Chapter 4

Functions

In this chapter we consider another important component of useful programs: *functions*. A programmer often needs to solve a problem or accomplish a desired task but may not be given a precise specification of *how* the problem is to be solved. Determining the "how" and implementing the solution is the job of the programmer. For all but the simplest programs, it is best to divide the task into smaller tasks and associate these subtasks with *functions*.

Most programs consist of a number of functions.[1] When the program is run, the functions are *called* or *invoked* to perform their particular part of the overall solution. We have already used a few of Python's built-in functions (e.g., `print()` and `input()`), but most programming languages, including Python, allow programmers to create their own functions.[2]

The ability to organize programs in terms of functions provides several advantages over using a monolithic collection of statements, i.e., a single long list of statements:

- Even before writing any code, functions allow a hierarchical approach to *thinking about* and solving the problem. As mentioned, one divides the overall problem or task into smaller tasks. Naturally it is easier to think about the details involved in solving these subtasks than it is to keep in mind all the details required to solve the entire problem. Said another way, the ability to use functions allows us to adopt a *divide and conquer* approach to solve a problem.

- After determining the functions that are appropriate for a solution, these functions can be implemented, tested, and debugged, individually. Working with individual functions is often far simpler than trying to implement and debug monolithic code.

- Functions provide *modularization* (or compartmentalization). Suppose a programmer associates a particular task with a particular function, but after implementing this function, the programmer determines a better way to implement what this function does. The programmer can rewrite (or replace) this function. It the programmer does not change the way data are entered into the function nor the way data are returned by the function, the programmer can replace this function without fear of affecting any other aspect of the overall program. (With monolithic code, this is generally not the case.)

From the file: `functions.tex`

[1] The term *function* can mean many things. We will not attempt to completely nail down a meaning here. Instead, we will introduce various aspects of what constitutes a function and hope the definition eventually becomes self-evident.

[2] What we are calling functions might be called subroutines or methods in other languages.

- Functions facilitate code reuse. Often some of the smaller subtasks that go into solving one particular problem are common to other problems. Once a function has been properly implemented, it can be reused in any number of programs.

To further motivate the use of functions, consider the BMI calculation performed in Listing 3.9. In this code the calculation is entered as an expression that is evaluated once. Surrounding code provides the input and output (I/O). But what if one wants to calculate the BMI for several people? Ideally one would *not* have to enter all the statements each time the calculation is performed. This is a situation where functions are useful: these statements can be placed in a function and then the function can be called (or invoked) whenever the calculation needs to be performed.

Broadly speaking, in Python, and in several other languages, functions are used in one of three ways:

1. A function may be called for the value it returns. In some ways, functions like this are similar to the mathematical functions with which you are familiar. Usually some values are passed into the function as arguments (or, more formally, parameters). The function generates new data based on these values and then *returns* this new data to the point in the program where the function was called. A function that returns data is said to be a *non-void function*. (We will soon see how a function returns data.)

2. A function may be called solely for its *side effects*. A side effect is an action the function takes that does not produce a return value. For example, a function may be called to modify the values in a list of data. As another example, a function may be called to generate some output using one or more `print()` statements. It is important not to confuse output, such as a `print()` statement produces, with the return value of a function. (We say more related to this point below.) If the function does not explicitly return data, we will say it is a *void function*.

3. A function may be called both for its side effects and its return value. Such a function is also non-void (since it explicitly returns something), but the code that calls the function does not necessarily use the value returned by the function. In some cases, when the side effect is the only action of interest, the return value is ignored.

In the following we consider both void and non-void functions.[3]

4.1 Void Functions and `None`

Before turning to the creation of our own functions, let us further consider the `print()` function. As we've seen, the `print()` function produces output, but it is, in fact, a *void function*. If a function doesn't explicitly return data and yet it is used in an expression, the function evaluates to `None` (`None` is one of the keywords listed in Listing 2.15). `None` is, in some ways, Python's way of saying, "I'm nothing. I don't exist."

To illustrate how `None` can appear in code, consider Listing 4.1.

[3] In the strictest sense, all Python functions are non-void in that even the ones that do not explicitly return a value will return `None`. This point is discussed in the following pages. Nevertheless, we will identify functions that only return `None` as void functions.

