Readings for Unit 4

Back to Top

The questions below are due on Friday July 14, 2023; 10:00:00 PM.

Licensing Information

Portions of these readings were modified or copied verbatim from the very nice book *Think Python 2e* by Allen Downey.

PDF of these readings also available to download: 6s090_reading4.pdf

Table of Contents

1) Introduction

So far, we have learned about a number of useful tools in Python. We have learned about:

- several types of Python *objects* that allow us to represent a variety of things in Python's memory (the types we have seen so far include numbers, strings, Booleans, the special `None` value, and compound objects like lists, tuples, dictionaries and sets); and
- several *control flow mechanisms* that allow us control over the order in which statements in a Python program are executed (the control flow mechanisms we have seen so far include `if/elif/else`, and `while` and `for` loops).

These are some really powerful tools, and, as you've seen through the exercises, they are enough to accomplish a wide variety of things![1] However, this week, we'll learn more about an incredibly powerful means of abstraction that will help you manage the complexity as the programs you write become more and more complicated: *functions*. Back to Top

Let's start our discussion of functions by considering the following code, which is designed to compute the result of evaluating a polynomial (represented as a list of coefficients) at a particular numerical value:

```
coeffs = [7, 9, 2, 3] # represents 7 + 9x + 2x^2 + 3x^3
x = 4.2

result = 0
for index in range(len(coeffs)):
   result = result + coeffs[index] * x ** index
print(result)
```

This is a nice piece of code in the context of a program that requires evaluating a polynomial, but in some sense it isn't as useful as it could be, in that it works only for the values of `coeffs` and `x` given above.

It is possible, however, to imagine a larger program that requires evaluating *several* polynomials. With the tools we have available to us so far, if we wanted to use the code above to this end, we would have to copy the code and paste it to new locations several times. Depending on what our program is doing, we might need to change the variable names `coeffs`, `x`, and `result` to prevent them from overwriting values we had computed for other polynomials. This is a pain! Not to mention, if we find a bug in our implementation, we would have to go back and fix it *in every copy-pasted copy we made of this code*!

It would be nice to be able to *generalize the notion of this computation* so that we could perform it on arbitrary inputs as part of a larger program. It turns out that a *function*, a type of Python object, is a great way to do this!

Functions are arguably the most powerful tool any programmer can have in their toolkit, but it can be a bit complicated to keep track of how Python handles them. As such, we're going to introduce relatively few new topics in this section, so that we can focus on functions, on how Python goes about evaluating code inside a function, and on how they can be used to increase the modularity of the programs we write.

2) Functions

A *function* is a type of Python object that represents an abstract computation. It can help to think of a function as a little program unto itself, which performs a specific task. Internally, that's what a function really *is*: it is a generalized sequence of statements that Python can evaluate to compute a result. This is perhaps not the most elegant definition in the world, so let's move on by way of example.

Python comes with several functions built in, and, in fact, we have already seen several examples of functions in Python. For example, we already learned about `len`, which computes the length of an input sequence. We could use `len` inside of a program, for example, by evaluating the following expression: `len("twine")`

In this example, the name of the function we are working with is `len`. The parentheses indicate that we want to "call" the function[2], which means to evaluate the sequence of statements it represents. The expression inside the parentheses (here, `"twine"`) is called an *argument*[3] to the function. In this case, the result is an integer representing the length of the argument.

It is common to say that a function "takes" one or more arguments as input and "returns" a result. This result is also called the *return value*. In this case, `len` is a `function` object, and the result of calling it is an `int`. But, importantly, functions can return values of *any* type![4]

We can treat the result of calling a function the same way we would treat any other Python object. For example, we could:

Back to Top

```
print(len("twine"))  # print the result
x = len("yarn")  # store the result in a variable
y = len("thread") + 27/2  # combine the result with other operations.
```

2.1) Multiple Arguments

Some functions take more than one argument. To specify this, we separate arguments with commas inside the parentheses associated with a function call. For example, consider Python's built-in `round` function, which can take two arguments:

```
print(round(3.14159, 2))  # round returns a number; this will print 3.14
```

> **Try Now:**
>
> This is something of an aside, but it is worth mentioning that Python provides documentation for all of its built-ins, which can be very helpful when determining how to use functions from Python. For example, see the section on `round`.

2.1.1) Alternate Forms of Previous Functions

It turns out that two of the functions we have already dealt with have alternate forms that take more than one argument, which may be useful in your programs moving forward:

- If `print` is given more than one argument, it will print all of its arguments on the same line, separated by spaces. If it is given *no* arguments (i.e., `print()`), it will simply make a blank line.
- `range` has three forms:
  - If `range` is given a single integer x, the object it returns contains the numbers from 0 to $x - 1$, inclusive.
  - If `range` is given *two* integers x and y, the object it returns contains the numbers from x to $y - 1$, inclusive.
  - If `range` is given *three* integers x, y, and z, the object it returns contains all values $x + i \times z$ such that $x \leq x + i \times z < y$, in increasing order of i. We can think of z as the step size that we take to go from x to y. This form is commonly used to iterate backwards: `range(10, 0, -1)`.

> **Try Now:**
>
> Experiment with these various forms to get a sense of how they behave. Try printing multiple values on a single line. Try printing several ranges. Since `range` does not return a list (but, rather, a special `range` object), you need to convert that object to a list or tuple to see the objects inside of it (for example, `list(range(9))` or `tuple(range(1, 4))`).

