# Fun With Functions



This reading is relatively new, and your feedback will help us improve it! If you notice mistakes (big or small), if you have questions, if anything is unclear, if there are things not covered here that you'd like to see covered, or if you have any other suggestions; please get in touch during office hours or open lab hours, or via e-mail at `6.101-help@mit.edu`.

## Table of Contents



## 1) Introduction

This week's reading focuses on **functions** and, in particular, on how functions fit into the environment model we started developing in last week's reading. We expect that functions are something you're all already familiar with, to some extent, from previous subjects; but we really think that functions are one of the most powerful tools we have for organizing and constructing complex programs. It's worth spending

the time to review some things about functions and to dig into the details and really understand how functions work under the hood so that we can use them effectively in our own programs.

## 2) The Power of Abstraction

One of our main goals in 6.101 is helping you develop new ways of managing the complexity of your programs. As we move to writing bigger and more complex programs (which may not fit all on one screen, and which may not fit nicely in your head all at once), we're going to need new tools and strategies to help us manage that complexity so that we can work with these kinds of programs effectively.

Although it may seem silly to say it, it's worth mentioning that reasoning about complicated systems can be... complicated. And a corollary is that reasoning about simpler systems is often simpler. As we move to thinking about (and designing!) bigger systems in terms of scope, scale, and complexity, it is important to have a way to think about breaking those complicated systems down into simpler pieces that are easier to understand.

To this end, one useful framework for thinking about complex systems involves thinking about:

- **Primitives**: What are the smallest and simplest building blocks of which the system is comprised?
- Means of **Combination**: How can we combine these building blocks together to build more complicated compound structures?
- Means of **Abstraction**: What mechanisms do we have for treating those complicated compound structures as building blocks themselves?

This third idea ("abstraction") is really critical, and it's the thing that allows us to keep things under control as we move to more complicated structures. The idea is that, once we have used the primitives and means of combination to build some new complicated structure, we can then give a name to that structure and treat it as though *it* were a primitive (including combining it with other pieces to make *even more complex* structures, which can then be given their own names and used to build even more complex structures, and so on...). The idea, then, is that we build up to big complicated systems layer by layer, each time designing, implementing, and testing new structures comprised of elements from the lower layers in such a way that each layer can be understood on its own. Then, when reasoning about the top-most (most complex) layer, we no longer need to think about it in terms of all of the underlying primitives, but merely in terms of the next layer down.

We can think about applying this framework to Python. In particular, considering *operations* that we can perform in Python:

- Python gives us several **primitive** operations, including arithmetic (`+`, `*`, etc.), comparisons (`==`, `!=`, etc.), Boolean operators (`and`, `or`, etc.), built-in functions (`abs`, `len`, etc.), and more.
- We also have ways to **combine** those pieces together, using things like conditionals (`if`, `elif`, `else`), looping structures (`for` and `while`), and function compostion (`f(g(x))`).
- But the *real* power comes when, after we've combined some pieces together to represent some new

operations, we can abstract away the details of those new operations and treat them as though they had been built-in to Python from the start. And one of our main means of **abstraction** is the ability to define functions using the `def` or `lambda` keywords.

Functions provide a means of representing the abstract notion of an arbitrary computation. We can specify the nature of that computation (what values we view as inputs, and how we combine those pieces together to produce the effects and/or outputs that we want), and then we can abstract away the details and give it a name, such that we can invoke that computation at a later time, or combine it with other functions to build up to bigger and more complex programs.

Let's take a look at an example from lab 1 (and 2!) involving our choice of representation for images. In particular, let's focus on the decision to store the pixel values as a 1-dimensional array. While there are some good reasons for this choice of representation, as you're working through the labs, you may have noticed that working with this representation directly is not always the easiest thing to do. Generally, when we think about modifying images, we want to think about locations in 2 dimensions (i.e., we want to think about modifying the brightness value at a 2-d $(row, col)$ location), and it takes a little bit of thought to figure out which index in the 1-d array of pixel values should be changed in order to make that kind of update.

But this is OK! One of the beautiful things about programming is that, if we are presented with a representation that we find difficult to work with, we can recognize that and create some functions that allow us to think about things using a more convenient representation. In this case, if we set up some functions that take care of some of the conversion between 2-d and 1-d representations, then we can use those things to allow us to think about our images using the more natural 2-d representation! For example, we could write a function like the following:

```
def flat_index(image, row, col):
    """
    Given an image and a (row, col) location, return the index associated
with
    that location in a 1-d list of pixel values stored in row-major
order.

    Parameters:
        * row (int): the row number we want to look up, with 0 being the
top-most
                     row
        * col (int): the column number we want to look up, with 0 being
the
                     left-most column


    Returns an integer representing the index into a row-major-order 1-d
list
    of pixel values that corresponds to the location (row, col)
```

```
    Raises an AssertionError if row or col are outside the bounds of the
given
    image
    """
    pass  # writing this code is left as an exercise to the reader!
```

We've not written out the body of this function for you (you are welcome to fill it in and use it in your lab if you would like!), but the beautiful thing about this function is that it allows us to think about our images in terms of 2-d structures; once we've written this function and we're sure it's working as expected, we no longer need to worry about the details of converting between 2-d and 1-d locations when thinking about higher-level operations we want to perform on images. In that sense, this function is more than just a way to help us avoid repetitious code by calling this function instead of writing out the calculation for this conversion each time we need it; it enables us to *think about later problems in a different, more natural way* by building on top of this new abstraction we've built for ourselves.

But, of course, this is not the only way we could have structured things to make our lives easier (another beautiful thing about programming is that there are always multiple ways to do things!). One example of a different approach we could have taken to the same problem (i.e., we want to be able to think about images as 2-dimensional structures, but the representation we're given is 1-dimensional) would be to define functions that convert back-and-forth between the given representation and some other, more natural representation. For example, some of you may have seen (in 6.100A or elsewhere) examples of representing 2-dimensional structures using lists of lists. If this representation feels more natural, we could have written functions like the following instead:

```
def image_to_2d_array(image):
    """
    Convert an image in 6.101's lab format to a 2-d array (list-of-lists)

    Parameters:
        * image (dict): a dictionary with keys 'height', 'width', and
'pixels',
                        in the format described in the lab 1 writeup

    Returns:
        A 2-dimensional array (list of lists) containing pixel values,
such
        that image_to_2d_array(image)[row][col] represents the pixel
value
        at a specific row and column
    """
    pass  # writing this code is left as an exercise to the reader!
```

```
def image_from_2d_array(array):
    """
    Convert a 2-d array of pixel values to an image in 6.101's lab
format.
    This function is the inverse of image_to_2d_array, such that, for any
image
    im, image_from_2d_array(image_to_2d_array(im)) should produce a new
image
    identical to im.

    Parameters:
        * array: A 2-dimensional array (list of lists) containing pixel
values,
        such that array[row][col] represents the pixel value at a
specific row and column

    Returns:
        An image in the 6.101's lab format (a dictionary containing
'height',
        'width', and 'pixels') equivalent to the input array.

    Raises an AssertionError if the rows in the input array are not all
the
    same length.
    """
    pass  # writing this code is left as an exercise to the reader!
```

With these functions in hand, our approach to the operations in the lab might have involved a few steps:

1. First, convert the image to a 2-d array using `image_to_2d_array`.
2. Then perform whatever operation we want using this new representation.
3. Finally, convert back to the lab's format using `image_from_2d_array`.

It's worth mentioning that these are not the only two ways we could have set up these abstractions, and there are pros and cons to each of these (as well as pros and cons to other ways we might think to set things up). But, the point as of right now is not to focus on the details of the specific functions written above but rather to demonstrate the usefulness of functions as a mechanism for building meaningful abstractions; and, as we saw in the example above, one way that these abstractions can help us out is, when we're working with data in a format that isn't particularly easy to work with, we can write ourselves some helper functions to make thinking about those data more natural and easier for ourselves.

# 3) Functions and the Environment Model

As we mentioned above, functions really are one of the most powerful tools we have at our disposal when designing programs. And, even better, all of the interesting and complex behavior that we get from functions all follows from a small set of rules governing functions. As such, the remainder of this reading will focus on those rules and how they fit into our environment model. And the really cool thing is that, if we understand those rules, then we can also understand all of the beautiful high-level behaviors associated with functions; and the more we understand about how functions work, the more able we'll be to make effective use of functions in our own programs.

And, again, the reason to be thinking about this model at all is that it really helps us as programmers if we don't have to resort to guess-and-check strategies for every decision we make in our programs. It helps to have a structured and robust way of thinking about what happens behind the scenes in response to code that we write.

So, throughout the semester, we'll keep expanding this model; and, for today, we're going to focus on how functions fit into our model. There are 2 distinct processes we're going to need to think about when introducing functions into our model: we will need to think about what happens when we *define* a function; and, separately, we'll need to think about what happens when we eventually *call* that function.

## 3.1) Function definition with `def`

The `def` keyword in Python does two things:

1. It creates a new function object in the heap. We're going to have a slightly simplified model as far as what we actually store inside of that object; but, in our model, this object contains:
    - The "formal parameters" of the function (i.e., the internal names by which we'll refer to the function's arguments).
    - The code in the body of the function.
    - A reference to the frame in which we were running when we encountered this `def` statement (we will refer to this as the "enclosing frame" of the function).
2. It associates that function object with a name in the frame in which we were running when we encountered the `def` statement.

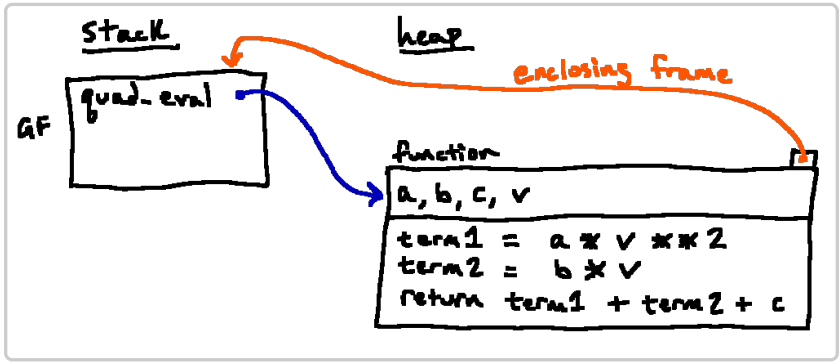
Let's take a look at how this works by looking at the diagram that results from running a piece of code. Here is a small function definition:

Show/Hide Line Numbers

```
def quad_eval(a, b, c, v):
    """
    Given coefficients a, b, and c representing a quadratic polynomial
    ax^2 + bx + c, as well as a value v, return the value of the
polynomial at
```

```
    the given point.
    """
    term1 = a * v ** 2
    term2 = b * v
    return term1 + term2 + c
```

This function may not be the epitome of the good style we're trying to encourage, but it will serve as an example by which we can demonstrate how functions behave in terms of our environment model. When we run this code and Python encounters the `def` statement here, it will follow the steps we outlined above, creating a new function object and associating that object with a name. We'll draw the resulting diagram like so:




Crucially, we don't run the code in the body of the function at this point; rather, we simply store away these pieces of information in an object, which we can later *call* to actually run the code. Inside of the function object, we can see the pieces we mentioned up above:

- The top region of the box contains the parameters of the function (i.e., we're calling them `a`, `b`, `c`, and `v`, respectively).
- The bottom region of the box contains the code representing the body of the function. Again, we're not running this code yet, just storing it away for later use.
- The little box in the top right corner shows a reference back to the frame that we were running in when we encountered the `def` statement (in this case, the global frame). Here, we've labeled this reference as "enclosing frame", but, once we get more comfortable with these diagrams, we may omit that label. Note also that this arrow is pointing to the global frame itself, not to any of the individual names or references in the global frame.

Note that we also associated this object with the name `quad_eval` in the global frame. It's also worth noting that the resulting function object above is what we call a "*first-class*" object. That is to say, this object is treated similarly to how we've treated other objects. The variable binding that we made via `def` is the same exact kind of reference we saw in last week's reading. If we look up the name `quad_eval`, we do so by following the arrow and finding the function object on the heap. If we want to bind this object to a different variable, or to use it as an element in a list, or to pass it as an argument to a different function, we're able to do that (and we'll see some examples of how that might be useful a little bit later on in this reading)!

### 3.2) Calling a Function

Function objects exist as abstract representations of particular computational processes, but they are really only useful when we *call* them (some people will call this "applying" or "invoking" the function; and, syntactically, this is accomplished by putting round brackets after some expression that evaluates to a function object). Calling a function is how we actually perform the computation represented by that function; and, when we call a function, Python performs the following four steps:

1. It evaluates the function to be called, followed by its arguments (in order), in the frame from which the call is being made.
2. It then creates a new frame for the function call, which will be used to store variables local to the function call. This new frame has a "parent pointer" back to the function's enclosing frame.
3. It then binds the names of the formal parameters of the function to the arguments that were passed in, inside of the new frame we just created.
4. And, finally, it executes the body of the function in this new frame. Importantly, during this step, if Python tries to evaluate a name that isn't bound in that frame, it will look in that frame's "parent" frame (the enclosing frame of the function we're calling).

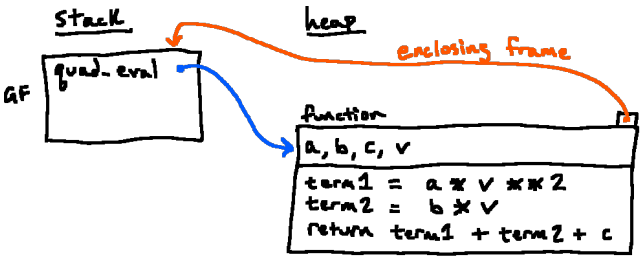
Let's see how this plays out using our example from above, when running the following code (which repeats the definition from above but also demonstrates a function call):

Show/Hide Line Numbers

```
def quad_eval(a, b, c, v):
    term1 = a * v ** 2
    term2 = b * v
    return term1 + term2 + c

n = quad_eval(7, 8, 9, 3)
print(n)
```

Use the buttons below to navigate through the process of drawing out the environment diagram for this program. We encourage you to follow along and draw the diagram for yourself alongside us as you're stepping through below!

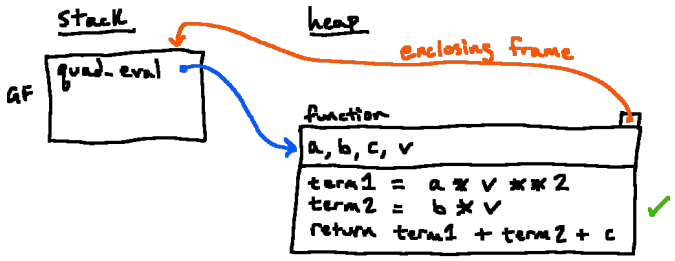
<< First Step | < Previous Step | Next Step > | Last Step >>



**STEP 1**

Here is the state of our diagram after executing the function definition on lines 1-4. Our next step will be to execute line 6: `n = quad_eval(7, 8, 9, 3)`. This is an assignment expression, so, as we saw last week, we start by evaluating the expression on the right-hand side of the equals sign. In this case, that expression is `quad_eval(7, 8, 9, 3)`, which looks like a function call (note the parentheses). So we'll follow our steps from above. The very first thing we do is evaluate the function that we're going to call by evaluating the expression to the left of the round brackets in the function call. In this case, that expression is `quad_eval`. We evaluate this name by following the associated arrow, where we find the function object we made earlier. We'll mark the function object with a small green checkmark on the next step, just so that we can keep track of it.

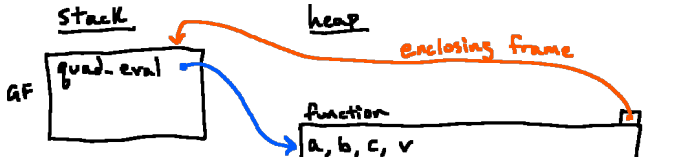
---

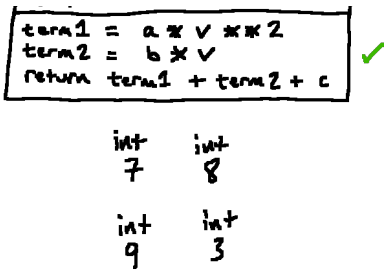



**STEP 2**

Now that we know what function we're calling, we proceed with evaluating the arguments to the function, one after the other. We'll show the result of that on the next step, but think ahead: what new objects (and how many) are going to be created when we evaluate the arguments?

---



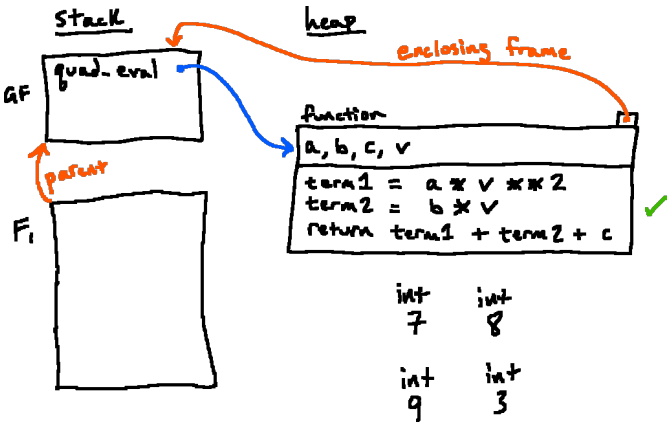



**STEP 3**

Here we see the result of evaluating the arguments: four new integer objects, representing `7`, `8`, `9`, and `3`, respectively. At this point, we know what function object we're calling and what arguments we're passing in, so now we can actually start the process of calling the function. So we'll carry on.

The next step is to create a new frame to keep track of the local variables for this function call, and to set its parent pointer appropriately. We'll see the result of this on the next step.

