

# Readings for Unit 3

[Back to Top](#)

The questions below are due on Saturday July 06, 2024; 10:00:00 PM.

## Licensing Information



The readings for 6.S090 are licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License](#). You are free to make and share verbatim copies (or modified versions) under the terms of that license.

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\\_reading3.pdf](#)

## Table of Contents

- 1) Introduction
- 2) Functions
  - 2.1) Multiple Arguments
  - 2.2) Substitution Model
- 3) Defining Custom Functions
  - 3.1) Function Environment Diagrams
- 4) Calling Custom Functions
  - 4.1) Abstraction
  - 4.2) Short Version
- 5) Another Environment Diagram Example
- 6) Built-Ins Frame
- 7) Print vs. Return
- 8) Why Functions?
- 9) More Loops
  - 9.1) `for element in iterable:`
  - 9.2) Nested Loops
- 10) Extras: Syntactic Sugar
  - 10.1) `+=` operator
  - 10.2) Type checking using `type` and `isinstance`
  - 10.3) String formatting
  - 10.4) Even more loops!
    - 10.4.1) Loop style conventions
    - 10.4.2) `for element in x:` vs. `for i in range(len(x)):`?
    - 10.4.3) Additional range arguments: `range(start, stop, step)`
- 11) Summary

## 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 ints, floats, Booleans, the special `None` value, and sequences like lists, tuples, and strings); 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!<sup>1</sup> 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*.

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. If we wanted to use the code above to this end, we could copy the code and paste it to new locations several times. Depending on what our program is doing, we might also 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<sup>2</sup>, which means to evaluate the sequence of statements it represents. The expression inside the parentheses (here,

"twine") is called an *argument*<sup>3</sup> 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 [Back to Top](#) *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!<sup>4</sup>

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

```
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 one or two arguments:

```
print(round(3.14159)) # round returns a number; this will print 3
print(round(3.14159, 2)) # round returns a number; this will print 3.14
```

### Try Now:

Side note: 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](#).

It turns out that 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.

**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 `print` in a variable, and then displaying it (again with `print`!). [Back to Top](#)

[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
```

However, not all functions can take extra arguments. For example, attempting to run `float(5, "5")` will result in the following error message: `TypeError: float expected at most 1 argument, got 2`.

## 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:

[Back to Top](#)

- 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

**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 Python 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 multiplication).

### 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`.<sup>5</sup>

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, [Back 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.

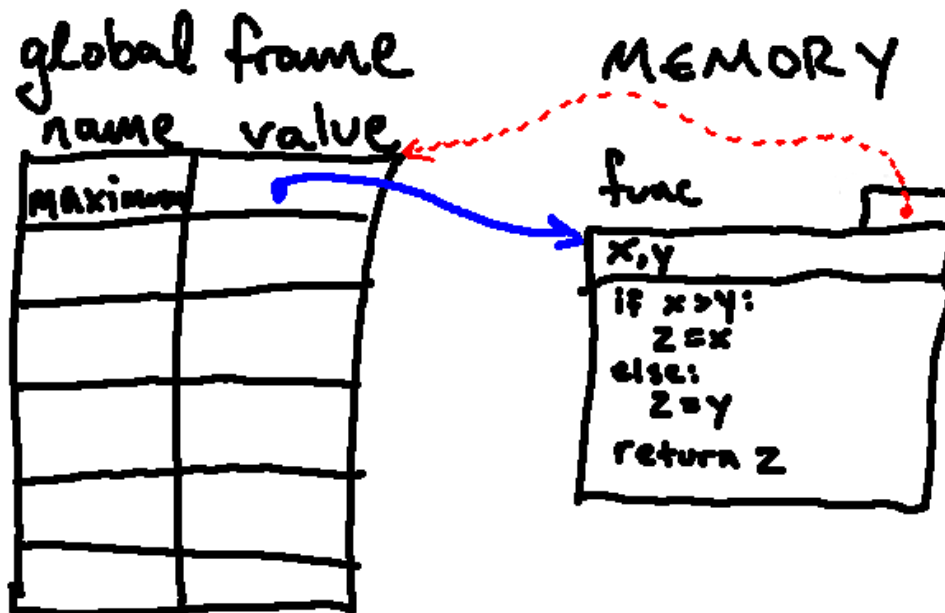
Importantly, this statement only *defines* the function; Python **does not run the code in the function body yet!** It simply makes an object that represents this function and associates the given name with that object.