Try Now:

As we see above, `print` is a function, as well! What is the return value of `print`? Try storing the result of a caBack to Top
`print` in a variable, and then displaying it (again with `print`!).

Show/Hide

Consider, for example, the following code:

```
x = print(7)
print(x)
```

The first line stores the result of calling `print(7)` in a variable `x`. Then, on the next line, when we print `x`, and we see the following: `None`. So `print` will display its arguments to the screen, but then it will return `None`.

`print` was our first example of an *impure* function (or, said another way, a function with "side effects"). Not only did it return `None`, but it also had the effect of displaying something to the screen.

What will the following piece of code print?

```
print(print(print(10)))
```

Show/Hide

In evaluating this code, the first function call Python actually evaluates is the inner-most `print(10)`. This will display a `10` to the screen, and will return `None`. So after evaluating that expression, we are left with: `print(print(None))`.

Python then evaluates the inner-most `print(None)`. This will display `None` to the screen and will *also* return `None`. After evaluating that expression, we are left with: `print(None)`, which displays yet another `None` to the screen.

All things considered, this code will have printed:

```
10
None
None
```

2.2) Substitution Model

Before we can go too much farther, we need to think about how Python evaluates functions in our substitution model. In order

to evaluate a function call, Python takes the following steps:

- Looks up the value to the left of the parentheses
- Evaluates each of the arguments from left to right
- Calls the function with the results of evaluating the arguments

Back to Top

Try Now:

Use the substitution model to predict the result of evaluating the following expression:

```
round(985.654321 + 2.0, len("ca" + "ts"))
```

Show/Hide

Python starts by evaluating `round` to find the built-in function. Then it moves on to evaluating the arguments, from left to right.

The first argument evaluates as we might expect: `985.654321 + 2.0` becomes `987.654321`. So after this evaluation, our overall expression looks like:

```
round(987.654321, len("ca" + "ts"))
```

In order to evaluate the second argument, we need to evaluate *another* function call! Here, Python looks up `len` and finds the built-in function, and then moves on to evaluating the arguments to `len`.

There is only one argument to `len`, the result of concatenating `"ca"` and `"ts"`, which makes our overall expression:

```
round(987.654321, len("cats"))
```

Still in the process of evaluating the second argument to `round`, Python *calls* `len` and replaces the function call with its return value:

```
round(987.654321, 4)
```

Finally Python can call `round` on these two arguments, which gives us:

```
987.6543
```

Try Now:

What happens when you try to run the following piece of code? Why did that happen?

Back to Top

```
x = 2
z = x(3.0 + 4.0)
print(z)
```

Show/Hide

If you ran the code exactly as it is typed above, you will have seen an error message: `TypeError: 'int' object is not callable`. In typical Pyton fashion, this is perhaps a little bit obtuse. But what Python is trying to say is: you tried to *call* something by using parentheses, but instead of being a function, the thing you were trying to call was actually an `int`. Since Python doesn't know what it means to call an `int`, we see this error.

So how did Python get to that point? Let's use our substitution model to find out. We'll start with `x(3.0 + 4.0)`. Python will start by looking up the value of `x`, and finding `2`, which leaves us with: `2(3.0 + 4.0)`. Next, Python will evaluate the value inside the parentheses, giving `2(7.0)`. Next, Python proceeds by attempting to *call* `2` with `7.0` as an argument. Since `2` is not a function, it is not clear exactly what this means, and so Python gives us an error.

Presumably, the code above was intended to *multiply* `2` and `3.0+4.0` rather than calling `2` with `3.0+4.0` as an argument. But Python isn't that smart, so we have to be very explicit if that's what we want (in this case, we have to include a `*` to indicate muiltiplication).

3) Defining Custom Functions

Using built-in functions or functions imported from Python modules is all well and good, but *real power* comes from being able to define functions of your own. This is accomplished via a Python statement called a *function definition statement*, which uses a Python keyword called `def`.[5]

This is perhaps best seen by example:

```
def maximum(x, y):
    if x > y:
        z = x
    else:
        z = y
    return z
```

This statement does two things:

- it creates a new `function` object in memory, *and*
- it associates the name `maximum` with that object in the current frame.

A function definition statement always starts with the keyword `def`, followed by an arbitrary name for the function. The sequence of names within the parentheses after the function's name are called *parameters* or, interchangeably, *arguments* (in this case, `x` and `y`), and the function describes a computation in terms of those parameters (as well as, potentially, cBack to Top
constants, etc.).

Like many of the structures we have seen, function definitions also have a body (all the code that is indented one level farther than `def`; in this case, the whole `if/else` statement). The `return` keyword, which is only usable inside a function definition, tells Python what the function should produce as its *return value* when it is called.