Listing 4.1 Demonstration that `print()` is a void function that returns `None`. This code also demonstrates that `None` generally cannot be used in expressions.

```
1  >>> type(None)  # Check None's type.
2  <class 'NoneType'>
3  >>> None + 2    # Try to use None in an arithmetic operation.
4  Traceback (most recent call last):
5    File "<stdin>", line 1, in <module>
6  TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
7  >>> # Assign return value of print() to x.  Since print() is a void
8  >>> # function, x is set to None.
9  >>> x = print("Salutations!")
10 Salutations!
11 >>> type(x)      # Check x's type.
12 <class 'NoneType'>
13 >>> x           # Nothing is echoed with entry of x.
14 >>> print(x)    # print() shows us x is None.
15 None
16 >>> x * 2       # Cannot use x (None) in arithmetic expression.
17 Traceback (most recent call last):
18   File "<stdin>", line 1, in <module>
19 TypeError: unsupported operand type(s) for *: 'NoneType' and 'int'
```

Line 1 uses `type()` to check the type of `None`. We see it is of type `NoneType`. Again, this essentially says "I don't exist." As line 3 indicates, `None` cannot be used in an arithmetic operation.

In line 9 the assignment operator is used to assign whatever the `print()` function returns to the variable `x`. However, *although the* `print()` *produces output, it does not explicitly return anything.* Since `print()` is a void function, the statement in line 9 sets `x` to `None`. This fact is demonstrated by checking `x`'s type as is done in line 11. In line 13, `x` is entered on a line by itself. In the interactive environment, this usually results in the value of the variable being echoed to the programmer. However, if the value of the variable is `None`, then no output is produced. If the variable is printed using a `print()` statement, as is done in line 14, then the output is the word `None`. Line 16 again illustrates that `None` (whether the literal or in the form of a variable) cannot be used in an arithmetic operation.

It may not be obvious at the moment, but the values returned by functions will be extremely important in much of what we will do later. Also, at some point in the future, you are likely to encounter a bug that involves `None` showing up in your code. So, although you do not need to have a deep understanding of `None`, you should have a feel for how it can appear in your code and what it represents.

4.2 Creating Void Functions

To create a function, one uses the template shown in Listing 4.2 where the terms in angle brackets are replaced with appropriate values as discussed below.

Listing 4.2 Template for defining a function.

```
def <function_name>(<parameter_list>):
    <body>
```

In this template the first line is known as the *header*. The first word of the header is the keyword `def` which you should think of as *define*, i.e., we are defining a new function. This keyword is followed by the function's name which is any identifier that follows the rules given in Sec. 2.6. The parentheses following the function's name are mandatory. They enclose a list of *formal* parameters that are identifiers separated by commas (i.e., a list of variable names). The actual values assigned to these parameters are established when the function is called; thus, expressions enclosed in parentheses when the function is called are known as the *actual* parameters. We will return to this point shortly. It should also be noted that a function is not required to have any parameters. If a function has no parameters, the parentheses enclose nothing. The parentheses are followed by a colon which signals the end of the header. The body of the function starts on the second line and is given in *indented code*.[4] The body can consist of any number of statements and it continues until the code is no longer indented. Indentation of statements must be at the same level. (In an integrated development environment such as IDLE, when one hits return, the proper indentation will generally be provided for you. To terminate the definition of the function in an interactive environment, you simply enter a blank line. When using IDLE to edit a file, you may have to use the delete key to remove the indentation that is automatically entered at the start of a line.)

The function can be terminated by a `return` statement, which ends execution of the function and returns execution to the location where the function was called.[5] If no `return` statement is present, execution continues until the end of the body, at which point execution returns to the point in the program where the function was called. (Thus, a `return` statement at the end of the body does not affect the execution in any way, but it is often included in the function definition to signal the end of the function.)

A function is not executed when it is defined. Rather, to execute (or call, or invoke) a function, one writes the function's name followed by parentheses with the requisite number of (actual) parameters.

Listing 4.3 provides a demonstration of the creation of two void functions. When a line ends with a colon, the Python interpreter knows more input is needed. Thus, in line 3, the prompt changes to indicate more input is expected. As you add lines of indented code, the interactive prompt will continue to be three dots. In the interactive environment you must use a blank line to terminate a function's body.[6]

[4] More formally, the body is known as the `suite` and the header and suite together are considered a *clause*.