### 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).
- The body of the function does not run when we define the function. Instead, it is simply saved inside the function object so that it can be used later when the function is called.

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:

[Back to Top](#)

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

c = maximum(a, b)
```

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, 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:

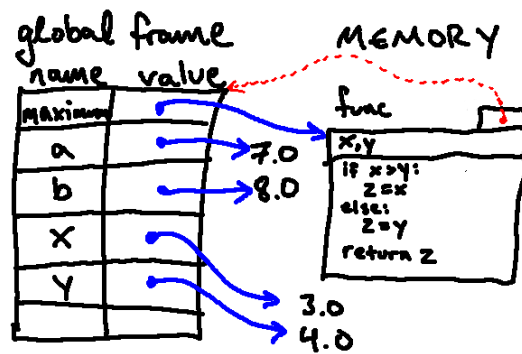
```
a = 7.0
b = 8.0
x = 3.0
y = 4.0

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](#)

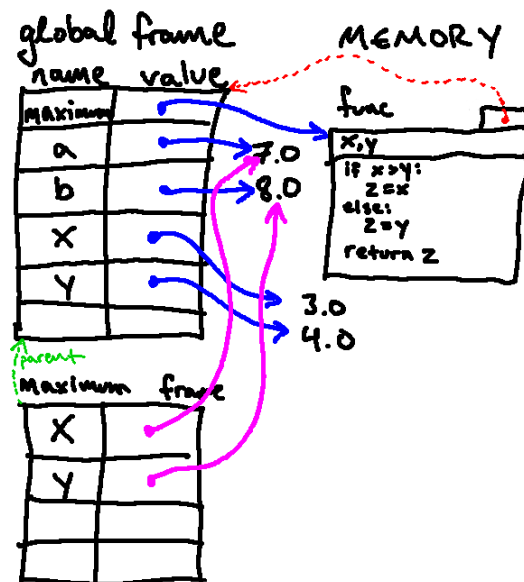
Now we are on to the line where the function call is evaluated. To accomplish this evaluation, Python completes the following steps:

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.<sup>6</sup>

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



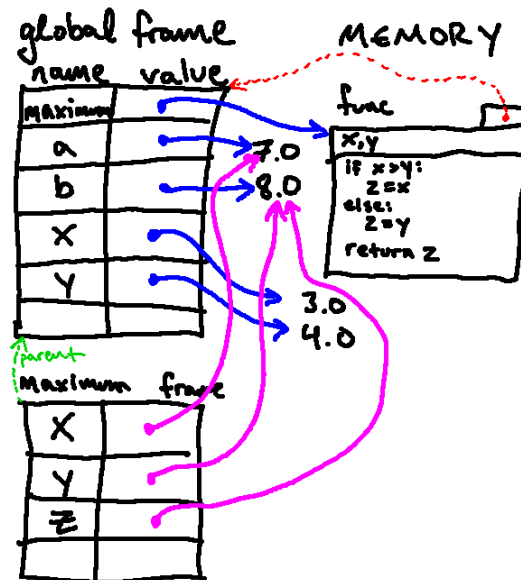
As promised, this is a bit complicated. The frame in the bottom-left contains the bindings that exist inside the function. The green arrow represents the "parent pointer."

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,

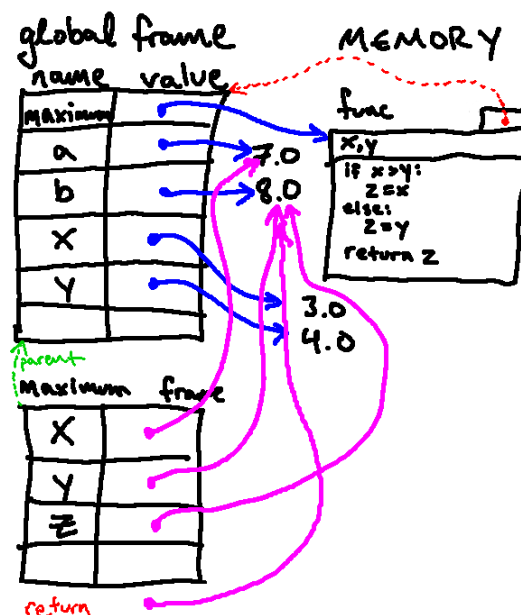
if it is not found in the current frame, Python then continues looking for that name in the parent frame before giving up.