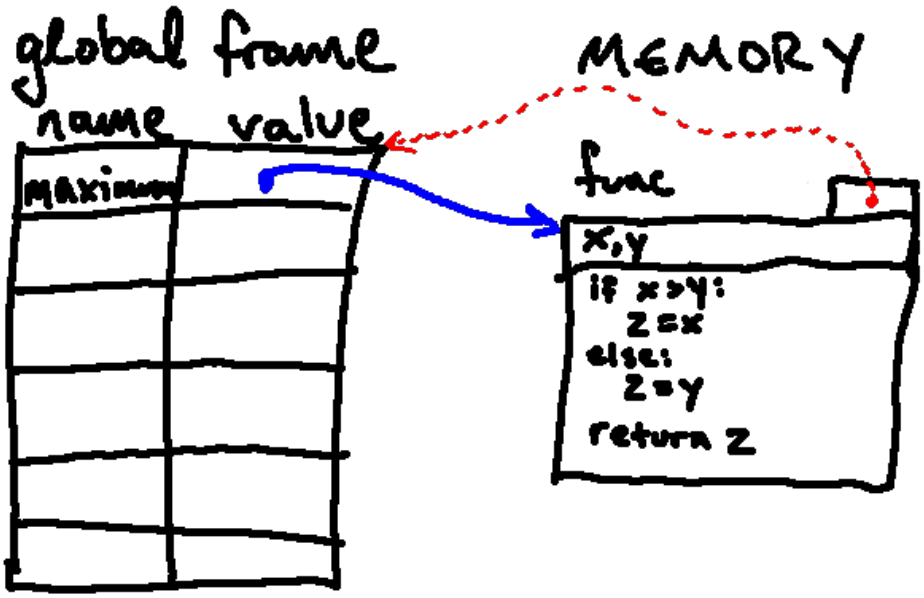
Importantly, this statement only *defines* the function; Python **does not run the code in the function body yet**! It simply makes an object that represents this function and associates the given name with that object.

3.1) Function Environment Diagrams

As with every new object we've introduced, we'll need a way to represent these objects in memory. Functions need to keep track of three pieces of information, and we'll try to depict all of those in our representation:

1. The names of the parameters to the function, in order;
2. The code in the body of the function; and
3. The frame in which the function was defined.

The following shows an example environment diagram that would result after executing the function definition statement above:



A few notes about this drawing:

- Note that the function definition did two things: it created a new function object, and it bound the name `maximum` to that object in the global frame.
- The names `x,y` on the top line of the function object represent the parameters of the function.
- The red arrow points back to the frame in which the function was originally defined (in this case, the global frame).

We can then call this function just as we would with any of the built-in functions (or imported functions) we've seen so far. For example:

```
a = 7.0
b = 8.0
x = 3.0
y = 4.0

c = maximum(a, b)
```

Back to Top

We'll now spend some time learning about what happens when this function is called. In the simplest terms, this is what happens:

1. First, Python looks up the name `maximum` and finds the function object in memory.
2. Next, Python runs the code in the function body with the parameters replaced with the values given (here, the body of the function would be evaluated with `x` replaced by `7.0` and `y` replaced by `8.0`) until either it reaches a `return` statement or the end of the body. If a `return` statement is reached, execution of the function stops and the associated value is returned; if the end of the function is reached (without hitting a `return` statement), the function returns `None`.

So after running the code above, `c` will have value `8.0`.

That said, it will be important for us to understand *how exactly Python got to that result*, and so we'll go into more detail on these steps in the next section. Grab a cup of tea and settle in! This will get complicated, but understanding it is crucial to understanding some of the behaviors we will see from Python in the future! Don't be afraid to re-read multiple times, and, of course, to **ask questions** if you are confused!

4) Calling Custom Functions

We now examine what happens when a user-defined function is called. We'll go through the example from above. In unit 5's readings, we'll explore more complex examples.

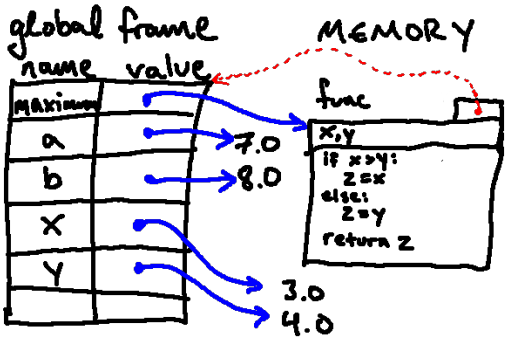
Here is the code (repeated from above) for our example:

```
a = 7.0
b = 8.0
x = 3.0
y = 4.0

c = maximum(a, b)
```

(Assume that `maximum` has already been defined as above)

We know how the first four lines will behave: Python will associate the names `a`, `b`, `x`, and `y` with the values `7.0`, `8.0`, `3.0`, and `4.0`, respectively, in memory, resulting in the following environment diagram:



Back to Top

Now we are on to the line where the function call is evaluated. To accomplish this evaluation, Python completes the following steps:

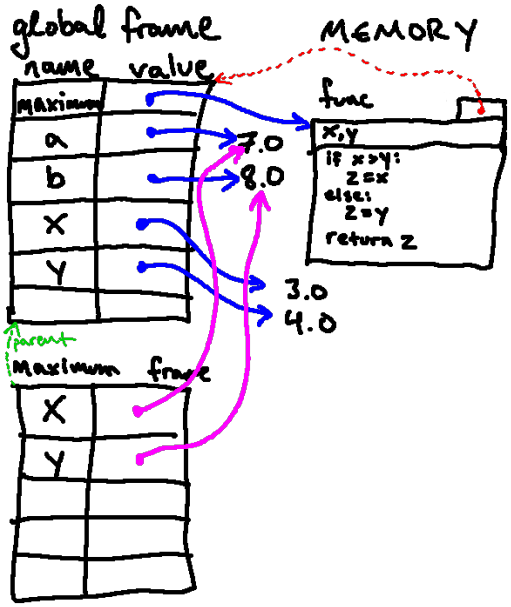
1. As we saw with built-in functions, Python first starts by evaluating the name `maximum` (finding the `function` object), followed by the arguments to the function (which evaluate to the `7.0` and `8.0` that `a` and `b`, respectively, point to).