[5] In Python, as with most computer languages, statements are executed sequentially, one after the other. However, when a program is run, in some sense the order in which statements are executed may not be the same as the order in which the statements were written. Later we will consider statements which explicitly alter the flow of execution (e.g., `if` statements or `for` loops which can tell Python to skip or repeat statements, respectively). However, when there is an expression in which a function is called, this effectively tells Python to not proceed further until the statements in the body of that particular function have been executed.

[6] In the command-line interactive environment, "semi-blank" lines can be included in the body of a function. In this case the indentation must be present, but the rest of the line is left blank. This can sometimes enhance the readability

Listing 4.3 Demonstration of the creation of two functions that do not explicitly return anything (hence they evaluate to `None` if used in an expression). Note that the function `say_hi()` is used in the body of the function `greet_user()`.

```
1  >>> # Create Hello-World function that takes no parameters.
2  >>> def say_hi():
3  ...     print("Hello World!")
4  ...
5  >>> say_hi()  # Check that function works properly.
6  Hello World!
7  >>> # Create function to greet a user.
8  >>> def greet_user(name):
9  ...     say_hi()
10 ...     print("Oh!  And hello ", name, "!", sep="")
11 ...     return
12 ...
13 >>> greet_user("Starbuck")  # Provide string literal as argument.
14 Hello World!
15 Oh!  And hello Starbuck!
16 >>> the_whale = "Moby Dick"
17 >>> greet_user(the_whale)   # Provide string variable as argument.
18 Hello World!
19 Oh!  And hello Moby Dick!
```

In lines 2 and 3 we define the function `say_hi()` which takes no parameters. The body of this function consists of a single `print()` statement. In line 5 this function is called to ensure that it works properly. The output on line 6 shows that it does.

In lines 8 through 11 the function `greet_user()` is defined. This function has a single parameter called `name`. The first line in the body of this function (line 9 in the overall listing) is a call to the function `say_hi()`. This is followed by a `print()` statement that has as one of its arguments the `name` parameter. The third line of the body is a `return` statement. When this function is called, the statements in the body are executed sequentially. Thus, first the `say_hi()` function is executed, then the `print()` statement is executed, then the `return` statement instructs the interpreter to return to the point in the code where this function was called. Although it is not an error to have statements in the body of the function after a `return` statement, such statements would never be executed.

The order in which the `say_hi()` and `greet_user()` functions are defined is not important. Either could have been defined first. However, it is important that every function is defined before it is called. Thus, both the `say_hi()` and `greet_user()` functions must be defined before the `greet_user()` function is called. (If one defines the `greet_user()` first and then called it before the `say_hi()` function was defined, an error would occur because the interpreter would not know what to do with the call to `say_hi()` that appears in the body of `greet_user()`.)

In line 13 the `greet_user()` function is called with the string literal `"Starbuck"` as the argument. This is the *actual* parameter for this particular invocation of the function. Whenever

of a multi-line function.

a function is called, the parameters that are given are the actual parameters. Recall that when the function was defined, the list of parameters in parentheses were the *formal* parameters, and these consisted of identifiers (i.e., variable names). Before the body of the function is executed, *the actual parameters are assigned to the formal parameters*. So, just before the body of the `greet_user()` function is executed, it is as if this statement were executed:

```
name = "Starbuck"
```

The output in lines 14 and 15 shows that the function works as intended.

In line 17 `greet_user()` is called with a variable that has been defined in line 16. So, for this particular invocation, before the body of the function executes, it is as if this statement had been executed:

```
name = the_whale
```

Since `the_whale` has been set to `Moby Dick`, we see this whale's name in the subsequent output in lines 18 and 19.

Now let's return to the calculation of the BMI which was previously considered in Listing 3.9. Listing 4.4 shows one way in which to implement a function that can calculate a person's BMI.

Listing 4.4 Calculation of BMI using a void function. Here all input and output are handled within the function `bmi()`.

```
1  >>> def bmi():          # Define function.
2  ...     weight = float(input("Enter weight [pounds]: "))
3  ...     height = float(input("Enter height [inches]: "))
4  ...     bmi = 703 * weight / (height * height)
5  ...     print("Your body mass index is:", bmi)
6  ...
7  >>> bmi()               # Invoke bmi() function.
8  Enter weight [pounds]: 160
9  Enter height [inches]: 72
10 Your body mass index is: 21.69753086419753
11 >>> bmi()               # Invoke bmi() function again.
12 Enter weight [pounds]: 123
13 Enter height [inches]: 66
14 Your body mass index is: 19.8505509642
```

In lines 1 through 5 the function `bmi()` is defined. It takes no parameters and handles all of its own input and output. This function is called in lines 7 and 11. Note the ease with which one can now calculate a BMI.