---

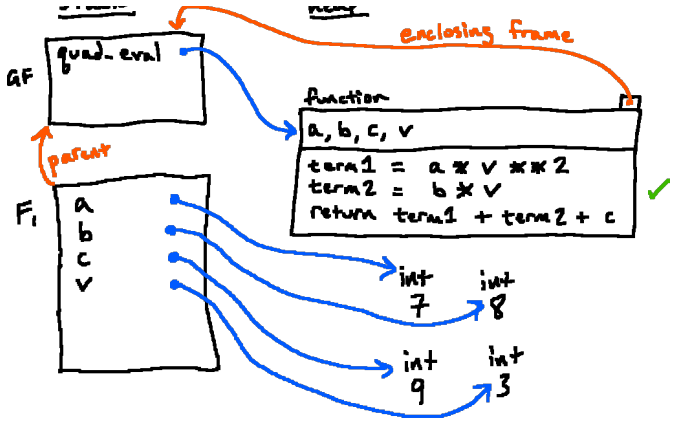



**STEP 4**

Here we see our new frame. We've given this frame a label so that we can refer to it easily (we'll call it **F1**). Note that it also has a parent pointer, which points back to the global frame (meaning that if we look up a name here and it isn't bound, we will continue looking for it in the global frame).

Now that we have set up our frame, the next step is to bind the parameters of the function to the arguments that we passed in. Continuing to the next step, we'll see the results of this process.

---



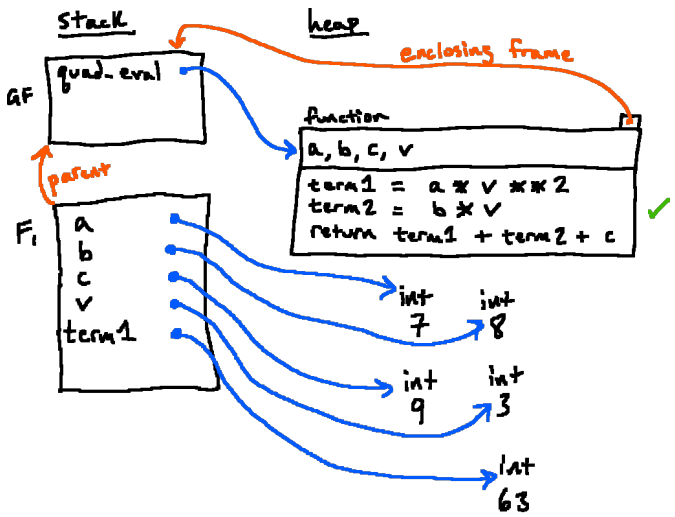

**STEP 5**

Here we see the effects of binding the function's parameters to the arguments we passed in: inside of **F1**, we now have a new variable for each parameter, each mapped to the corresponding argument we passed in.

At this point, our frame is completely set up, and so we are ready to carry on and run the body of this function inside of **F1**. The fact that we're running this code inside of **F1** (rather than in the global frame) is important. Given the structure we have here, our function can access variables both from **F1** and from the global frame; and any variables we define inside of the body of the function will be bound in **F1** rather than the global frame. This is the mechanism by which the notion of *local* variables is enforced, and it allows us to define our functions using whatever parameters and local variable names we want, without fear of Python getting confused if those names are also used in other contexts outside the body of our function.

Now that we're ready to run our function, the first step will be to execute the first line of the function's body, `term1 = a * v ** 2`. Before clicking through to the next step, take a moment to try to draw the effects of this change for yourself. Once you've walked through that process, click through to the next step to see whether your result matches ours.

---



**STEP 6**

Here we see the net effect of running that first line of code: we now have a variable `term1` in **F1**, and it's bound to an `int` object representing `63` on the heap.

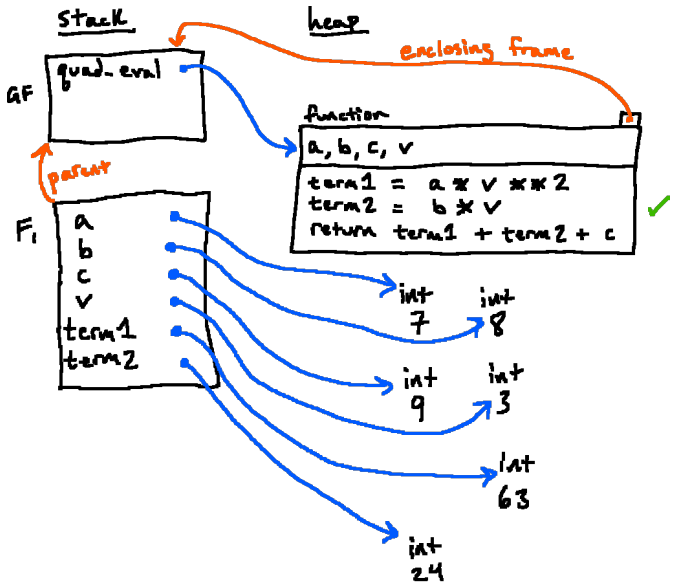
It's worth noting that it actually took several steps to get to this point: we first notice that `term1 = a * v ** 2` is an assignment statement, so we start by evaluating the expression on the right-hand side of the equals sign. Doing so involves several steps:

- we look up `a` in **F1**, following its reference to find the value `7` on the heap
- we look up `v` in **F1**, following its reference to find the value `3` on the heap
- we allocate a new `int` object representing `2` so that we can raise `v` to that power
- we perform exponentiation on the `3` and the `2` from the last two steps, resulting in a new `int` object representing `9`
- we perform multiplication on the `9` from the last step and the `7` we got from evaluating `a` in the first step to get a new integer `63`, the ultimate result of evaluating the expression
- we bind the name `term1` to that value in **F1** (creating a new *local* variable)

If we were being really careful ("pedantic," some might say...), we would have written out those intermediate results (e.g., the `2` and the `9` we got from exponentiation above). But we will often skip those kinds of steps and jump right to the end result, in part because, without any references pointing to them, they will be garbage-collected as soon as we're done with them.

In the next step, we're going to evaluate `term2 = b * v`. Try walking through this process for that assignment statement and adding its effects to your diagram, and then click on to the next step!

---




**STEP 7**

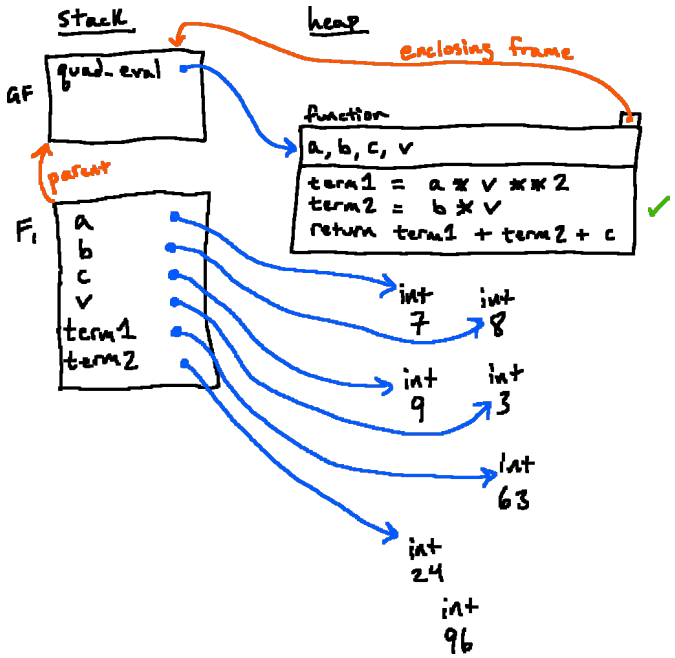
Here we see a new local variable `term2`, which is bound to a new integer object representing

the value `24`. The intermediate steps we took to get here are similar to what we saw in the previous step:

- we looked up `b` in **F1**, following its reference to find the value `8` on the heap
- we looked up `v` in **F1**, following its reference to find the value `3` on the heap
- we performed multiplication on those two `int` objects, resulting in a new `int` object representing `24`
- we bound the name `term2` in **F1** to that value

Once we've done this, the next statement in the function is the `return` statement at the bottom. Over the next couple of steps, we'll see how this statement behaves; but let's start by evaluating the associated expression `term1 + term2 + c`. Note that evaluating this will produce a new object representing the result of that computation. Think through what that value should be (and the process by which we arrive at that result) and then click on to the next step to see the result.

---



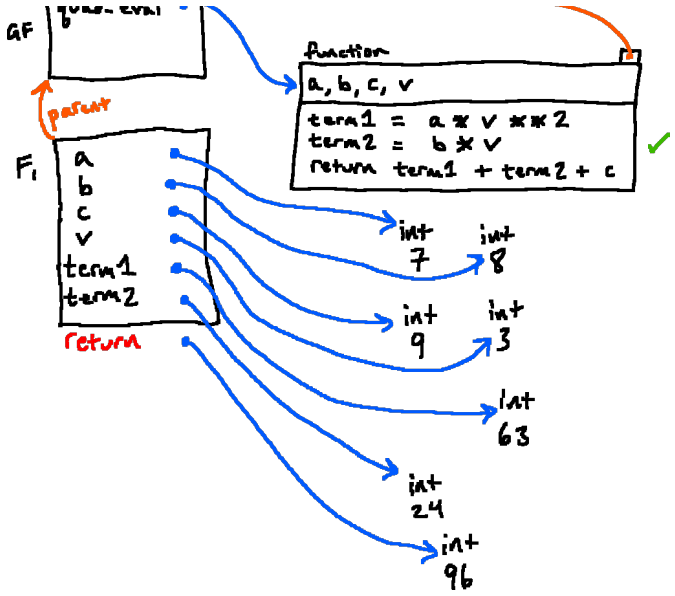

**STEP 8**

Here we see the result of that computation: an `int` representing `96`. We got here by:

- looking up `term1` in **F1**, following its reference to find the value `63` on the heap
- looking up `term2` in **F1**, following its reference to find the value `24` on the heap
- performing addition on those two values, resulting in a new `int` object, `87`
- looking up `c` in **F1**, following its reference to find the value `9`
- performing addition on the `9` and the `87`, resulting in a new `int` object, `96`

The next step we will take is a small one, but an important one for bookkeeping: we're going to make a note of the fact that this `96` object we just made is the value we're returning from the function call. Click through to the next step to see the notation we'll use for that.

---




**STEP 9**

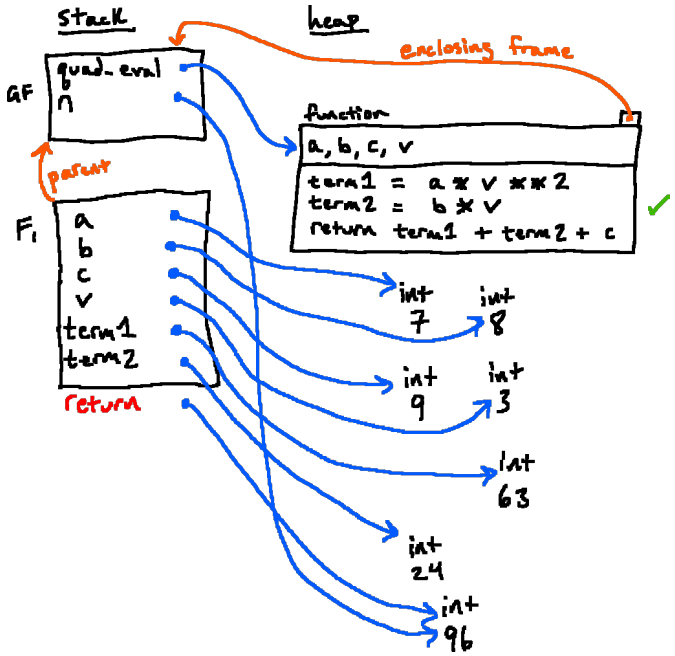
Here, we've labeled the return value by adding a little label near the frame; we're done with calling this function, and we have our result!

Importantly, we don't put this value in the frame (which would imply that we've made a new local variable called `return`!); but we'll note this down, because the next thing we need to do is to remember how we got here, i.e., what was it we were supposed to do with the result once we had it?

Looking back at how we got here, we remember that we started the process of calling this function as part of executing `n = quad_eval(7, 8, 9, 3)` in the global frame; and this result `96` is the result of evaluating the expression on the right-hand side of the equals sign in that expression, so the next thing to do is to associate that value with the name `n` in the global frame!

Make that change to your diagram, and then click through to see it in ours.

---

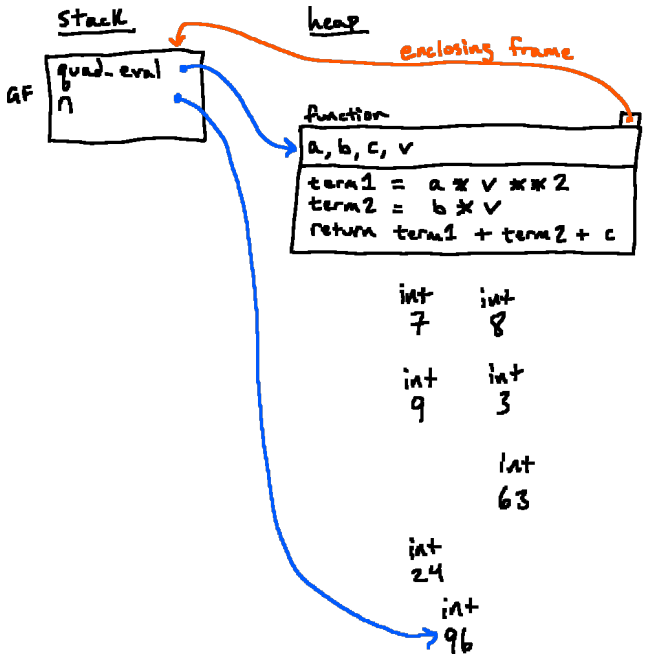



**STEP 10**

The arrows are starting to get a little unwieldy here, but the important thing to see is that we can now refer to the result from this function call (our new `96` object) by the name `n` in the global frame.

At this point, we can also do some cleanup. Since we are done executing in the frame **F1**, and since there are no arrows pointing to **F1** from other objects, we can delete it so that Python can reuse that memory for other things later on. Click through to the next step to see the result of cleaning up the frame.

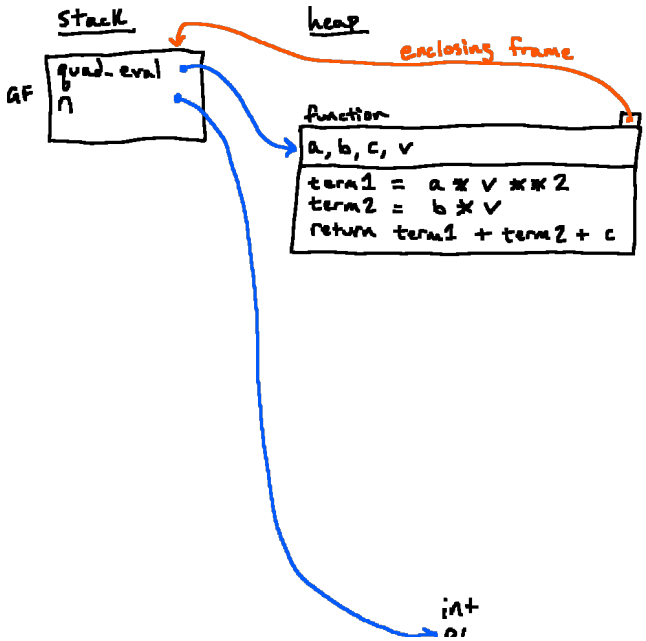
---




**STEP 11**

With our frame cleaned up, our diagram looks a little bit nicer again! But now we also have several objects out on the heap that don't have any references left pointing to them (these objects were only accessible from inside the body of the function, by way of **F1**), so they can be garbage collected as well. Let's get rid of them by clicking through to the next step.

---



**STEP 12**

And here we are, at the diagram representing the final state of our program. There's something beautiful about this end result: even though we just went through quite a process to get here (making a new frame, setting up local variables, allocating a bunch of new objects, garbage collecting things, etc.), the end result is quite clean: we have `n` now bound to the result of that function call, and all of the intermediate machinery used to complete the function call is now gone!

Now that you've worked through the example above, answer the following question about it:

In step 4 above, when we set up the frame **F1** for this function call, we set its parent pointer to be the global frame. For what reason was the parent pointer set this way?

- The function we're calling was being called from the global frame.
- The function we're calling was originally defined in the global frame.
- The name we used to refer to the function, `quad_eval`, was bound in the global frame.
- Something else

## 3.3) Arguments vs. Parameters

Note that we've made a distinction here between:

- *parameters*, which are variables used inside the function, and
- *arguments*, which are the values that are passed when the function is called, becoming the values of the parameters as the body of the function starts executing.

It's important to keep this distinction straight in your mind. When calling `quad_eval(7, 8, 9, 3)`, the arguments of the call are 7, 8, 9, and 3, and they end up assigned to the parameters `a`, `b`, `c`, `v`, respectively.

Parameters and arguments are sometimes called *formal parameters* and *actual parameters*, respectively, or just *formals* and *actuals*.

## 3.4) Check Yourself

Now that we've talked through the process by which functions are defined and called, we'll take a look at how they play out in a small example. This small program is, admittedly, a little bit contrived... but it should still illustrate how these rules give rise to the behaviors that we expect to see from functions.

Here is the piece of code we'll be considering (note that the names `foo` and `bar` don't have any real

meaning, but they're conventionally used to refer to functions for which we don't have nicer names, often in silly examples like this):

Show/Hide Line Numbers

```
x = 500

def foo(y):
    return x+y

z = foo(307)

print('x:', x)
print('z:', z)
print('foo:', foo)

def bar(x):
    x = 1000
    return foo(308)

w = bar(349)

print('x:', x)
print('w:', w)
```

**Check Yourself:**

**Without running the code above**, try to predict what will be printed to the screen and fill in your guesses below. If you aren't able to predict these values correctly, that's OK (mistakes are a great way to learn!), but please make an earnest attempt to answer without relying on the description that follows, before moving on to our explanation.