2. This is where things get different. Python's next step when calling a user-defined function is to *create a new frame*. This frame will be similar to the global frame in that it will map names to variables, but these variables will be *local* to the function (i.e., they will only be accessible inside the function being called).

   Once this new frame is created, Python binds the names of the parameters to the argument that were passed in to the function. From this point on, variable lookups will happen inside this new frame (until the function is done executing).

   This frame also contains a "parent pointer" to *the environment in which the function being called was defined*.[6]

   Once this step is done, we will have an environment diagram like the following:



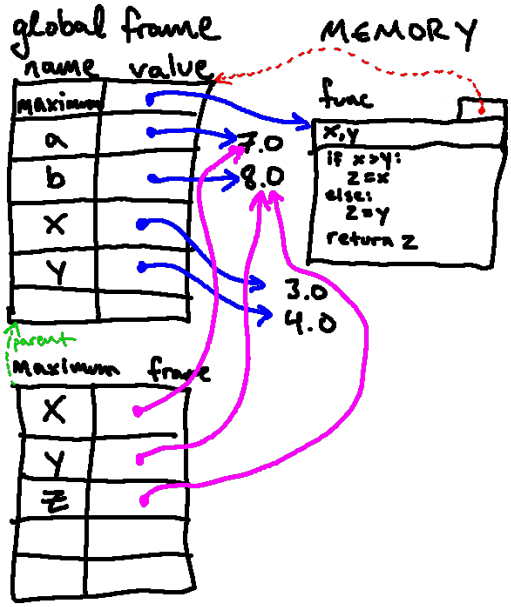
   As promised, this is a bit complicated. The frame in the bottom-left contains the bindings that exist inside the function. The green arrow represents the "parent pointer."

3. Python then executes the body of the function within this new frame. This means that, if Python looks up `x`, it finds the value `7.0` (bound in this frame), rather than the `3.0` value that is bound in the global frame. When looking up a variable, if it is not found in the current frame, Python then continues looking for that name in the parent frame before giving up.

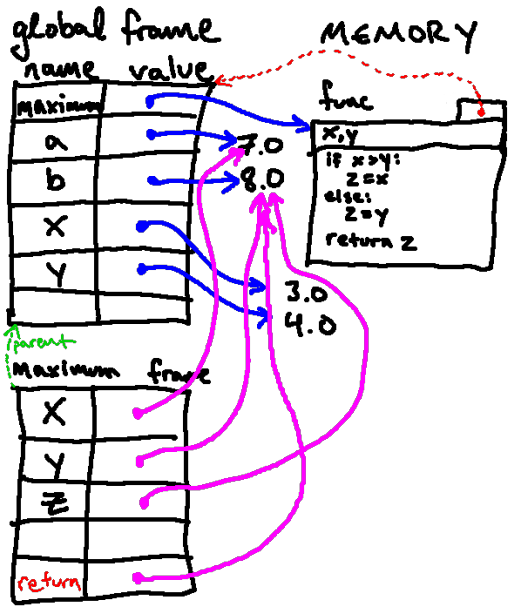
In the process of executing the function body, if Python makes any new assignments, those are *also* made in the current frame.

Back to Top

So when our example code assigns a value to the variable `z`, that binding is made in the current frame. In the course of executing the conditional, we assign `z` to the same value as `y`, which results in the following environment diagram:



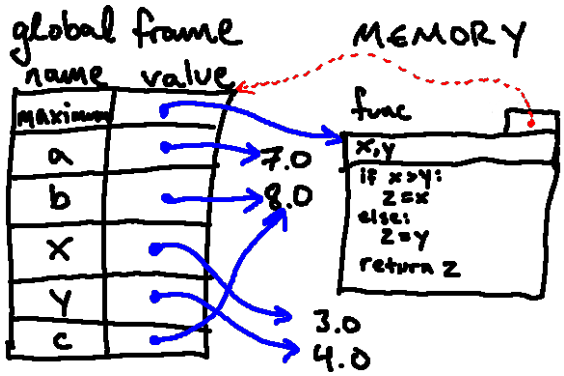
4. When the body is over or a `return` statement is reached, Python notes the value to be returned. (In the below diagram, the red "return" does not indicate an actual new variables called "return"; rather, it simply indicates the value that is to be returned from the function). Here, the return value was `z`, which pointed to the `8.0` currently in memory, so that is the value that will be available as the result of calling this function:



Python then stops executing this function and returns to executing in the frame from which the function was called, after

returning the value in question. It also cleans up the new frame it created[7]. In our example from above, the return value is then associated with the name `c` in the global frame, leaving us with the following:

Back to Top



4.1) Abstraction

Notice that above, the process of creating the new frame gave us a wonderful feature: the variables *inside* the function body don't affect the variables *outside* the function body. This allowed us to call something `x` inside the function body and not have that cause problems with the thing called `x` *outside* the function (when the function is done executing, if we print `x`, we see the `3.0` that was assigned to `x` in the global frame)!