4.3 Non-Void Functions

Non-void functions are functions that return something other than `None`. The template for creating a non-void function is identical to the template given in Listing 4.2 for a void function. What

distinguishes the two types of functions is how `return` statements that appear in the body of the function are implemented. In the previous section we mentioned that a `return` statement can be used to explicitly terminate the execution of a function. When the `return` statement appears by itself, the function returns `None`, i.e., it behaves as a void function. However, to return some other value from a function, one simply puts the value (or an expression that evaluates to the desired value) following the keyword `return`.

Listing 4.5 demonstrates the creation and use of a non-void function. Here, as mentioned in the comments in the first two lines, the function calculates a (baseball) batting average given the number of hits and at-bats.

Listing 4.5 Demonstration of the creation of a non-void function.

```
1  >>> # Function to calculate a batting average given the number of
2  >>> # hits and at-bats.
3  >>> def batting_average(hits, at_bats):
4  ...     return int(1000 * hits / at_bats)
5  ...
6  >>> batting_average(85, 410)
7  207
8  >>> # See what sort of help we get on this function.  Not much...
9  >>> help(batting_average)
10 Help on function batting_average in module __main__:
11
12 batting_average(hits, at_bats)
```

Note that the `return` statement in line 4 constitutes the complete body of the function. In this case the function returns the value of the expression immediately following the keyword `return`. Lines 6 and 7 show what happens when the function is invoked with 85 hits and 410 at-bats. As mentioned, before the body of the function is executed, the actual parameters are assigned to the formal parameters. So, for the statement in line 6, before the body of `batting_average()` is executed, it is as if these two statements had been issued:

```
hits = 85
at_bats = 410
```

The expression following the keyword `return` in the body of the function evaluates to `207` for this input (a batting average of "207" means if this ratio of hits to at-bats continues, one can anticipate 207 hits out of 1,000 at-bats). Because the `batting_average()` function was invoked in the interactive environment and its return value was not assigned to a variable, the interactive environment displays the return value (as given on line 7).

In line 9 the `help()` function is called with an argument of `batting_average` (i.e., the function's name without any parentheses). The subsequent output in lines 10 through 12 isn't especially helpful although we are told the parameter names that were used in defining the function. If these are sufficiently descriptive, this may be useful.

Listing 4.6 demonstrates the creation of a slightly more complicated non-void function. The goal of this function is to determine the number of dollars, quarters, dimes, nickels, and pennies in

a given "total" (where the total is assumed to be in pennies). Immediately following the function header, a multi-line string literal appears that describes what the function does. This string isn't assigned to anything and thus it is seemingly discarded by the interpreter. However, when a string literal is given immediately following a function header, it becomes what is known as a "docstring" and will be displayed when help is requested for the function. This is demonstrated in lines 16 through 21. So, rather than describing what a function does in comments given just prior to the definition of a function, we will often use a docstring to describe what a function does. When more than one sentence is needed to describe what a function does, it is recommended that the docstring be started by a single summary sentence. Next there should be a blank line which is then followed by the rest of the text.

Listing 4.6 A function to calculate the change in terms of dollars, quarters, dimes, nickels, and pennies for a given "total" number of pennies

```
1  >>> # Function to calculate change.
2  >>> def make_change(total):
3  ...     """
4  ...     Calculate the change in terms of dollars, quarters,
5  ...     dimes, nickels, and pennies in 'total' pennies.
6  ...     """
7  ...     dollars, remainder  = divmod(total, 100)
8  ...     quarters, remainder = divmod(remainder, 25)
9  ...     dimes, remainder    = divmod(remainder, 10)
10 ...     nickels, pennies    = divmod(remainder, 5)
11 ...     return dollars, quarters, dimes, nickels, pennies
12 ...
13 >>> make_change(769)
14 (7, 2, 1, 1, 4)
15 >>> # Try to get help on this function.  "Docstring" is given!
16 >>> help(make_change)
17 Help on function make_change in module __main__:
18
19 make_change(total)
20     Calculate the change in terms of dollars, quarters,
21     dimes, nickels, and pennies in 'total' pennies.
```