What value will be printed for `x` on line 8?

What value will be printed for `z` on line 9?

What value will be printed for `x` on line 18?

What value will be printed for `w` on line 19?

Now that you have made a guess, let's draw an environment diagram as a way to help us reason about what happens when we run this code; and, if we're careful with following the associated rules, we can use the diagram to accurately predict the output of the program above. But, this is a complicated example, so grab a cup of tea or coffee and settle in. If you are confused by anything that follows, please feel free to ask for help!

Show/Hide Line Numbers

```
x = 500

def foo(y):
    return x+y

z = foo(307)

print('x:', x)
print('z:', z)
print('foo:', foo)

def bar(x):
    x = 1000
    return foo(308)

w = bar(349)

print('x:', x)
print('w:', w)
```

<< First Step | < Previous Step | Next Step > | Last Step >>


**STEP 1**

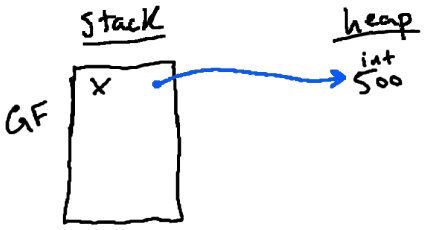
Before we start running this program, we set up our initial environment structure. Here we have set up our "global frame" (**GF**), in which our program will be executed.

The next step to be executed is the assignment statement on line 1 (`x = 500`). What should the diagram look like after that statement has been executed?

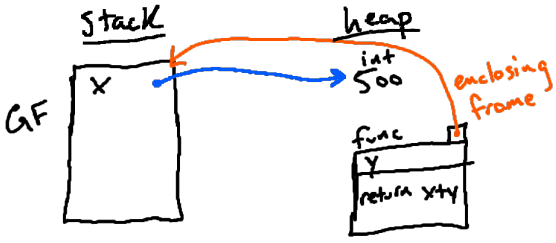
**STEP 2**

Evaluting `500` gives us a new object that represents the integer 500. After creating this object, we associate the name `x` with it (in the global frame).

The next statement we encounter is a *function definition*. As mentioned above, Python will do two things here: firstly, it will make a new object to represent this function; then, it will associate that new function with a name (in this case `foo`).




---




**STEP 3**

Here we see the function object itself. Note the three things stored inside of the function: the names of the parameters (in this case, a single parameter called `y`), the body of the function (`return x+y`), and a reference to the enclosing frame (i.e., the frame we were running in when we executed this `def` statement).

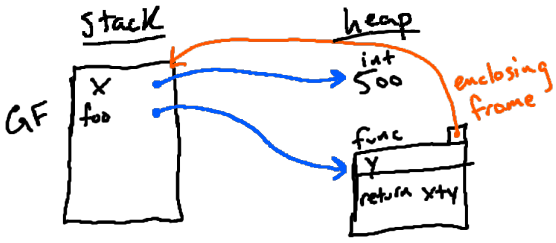
Here we have labeled the reference to the enclosing environment with those words, but we will not always write that out.

---

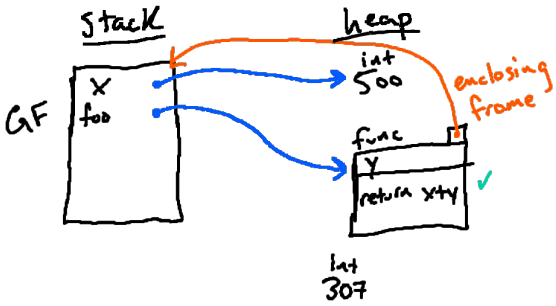
**STEP 4**

Now we have also associated that function with the name `foo` in the global environment.

Note that we have not run any part of this function's body yet; we have just stored away the relevant information so that we can eventually call this function.




---




**STEP 5**

After finishing with the `def` statement, we are moving on to line 6, `z = foo(307)`. This represents a function call, and so we need to follow the steps outlined under the "Calling a Function" header above.

We start by figuring out what function to call and what to pass to it. In this case, the function we're calling (the thing before the round brackets) is given by the expression `foo`. Evaluating that name in the global frame, we follow the associated arrow and find the function object on the right (marked with a small green check mark so that we can keep track of it).

Then we evaluate the arguments to the function. In this case, we have a single argument `307`. Evaluating that expression produces a new object representing the integer 307.

Once we know what function we're calling and what arguments we're passing in to it, we are ready to proceed with the function call.

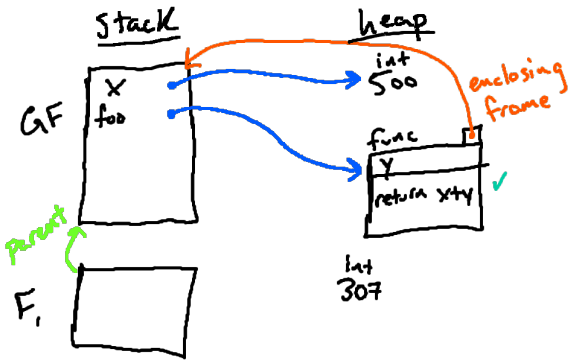
---

**STEP 6**

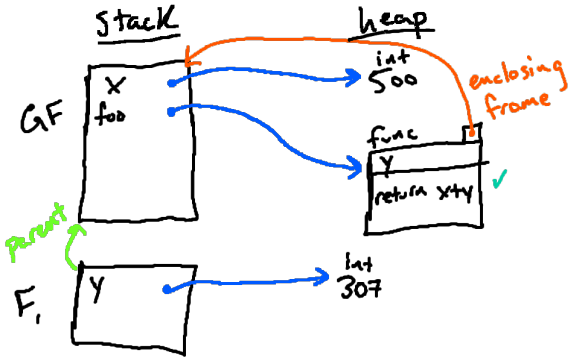
Our next step is creating a new frame to store the local variables associated with this function call. So we have made a new frame here. We also give it a name (just so that it is easier to refer to it), in this case **F1**.

Note that we have already drawn **F1**'s parent pointer. How did we know that **GF** was the right parent? When we made **F1**, we looked at the function we were calling and used its enclosing frame as **F1**'s parent.

Now that we've got our new frame set up, we proceed with our next step: binding the parameters of the function to the arguments that were passed in.




---



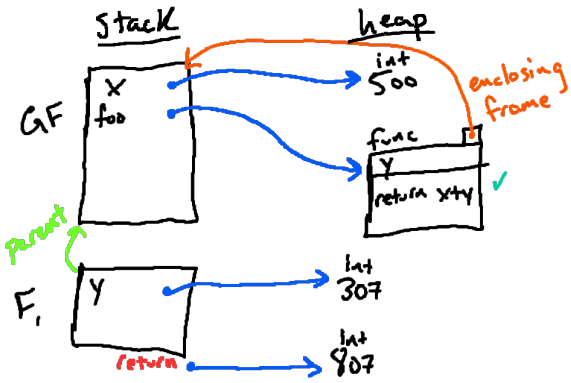

**STEP 7**

Here we see the end result of binding the parameters of our function to the arguments that were passed in. In this case, the function we're calling is a function of a single parameter (`y`), and the argument that was passed in is the object we see containing 307.

After this step, we have our local frame all set up, and we are now ready to actually run the body of the function.

actually run the body of the function.

---




**STEP 8**

After we've set up the frame associated wth this function call, we proceed by executing the body of the function inside of **F1**. In this case, the body of the function is `return x+y`.

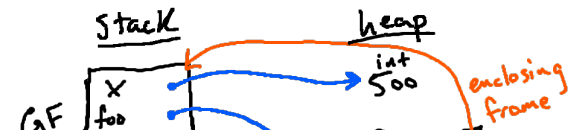
Executing `return x+y` with respect to **F1** involves evaluating `x` and `y` with respect to **F1** and then adding those results together.

Evaluating `y` with respect to **F1**, we follow the arrow from `y` to the associated object, finding the integer 307 that was passed in as argument.

Evaluating `x` with respect to **F1** is a little more complicated, since the name `x` is not bound in **F1**. But here we see the nature of the parent pointer. When evaluating a variable name in a given frame, if we don't find the name bound in that frame, we will follow the frame's parent pointer and look for the name there. If we find the name there, the associated object is the result. If we still don't find the name, we follow the next parent pointer; and we keep doing this until we either find the name we're looking for, or we run out of frames to look in. In this case, we don't find `x` in **F1**, so we follow its parent pointer and look for `x` in **GF**, where we find the associated value of 500.

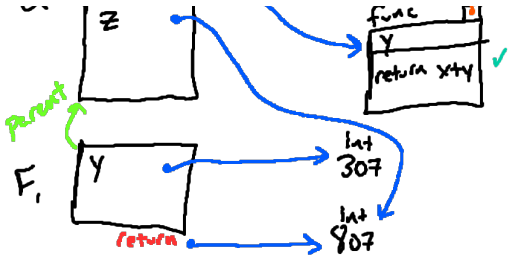
Once we have those two pieces, we add them together, producing a new object representing the integer 807. This new object is the return value from our function call, which we will sometimes denote as above. Our function call is now done, and we need to look back and remember how we got here. In this case, we started this function call in the process of evaluating line 6, where our goal was to associate the result of the function call with the name `z` in the global frame.

---



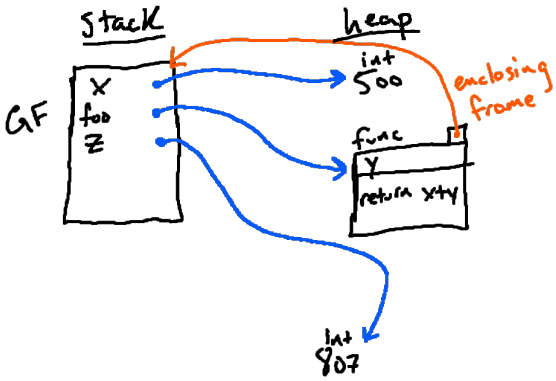

**STEP 9**

Here we see the result of that assignment. At




this point, we are done with the function call, and we can clean up all of the machinery that we used to figure out how the function call behaved and move back to executing code in the global frame. The result of that cleanup will be shown in the next step.

---




**STEP 10**

Here we see the result after line 6 has been executed. Next, we hit three print statements. Following the arrows in our diagram, we can now see that `x` is associated with `500` and `z` is associated with `807`; so we can predict that those are the values that will be printed for `x` and `z`. But an important question remains: what will be printed for `foo`?

Even though it may seem strange at first, we can determine what will be printed for `foo` in the exact same way that we figured out what to print for `x` and `z`: by following the arrow in our diagram from the name `foo`. In this case, we find the function object on the right side of the diagram, which is what Python will print.

Python prints this in a somewhat strange way, but it's trying its best. It prints out a little representation that includes the function object's location in memory, something like `<function foo at 0x7f30d9080040>`, where the exact number on the right side will be different each time we run the program.

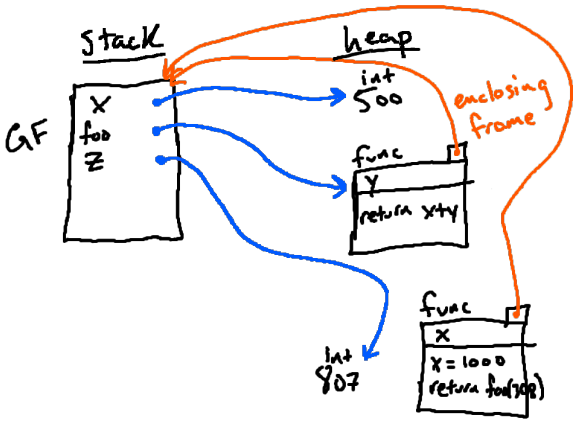
**Importantly**, what we are printing when we `print(foo)` is *the function object itself*, not the result of calling that function. Line 10 does not cause `foo` to be called; we simply look up the name `foo` and print the associated object.

Ultimately, we see something like the following as the output of lines 8-10:

```
x: 500
z: 807
foo: <function foo at 0x7f30d9080040>;
```

Now, we are ready to move on to our next statement, the second `def` on line 12.

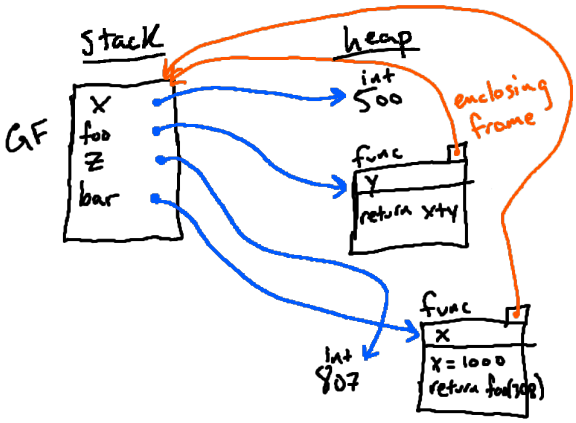
---




**STEP 11**

We follow the same rules when executing this `def` statement as we did before, and the first part of that process results in a new function object, shown in the diagram above. Note again that we have not run any of the code in the body of this new function yet; we are just storing it away to be called later.

---




**STEP 12**

The `def` statement also associates the name `bar` with this function, in the global frame. The end result is shown in the diagram above.

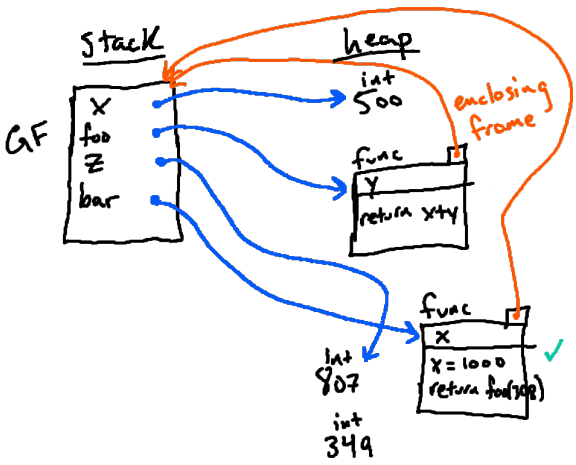
With that, we are done with evaluating the `def` statement that starts on line 12, and we are ready to move on to the next statement, the assignment on line 16.

---

**STEP 13**

Evaluating the assignment statement on line 16 involves first evaluating the expression on the right-hand side of the equals sign, which is a function call. We start that process by determining what function we're calling and what arguments we're passing in to it.

In this case, the function we're calling is given by `bar`, and a single argument `349` is given. So we evaluate the name `bar`, finding the function object indicated with the green checkmark above; this is the function we're going to call. And evaluating `349` produces a new object representing the integer 349, which is the object we are passing in to the function.

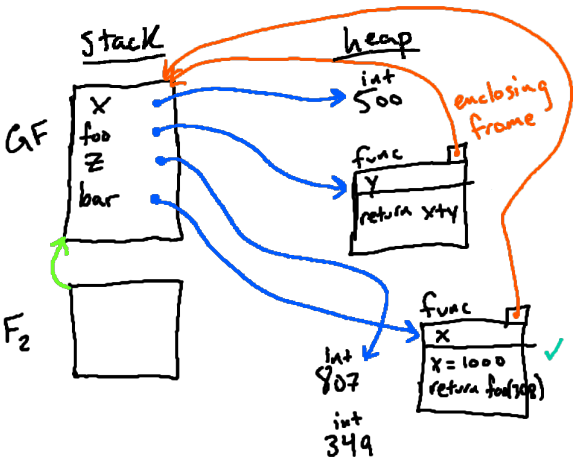



Once we know what function we're calling and what we're passing as inputs, we can proceed with the function call using the rules described earlier.

---

**STEP 14**

We start by making a new frame to store this function's local variables. Above, we've made this frame and labeled it as **F2**. (We garbage-collected F1, but let's give this frame a fresh name, to avoid confusion.) Note that **F2** has a parent pointer to our function's enclosing frame.



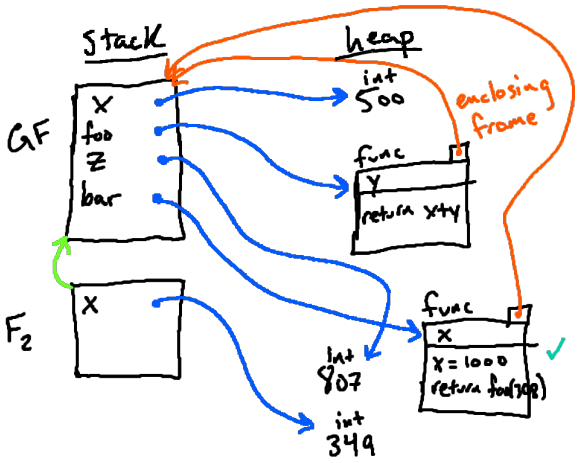

After we have created the frame, our next step will be to bind the parameters of the function to the objects passed in as arguments; the

result of this process will be shown in the next step.

---

**STEP 15**

Here, we show the result of binding the name `x` to the value `349` inside of our new frame **F2**.

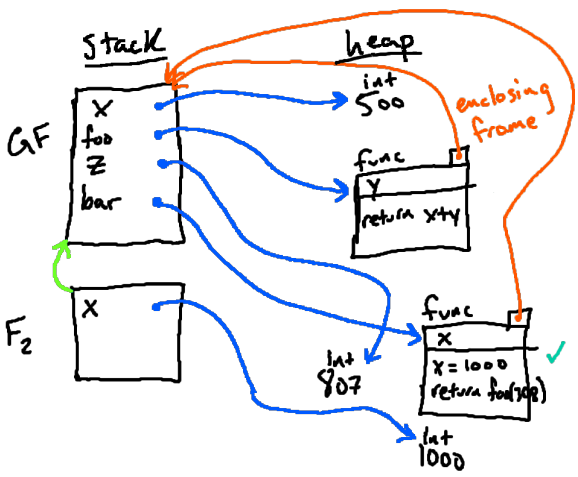



Note that this binding did *not* affect the value associated with the name `x` in the global frame. Even though `x` refers to `349` in **F2**, `x` still refers to the value `500` in the global frame. And since we will ultimately evaluate the body of this function inside of **F2**, referring to `x` in the function will not find the global binding of `x`.