In the process of executing the function body, if Python makes any new assignments, those are *also* made in [Back to Top](#) frame.

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:

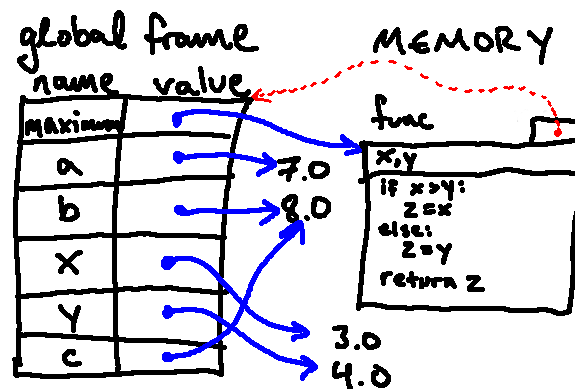


4. When the body is over or a return statement is reached, Python notes the value to be returned. (In the below diagram, the red "return" does not indicate an actual new variables called "return"; rather, it simply indicates the value that is to be returned from the function). Here, the return value 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<sup>7</sup>. 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) 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`.<sup>8</sup>

Show/Hide Line Numbers

```

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

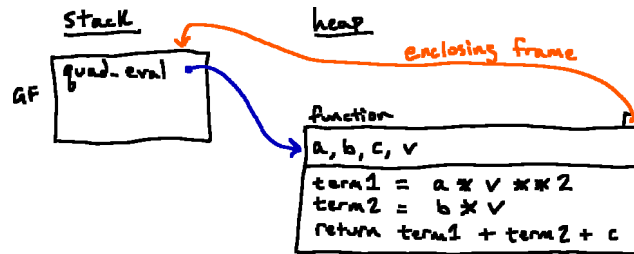
```

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 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 equal 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.

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

SaveSubmitClear Answer**100.00%**

As staff, you are always allowed to submit. If you were a student, you would see the following:

*You have infinitely many submissions remaining.*

**Solution:** The function we're calling was originally defined in the global frame.

## 6) Built-Ins Frame

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 `prtn`. In looking this up, Python would first look in the global frame; when it doesn't find `prtn` there, it would then look in the built-ins frame; and when it doesn't find `prtn` 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")

```

[Back to Top](#)

```

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

```

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 the last unit, 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.

## 8) Why Functions?

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

[Back to Top](#)

- 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 function, you can reuse it.

## 9) More Loops

Now that we have finished our deep dive into how functions work, let's take another deep dive into another concept we introduced in the previous unit: loops! It turns out there are many different ways to structure loops in Python, but for right now we are going to introduce two more structures: looping over elements and nested loops.

### 9.1) for element in iterable:

In the last unit's readings, we saw that we could construct a for loop to print out all of the individual characters in a string:

```
word = "hi"  
for i in range(len(word)):  
    print(word[i])  
print("done!")
```

It turns out that we could write this program more simply as follows:

```
word = "hi"  
for i in word:  
    print(i)  
print("done")
```

More generally, a for loop can iterate (loop) over any *iterable* object (an object capable of returning its members). Sequences such as list, tuple, string, and range objects are all iterable.

**Try Now:**

How many print statements will the following program execute? What will the output of the program look like? [Back to Top](#)

```
my_list = [1, (2, 3)]

for element in my_list:
    print(element)

my_tuple = (0, my_list, (-2))

for element in my_tuple:
    print(element)

my_string = ""

for element in my_string:
    print(element)

print("done")
```

[Show/Hide](#)

This program prints the following six lines:

```
1
(2, 3)
0
[1, (2, 3)]
-2
done
```

Note that this corresponds to 2 elements in my\_list, 3 elements in my\_tuple (note the -2 surrounded by parentheses is interpreted as an int because there was no comma!), 0 elements in my\_string, and the final "done" print statement.

Primitive types such as float, int, None, or bool objects are not iterable.