Technically `return` statements can only return a single value. It might appear that the code in Listing 4.6 violates this since there are multiple values (separated by commas) in the `return` statement in line 11. However, in fact, these values are collected together and returned in a single collection known as a *tuple*. Later we will study tuples and other collections of data. For now it suffices to know that it is possible to get multiple values from a function simply by putting them in a comma-separated list in the `return` statement. The output in line 14 shows that `769` pennies is the equivalent of seven dollars, two quarters, one dime, one nickel, and four pennies. (One could use simultaneous assignment to store these values to individual variables.)

Now let us again revisit the calculation of a BMI. In the `bmi()` function of Listing 4.4, the user was prompted for a weight and height for each invocation of the function. However, there may

be applications for which weight and height data are stored in a file and thus it doesn't make sense to prompt for these values. In anticipation of situations such as this, it may be better to separate the calculation of the BMI from the code that handles the input (before the calculation) and the output (after the calculation).

Listing 4.7 demonstrates one way this might be done. Here three functions are implemented. `get_wh()` gets the weight and height; `calc_bmi()` calculates the BMI (given the weight and height as parameters); and `show_bmi()` displays the BMI (which it is given as a parameter).

Listing 4.7 Reimplementation of the BMI calculation where the calculation itself has been separated from the handling of the input and output.

```
1  >>> def get_wh():
2  ...     """Obtain weight and height from user."""
3  ...     weight = float(input("Enter weight [pounds]: "))
4  ...     height = float(input("Enter height [inches]: "))
5  ...     return weight, height
6  ...
7  >>> def calc_bmi(weight, height):
8  ...     """
9  ...     Calculate body mass index (BMI) for weight in
10 ...     pounds and height in inches.
11 ...     """
12 ...     return 703 * weight / (height * height)
13 ...
14 >>> def show_bmi(bmi):
15 ...     print("Your body mass index is:", bmi)
16 ...
17 >>> w, h = get_wh()
18 Enter weight [pounds]: 280
19 Enter height [inches]: 72
20 >>> bmi = calc_bmi(w, h)
21 >>> show_bmi(bmi)
22 Your body mass index is: 37.9706790123
```

Lines 1 through 16 define the three functions `get_wh()`, `calc_bmi()`, and `show_bmi()`. In line 17, `get_wh()` is called to obtain the weight and height. This leads to the user being prompted for input as shown in lines 18 and 19. The values the user enters are returned by `get_wh()` and assigned, simultaneously, to the variables `w` and `h`. (As will be discussed in the next section, the `weight` and `height` variables defined in the function `get_wh()` are not defined outside of that function. Thus, we need to store the values returned by the function. By "store" we mean assign the values to variables for later use.)

In line 20, `calc_bmi()` is called to calculate the BMI. If we have the weight and height (whether read from a file or obtained via the keyboard), this function can be used to calculate the BMI. The value this function returns is stored in the variable `bmi` which is subsequently passed as a parameter to `show_bmi()` in line 21 to obtain the desired output.

4.4 Scope of Variables

For now, you should always use parameters to pass data into a function and always use the `return` statement to obtain data from a function. (There are other ways to exchange data with a function using what are known as *global variables*, but the use of global variables is generally considered to be a dangerous programming practice.) Something that you need to keep in mind is that all the parameters and variables that are defined in a function are *local* to a function, meaning that these variables cannot be "seen" by code outside of the function.

The code over which a variable is accessible or visible is known as the variable's scope. The scope of a variable within a function is from the point it is created (either in the parameter list or in the body via an assignment operation) to the end of the function.

Although scope is an important and, at times, complicated issue, it is not something we want to belabor here. Nevertheless, there are a few issues related to scope about which you should be aware as this awareness may help prevent bugs from appearing in your code.

Listing 4.8 again uses the `get_wh()` function that was used in Listing 4.7.