This is the real reason functions are so powerful: they offer us a means of *abstraction* (meaning, once we have defined a function, we can use it knowing only its end-to-end behavior, without worrying about exactly *how* that behavior was implemented in the function body, what variable names were used, etc.).

4.2) Short Version

Here is the short version of Python's process for calling a user-defined function (again, for details, please see above):

1. Evaluate the arguments. (If an argument is itself a function call, apply these steps to it.)
2. Make a new frame. It stores a pointer to the frame's parent (the frame in which the function was defined).
3. Bind the arguments. In step 1, you already evaluated and simplified the arguments. Now you just have to bind variables to those values in the new frame.
4. Execute the body of the function in this new frame. Depending on what you see, you may be drawing more bindings and/or drawing new frames.
5. Note the return value of the function.
6. When execution has finished, remove the frame and resume execution in the calling frame.

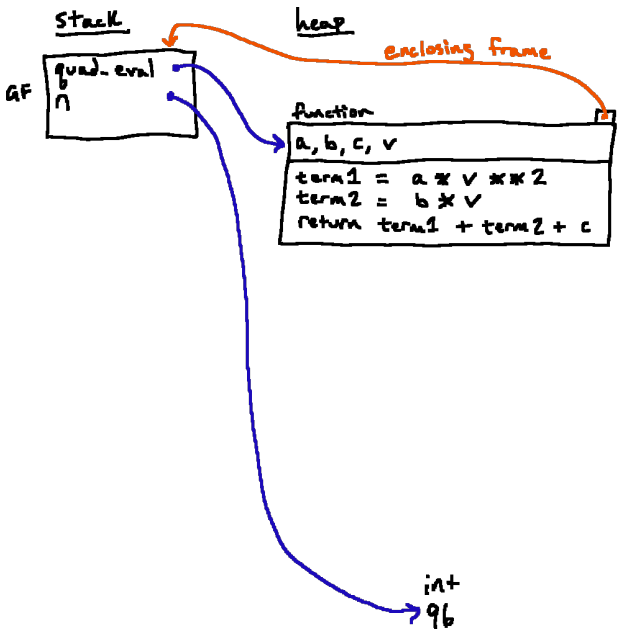
5) More Environment Diagram Examples

Let's look at another environment diagram example. The code below, defines and calls a function named `quad_eval`, which evaluates the value of a quadratic formula $av^2 + bv + c$ at a particular value, `v`.[8]

```
1 def quad_eval(a, b, c, v):
2     term1 = a * v ** 2
3     term2 = b * v
4     return term1 + term2 + c
5
6 n = quad_eval(7, 8, 9, 3)
7 print(n)
```

Back to Top

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 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?

Back to Top

- ○ 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

Save Submit View Answer

As staff, you are always allowed to submit. If you were a student, you would see the following:
You have infinitely many submissions remaining.

Another Example

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):

```
1  x = 500
2
3  def foo(y):
4      return x+y
5
6  z = foo(307)
7
8  print('x:', x)
9  print('z:', z)
10 print('foo:', foo)
11
12 def bar(x):
13     x = 1000
14     return foo(308)
15
16 w = bar(349)
17
18 print('x:', x)
19 print('w:', w)
```

Try Now:

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?

Check Formatting | Submit | View Answer

As staff, you are always allowed to submit. If you were a student, you would see the following:
You have infinitely many submissions remaining.

Back to Top

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

Check Formatting | Submit | View Answer

As staff, you are always allowed to submit. If you were a student, you would see the following:
You have infinitely many submissions remaining.

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

Check Formatting | Submit | View Answer

As staff, you are always allowed to submit. If you were a student, you would see the following:
You have infinitely many submissions remaining.

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

Check Formatting | Submit | View Answer

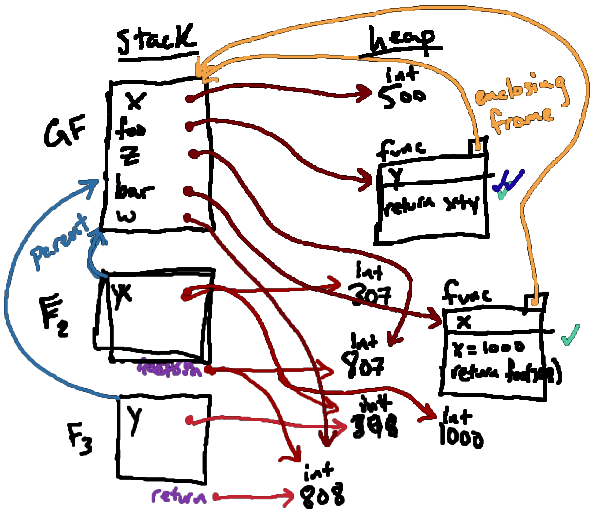
As staff, you are always allowed to submit. If you were a student, you would see the following:
You have infinitely many submissions remaining.

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!