This setup (where each function call gets its own frame to store its local variables) is generally really nice and quite powerful, in that it allows us to call the parameters of our functions (and other local variables inside of our functions) by any names we wish, without needing to worry about accidentally messing up the variable bindings in other frames.

Anyway, once we have made that binding, we are ready to execute the body of the function inside of **F2**. And so the next statement we encounter is `x = 1000`, which we evaluate with respect to **F2**. Can you predict what the result will look like? We will show it in the next step.

---



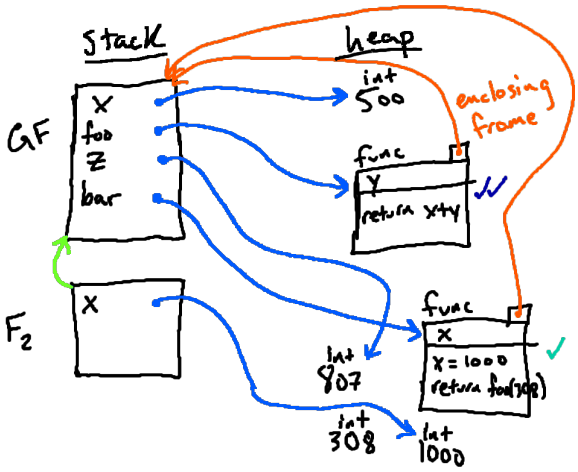

**STEP 16**

Since we were executing `x = 1000` with respect to **F2**, we modify the arrow from `x` inside of **F2**. Note again that the global binding of `x` is unaffected. At this point, we're done with `x = 1000`, so we can move

on to the next statement.

---

**STEP 17**




Remember that at this point, we are still executing the body of our function inside of **F2**. And the next statement we encounter is `return foo(308)`. This is another function call! So what do we do?

Conveniently, even though we are now going to be two function calls deep, we evaluate this second function call in precisely the same way we evaluated the first one, by following the rules laid out earlier on this page. And so the first thing we need to do is to determine what function we're going to be calling and what arguments we're going to pass in. But there is a little bit of subtlety here that we need to be careful of.

Because this function call is being made from **F2**, figuring out what function to call involves looking up the name `foo` inside of **F2**. We don't find the name `foo` there, so we follow the parent pointer and look in the global frame, where we find that the name `foo` is bound to the function object that now has two small blue checkmarks next to it; so that's the function object we'll be calling.

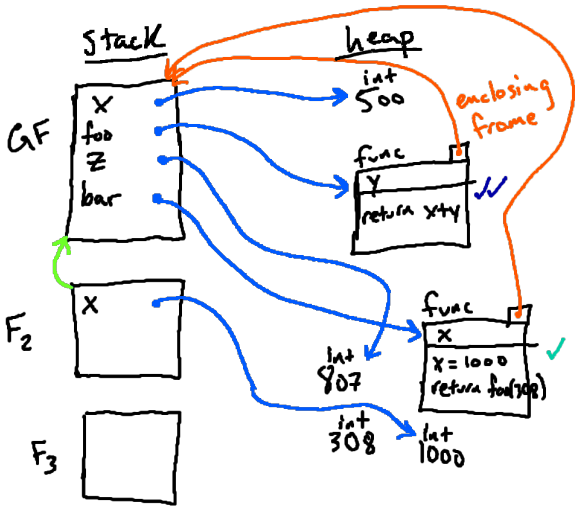
Then we evaluate the argument and get a new integer 308, which has been drawn in above as well. And, now that we know what function we're calling and what arguments to pass in, we can proceed with actually calling the function. Our first step will be to make a new frame for this new function call.

---

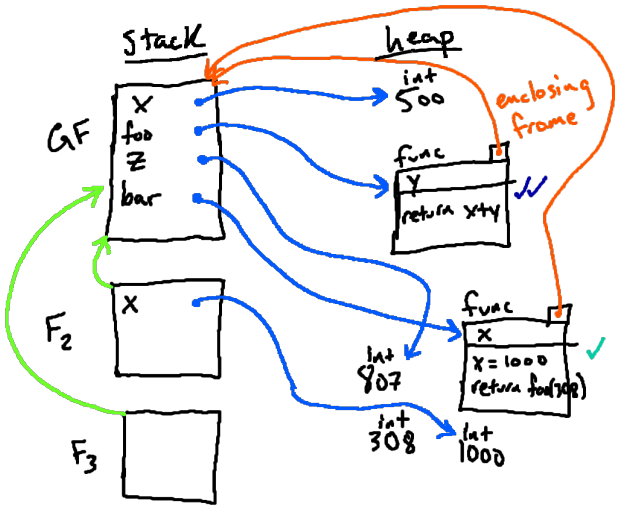
**STEP 18**

Here we have drawn in our new frame (called **F3**), but we're actually not quite done with that part yet because we have not yet given this frame a parent pointer. **Please take a moment and try to predict**, given the rules

parent pointer. **Please take a moment and try to predict**, given the rules we have outlined here, where should **F3**'s parent pointer go?




---




**STEP 19**

The key rule for determining a new frame's parent pointer is: **what is the enclosing frame of the function we're calling?**

In this case, we're calling the function drawn toward the top of the diagram (with the two blue checks next to it, which is referred to as `foo` in the global frame); and that function's enclosing environment is **GF**, so that's where our new frame's parent pointer goes.

Importantly, this is true even though we made this function call from **F2**! What matters for determining this structure is not where we're making the call from, but rather where the associated function was originally created

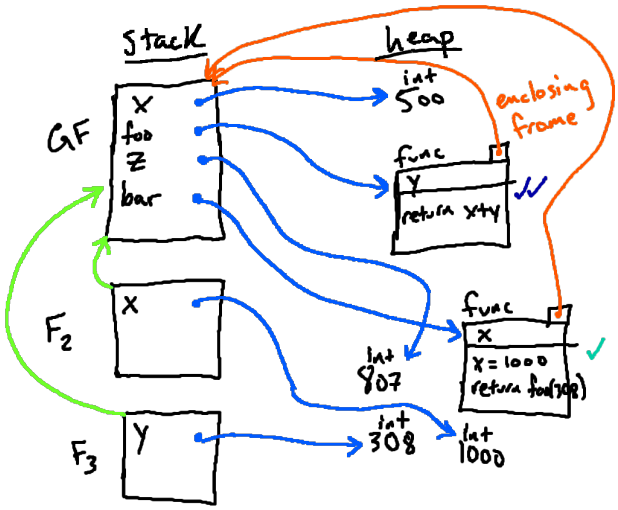
call from, but rather where the associated function was originally created. This may seem like a strange rule at first, but it leads to a lot of nice properties that we will discuss in more depth during class.

And now that we've set up our new frame, we can move on to our next step, binding the parameters of the function to the arguments that were passed in.

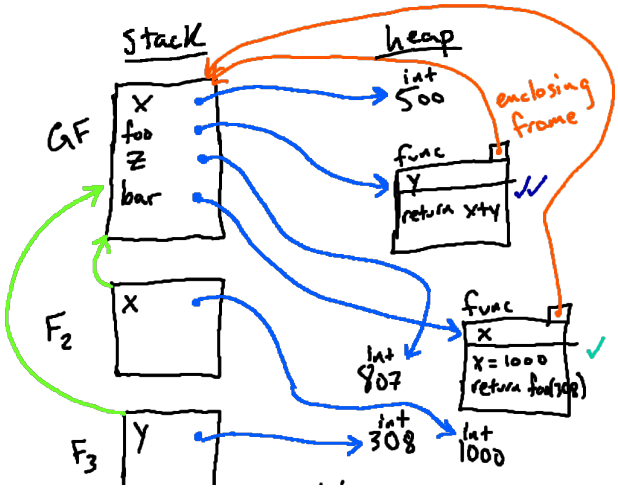
---

**STEP 20**

The function we're calling is a function of a single parameter called `y`, and so inside of **F3**, we bind the name `y` to the 308 that was passed in. Having done so, we are ready to execute the body of the function inside **F3**.




---




**STEP 21**

Evaluating the body of the function with respect to **F3** involves evaluating both `x` and `y` with respect to **F3** and then adding the results together to produce the return value of this function call.

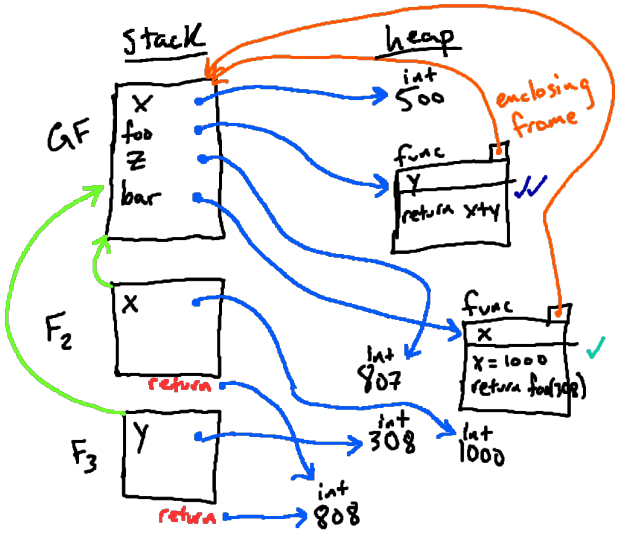

Evaluating `y` with respect to **F3** is relatively straightforward: `y` is bound in that frame, and so we follow its arrow to find the 308 that we passed in as an argument.

Evaluating `x` with respect to **F3** is more involved. We look for `x` in **F3**, but we don't find it there, so what do we do next? We follow **F3**'s parent pointer and look for `x` there. So, importantly, we find the value 500, to which `x` is bound in the global frame (and *not* the 1000 to which `x` is bound in **F2**).

Adding these values produces a new value `808`, which will be our return value.

Now we need to think a little bit. How did we get to this function call, and where should we return back to? Remember that we started this function call in the process of figuring out what to return from our original call to `bar`. So we'll jump back to that point in the execution, with our new `808` value in hand.

---

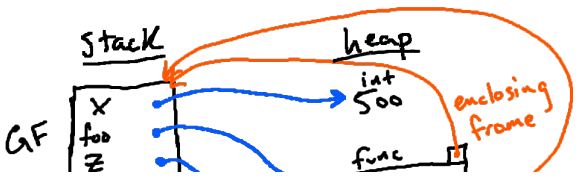



**STEP 22**

Since the line we had been executing in **F2** was `return foo(308)`, we're simply going to take the result of that function call (the 808 we just figured out) and mark it as the return value from **F2**, which has been reflected above.

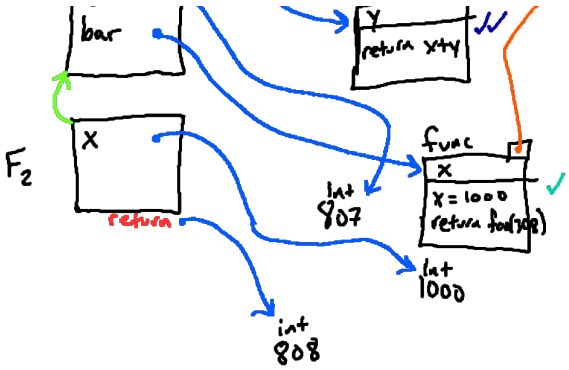
At this point, we are also done with that second function call. So on the next step, we'll clean up some of the machinery that we created purely for determining that result.

---




**STEP 23**

And we also need to remember: how did we

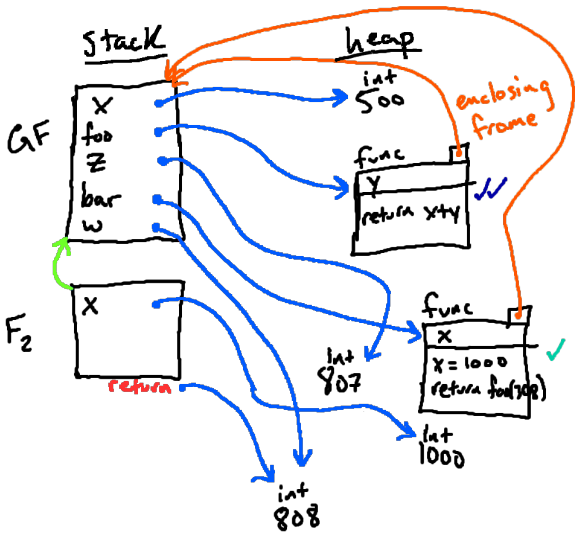



come to be calling `bar` in the first place? That is, now that we have the output of the function call, where do we return to?

We first got here by executing `w = bar(349)` in the global frame. So that's where we'll go next, binding the name `w` to 808 inside of the global frame.

---

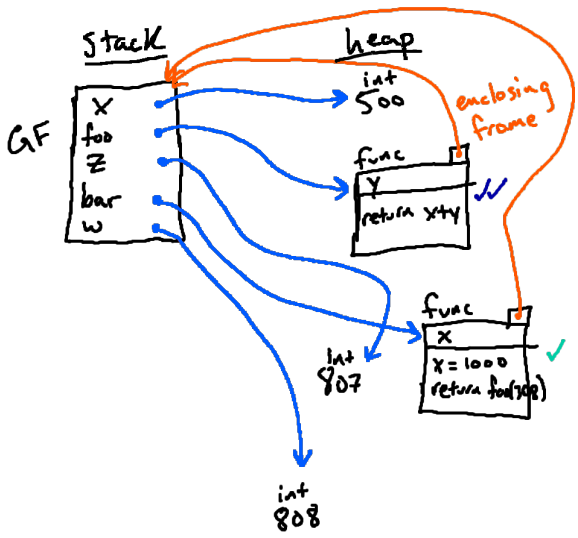
**STEP 24**




Phew, that was quite a journey! But now we're done with executing line 16 in our original program!

At this point, we can clean up **F2**, since we're done with that function call, leaving us with one final diagram.

---




**STEP 25**

Now the only thing left to do is to figure out what prints when lines 18 and 19 are run. Tracing the arrows from `x` and `w` from the global frame, we find 500 and 808 (respectively), and so those are the values that will be printed:

x: 500
w: 808

After the value 808 has been returned from **F2**, we no longer have a need to keep **F2** around, and so it (as well as the integer `1000` that was only reachable from **F2**) will be garbage collected, leading to the final diagram.

This diagram represents the state of memory at the end of our little program. Notice that the value of `808` that we got from calling our function is still available to us, via the name `w` in the global frame.

One last question for this section: imagine adding `x = 2000` to line 5 of the program above and running it again. Which of the following printed values would be different from running the code without that change?

- [ ] `x`
- [ ] `z`
- [ ] `foo`
- [ ] `w`

If you're unsure of why this answer is what it is, think about how things would change in terms of our environment diagrams. And, of course, if you're stuck, please don't hesitate to ask for help!

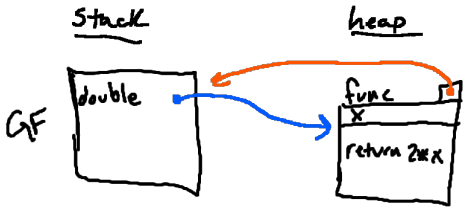
## 4) Functions Are "First-Class" Objects

One powerful feature of Python is that it treats functions as first-class objects, which means that functions in Python can be manipulated in many ways that other data can (they can be passed as arguments to other functions, defined inside of other functions, returned from other functions, added to collections, and bound to variables, among other things). This is reflected in our environment diagrams by the fact that functions are represented by objects on the right sides of our diagrams, just like other data (integers, lists, etc.) are!

Some of the sections below will talk more about the consequences of this feature. But, to start, let's look at a small example:

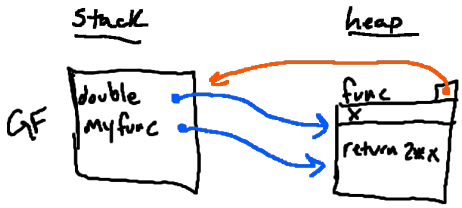
```
def double(x):
    return 2 * x

myfunc = double
```

After executing the first `def` statement in the little code block above, our environment diagram looks something like this:

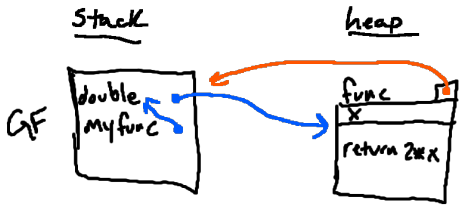



Below are four possible diagrams that could result after executing the second statement (`myfunc = double`). Which of these, if any, represents the result of executing that statement?

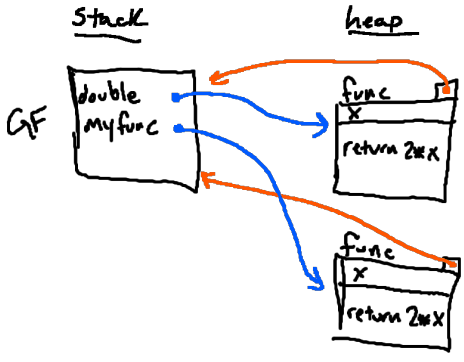
**A**



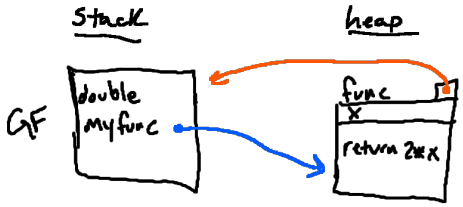

**B**




**C**




**D**



Which diagram shows the correct result? --

What is the result of evaluating `myfunc(21)`, starting from the diagram above? Enter a value below, or enter `error` if this expression causes a Python error.

