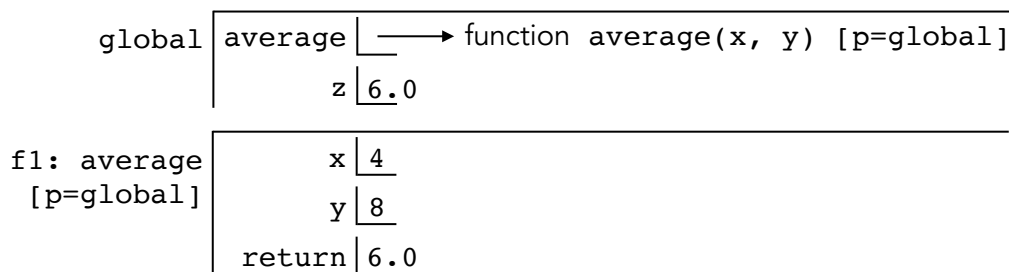


Then we continue normally, until we finish everything that happens during the call to `average`. If it's too much at once, use the 3-step process as before. Here's what we have once it finishes.



Lastly, we go back up to the global frame and finish assigning `z`.

1. Cover the left	2. Evaluate the right	3. Assign the variable
<code>z = average(4, 8)</code>	From our diagram, <code>average(4, 8)</code> is 6.0.	<code>z : 6.0</code>



Notice that in the example above, the function's parameters get evaluated and assigned in the frame, before we even look at anything the function does. This is always the case, even if the function never uses its parameters! So, if the parameter causes an error then the function never gets executed at all.

```
>>> def example(x, y):  
...     return 10  
>>> example(4, 1/0)  
Error
```

This can be a lot to absorb, but it's very important that you understand it. If it helps, here's a general procedure for doing a function call in an environment diagram. Use it as a guide while you're just learning these concepts, but don't become reliant on it:

1. Start a new frame for the function you're calling. The frame should have the same parent frame as the function.
2. In the new frame, evaluate and assign the parameters passed in to the function call.
3. Fill in the frame with everything that happens inside the function call.
4. At the end of the frame, make a box for the function's `return` value.
5. Now that the function call is done, go back up to the frame it was called from.

Impure Functions

We saw earlier that every function ever has an output — a `return` value. Well, some functions also have side-effects. When you call an impure function, remember two things:

- * The side effect happens. (Side effects are events that occur, not values to be evaluated.)
- * The actual function call evaluates to its `return` value.

For example, imagine an impure function called `explode_the_moon`, as described below:

- * side effect: The moon explodes.
- * `return` value: The string `"The moon has exploded"`.

```
>>> explode_the_moon()  
"The moon has exploded."
```

As of this moment, the moon has exploded.

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

- * side effect: All the parameters are displayed on the terminal screen.
- * `return` value: `None`.

```
>>> x = print(4)  
4  
>>> x  
None
```

```
>>> y = print(4, 5, 6)  
4 5 6  
>>> y  
None
```


Make sure you understand the examples on the previous page, before reading on. We're going to go over a few more little details that you should also know.

First of all, when a function call evaluates to `None`, Python doesn't display the `return` value. This is just something the creators of Python decided to do, to make things less cluttered.

```
>>> def f():
...     return None
>>> f()
>>>
```

```
>>> print(2)
2
>>>
```

Also, when you `print` a string, it gets displayed without quotes. This just makes it look pretty.

```
>>> print('61A')      >>> print(4, 'ever')    >>> print("Say 'Hi'")
61A                   4 ever              Say 'Hi'
```

That's all there is to know for `print`! But sometimes it can get tricky, so let's finish off with a few examples just to make sure you have all the little details right. Look at each of the examples below very carefully, and check that they all match what you expect. If something doesn't make sense, back up a little and find out why. `print` is an important function, and it will certainly appear on exams.

```
>>> def fav_numbers():
...     print(3.14)
...     print(2.718)
...     return 42
>>> fav_numbers()
3.14
2.718
42
>>> x = fav_numbers()
3.14
2.718
>>> x
42

>>> print(print(61), print("A"))
61
A
None None
```

```
>>> def printer(paper, ink):
...     x = print(paper)
...     print(ink)
...     return bool(x)
>>> printer(61, "A")
61
A
False

>>> x, y = str(print(1)), 5
1
>>> x + str(y)
'None5'
>>> print(1, 'cat', print(2))
2
1 cat None
>>> print(1 / 0)
Error
```

This was a long chapter, and a very important one. It lays the groundwork for most of the class, and covers a lot of material that will appear on the midterm. I won't ask this often, but please read it again later today, or perhaps tomorrow. I think you'll catch some stuff you missed the first time through, and you'll cement these ideas in your mind a lot better. At the very least, skim it to review before bed. And as always, happy coding!