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

```
1  x = 500
2
3  def foo(y):
4      return x+y
5
6  z = foo(307)
7
8  print('x:', x)
9  print('z:', z)
10 print('foo:', foo)
11
12 def bar(x):
```

```
13      x = 1000
14      return foo(308)
15
16  w = bar(349)
17
18  print('x:', x)
19  print('w:', w)
```



STEP 8

[Back to Top]

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, 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.

```
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.

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?

Back to Top

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

Save Submit View Answer

As staff, you are always allowed to submit. If you were a student, you would see the following:
You have infinitely many submissions remaining.

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!

6) Built-Ins

Now that we have talked about the notion of multiple frames, we can clear something up about Python's built-ins. We have talked a lot now about how Python looks up the values associated with variable names, and you may have wondered how Python found the built-in values.

How did Python know what function to call when we referenced `print`? It is true that `print` and the other built-ins do not exist in the global frame. Rather, we can think of the global frame itself as having a parent pointer to a special "built-ins" frame: when Python looks up a name in the global frame and doesn't find it, it then looks in this special "built-ins" frame before throwing a `NameError`.

This is how the lookups for, for example, `print` and `len` proceed: Python first looks for them in the global frame. Since it doesn't find them there, it looks in the built-ins frame, where it finds them.

Failed variable lookups also proceed in this same way. Say we made a typo and were accidentally looking up the value `pritn`. In looking this up, Python would first look in the global frame; when it doesn't find `pritn` there, it would then look in the built-ins frame; and when it doesn't find `pritn` there either, it will give up and raise a `NameError`.

7) Print vs. Return

In general, print statements are useful for displaying information to the user, and return statements allow the results of function calls to be stored and passed around within the program, which turns out to be quite useful since you can then look up (or even print) those values again later.

One important difference between print statements and return statements is that once Python reaches a return statement inside a function it immediately stops and exits the function after returning the value. While one function can have multiple print statements that display multiple lines to the screen, return statements are considered a 'hard stop' to the function.

For example, consider the outputs of the following two functions:

```
def without_return():
    print("without return")
    print("this line gets printed")
```

```
def with_return():
    print("with return")
    return "STOP"
    return "this line never runs"
    print("this line never gets printed")

without_result = without_return()
'''   the above line results in the following output:
without return
this line gets printed
'''
print(without_result)  # None


result = with_return()
'''   output:
with return
'''
print(result) # STOP
```

Back to Top

It is also important to note that in the absence of a return statement, all functions by default return None. For example, consider this function that returns 'positive' if a number is greater than 0:

```
def is_positive(x):
    if x > 0:
        return 'positive'

result = is_positive(5)
print(result) # prints 'positive'
result2 = is_positive(0)
print(result2) # prints None
```

Just like print allows you to display multiple values, return statements can also output multiple values:

```
def divide(dividend, divisor):
    # calc the quotient and remainder of dividing two numbers
    return dividend // divisor, dividend % divisor

result = divide(10, 3) # returns a tuple
print(result) # (3, 1)
print("result is", result[0], "remainder", result[1]) # result is 3 remainder 1
num, remainder = divide(19, 4)  # unpacking the returned tuple
print(num) # 4
print(remainder) # 3
```

Unlike print, return statements do not need round brackets. If we want to return multiple values, we simply separate them by a comma. If you recall from a couple of units ago, this creates a tuple, which we can clearly see when we print out the result. To access the different returned values, we can simply index into the result, or unpack them into separate variables, as shown above.

7.1) Return to Refactoring

For an example of why using return statements are useful, consider the following problem which we solved earlier irBack to Top:

```
def simple_full_packages(order, package_limit):
    '''
    Calculate the number of full packages that can be delivered, assuming near
    infinite inventory.

    Inputs:
        order (int) - number of items requested
        package_limit (int) - number of items that can fit in a shipment
          container

    Returns:
        The integer number of completely filled packages.
    '''
    if package_limit == 0:
        num_packages = 0  # avoid Division by 0 error
    else:
        num_packages = order // package_limit

    return num_packages


def num_packages(order, package_limit):
    '''
    Calculate the fewest number of packages that would be required to ship
    all items in an order.

    Inputs:
        order (int) - number of items requested
        package_limit (int) - positive number of items that can fit in a
                              shipment container

    Returns:
        The integer number of packages required.
    '''
    if package_limit == 0:
        num_boxes = 0  # avoid Division by 0 error
    else:
        num_boxes = order // package_limit

    if package_limit*num_boxes < order:
        num_boxes =  num_boxes + 1
    return num_boxes

if __name__ == "__main__":
    # Local testing -- feel free to add your own tests as well!
    print("testing simple_full_packages:")
    print(f"Got {simple_full_packages(10, 0)=}, expected 0")
    print(f"Got {simple_full_packages(10, 3)=}, expected 3")
    print(f"Got {simple_full_packages(5, 5)=}, expected 1")
```

```
print("\ntesting num_packages:")
print(f"Got {num_packages(10, 1)=}, expected 10")
print(f"Got {num_packages(10, 3)=}, expected 4")
print(f"Got {num_packages(10, 5)=}, expected 2")
```

Back to Top

Notice how the conditional logic is very similar for both functions. In fact, the only difference is a single `if` statement in `num_packages`. Because we used return statements instead of print, we can *refactor* our code as follows, to make it simpler and more concise:

```
def simple_full_packages(order, package_limit):
    '''
    Calculate the number of full packages that can be delivered, assuming near
    infinite inventory.

