As you know if I type an expression into the interactive Python interpreter it will then display the value of that expression . It’s also the case that if I call print on the value negative two . I’ll see the same output , but two different things have happened here. Let’s try to understand the difference . so if I type ‘Go Bears’, I see ‘GoBears’ is the same as exactly the expression that I typed in because this is a string literal and so we see the string value, if on the other hand I print(‘Go Bears’), let’s see almost the same thing but no quotation marks, so there must be a difference between printing and just evaluating something

We can’t see more of this difference when we look at the special value called None

None represents nothing if , if I just evaluate it I see nothing ,

If I print it out I actually see None

So what’s going on here is that Python has rules for automatically displaying the value of any expression you type in,

You type in so automatically displays this negative two or this Go Bears

None is a special case for nothing gets displayed automatically but if I print it I can make it appear

Okay, what else can print do, well print can print multiple values separated by spaces and it can print None, so I could print

Here’s an interesting case what happens if I nest calls to print

Spend a moment thinking about what will happen and then I’ll show you in three, two, one

Let’s understand exactly what happened there , first what is None, None indicates that nothing has been returned from some function

so the special value None is just there to represent nothing

In Python it’s called None, in other languages it has other names

A function that does not explicitly return a value will return None

Careful: None is not displayed by the interpreter as the value of an expression

If I try to define a function does\_not\_square(x) which just computes x times x but doesn’t return it , then

I’ve created a function that returns nothing because there’s no return there

So when I call does\_not\_square(4), I don’t get 16 back instead I get None

However according to this third rule : None is not displayed by the interpreter as the value of an expression

Here’s a call expression (does\_not\_square(4)). If I evaluate it 4 does get multiplied by four but it doesn’t get returned what gets returned is None and None doesn’t appear at all

Next I could say sixteen equals does\_not\_square(4)

What’s going to happen here is we’ll compute the value of this call expression

It will evaluate to None which will be bound to the name sixteen,

So the name sixteen is now bound to the value None . If I just type in sixteen nothing will be displayed

If I type in sixteen plus 4 I will not get 20, I will not get a type 20, I will get a type error that says : unsupported operand type(s) for +: ‘NoneType’ and ‘int’ .

What does that mean? Well , this plus is the same as that plus and what it’s saying is that I tried to add together a None and an int that stands for integer, 4 is an integer, so this kind of error might arise when we’ve tried to add something to nothing which you cannot do

Okay, back to print , well it turns out there are two kinds of functions : pure functions that just return values and Non-pure Functions that have side effects .

A pure function is something like **abs** which computes the absolute value of its argument . The absolute value of negative two is two and the only thing that **abs** does is return the number two whenever you call it on negative two that’s the argument that’s the return value

Here’s a picture of a pure function, it’s a closed pipe that goes from inputs to outputs, **Pow** is another pure function I pass in two arguments , it get out one return value

**Non-Pure Functions** are completely different, they take in some input , they also have some output . In this case, print has the output None or the return value of None. But in addition instead of being a closed system, it also has a side effect that’s what’s coming out of here , is that in addition to returning None, print displays whatever was passed into it, so in this case we passed in negative two is an argument, it displays the output negative two, the return value is None. The side effect isn’t a value at all, it’s just something that happens its behavior , that’s a consequence of calling the function, so print is a different thing then abs or pow because it has side effects and that’s why we get such interesting behavior when we nest print calls withing each other .

So let’s look at that print (print(1), print(2)) nested expression again where what we saw afterwards was one two and then None None, we can’t understand what’s going on here and we can do it using an expression tree,

So in order to get the value of this nested call expression

We first evaluate the operator which is a function that prints , the first operand is another call expression which we evaluate by evaluating its operator and its upper end now we have the function print applied to one when we actually apply print to th number one we have a side effect of displaying one that’s where this first one came from it was while we were evaluating this operand sub expression we got a walk displayed but it’s not the value of anything it just appeared there because that’s what print does is it makes things appear now we got a value for this expression as well which was None because that’s what print always returns next we evaluate this operand which is itself a call expression , an applying print to the number 2 displays the number two which we see here and the value of that sub expression is None, so now we have print applied to None and None

When print to None and None, a side effect of that is to display None, None which we see on this line and eventually we get the value for the whole thing which is None output here that None doesn’t get displayed because the interactive interpreter for Python doesn’t automatically display None when it’s the value of an expression typed at the prompt