Listing 4.8 Demonstration that variables are local to a function.

```
1  >>> def get_wh():
2  ...     """Obtain weight and height from user."""
3  ...     weight = float(input("Enter weight [pounds]: "))
4  ...     height = float(input("Enter height [inches]: "))
5  ...     return weight, height
6  ...
7  >>> get_wh()
8  Enter weight [pounds]: 120
9  Enter height [inches]: 60
10 (120.0, 60.0)
11 >>> print(weight)
12 Traceback (most recent call last):
13   File "<stdin>", line 1, in <module>
14 NameError: name 'weight' is not defined
15 >>> print(height)
16 Traceback (most recent call last):
17   File "<stdin>", line 1, in <module>
18 NameError: name 'height' is not defined
```

Lines 1 through 5 define `get_wh()` as before. This function is invoked in line 7. The user enters `120` for the weight and `60` for the `height`. Since the return value of this function is not assigned to anything, the interactive environment displays the return value in line 10 (were this code run from within a file, the return value would simply be ignored—it would not appear as output). In lines 11 and 15 an attempt is made to print the `weight` and `height`, respectively, that exist in the `get_wh()` function. Both these attempts produce `NameErrors` with a statement that the variables are not defined. Again, these variables exist in the `get_wh()` function and only in `get_wh()`.

In Sec. 2.7 we discussed namespaces and how Python uses them to associate identifiers with values. Python actually maintains multiple namespaces. When a function is defined, there is a namespace created for that particular function. When a identifier is referenced within a function, Python will look in that function's namespace for the associated value. If it cannot find that identifier, it will look in the surrounding scope, i.e., in the namespace associated with place from which the function was called. (For example, if we call a function from the interactive environment, the *global* namespace is the surrounding scope for that function call.) However, if an identifier is referenced outside of a function, Python will never look inside the namespaces of functions to try to determine the value of that identifier. Similarly, if an identifier is referenced in one function, Python will not look in another function's namespace to try to determine the value of that identifier.

Now consider the code shown in Listing 4.9 which further demonstrates the local nature of variables. This code is quite confusing in that the variable name `weight` is used throughout the code. Nevertheless, the variable `weight` in the function `change_weight()` is different from the variable `weight` defined outside the function!

Listing 4.9 Another demonstration of the local scope of variables within a function.

```
1 >>> weight = 160                    # Set initial weight to 160.
2 >>> def change_weight(weight):     # Define function to change weight.
3 ...     weight = weight - 10
4 ...     print("weight is now", weight)
5 ...
6 >>> change_weight(weight)          # Claim is weight is now 150.
7 weight is now 150
8 >>> print(weight)                  # Check on weight.  Still 160!
9 160
```

In line 1 `weight` is set to `160`. In lines 2 through 4 a function is defined whose name implies it will change the `weight`. The function is invoked in line 6. The output in line 7 shows that `weight` is now 150, i.e., 10 less than the value that was passed to the function. *However*, one needs to realize that the variable `weight` within the `change_weight()` function is different from the variable `weight` that lives outside the function. The proof of this is in lines 8 and 9 where we see the value of `weight` outside the function is unchanged from its initial value, i.e., it is still `160`.

So, how can one employ a function to change the value of a variable? To change the value of a `float`, `int`, or `str` variable, you must assign a new value to the variable. This assignment must be done within the desired scope. To use a function to change a variable, the function should return the desired value and then this should be assigned to the variable. The code in Listing 4.10 illustrates this (note that the `change_weight()` function defined in Listing 4.10 is different from the one defined in Listing 4.9).

Listing 4.10 Demonstration of a way in which a function can be used to change the value of an `int` variable. Here one must assign the return value of the function back to the variable.

```
>>> weight = 160                # Initialize weight to 160.
>>> def change_weight(weight): # Define function to return new weight.
...     return weight - 10
...
>>> weight = change_weight(weight) # Assign return value to weight.
>>> print(weight)                  # Confirm that weight has changed.
150
```

4.5 Scope of Functions

As described in Secs. 4.2 and 4.3, a `def` statement is used to create a function. So far, all the functions we have created have been defined outside the body of any other function. Defining a function in this way gives the function *global scope* meaning it is visible everywhere—we can call a function with global scope from within another function or call it while outside of any function (i.e., directly from the interactive prompt).

However, it is also possible to define a function within the body of another. When this is done, the inner function can only be invoked from within the function in which it was defined.[7] Thus the scope of the inner function is local to the outer function. This is demonstrated in Listing 4.11 where the function `g0()` is defined in the body of `f0()`. `g0()` returns the cube of its argument. `f0()` passes its argument to `g0()`, raises that to the fourth power, adds `4`, and returns the resulting value.