    Inputs:
        order (int) - number of items requested
        package_limit (int) - number of items that can fit in a shipment
          container

    Returns:
        The integer number of completely filled packages.
    '''
    if package_limit == 0:
        num_packages = 0  # avoid Division by 0 error
    else:
        num_packages = order // package_limit

    return num_packages


def num_packages(order, package_limit):
    '''
    Calculate the fewest number of packages that would be required to ship
    all items in an order.

    Inputs:
        order (int) - number of items requested
        package_limit (int) - positive number of items that can fit in a
                              shipment container

    Returns:
        The integer number of packages required.
    '''
    num_boxes = simple_full_packages(order, package_limit)

    if package_limit*num_boxes < order:
        num_boxes =  num_boxes + 1
    return num_boxes
```

Now instead of repeating code, `num_packages` can use `simple_full_packages` to calculate the initial number of boxes, and then decide whether a partially full box will be needed.

Code that is simple, concise, and easy to read tends to be both easier to understand and debug. That is why *refactoring*, the process of revising code to have better style (e.g., descriptive variable names, better high-level comments, consistent spacing), reduced repetition, and less complexity, is an important programming skill that we will continue to practice this weeBack to Top

8) Why Functions?

To close this section, here are some reasons for using functions:

- Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read and debug.
- Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.
- Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.
- Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

9) Extras: Syntactic Sugar

Before closing fully, we briefly describe some of Python's "syntactic sugar" to which you've been exposed. Wikipedia explains that syntactic sugar is "syntax ... that is designed to make things easier to read or to express. It makes the language 'sweeter' for human use: things can be expressed more clearly, more concisely, or in an alternative style that some may prefer."

Last week, we saw a lot of examples of syntactic sugar in the form of ternary statements, comprehensions, and the built-in functions min, max, sum and sorted. This week we're going to dive into one more useful piece of syntax that we've been using throughout the course but haven't explained: f-strings.

9.1) String formatting

Sometimes we want to print a combination of text and variables. For example, say we were a grocery store that wanted to display the cost of an item a user selected. With what we know about type-casting and string concatenation we could accomplish this as shown in the example below:

```
item = 'bread'
cost = 1.99
print("The " + item + " costs $" + str(cost) + ".")
# "The bread costs $1.99."
```

But what if we wanted to print the cost of five different items? It would be a pain to make sure all the `+` signs and quotation marks were in the right places. That's why more recent versions of python have introduced more convenient ways of formatting strings.

One particularly useful string formatting method is called the f-string. For example:

```
print(f"The {item} costs ${cost}.")
# "The bread costs $1.99."
```

Placing an `f` at the beginning of the string lets python know to format it so that any value enclosed in curly brackets is evaluated, cast to a string, and then automatically concatenated in the right place.

Using an `=` sign before the closing curly brackets indicates that you want to display both the code as a string, and the result of evaluating that code. For example:

Back to Top

```
nums = [1, 2, 3, 4, 5]
print("len(nums) = " + str(len(nums)))
```

Can be more concisely written with the following f-string:

```
nums = [1, 2, 3, 4, 5]
print(f"{len(nums) = }")
```

An older python string formatting method that you might encounter in the "wild" uses `%` signs to act as a placeholder for a value.

```
print("The %s costs $%g." % (item, cost))
# "The bread costs $1.99."
```

The `%s` indicates that a string value will be inserted, `%g` means a floating-point number will be inserted. At the end of the string, the values are listed in order. You can find more information about formatting data types using the `%` notation in python's documentation.

So far, we've used f-strings to help us manually check our code to see if it's getting the expected result. In future units, we'll learn about other tools we can use to get Python to test our code for us!

10) Reading and Writing csv Files

Finally, let's talk about one more useful built-in module.

Sometimes the input to your program might be stored in a file. For example, if I am writing a program to process items in my grocery list, and my grocery list lives in a comma-separated values file called grocery_list.csv, it'd be convenient if my program could access the contents of that file. Python can do just that!

The first thing to do in order to use a file is to *open* it. This is done by invoking the built-in function `open`

We can open a file in different modes, like read mode or write mode. Since we're just reading from the file for now, we'll tell that to the function `open` by writing the string `'r'`. If my grocery_list.csv file is in the same directory as my Python program[9], I can write

```
opened_file = open("grocery_list.csv", "r")
```

`opened_file` is now an object. With `opened_file` in hand, we can use the csv module to read the file contents.

We must tell Python that we plan to use it by writing `import csv` at the top of our file. Then we can create a reader object by calling a function `reader` which the `csv` module makes available to us. Our code is now:

```
import csv
```

```
opened_file = open("grocery_list.csv", "r")
reader = csv.reader(opened_file)
```

Back to Top

`reader` is an object that allows for iteration. We can print out all the rows in the file, which the reader stores as lists of strings, by looping:

```
for row in reader:
    print(row)
```

For a few reasons[10] (which admittedly aren't likely to be critical for us), if we open a file, we should close it as well, after we're done with it:

```
opened_file.close()
```

Since it's very easy to forget to close a file, Python has some great syntactic sugar which automatically does it for us. We can create a `with/as` block, inside of which the opened file will be open, but outside of which it is automatically closed.

The block doesn't explicitly use the `=` assignment operator to set the `opened_file` variable, but it still gives `opened_file` the same value as before.

Our final program would look like this:

