When Python execute a program, different expressions can be evaluated in different environments . So there can actually be multiple environments in the same environment diagram . we’re going to look at one of the examples that I showed you last time in detail with diagrams , so that we know exactly what’s going on, but first let’s review everything that I told you before about user-defined functions.

Here on one slide in summary form with all the vocabulary that you need .

So, first we have a def statement, a def statement is what creates functions . it looks like this that **whole thing is a def statement** which spans multiple lines .

It has a name for the function you’re defining , a formal parameter which is the name you give to the argument of the function, there can be more than one formal parameter in which case they’re separated by commas one for each argument. The body of the def statement is everything indented after the first line , and in this case, it’s just a single return statement , though it is possible to have more than one line , this return statement has a return expression which multiplies x and x together.

When a def statement is executed, a new function is created , the name which we see here is square is bound to that function in the current frame .Now when I’ve defined this function that squares things, I still haven’t actually multiplied anything by anything else because I haven’t called that function yet, that happens with a call expression

So later on in my program I might say square two plus two that’s a call expression the operator is the name square it’s value is the function that squares, the one that we just defined, there’s an operand in between parentheses , an operand is an expression in this case two plus two. It evaluates to a value 4 which is the argument of the function.

So when we reach a call expression

We evaluate it by evaluating the operators and operands and then calling the function on the arguments . calling or applying a user-defined function is also a process that we need to spell out

So I showed you a diagram that looked like this which was the function signature inside of a little tube, in comes the argument , outcomes the return value

How does this happen, well we start by creating a new frame in which the formal parameters of the function we’re calling are bound to the arguments for passing it, in this case x would be bound to 4 and then the body of the function that we’re calling is executed in that new environment. So we would in this case compute the multiplication of x and x where x is bound to 4, 4 times 4 is 16, so that’s the simplest possible example, a slightly more complicated one is when we import mul we define square just as we did before, but now we’re going to square the square of 3 .

This diagram indicates that we’ve already executed the Def statement which bound the name Square to newly created square function

And we’re about to square the square of 3, how do we square the square of 3, it’s a call expresion , so we just use our rule for evaluating call expressions which means the operator is evaluated , it means the function that squares , the operand is also a call expression, so we need to apply that procedure again.

Squares the function that squares 3 is the number 3, and now we can apply our user-defined function square to the number 3, how do we do that, well,

It has there steps :

First : I creat a new frame then I bind the formal parameter x to the argument value 3 and there’s the binding right there ,

Finally I execute the body of the function which in this case has returned mul x, x. so I have to multiply x and x together I get a return value of 9 and that’s how I get the value of this call expression , now that I know what the value of this operand expression is, I can apply this function square to the argument 9 and repeat the process again . so that means introducing a frame binding the formal parameter x to the argument value 9 and then executing the body which multiplies 9 times 9 and gives me 81 . so let’s look around here for a second. We have one square function. We created two frames from that function by calling the same function twice . Each of those frames is different which is why we gave it a different frame label f1 or f2 it’s also different because we passed in different arguments so we got different bindings from the formal parameter to the argument and that led to different return values.

An environment is a sequence of frames.

So far we’ve had an environment which was just the global frame alone, we had those before we ever even had def statements

But once we started calling user-defined functions we started getting multiframe environment ones that have a local frame and then a global frame

Let’s try to find all the different environments in this diagram, there’s one just the global frame alone. There’s another f1 followed by the global frame and there’s a third the frame f2 followed by the global frame so there’s three different environments here None of them include all three frames but there’s one environment per frame if you start with a particular frame you can always find the whole environment just by following the parents of the frames. So let’s say we’re interested in the environment that starts with the frame f2 . well we know the next frame is the parent of this frame which is the global frame , says right there, global frames don’t have any parents , they’re always the last frame in an environment in very important point is that names have no meaning without these environments these are the things that endow mall and x and square with some sort of meaning some value

every expression is evaluated in the context of an environment which allows us to figure out what names mean and a name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found,

in this case when we evaluate mul(x,x) for the second time, it’s in an environment that starts with the frame f2 followed by the global frame and we have two different names we have to look up , mul and x, so we look up x the first thing we do is look in the first frame of the current environment and we find x there so this is the earliest frame of the environment in which x is found, so x is 9, when we look up mul we first look at f2 to see if mul is there, it’s not so then we look in the next frame of the environment

and the low and behold, there is mul bound to the function that multiplies , so that’s another case where we found the name in the earliest frame of the current environment in which that name was found , it just happened to be the global frame but we did check f2 first

Names can have different meanings in different environments because each frame can have a different binding for the same name . In particular, a call expression and the body of the function being called are evaluated in different environments.

Here’s an example where we use the name square for both the name of a function and the name of the formal parameter. I’m not recommending you do this, but if you square 4 in this case, you really do get 16, why is that, well in the environment diagram for this example, there’s a global frame in which square is bound to the squaring function, there’s a local frame in which square is bound to 4 because we call the squaring function on 4 , the difference is that when we evaluate square 4 operator, we’re evaluating that expression in the global frame that’s the call expression , it’s not indented at all which indicates that it gets evaluated in the global frame ,

on the other hand, this line is indented,it’s part of the body of square function and so it’s always going to be executed in an environment that starts with a square frame because we create this frame and then we execute the body , so this square is evaluated an environment that starts with f1 followed by the global frame but when we go looking for what square means we always look in f1 first , there it is and we found 4, so we never find this binding because we’re only interested in the earliest frames of the current environment