Starting with the definition of `double` from above, if we *wanted* to create a structure like the one shown in environment diagram option **C** above, what code should we write? Type your code in the box below:

```
def double(x):
    return 2 * x

```

### 4.1) Example: Thinking About Types

Consider the following small piece of code:

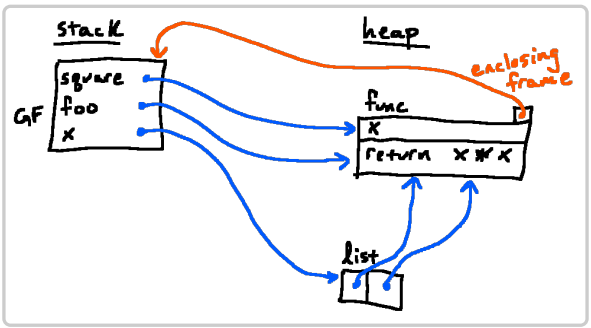
Show/Hide Line Numbers

```
def square(x):
    return x * x
foo = square
x = [square, foo]
```

Start by drawing an environment diagram for this code, and, once you have done so, click below to show our solution:

Show Our Solution

Here is the diagram we drew:




Does your drawing match this one? If not, pay careful attention to the differences. What assumptions about the code above (or the nature of the diagrams) led to those differences?

Make sure you understand this diagram before moving on! If not, please come and ask someone during instructor office hours or open lab hours, or via the `6.101-help@mit.edu` mailing list.

Now, using your diagram from above, predict the type of each of the following expressions and enter your answer below. For each, also think about what would be printed to the screen if we were to print that object. After you have entered the correct type for each expression, viewing the associated answer will provide some more details and explanation.

Using the global frame of the code/diagram above, what is the type of `square(2)`?

--

Using the global frame of the code/diagram above, what is the type of `foo(0.7)`?

--

Using the global frame of the code/diagram above, what is the type of `foo`?

--

Using the global frame of the code/diagram above, what is the type of `x`?

--

Using the global frame of the code/diagram above, what is the type of `foo[1]`?

--

Using the global frame of the code/diagram above, what is the type of `x[0]`?

--

Using the global frame of the code/diagram above, what is the type of `square()`?

--

Using the global frame of the code/diagram above, what is the type of `x[1](3.1)`?

--

Using the global frame of the code/diagram above, what is the type of `(square + foo)(7)`?

--

### 4.2) Example: Functions As Arguments

We'll see a more practical example in a moment, but, for now, let's take a look at a small example illustrating an interesting use of first-class functions. Consider the following code:

```
def apply_n_times(f, n, x):
    out = x
    for i in range(n):
        out = f(out)
    return out
```

This code is, sadly, undocumented right now, but we can figure out some things about what it's doing from looking at how its arguments are used within the body of the function. Take a moment and see if you can figure out anything about the type(s) or value(s) of the arguments before reading on.

Looking at the code, we can see where `n` is being used: it's being used as the sole argument to the `range` function. Since `range` accepts integers as inputs, we can assume that `n` must be an `int`. And the looping structure there tells us that `n` specifies the number of times that we're repeating some action. What are we doing inside of the body of that loop?

The syntax there shows us that we are *calling* `f` with `out` passed in, binding the result of that call back to the name `out`. This tells us that `f` probably needs to be a *function*! Indeed, `apply_n_times` is a function itself, but it takes *another function* as one of its inputs! This is pretty trippy, but we can see it in action.

Consider the following function definition:

```
def double(y):
    return y * 2
```

What can we put in the blank in the code below to make the result equal to `48` (with no exceptions being raised)?

```
apply_n_times(_________, 4, 3)
```

--

#### 4.2.1) Aside: Using _ For Intentionally-Unused Variables

A quick aside about Python programming style. If we took this code:

Show/Hide Line Numbers

```
def apply_n_times(f, n, x):
    out = x
    for i in range(n):
        out = f(out)
    return out
```

... and gave it to a Python style checker like `pylint`, one of its complaints would be:

```
line 3: Unused variable 'i' (unused-variable)
```

... because, indeed, the `for` loop is defining `i` but never actually needs the value of `i`.

Style checkers complain about variables that are created but never used because it might be a bug (you meant to use it but forgot to), or at least it's a waste of effort.

But it's hard to get rid of the warning in this case, because the Python `for` loop requires *some* variable name.

For these kinds of situations, when you have to provide a variable name but don't actually need the value that gets assigned to it, Python has a convention of using `_` for the variable name. A single underscore is a valid variable name, but it looks like a placeholder or "I don't care" name.

So if the loop were written this way:

```
for _ in range(n):
    out = f(out)
```

...then `pylint`, and other Python programmers, would know that you are intentionally not using that loop variable. All that you care about is running the loop `n` times.

### 4.3) Function definition with `lambda`

For just a bit longer, let's keep looking at our small example function from the last section:

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

As discussed above, this statement does two things: it creates a new function object, and it associates that

function object with a name, such that we can refer to it later by that name.

Python also has another way of defining functions, which can be useful in some situations: the `lambda` keyword. This may seem a bizarre name, but it comes from a mathematical system for expressing computation, called the Lambda calculus. Another way to make a function that squares its input is:

```
lambda x: x * x
```

This makes a function that is almost exactly the same as `square`, except that it does not have a name. `lambda` is never necessary (you can always use `def` instead), but it is a nice convenience in some situations. The most common place where `lambda` is useful is when we need to provide a small function as argument to another function, but we don't have that function around yet (and we don't want to keep it around for other purposes). For example, in our `apply_n_times` example from above, if we had no use for `double` outside of passing it in as argument to `apply_n_times`, we could have produced an equivalent function call with the following:

```
apply_n_times(lambda x: x * 2, 4, 3)
```

Note that `lambda` is intended for very short functions, so it is more limited than `def`. It can only evaluate and return a single expression, so you can't use Python constructs that need multiple lines, like `for`, `while`, or `if` statements. You also can't define additional local variables inside a `lambda`.

### 4.4) Example: Plotting Function

We've seen a couple of examples above that made use of functions as first-class objects, but they were somewhat contrived, small examples that only existed to demonstrate the rules by which function objects operate. But right about now you might be asking yourself: is this actually *good for anything*? Is there any *practical* use? And the short answer is: yes! Not only is this a cool feature, but it turns out to be useful for organizational reasons. For now, let's just take a look at one example of how we can leverage this idea to improve the style of a piece of code.

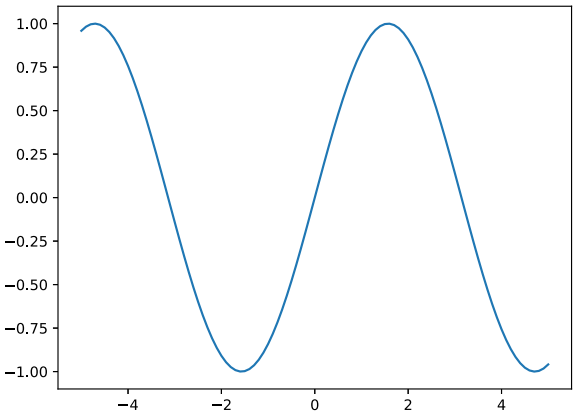
To start, let's consider the following piece of code, which uses the `matplotlib` package to generate a plot of a sine wave:

Show/Hide Line Numbers

```
import math
import matplotlib.pyplot as plt

def sine_response(lo, hi, step):
    xs = []
    ys = []
```

```
    cur = lo
    while cur <= hi:
        xs.append(cur)
        ys.append(math.sin(cur))
        cur += step
    plt.plot(xs, ys)
```

If we run call this function using `sine_response(-5, 5, .1)`, matplotlib produces a plot for us showing a sine wave evaluated at several points between -5 and 5, spaced .1 apart:




But, now, imagine we want to plot other functions. What can we do? Well, the temptation will exist to copy/paste the function and make some slight changes:

Show/Hide Line Numbers

```
import math
import matplotlib.pyplot as plt

def cosine_response(lo, hi, step):
    xs = []
    ys = []
    cur = lo
    while cur <= hi:
        xs.append(cur)
        ys.append(math.cos(cur))
        cur += step
    plt.plot(xs, ys)
```

Hooray, now we can plot cosines, too! Oh, but maybe there's another thing we want to plot, so let's do it again:

Show/Hide Line Numbers

```
import math
import matplotlib.pyplot as plt

def square_response(lo, hi, step):
    xs = []
    ys = []
    cur = lo
    while cur <= hi:
        xs.append(cur)
        ys.append(cur ** 2)
        cur += step
    plt.plot(xs, ys)
```

And so on and so on...

This process is very tempting, but it results in a **lot** of repeated code! And then what happens when we want to change the behavior of this function, or we discover a bug? Now we have to remember to make the necessary changes in *each* of the similar functions we've defined above!

But, we can do better here! An important thing to notice here is that each of the above functions is doing the same operation: it is looping over input values, *performing some operation on each input* and storing the results in a list; and then, finally, plotting the results.

The high-level idea here is that we would like to reorganize things so that the common behavior exists only in one place that we can repeatedly invoke but where the specifics can be altered. One way to do this is shown below:

Show/Hide Line Numbers

```
import math
import matplotlib.pyplot as plt

def response(f, lo, hi, step):
    xs = []
    ys = []
    cur = lo
    while cur <= hi:
        xs.append(cur)
        ys.append(f(cur))
        cur += step
    plt.plot(xs, ys)
```

Notice that, here, we're specifying the function we want to plot as a Python function object! With this structure, the process of plotting a new function is easier, more concise, and far less error-prone than our

original version! For example, with that helper function in hand, the following code is all we need to plot several different functions:

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

response(square, 0, 5, 0.001)
response(lambda x: x ** 3, 0, 5, 0.001)
response(math.sin, 0, 10, 0.01)
```

Which is a huge improvement over the repetitious code we started with!

So, here, we've seen an example of leveraging first-class functions as an organizational tool, and we'll see more examples of this idea in an upcoming lab and recitation.

## 5) Understanding Last Time's Mystery

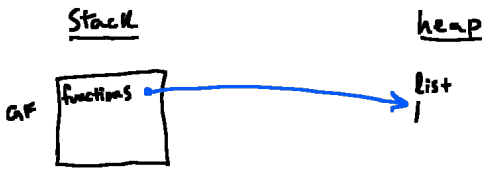
In last week's reading, we looked at the following piece of code as an example of code that is difficult to understand:

Show/Hide Line Numbers

```
functions = []
for i in range(5):
    def func(x):
        return x + i
    functions.append(func)