```
import csv

with open("grocery_list.csv", "r") as opened_file:
    reader = csv.reader(opened_file)
    for row in reader:
        print(row)
```

Try Now:

Below is an example grocery list and the Python code we just wrote. Save them into the same directory, and run `read.py` to see the printed list output. Experiment with what happens if you add more columns to the CSV file.[11]

grocery_list.csv

read.py

Try Now:

What do you expect to be printed if we run the following code, which just repeats the printing for loop? Try it to check.

Back to Top

```
import csv

with open("grocery_list.csv", "r") as opened_file:
    reader = csv.reader(opened_file)
    for row in reader:
        print(row)
    for row in reader:
        print(row)
```

Show/Hide

The reason you get this unexpected result is subtle. The relevant mental model is that the `reader` object is a sort of one-directional pointer inside the file. That is, it starts at the beginning of the file when you open it, and it advances forward row by row when it is looped over, but it does not automatically go back to the beginning. If you wish to use data in a file multiple times, a good approach is to store the data from the reader into a variable (likely a list or other sequence) just once, then manipulate the data stored in that variable, instead of going back to the file directly to get the data again:

```
import csv

data_rows = []
with open("grocery_list.csv", "r") as opened_file:
    reader = csv.reader(opened_file)
    for row in reader:
        data_rows.append(row)

# Can now use data_rows multiple times
```

Try Now:

As an exercise, try to write code that will read in the grocery_list.csv file and create a dictionary that maps the of the item to the integer quantity of the item (not including the first row). Back to Top

Show/Hide

This can be accomplished with the following code:

```
import csv

groceries = {}

with open("grocery_list.csv", "r") as opened_file:
    csv_reader = csv.reader(opened_file)
    for row in csv_reader:
        if row[1] != "Quantity":
            groceries[row[0]] = int(row[1])
            # note all values are read as strings. If we want a value we
            # read to be treated as a number, we need to cast it to an
            # int or a float!


print(groceries)
```

Now let's say we wanted to add something to our grocery list, like oranges. While we could append this to our `data_rows` while running our program, this would not actually change the grocery_list.csv file. Luckily, Python also allows us to write to files. Check out the write.py file below, which does just that:

```
import csv

data_rows = []
with open("grocery_list.csv", "r") as opened_file:
    reader = csv.reader(opened_file)
    for row in reader:
        data_rows.append(row)

data_rows.append(['Oranges', 3])

# make a new grocery list file
with open('new_grocery_list.csv', 'w', newline='') as csvfile:
    writer = csv.writer(csvfile, delimiter=',')
    for row in data_rows:
        writer.writerow(row)
```

Note how when we make the new grocery list, we opened the file in 'w' or write mode. Instead of using the csv library to read the file, now we create an object that can write individual rows one at a time. Note that each row is a lists of values, where each

element represents the value of a single column.

For more information on the csv library, see the Python documentation.

Back to Top

While working with spreadsheets directly can often be useful, as we'll see in the exercises, using Python to do data analysis can allow us to more easily analyze large amounts of data, especially when it is in a raw or unprocessed format.

11) Summary

In this reading, we (mainly) introduced *functions*. Importantly, we also talked about *defining new functions of your own creation* as a means of abstracting away details of a particular computation so that it can be reused. We spent a good deal of time and effort focusing on how defining and calling these functions fits in to our mental models of Python (substitution model and environment diagrams). In unit 5, we'll solidify this understanding of functions further, with more examples, and we'll introduce some more related function features.

In this set of exercises, you will get more practice with simulating the evaluation of functions and with defining functions of your own. You will also get the chance to practice using string-formatting and reading and writing files, which is useful for data analysis and processing, among other things.

Next Exercise: Fun with Functions

Back to exercises

Footnotes

[1] In fact, with the subset of Python we have learned so far, it is possible to prove that we have everything we need to Back to Top *problem that can be solved via computation*! It's a little bit dense, but the Wikipedia article for Turing Completeness can provide small window into this area of computer science theory (called *computability theory*).

[2] Some would say "invoke" the function, and we may use both terms interchangeably here

[3] or, interchangeably, a *parameter*

[4] In the next set of readings, we'll see that the return value of a function can even be a function itself!

[5] `def` is short for **def**ine, or **de**fine **f**unction, depending on whom you ask.

[6] In this case, because `maximum` was defined inside the global frame, the parent pointer of this new frame will point to the global frame. But because it is possible for functions to be defined inside of functions, this "parent" of this new frame will not necessarily be the global frame. Specifically, it will be the frame in which the function being called was defined.

[7] And, as before, if this cleanup results in any objects in memory not having any pointers left to them, they would be garbage collected.

[8] Note this section was adapted from 6.101's readings

[9] If the files were *not* in the same directory, we may need to give a lengthier absolute path to the grocery_list.csv file, so Python knew where to look for it.

[10] Some of those reasons: there could be limits on the number of files you can open at a time, opened files might not be accessible elsewhere, file changes (if we were writing, not reading) might not go into effect until the file is closed, open files can slow down your program, and it's just cleaner programming.

[11] Note this is not my actual grocery list; this part of the reading was written by a previous instructor.