```
x = 5

for element in x:
    print(element)
```

Running the program above results in an error, `TypeError: 'int' object is not iterable`, which is Python's way of saying it does not know how to loop over a single number. However, if we change x to a range object like in the program below, the program will work and display the numbers 0-4.



```
x = range(5)
```

```
for element in x:  
    print(element)
```

[Back to Top](#)

## 9.2) Nested Loops

Just like you can have conditional statements within conditional statements, you can have loops inside of loops!

### Try Now:

What does the following program print?

```
nested_sentence = [ ["parts.", "into separate"], ["and split"],  
                   ["has been reversed", "This sentence", ] ]  
new_sentence = ""  
  
for word_list in nested_sentence:  
    for word in word_list:  
        new_sentence = word + " " + new_sentence  
  
print(new_sentence)
```

**Try Now:**

What does the following program print?

[Back to Top](#)

```
total = 0
count = 0
nested_nums = [[-3, 7.8, 9], [-500, 100, 0], [32, 1, -1]]
for num_list in nested_nums:
    for num in num_list:
        if num >= 0:
            total = total + num
            count = count + 1

print(total / count)
```

[Show/Hide](#)

Output:

24.966666666666667

Note this is very similar to an exercise in the 2D-Arrays problem from unit 2 that was intended to be completed in one loop by calling a helper function that had a separate loop. We could similarly make a helper function like this:

```
total = 0
count = 0
nested_nums = [[-3, 7.8, 9], [-500, 100, 0], [32, 1, -1]]

def get_total_count(num_list):
    total = 0
    count = 0

    for num in num_list:
        if num >= 0:
            total = total + num
            count = count + 1
    return total, count

for num_list in nested_nums:
    result = get_total_count(num_list)
    total = total + result[0]
    count = count + result[1]

print(total / count)
```

Running this program produces the same output as the original program! Can you see how even though we now have two single loops, it is effectively still a nested loop?

[Back to Top](#)

## 10) Extras: Syntactic Sugar

Before closing fully, we briefly describe some more of Python's "syntactic sugar". Last week, we saw a lot of examples of syntactic sugar in the form of sequence operations like comparisons, string methods, and casting sequence types. This week we're going to dive into some more helpful shortcuts involving adding, string formatting, type checking, and loops.

### 10.1) += operator

We commonly want to add something to an existing variable in Python, so Python provides a shortcut in the form of += operator:

```
x = 5
x += 1 # equivalent to x = x + 1
```

In addition to numeric operations, the += operator can also be used to concatenate sequences.

```
a = "hello"
b = a
b += " again" # equivalent to b = b + a
print(a, b)  # hello  hello again
```

Note how in the example above with the strings, that concatenating `b += a` created a new string (so the original string that `a` references is unmodified). This makes sense because as we covered in the last unit, strings are immutable.

However, the += operator does something a little unusual for lists:

```
a = [1]
b = a
b += [2] # NOT equivalent to b = b + a
print(a, b) # [1, 2]  [1, 2]
```

It turns out, the += operator *mutates* the given list object instead of concatenating and creating a new list, like in the program below.

```
a = [1]
b = a
b = b + [2] # NOT equivalent to b += a
print(a, b) # [1]  [1, 2]
```

### 10.2) Type checking using type and isinstance

It is sometimes desirable to be able to determine the type of a given object.

For example, say that we had a list of assorted types and we wanted to add all of the numbers (ints and floats) together. If we tried to naively add everything like in the program below, we would quickly encounter an error: `TypeError: unsupported operand type(s) for +=: 'int' and 'NoneType'`. [Back to Top](#)

```
total = 0
my_list = [-5, 1, True, None, "5", 2.4]

for item in my_list:
    total += item

print(total)
```

### Try Now:

On what items would the error occur? Why does adding `True` not produce an error?

Show/Hide

Trying to add `None` or `"5"` to `total` would result in an error. Python cannot combine numbers with non-numeric types like `None` or strings. However, booleans are implicitly represented as numbers (`True` is 1 and `False` is 0) so we can perform numeric operations with them.

We can check if the type of the item is an `int` or a `float` as follows using the builtin `type` function:

```
total = 0
my_list = [-5, 1, True, None, "5", 2.4]

for item in my_list:
    if type(item) == int or type(item) == float:
        total += item

print(total) # -1.6
```

Alternatively, we can use `isinstance` to check if an object is one of some number of types:

```
total = 0
my_list = [-5, 1, True, None, "5", 2.4]

for item in my_list:
    if isinstance(item, (float, int)):
        total += item

print(total) # -.6
```

Surprisingly, these programs output different results! This is because there are some slight differences between `type` and `isinstance`. `isinstance` says that `True` is an `int` because `True` is implicitly represented as the integer 1. While this may be a bit confusing, just know that `type` checks only the explicit type of an object, while `isinstance` checks both the explicit and implicit types of an object. In most cases, it does not make difference whether you use `type` or `isinstance`. [Back to Top](#)

## 10.3) 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:

```
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](#). [Back to Top](#)

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!

## 10.4) Even more loops!

We also wanted to discuss some more loop style conventions and extra loop operations involving range.

### 10.4.1) Loop style conventions

There are a couple of style conventions regarding loops that are worth mentioning. First, when looping over integers that are increasing one by one, we generally use a single letter variable name like `i`, which stands for integer or index.

However, when looping over the elements of an iterable, it is generally good style to use a more descriptive variable name (so that another human reading your code can easily tell what kind of object you are looping over.) While the variable name `element` (or `elt` for short) is fine for when you are iterating over assorted types, it is very generic. The following loops show some more examples of good looping variable name convention.

```
nested_nums = [[1, 2], [3, 4], [5, 6, 7]]
num_len = 0

for num_list in nested_nums:
    num_len = num_len + len(num_list)

nums = [1, 2, 5.4, 2.1]
total = 0

for num in nums:
    total = total + num

word = "hello"
new_word = ""
for char in word: # char is short for character
    new_word = char + " "
print(new_word)
```

Note that it is also good style to leave a blank line before and after an indented block of code like a for loop, conditional statement, or function. This makes separate pieces of your program easy to separate, and increases readability (think of it like the blank line or indentation between paragraphs in an essay.)

### 10.4.2) for element in x: vs. for i in range(len(x)):

Just like there is a general rule for when to use a while loop versus a for loop (basically always use a for loop unless you do not know how many times the program need to iterate), there is a general rule for using something like `for element in x:` versus `for i in range(len(x)):`. Generally, it is better to use the former (loop over the elements directly) *unless* you need the index for some purpose.

While you can always loop over the indices to imitate looping over the elements:

[Back to Top](#)

```
word = "hello"
new_word = ""
for i in range(len(word)):
    char = word[i]
    new_word = char + " "
print(new_word)
```

This requires an extra variable. And if we removed the `char` variable, like in the program below, then we are left with using `word[i]`, which is fine, but is not as clear and descriptive as the variable `char`.

```
word = "hello"
new_word = ""
for i in range(len(word)):
    new_word = word[i] + " "
print(new_word)
```

We can also loop over elements to imitate looping over the indices. For example, if we were interested in displaying the index and the character in a string, we could technically use a separate variable to keep track of the index:

```
word = "hello"
i = 0
for char in word:
    print(char, i)
    i = i + 1
```

But then we would have to remember to update the value of `i`, and so it would be simpler to loop over the indices as follows:

```
word = "hello"
for i in range(len(word)):
    print(word[i], i)
```

### 10.4.3) Additional range arguments: `range(start, stop, step)`

It turns out that `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))`).

[Back to Top](#)

## 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

<sup>1</sup> In fact, with the subset of Python we have learned so far, it is possible to prove that we have everything we need to solve *any 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*).

<sup>2</sup> Some would say "invoke" the function, and we may use both terms interchangeably here

<sup>3</sup> or, interchangeably, a *parameter*

<sup>4</sup> In the next set of readings, we'll see that the return value of a function can even be a function itself!

<sup>5</sup> `def` is short for **define**, or **define function**, depending on whom you ask.

<sup>6</sup> 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.

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

<sup>8</sup> Note this section was adapted from [6.101's readings](#)