for f in functions:
    print(f(12))
```

It is somewhat surprising that, despite the looping structure here, when we run this code, we see five `16`'s printed to the screen! Despite the surprising nature of this example, though, we now have all of the tools we need in order to make sense of this example and to understand *why* it behaves the way it does. We'll walk through an environment diagram to explain this behavior, and you're strongly encouraged to follow along (and to reach out for help if any of the steps are unclear!).

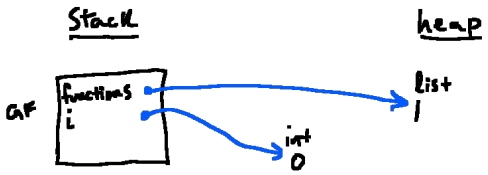
We'll start by drawing the diagram just for the first segment of the code (lines 1-5, where we are building up the `functions` list). Again, we encourage you to try to stay one step ahead of the drawings below (that is, try to draw out how things will change during each step, then click ahead and compare your work against our diagram).




**STEP 1**

Here is the state of our diagram as we are first getting started. After having run line 1 (`functions = []`), we have a single empty list on the heap, and the name `functions` is bound to that object.

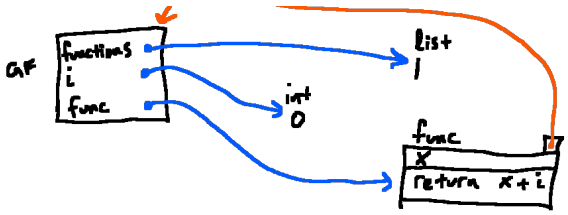
On the next step, we'll start the process of looping. For now, we'll not draw the `range` object created by calling `range(5)`; rather, we'll jump ahead to when we first enter the body of the loop. When we do that, `i` will be bound to the first element from that `range` object *in the frame where we're executing the loop*. No new frames will be made at this point, so `i` will be bound to `0` in the global frame. Click ahead to see the result of that binding.




**STEP 2**

Here we've bound the name `i` in the global frame to an `int` object representing `0` (the first element produced by the `range` over which we're looping).

Now we're inside the body of the `for` loop, and the next thing we see is a function definition (`def`) on lines 3-4. Executing this statement, we will end up with a new function object on the heap (including all the parts we've seen before), and it will be bound to a name. Try drawing that out now, before clicking through to the next step.



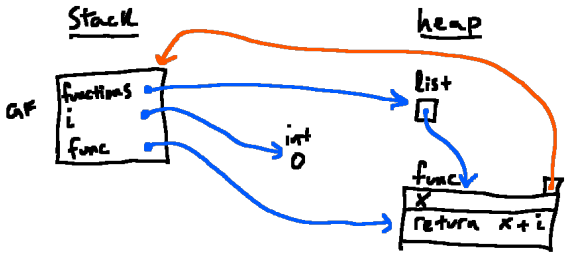

**STEP 3**

Here we see the result of that function definition. We've made our function object and stored away the relevant information inside of it. Because we're still running in the global frame, this new function's enclosing frame is the global frame, and the name `func` is bound in the global frame as well.

It's worth reminding ourselves of a critical fact here: at this point, when we're defining this function, *we do not execute the body of the function*. So we don't do any variable lookups or any additions; we're done with defining our function, so we simply carry on to the next statement.

The next statement is `functions.append(func)`. Take a moment to update your diagram to show the results of executing that statement before clicking through to see our result.

---

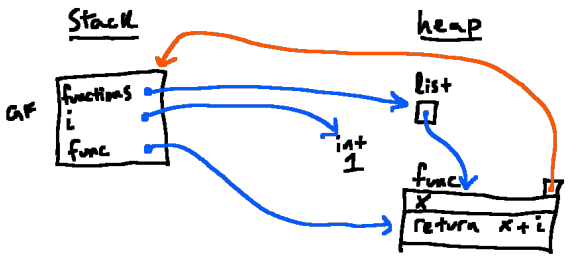



**STEP 4**

This is not a big change, but our list (which we've called `functions` in the global frame) is no longer empty: it now holds a reference to the function we just created.

After this point, we've hit the bottom of the body of our `for` loop, and so we're ready to move on to the next iteration. The next time through the loop, our looping variable `i` will be bound to the *next* element coming from the `range` object we're looping over. Once you've thought through the effects and tried to update your diagram accordingly, go ahead and click through to the next step to see the result.

This next step is **important** to describing the behavior of this piece of code, so pay careful attention!!

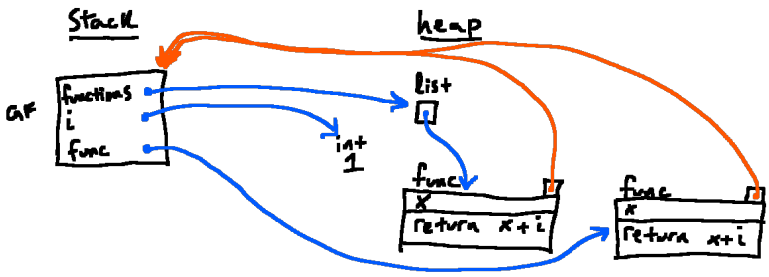



**STEP 5**

This might seem like a small thing, but now, in our second time through the loop, `i` has been rebound to `1` in the global frame. And because the `0` from before was left with no references pointing to it, it got garbage collected.

This will be a real problem for us later on, since the function we created never evaluated `i` to *find* that value of `0`; it simply stored away an instruction that, *when it is called*, it should look up `i` and add the associated value to its argument (it doesn't know that `i` *was* `0` when it was defined, because we never evaluated `i`)!

Let's keep going, though, to get a fuller picture of how things ultimately play out. Back inside the loop after we've set `i` to `1`, we again hit the function definition statement on lines 3-4; and nothing is different about how we respond to it: we're going to create a new function object holding the relevant information, and we will associate that function with the name `func` in the global frame (overwriting the previous binding). Take a moment to sketch out the changes to the diagram before moving on.




**STEP 6**

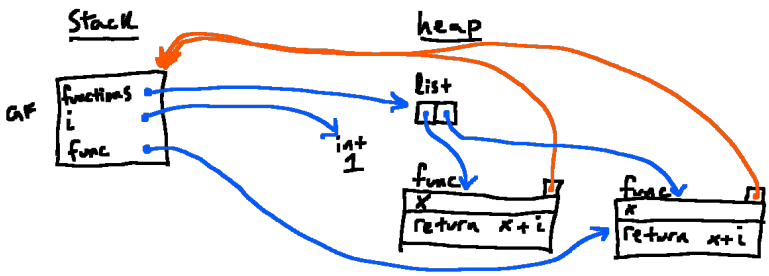
We now have a second function object, created during our second time through the loop. Everything about it (its parameters, its body, and its enclosing frame) is the same as the first function object we made, but this *is* a distinct object in memory despite those similarities. We

also rebound the name `func` to point to this new function object. Because our original function

object still has a reference pointing to it (from index `0` of the `functions` list), it will not be garbage collected.

The next step we'll take is `functions.append(func)`.

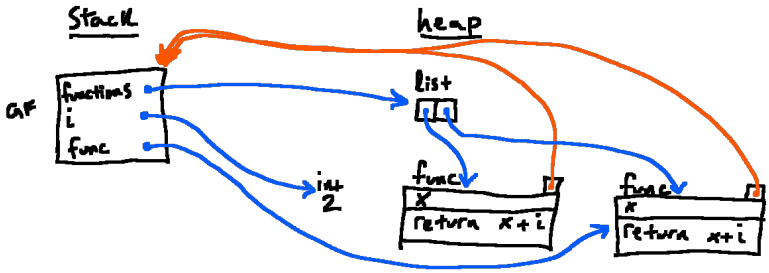
---




**STEP 7**

Once again, a small change. Our `functions` list is now *two* elements long; it contains a reference to each of the function objects we've created. We've reached the bottom of the `for` loop again, so it's time to go back to the top. Our next time through the loop, `i` will have a different value. Go ahead and update your diagram, and then click through.

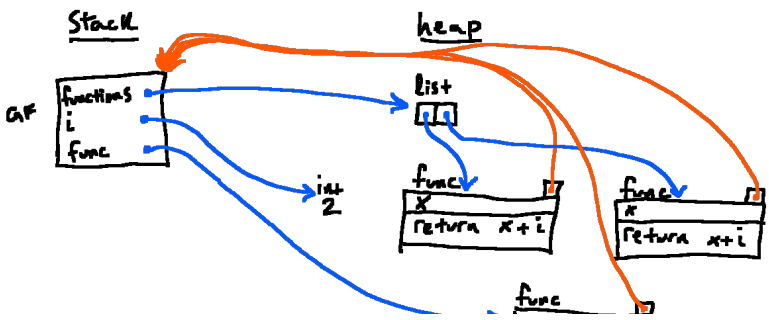
---




**STEP 8**

Again, we've rebound `i` in the global frame, and its old value has been garbage collected. Next, we'll hit the function definition again. Go ahead and update your diagram before moving on (I know it's starting to get a little busy/tedious, but bear with me!).

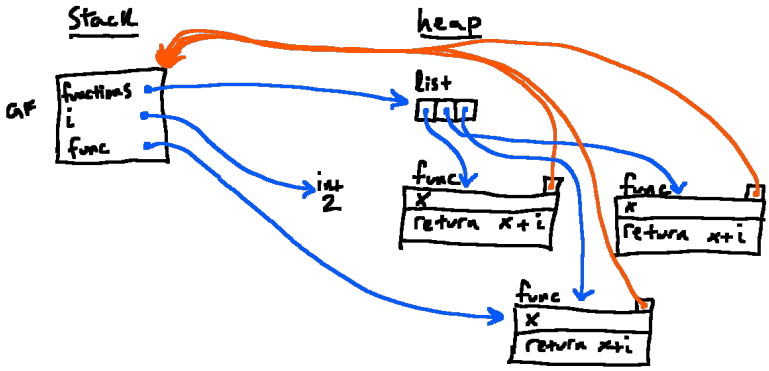
---




**STEP 9**

Once again, we get a brand-new function object containing exactly the same information as our other function objects, and we rebind the name `func` to point to it. Next, we'll append it to our list (click through to see the effect).

---

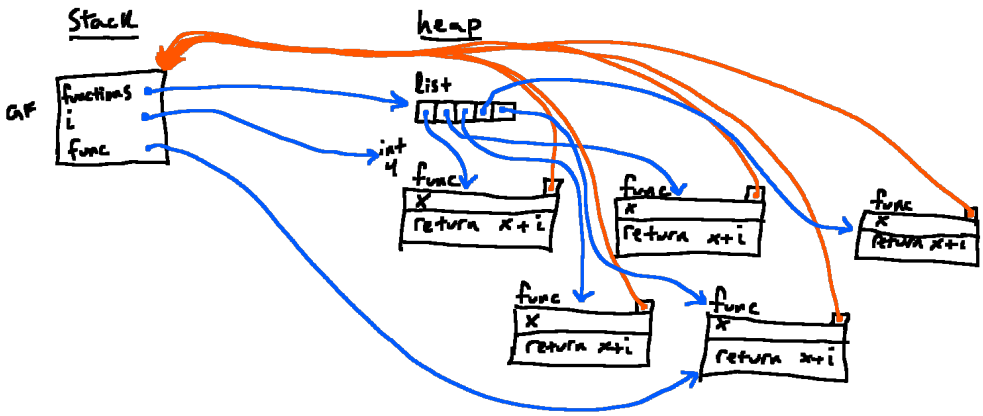



**STEP 10**

No surprises here, our list (called `functions` in the global frame) is now 3 elements long; it contains reference to three equivalent (but distinct) function objects.

At this point, we've maybe started to see a pattern emerge. So let's jump ahead to when we *exit* the `for` loop. Take a moment to think about what the diagram is going to look like before clicking ahead to the next step.

---




**STEP 11**

Here we are!!! Remember that we skipped ahead by a couple of times through the `for` loop's body, so this diagram represents the state of the program after we're done with the loop. This diagram is a little bit complicated, but we can summarize some of the key points that will affect our output:

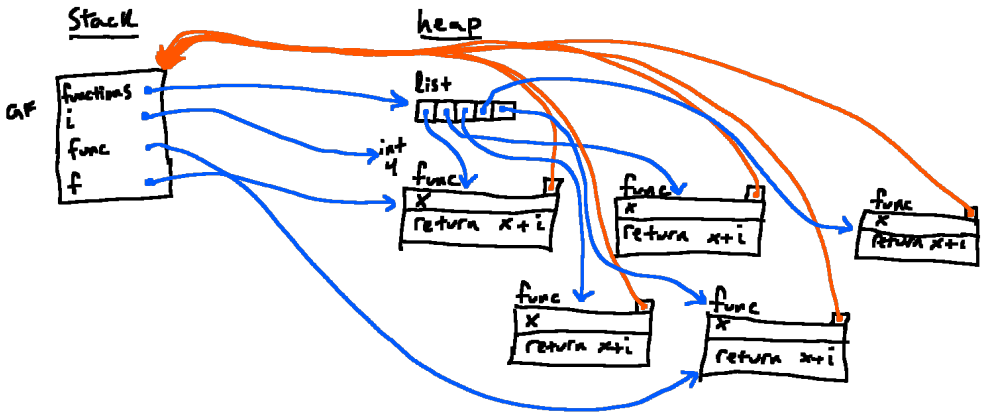
- the name `i` is bound to the value `4` in the global frame (this binding remains even after we

exit the loop)
- we have five distinct function objects in our `functions` list
- each of those functions has the same body, and executing that body involves looking up `i`

Thinking back to the rules we set up for calling functions earlier in the reading, can you see what the issue is going to be? We'll continue to step through the diagram, but this is a great place to **pause** before continuing. Do you understand all of the steps we took to this point? If not, it's worth going back (and reaching out for help if you're stuck!).

Once you've had a moment to reflect, we can continue on. We're just about ready to move into the bottom part of the code, the second `for` loop (`for f in functions`). The first thing that happens is that, before we run the body of the loop for the first time, we'll bind the name `f` to the first element from the thing we're looping over (the `functions` list in this case). Onward we go! Update your diagram and then go ahead and click through to the next step.

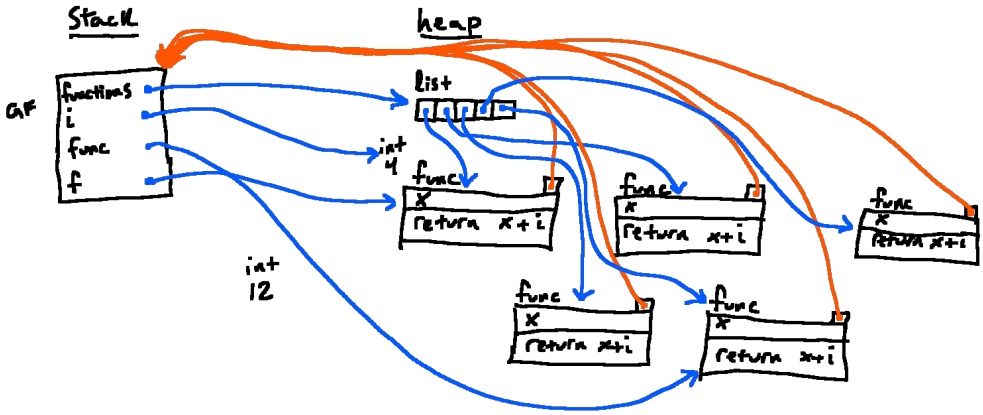
---




**STEP 12**

Here we see that we have bound `f` to the function object at index `0` in our list. Once that binding is made, we're ready to execute the body of the `for` loop. And the first thing we do there is to make a function call `f(12)`. We'll follow our normal steps here. Firstly, we'll evaluate the expression `f` to find out what function to call, and we'll evaluate `12` to find out what to pass to it. Clicking ahead will show the effect of this.

---

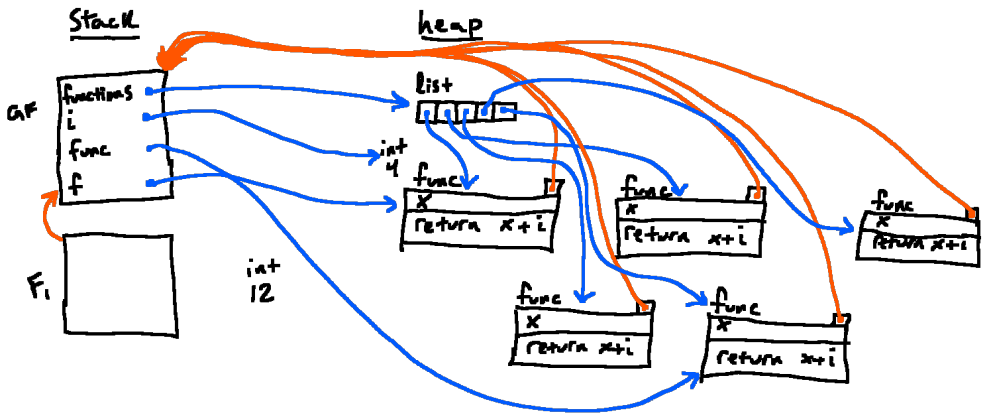



**STEP 13**

Now that we have figured out what function we're calling and what we're passing to it, the next

step is to set up our new frame (and set its parent pointer appropriately). Update your drawing to include a new frame and set its parent pointer to point to the enclosing frame of the function we're calling.

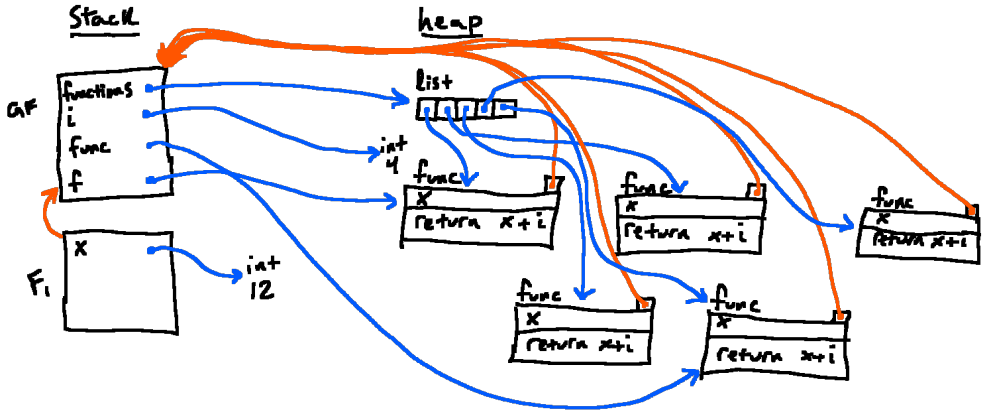
---




**STEP 14**

Here we have our new frame (we've labeled it as **F1**), and our next step is to bind the parameters of the function (in this case, a single parameter `x`) to the value that was passed in.

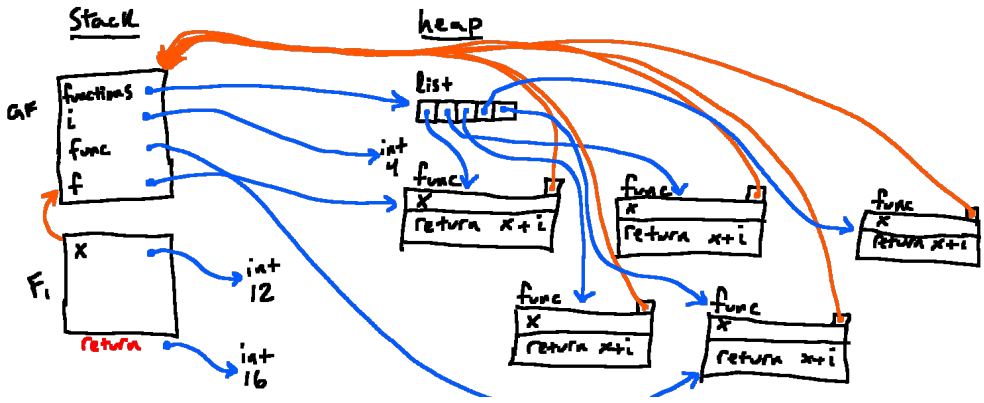
---




**STEP 15**

With that binding made, we're ready to run the body of the function inside of **F1**. Remember that the body of the function that we're calling is `return x + i`. As part of executing that return statement, we'll need to evaluate `x + i`, which involves looking up the names `x` and `i`. What value will we find for each? What will our function ultimately return?

---



**STEP 16**

Looking up `x` inside of **F1**, we find the `12` that is bound locally. But `i` is *not* bound locally. So what do we do? We follow the parent pointer, and we *do* find the name `i` in the global frame. It references a value of `4` for `i`, so that's what we'll use.

Then we add those two values together to get a new `int` object representing `16`, which we return.

We'll stop here with the diagram, but note that this result would have been the same regardless of which of these function objects we called. None of them remembers the value that `i` held when it was created; they all simply say to look up the *current* value of `i` and add it to their inputs! So as we continue to loop and call each of these function objects in turn, they all produce the same output!

## 6) Closures

Now that we've explained the interesting phenomenon from last week's reading, we'd like to wrap up by trying to fix it. Presumably, the person who wrote that code did not intend to see five `16`'s, but rather some number that is changing. The intent was to create five noticeably different functions: one that adds `0` to its input, one that adds `1` to its input, one that adds `2` to its input, and so on....

Before we can get there, though, we're going to introduce one more bit of terminology. This section is not about introducing a new *rule* for how function objects behave (we've already covered all of them, in fact!) but rather a powerful effect of those rules.

Importantly, a function object "remembers" the frame in which it was defined (its enclosing frame), so that, later, when the function is being called, it has access to the variables defined in that frame and can reference them from within its own body.

We call this combination of a function and its enclosing frame a **closure**, and it turns out to be a really useful structure, particularly when we define functions inside the bodies of other functions.

Let's explore this idea using examples, starting with the following piece of code:

```
x = 0

def outer():
    x = 1
    def inner():
        print('inner:', x)
    inner()
```

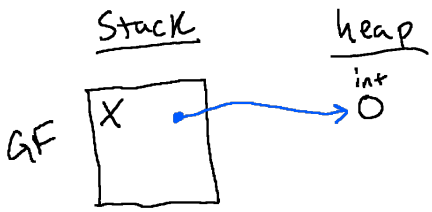
```
    print('outer:', x)

print('global:', x)
outer()
inner()
print('global:', x)
```

Notice that here we are defining a function, `inner`, *inside* of another function, `outer`. But it's worth mentioning up front -- and we'll see this play out concretely -- that Python follows exactly the same rules for defining functions and calling those functions in this situation, as it would in any other situation. Applying those rules when we have a function defined inside another function leads to some really interesting results and behaviors.

Show/Hide Line Numbers

```
x = 0

def outer():
    x = 1
    def inner():
        print('inner:',
x)
    inner()
    print('outer:', x)

print('global:', x)
outer()
inner()
print('global:', x)
```

<< First Step | < Previous Step | Next Step > | Last Step >>



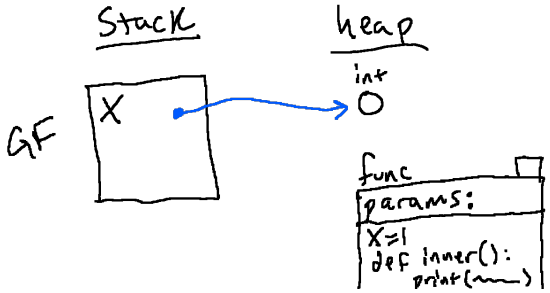

**STEP 1**

In this example, the first thing we hit is `x = 0`, we get an integer `0` out on the heap, and we associate `x` with that `0`.

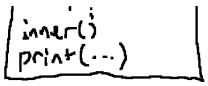
**STEP 2**

Then we hit this function definition, which encapsulates all the highlighted lines of code.

We're going to make a new function object, just the same way we do with any `def`. This is a function of no parameters, and its body is complicated, but we're not evaluating the body at this point in time.

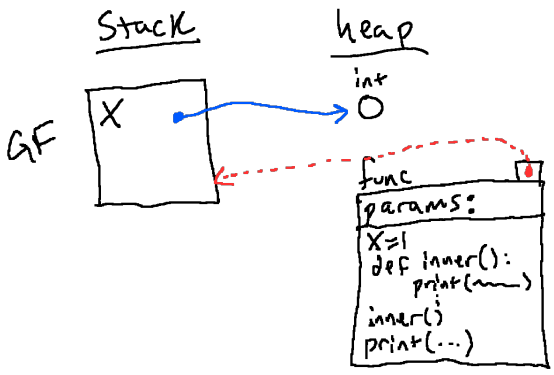



We've abbreviated its body a bit in the diagram (leaving out the details of the `print` statements).




This is important: we're defining the `outer` function but not evaluating any of the code inside it yet.

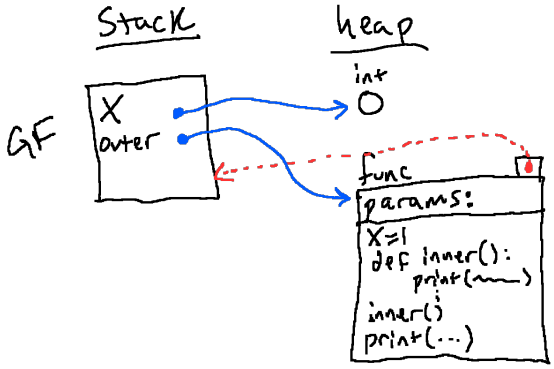
---




**STEP 3**

And this new function object will have an enclosing frame pointer that goes back to the global frame, because that's where we were running when we made this object.

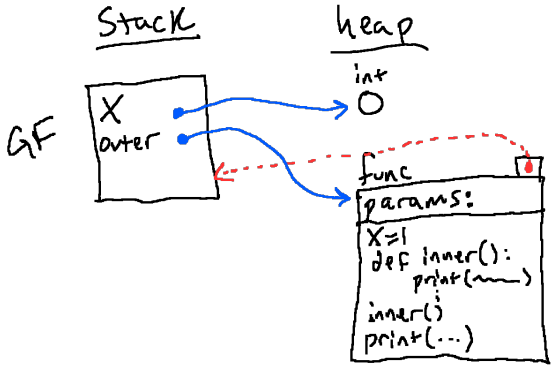
---




**STEP 4**

And `def` also associates a name with that function: `outer`.

---




**STEP 5**

Then we see this statement, which needs to evaluate and print `x`.

Since we're running in the global frame, `x`

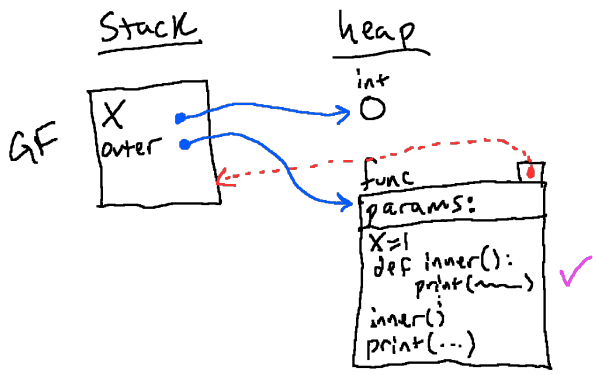
has value `0`.

(The `x=1` in the body of `outer` hasn't happened yet! And it wouldn't affect the global frame anyway, as we'll see when we get to it.)

So we print:

```
global: 0
```

---




**STEP 6**

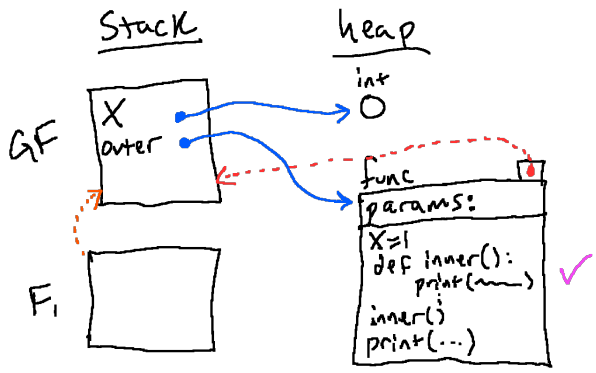
Now we're ready to call `outer`. Let's follow our usual process.

First we figure out which function we're going to call. We evaluate `outer` in the global frame, and that finds the function object we just created. Let's mark it with a purple checkmark to remember which function we're about to call.

Then we would also evaluate the arguments, but there are no

arguments in this case, so we just move on to the next step.

---



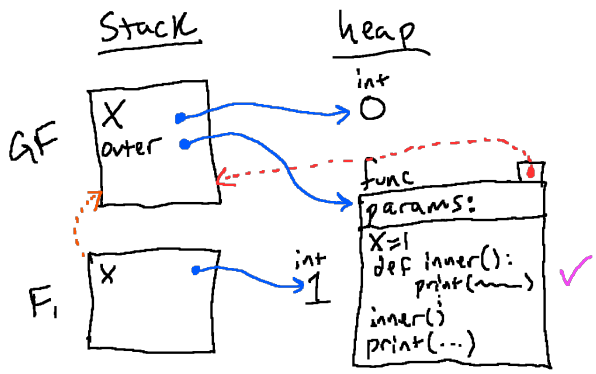

**STEP 7**

The next step is to create a new frame, which we'll call **F1**. The frame will have a parent pointer back to the global frame -- not because we were evaluating in the global frame,

remember, but because the *function object* we're calling has its enclosing frame pointing back to the global frame.

Now we bind our parameters to the arguments, but there are no parameters, so we're done with that step already.

---




**STEP 8**

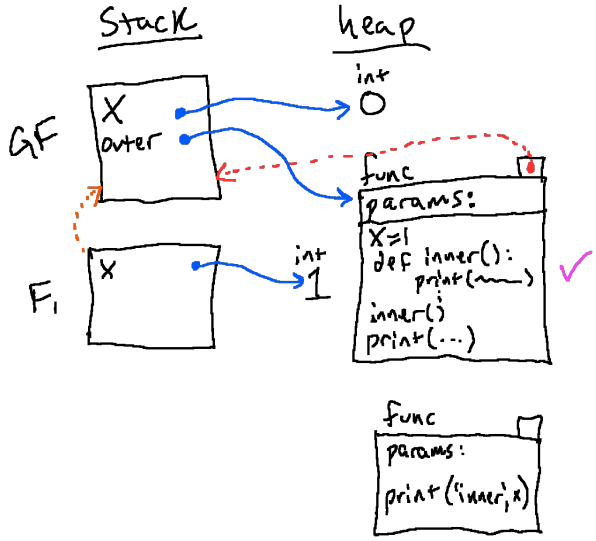
So we can move on to evaluate the body of the function now, inside the frame **F1**.

The first thing we hit is `x = 1`, so we evaluate that `1` and get a new integer and associate it with `x` inside of **F1**.

---

**STEP 9**

Then we hit `def inner`, and this is a function definition, so we follow the same steps for a function definition. We get a new function object out on the heap, again with no parameters, and its body is

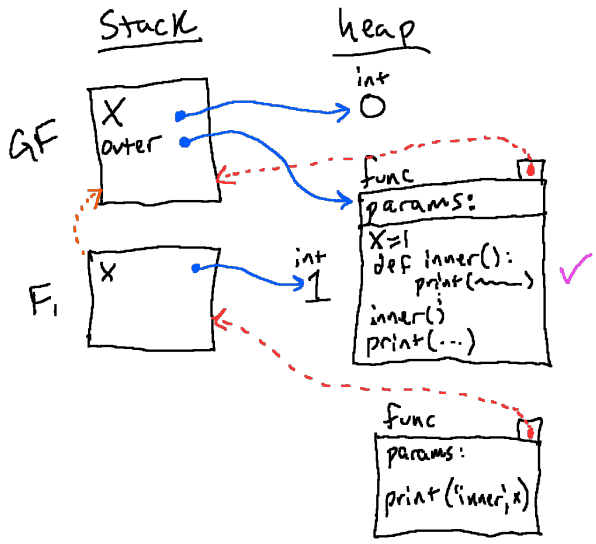



`print('inner:', x)`.

Check yourself: where is this function's enclosing frame going to

point?

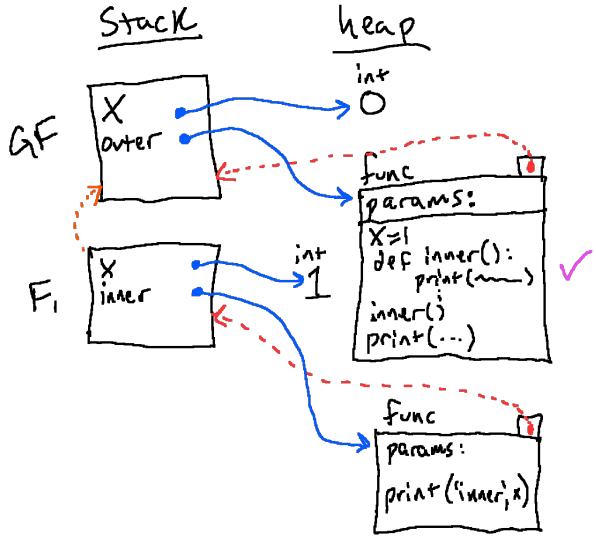
---




**STEP 10**

Remember that our rule is: what frame were we working in when we created this function object? We were evaluating this `def inner` inside of **F1**. So this function's enclosing frame is *not* the global frame, but **F1**.

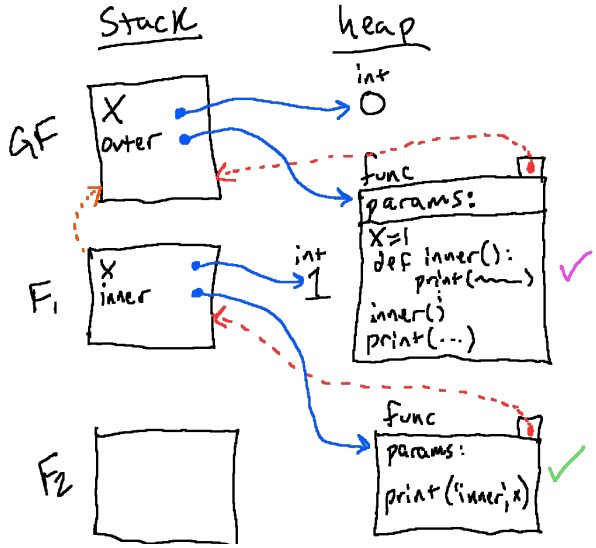
---




**STEP 11**

And finally we associate the variable `inner`, in the frame **F1** where we are evaluating, with that new function object.

---




**STEP 12**

Now we come down and hit this line, where we are going to call `inner()`.

We follow the same process we always follow. First we evaluate the function we want

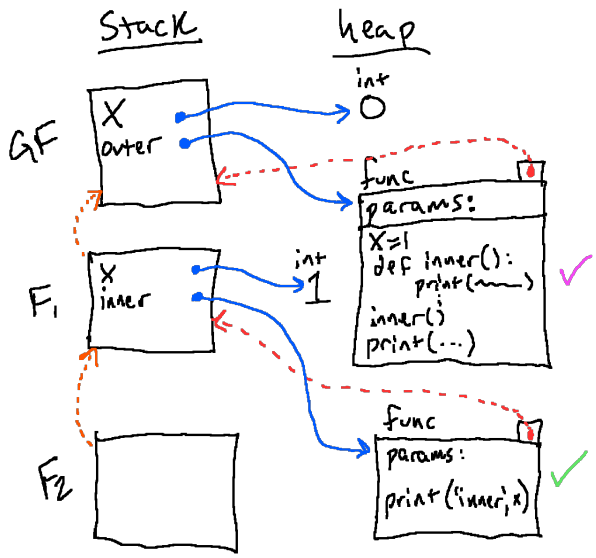
to call, by looking up the name `inner` and finding the function we just created -- let's mark it with a green checkmark to remember which one we're about to call.

Then evaluate the arguments -- but again, there are no arguments.

Now we create a new frame **F2**. This frame needs a parent pointer, so there's the question about where its parent pointer goes.

Check yourself before going to the next step: where will the parent pointer of **F2** point?

---

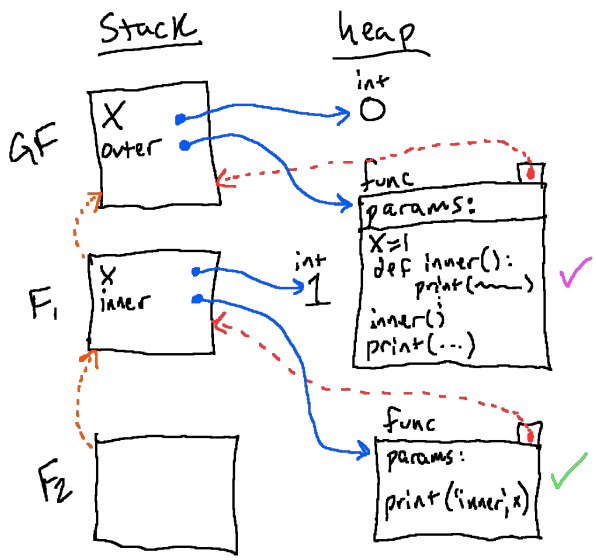



**STEP 13**

The important thing is which function we're calling -- the function object marked with a green check, in the lower right. That function's parent pointer goes back to **F1**. So that's why the new frame **F2** also points back to **F1**.

Next we would bind the names of the parameters to the arguments that were passed in, but there are no parameters here.

---




**STEP 14**

So we can just run the body of the function.

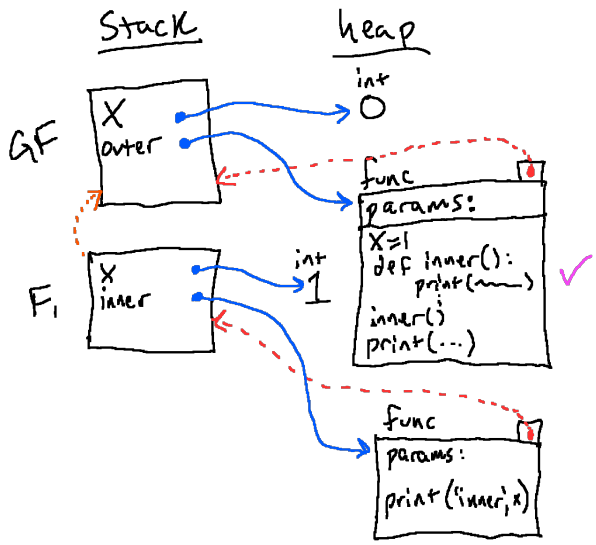
Running it inside of the new frame **F2**, we see

`print("inner:", x)`, so we look up `x` in **F2**. We don't find `x` in **F2**, so we trace up to its parent frame **F1** and find `x` bound to `1`. (Note that we don't reach the global frame, where `x` is 0. We have to follow parent pointers and stop at the first frame where `x` is defined.)

So we'll print:

```
inner: 1
```

---



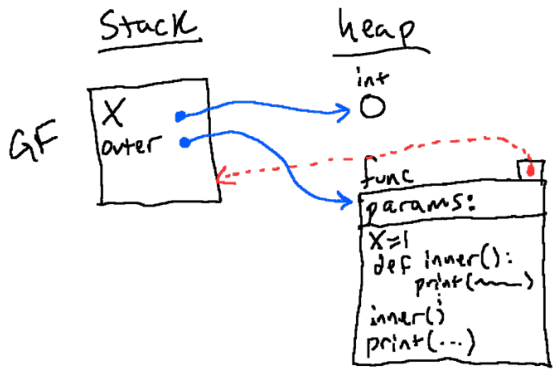

**STEP 15**

Okay, we're done with that function call to `inner()` that used **F2**, so we can garbage-collect it and come back up inside **F1**, back to evaluating the body of `outer` inside **F1**.

And now we evaluate `print('outer:', x)`. Looking up `x` finds the value `1`, so we print:

```
outer: 1
```

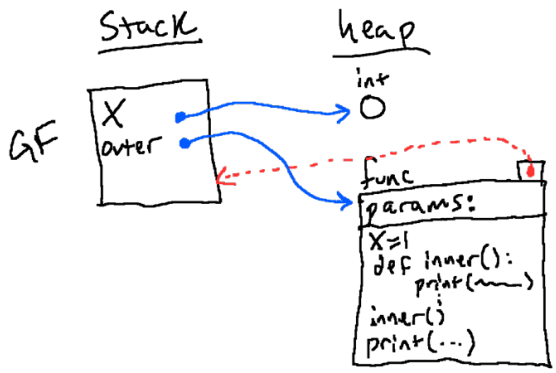
---




**STEP 16**

And now finally we are done with our call to `outer()`, so we can garbage-collect **F1** as well, along with the `inner` function object we created.

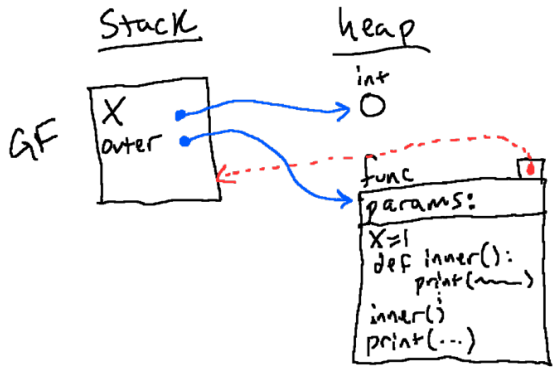
---




**STEP 17**

Let's then move on to the next line, which says to call `inner()`. But note that we are back to running in the global frame -- where it turns out there is nothing called `inner`! This will give us an error, a `NameError`, complaining that `inner` is not defined. Remember that `inner` existed only in frame **F1**, not in the global frame where we are now.

---



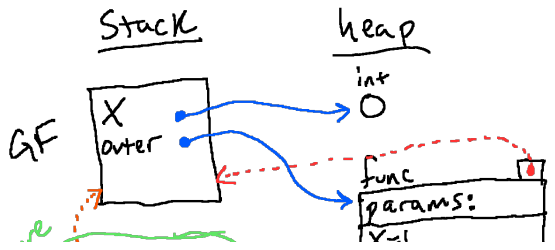

**STEP 18**

Suppose we ignore that bad call to `inner()` for now, and assume we move on to `print('global:', x)`, which will find the `x` in the global frame bound to 0 and print:

```
global: 0
```

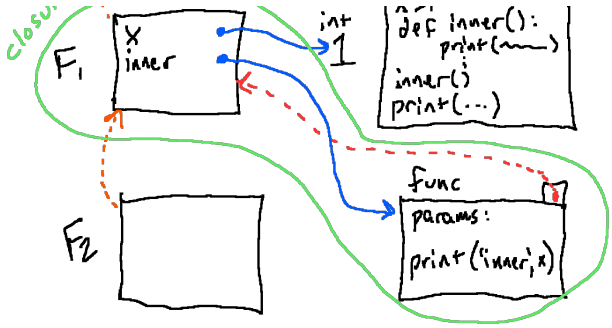
And we're done evaluating the program.

---




**STEP 19**

But there's some interesting behavior going on that we should




highlight, so let's back up to a middle step of this process, when all the frames and function objects still existed, and we were calling this `inner()` function.

When we created the `inner` function (marked with the green checkmark in the lower right), its parent reference went back to **F1**, the frame associated with calling `outer`.

So whenever we call this function, the frame that we make for that call will have **F1** as its parent. So this `inner` function, whenever we call it, has access to all the variables in that parent frame.

So it's useful to think about that `inner` function object not only as an object unto itself but as the *combination* of the function and the frame where it was created (its "enclosing environment") as a single entity, which we've circled in green here. Why? Because the function has access to all of these variables defined in the parent frame. That entity is what we refer to as a **closure** -- the combination of a function object and its enclosing environment.

The previous example demonstrated a closure but didn't really do anything interesting with it. The next example shows how we can *use* closures to get some interesting and useful behavior. Look at the code on the left as you step through the explanation:

Show/Hide Line Numbers

```
1 def add_n(n):
2     def
  inner(x):
3         return x
  + n
4     return inner
5
6 add1 = add_n(1)
7 add2 = add_n(2)
8
9 print(add2(3))
```

<< First Step | < Previous Step | Next Step > | Last Step >>

---

**STEP 1**

Maybe we can first try and predict what's going to happen here. (Hopefully the diagram is drawing itself in your mind, or you are following along with pencil and paper. We'll back up and draw the environment diagram in a moment.)

We've got a function `add_n`, which takes an argument `n`.

Inside it we've got a function `inner`.