Listing 4.11 Demonstration that one function can be defined within another. However, when this is done, the scope of the inner function is restricted to the outer function in which it was defined.

```
>>> def f0(x):
...     def g0(y):
...         return y ** 3 + 3
...     return g0(x) ** 4 + 4
...
>>> f0(4)
20151125
>>> g0(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'g0' is not defined
>>> def f1(x):
...     return g1(x) ** 4 + 4
...
>>> def g1(x):
...     return x ** 3 + 3
...
```

[7] Also, the function can only be called *after* it is defined.

```
18 >>> f1(4)
19 20151125
20 >>> g1(4)
21 67
```

The `f0` and `g0()` functions are defined in lines 1 through 4. The body of `g0()` consists of a single return statement (line 3). The body of `f0()` consists of the definition of `g0()` and then its return statement in line 4. Lines 6 and 7 show the result produced by calling `f0()` with an argument of 4.

In line 8 an attempt is made to call `g0()`. This produces a `NameError` with the statement that `g0()` is not defined. Again, the scope of `g0()` is restricted to `f0()` and thus there is no other way to access this function. There are situations in which this sort of *encapsulation* is considered advantageous. Encapsulation is the practice of restricting the scope of programming elements, such as variables and functions, so that these elements are accessible only where they are needed. If a function, such as `g0()`, is only needed within another function, such as `f0()`, then this type of nesting can be considered a good programming practice.

However, many languages do not allow the nesting of function definitions as Python does. Thus, we will generally restrict our function definitions to be in the *global scope*, external to any function. Lines 12 through 21 demonstrate an alternative way of implementing `f0()`. This alternative approach creates the functions `f1()` and `g1()` (which perform calculations equivalent to `f0()` and `g0()`, respectively). Since `g1()` is defined in the global scope, it can be called from within `f1()` but also external to `f1()` as line 20 demonstrates.

Ultimately the scoping rules for functions are no different than for variables: anything defined inside a function is local to that function. Variables and functions defined external to any function have global scope and are visible "everywhere."

4.6 print() vs. return

In some situations `print()` statements and `return` statements appear to behave the same way. *However,* `print()` *and* `return` *are fundamentally different and a failure to understand how they differ can lead to errors.* To illustrate this, consider the code in Listing 4.12. Two functions are defined, `p()` and `r()`. Both take a single argument and both multiply this argument by `2`. `p()` *prints* the result of this multiplication whereas `r()` *returns* this result.

Listing 4.12 Demonstration of the difference between a function that prints a value and one that returns the same value.

```
1 >>> def p(x):              # p() prints a result
2 ...     print(2 * x)
3 ...
4 >>> def r(x):              # r() returns a result
5 ...     return 2 * x
6 ...
7 >>> p(10)
```

```
8  20
9  >>> r(10)
10 20
11 >>> p_result = p(10)    # Use p() in assignment operation.  Note output.
12 20
13 >>> r_result = r(10)    # Use r() in assignment operation.  No output!
14 >>> p_result, r_result
15 (None, 20)
16 >>> p(10) + 7
17 20
18 Traceback (most recent call last):
19   File "<stdin>", line 1, in <module>
20 TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
21 >>> r(10) + 7
22 27
```

After defining the two function, `p()` is called in line 7 with an argument of `10`. On line 8 we see the value that is *printed* by this function, i.e., 20. Since `p()` doesn't explicitly return a value, it is a void function.

In line 9 `r()` is called, also with an argument of `10`. In this case `r()` *returns* the result, which is also `20`. We see the result on line 10 *only* because the interactive environment displays the value of an expression if the expression is the only thing on the line. To further illustrate this point, consider the next few lines.

An assignment statement appears in line 11 where `p()` is to the right of the assignment operator and `p_result` is the variable to the left. Note that `20` appears on line 12. This is because when the `p()` function is called, it prints two times its argument. So, what is `p_result`? Knowing that `p()` is a void function provides the answer (but the answer also appears later in Listing 4.12 itself).

Another assignment statement is given in line 13. In this case `r()` appears to the right of the operator and `r_result` is to the left. Note that, unlike when `p()` was used in a similar statement, this statement produces no output! This is a consequence of the fact that `r()` does not print anything. The value this function returns is simply assigned to `r_result` and the interactive environment doesn't display anything for such statements. Line 14 displays the values of `p_result` and `r_result`, which are `None` and `20`, respectively.