```
print(add1(7))
print(add_n(8)(9))
```

And the *return value* of `add_n` is that function object `inner`. But the important thing is that the function object is going to remember the environment in which it was created. In that environment, `n` is bound to whatever value we passed to `add_n`.

So the function that is returned is going to remember the value of `n` that we provided when we created it.

---

**STEP 2**

So down here, when we call `add_n(1)`, it ends up binding `n` to `1` in a new frame and creating a function object that captures that frame. That function ends up bound to `add1`. So `add1` will be a function that adds 1 to whatever value we pass to it. (Remember the return value of `add_n` is a function, so what we're storing in `add1` is a function that we can call.)

---

**STEP 3**

And then separately we're making `add2`, which is a separate function that remembers `n` bound to 2, so it will be a function that adds 2 to whatever value we pass to it.

---

**STEP 4**

So down here, when we call `add2` with a `3` passed in, we should print `5`.

And on the next line, `add1(7)` should give us `8`, because `add1` is a function that adds `1` to whatever number we pass into it.

---

**STEP 5**

The syntax on the last line looks a little weird, so let's unpack it.

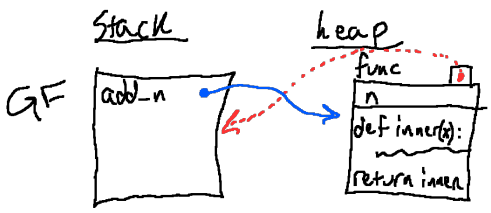
Remember that `add_n` called with some number gives us a function back, and we created two of those functions and stored them into `add1` and `add2`.

But it's equally valid *not* to store that function away with a name but to call it immediately. That's what we're doing on this last line -- `add_n(8)` gives us back a function (which adds 8 to whatever you give it), and then we immediately call that function with `9`, so we should get `17`.

---

**STEP 6**

Now let's go back to the beginning and talk through the environment diagram, at least for some of this program, so we can see how this closure

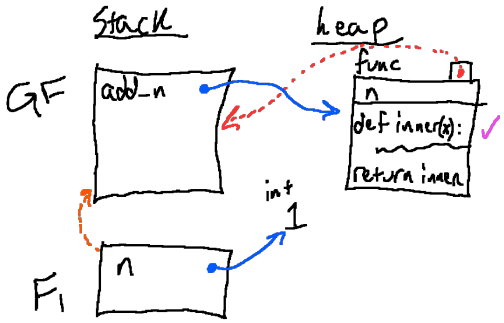



idea plays out in this example.

We start out defining `add_n`, a function of a single parameter `n`, whose enclosing frame is the global frame, and we bind it to `add_n` in the global frame.

The body of this function, importantly, is the entire body of `add_n`, shown here.

---




**STEP 7**

On this line, we're going to call `add_n` with the argument `1`, and whatever it returns, we're going to associate with the name `add1`.

We evaluate `add_n` to find the function object we're going to call, which we've marked with a purple check.

We evaluate the argument to get the integer `1`.

We create a new frame **F1** and bind the parameter `n` to the argument `1`.

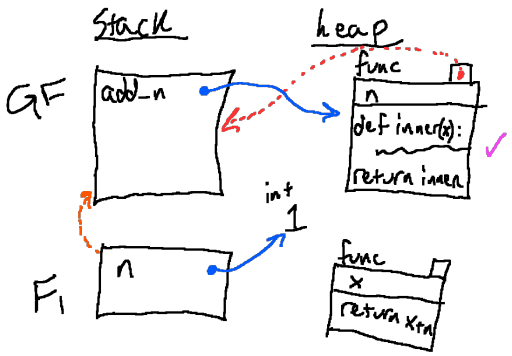
And finally, the parent of that new frame **F1** is the global frame, because that was the enclosing frame of the function object.

---

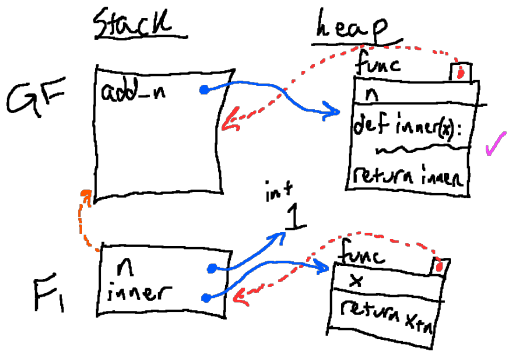
**STEP 8**

Then we run the body of `add_n` inside of **F1**. The first thing we hit is `def inner(x)`, so we create a new function object, with parameter `x`, and `return x+n` as its body.

Now this function object has an enclosing frame --- where does its enclosing frame pointer go?




---

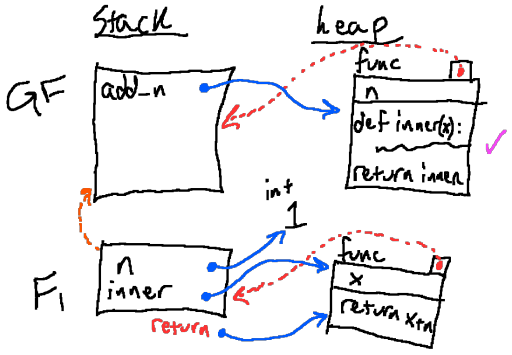



**STEP 9**

The parent reference goes to **F1**, because we evaluated this `def` instead of `F1`.

And then we associate the name `inner` with that function object.

---



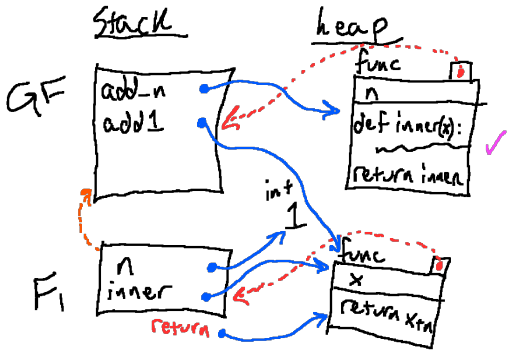

**STEP 10**

Now the last thing we do in the body of `add_n` is `return inner`. We'll denote the return value by pointing over at the function object we just created.

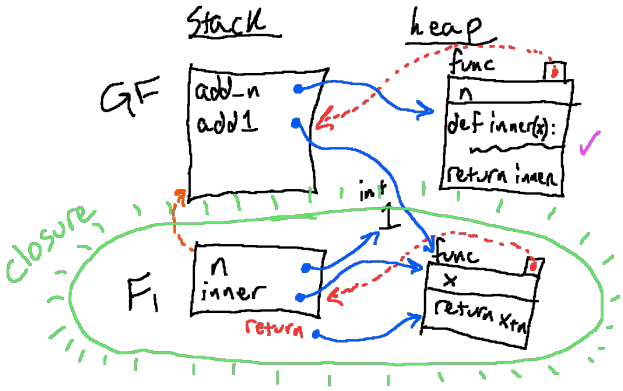
---

**STEP 11**

Now that we have the return value of `add_n(1)` -- that function object -- we bind it to `add1` in the global frame.




---




**STEP 12**

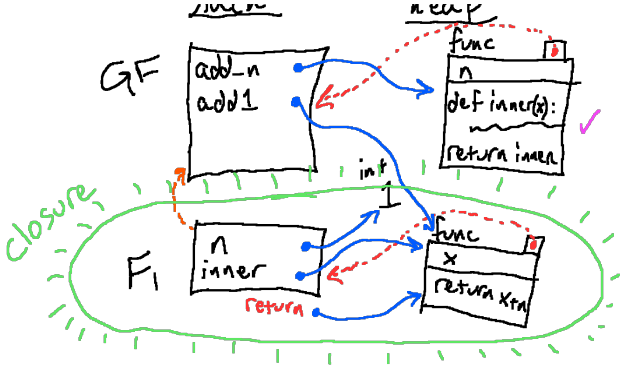
It's worth mentioning here that normally, in previous diagrams we've drawn, when we're done with and returned from a function call like `add_n`, that function call and its frame go away (and a bunch of stuff gets garbage-collected).

But here it turns out we can't do that! We can't get rid of **F1**. Why not? Because, from the global frame, we have a way to get to that function object (`add1` is pointing to it), and that function object relies on the frame **F1**. It still needs the values inside the frame, so the frame can't be thrown away (and Python won't throw it away).

That whole combination --- the function object and its enclosing environment, circled in green --- we refer to as a closure, and we can think of it as a single entity, a function that can rely on variables defined in that enclosing frame.

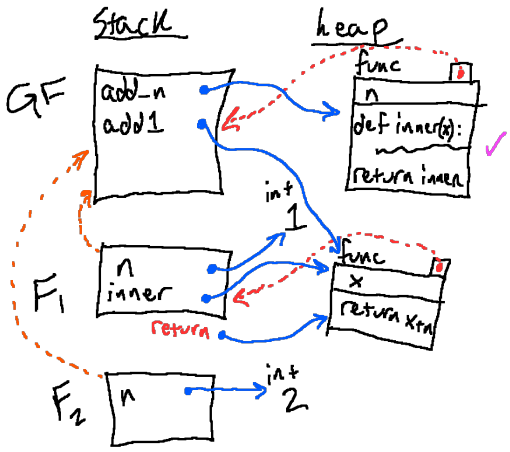
---

**STEP 13**




And it's also worth noting that when you call `add1` -- meaning the function object, since `add1` in the global frame points to that function object -- when we call that function object, it has access to the fact that `n` was bound to `1` when we created the function. So every time we call that function and run its body `return x+n`, it's going to see `1` for the value of `n`. Which is why this function will always add `1` to the value passed into it.

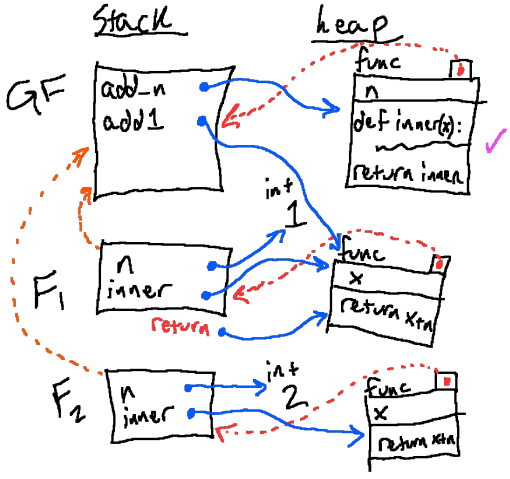
---




**STEP 14**

Now we call `add_n(2)`, so we'll get a new frame **F2** with `n` bound to `2`, and its parent pointer will point to the global frame.

---




**STEP 15**

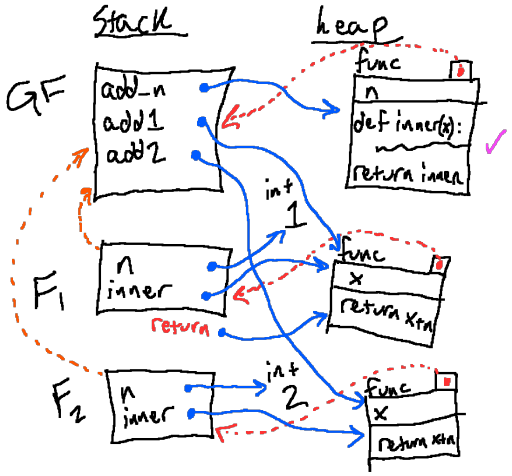
We'll evaluate the body of `add_n` again, starting with `def inner(x)` again, which creates a fresh function object bound to `inner`. But this function object's enclosing frame goes back to **F2**, in which `n` is bound to `2`. So this is a function that will always add `2` to the value passed to it.

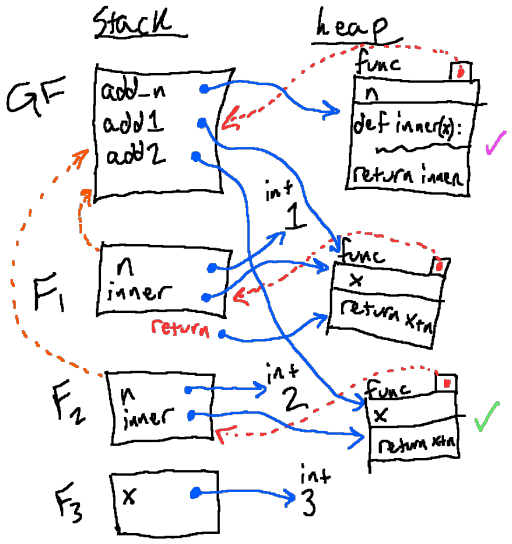
Then we'll return this new function object.

**STEP 16**

And now we have bound `add2` in the global frame to this second function object.

With the diagram in this state, we can see how it's useful to bind a function to its enclosing frame as a single entity. These functions that we're now calling `add1` and `add2` (in the global frame) *do different things*. One adds 1 to what we pass in; the other adds 2. But if we look just at their function objects themselves -- parameters and bodies -- they look the same! The thing that differentiates them is their enclosing frames, which are binding different values for `n`.



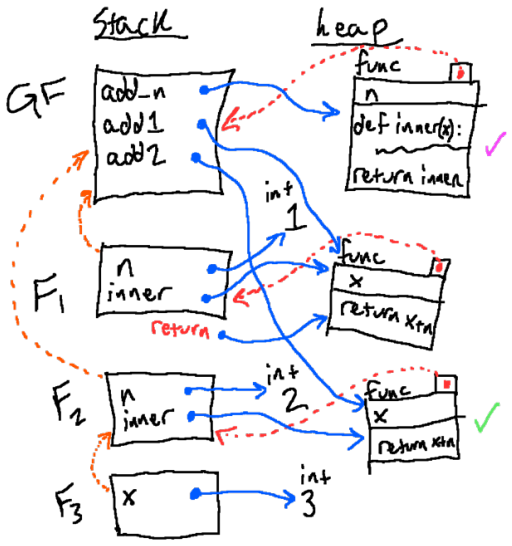

**STEP 17**

We won't work through all the remaining lines of code, but let's at least look at how calling `add2` works, with `3` passed as its argument.

We start by evaluating `add2` in the global frame (where we're running this line) and find the function object at the bottom right, marked with a green checkmark.

Then we evaluate the argument (`3`), create a new frame (**F3**), and bind the parameter `x` to the argument `3`.

Where does the new frame's parent point?




**STEP 18**

Remember that our rule is that the new frame's parent comes from the function object's enclosing frame -- the frame that the function was defined in. Because the function we're calling was defined in **F2**, **F3** points to **F2**.

This is important! This is where it will get the right value of `n`. Because now we execute the body of the function object, `return x + n`, and it finds that `x` is bound to `3` (in frame **F3**), that `n` is `2` (in frame **F2**), and returns `5`.

## 7) Fixing the Mysterious Code

Using this idea of a closure, we can fix the mystery code from earlier! The code below resolves the issue (at least insofar as preventing the output from all of the functions from being identical!) by *evaluating* `i` each time through the loop and setting up a *closure* for each of the functions we add to the `functions` list, such that, when each is called, it has access to a variable storing the value that `i` had when that function was created.

Show/Hide Line Numbers

```
1 def make_adder(n):
2     return lambda x: x + n
3
4 functions = []
5 for i in range(5):
6     functions.append(make_adder(i))
7
8 for f in functions:
9     print(f(12))
```

When this program is run, what will it print out?

## 8) Summary

We've covered a lot of ground in this reading, and we've talked about a few different things, but our main focus throughout was on functions. We started by seeing a couple of examples of how we can use functions as an organizational tool, by building up meaningful abstractions for ourselves.

Next, we went down into the weeds a little bit, going into detail about the rules that function objects follow, including discussing both what happens when we *define* a function and what happens when we *call* that function. To summarize some of the key rules:

- When we define a function, the new function object's enclosing frame is the frame where it was defined
- The body of a function is not evaluated when it is defined, only when it is called
- When we call a function, the new frame's parent is the enclosing frame of the function object
- Functions are first-class objects, and you can pass them as arguments to other functions, return them as results, and bind variable names to them, just like other data types

Finally, we did pull back a little bit and look at some of the interesting behaviors that follow from these rules, as well as how we can take advantage of these behaviors in our own programs.