Look closely at the statement in line 16 and the subsequent result. This statement is the sum of `p(10)` and `7`. To calculate this sum, Python must first evaluate `p(10)`. Thus the `p()` function is called with an argument of `10`. Since the body of `p()` contains a `print()` statement, we see the output of that `print()` statement in line 17. Since `p()` is a void function, line 16 is ultimately equivalent to: `None + 7`. An integer cannot be added to `None` and hence this produces an error as the text in lines 18 through 20 shows. Finally, line 21 shows that `r(10)` can be used in a sum since it *returns* an integer.

The code in Listing 4.12 might leave you with the impression that `print()` statements cannot appear together with a `return` statement in a function, but this is *not* the case. For debugging purposes `print()` statements are often the quickest and easiest way to ascertain where a problem lies. Thus, when developing your code, it can often be useful to "sprinkle" your code with

`print()` statements that enable you to check that values are what they should be. There are also many other circumstances where a function should have one or more `print()` statements *and* a `return` statement.

As an example of mixing `print()` and `return` statements, consider the code shown in Listing 4.13. A function `f()` is defined that calculates a value, prints that value, and then returns it.

Listing 4.13 Demonstration that `print()` and `return` statements can appear in the same function.

```
1 >>> def f(x):
2 ...     y = 2 * (x - 3) // 14
3 ...     print("f(", x ,") = ", y, sep="")
4 ...     return y
5 ...
6 >>> z = (7 + f(94)) * 2
7 f(94) = 13
8 >>> z
9 40
```

In line 6 the `f()` function is used in an expression. When this function is evaluated, as part of that evaluation, the `print()` statement in its body is executed. Thus, in line 7, the output we see is the output produced by the `print()` statement in `f()` (line 3 of the listing). The value that `f(94)` *returns* (which is `13`) is used in the arithmetic expression in line 6 and `z` is assigned the value `40`.

4.7 Using a `main()` Function

In some popular computer languages it is required that every program has a function called `main()`. This function tells the computer where to start execution. Since Python is a *scripted* language in which execution starts at the beginning of a file and subsequent statements are executed sequentially, there is no requirement for a `main()` function. Nevertheless, many programmers carry over this convention to Python—they write programs in which, other than function definitions themselves, nearly all statements in the program are contained within one function or another. The programmers create a `main()` function and often call this in the final line of code in the program. Thus, after all the functions have been defined, including `main()` itself, the `main()` function is called.

Listing 4.14 provides an example of a program that employs this type of construction. Here it is assumed this code is stored in a file (hence the interactive prompt is not shown). When this file is executed, for example, in an IDLE session, the last statement is a call to the `main()` function. Everything prior to that final invocation of `main()` is a function definition. `main()` serves to call the other three functions that are used to prompt for input, calculate a BMI, and display that BMI.

Listing 4.14 A BMI program that employs a `main()` function which indicates where computation starts. `main()` is invoked as the final statement of the program. (Since the definitions of the first three functions are unchanged from before, the comments and docstrings have been removed.)

```
def get_wh():
    weight = float(input("Enter weight [pounds]: "))
    height = float(input("Enter height [inches]: "))
    return weight, height

def calc_bmi(weight, height):
    return 703 * weight / (height * height)

def show_bmi(bmi):
    print("Your body mass index is:", bmi)

def main():
    w, h = get_wh()
    bmi = calc_bmi(w, h)
    show_bmi(bmi)

main()  # Start the calculation.
```

In an IDLE session, after this file has been run and one BMI calculation has been performed, you can perform any number of subsequent BMI calculations simply by typing `main()` at the interactive prompt.

4.8 Optional Parameters

Python provides many built-in functions. The first function we introduced in this book was the built-in function `print()`. The `print()` function possesses a couple of interesting features that we don't yet know how to incorporate into our own functions: `print()` can take a variable number of parameters, and it also accepts optional parameters. The optional parameters are `sep` and `end` which specify the string used to separate arguments and what should appear at the end of the output, respectively. To demonstrate this, consider the code shown in Listing 4.15.

Listing 4.15 Demonstration of the use of the `sep` and `end` optional parameters for `print()`.

```
>>> # Separator defaults to a blank space.
>>> print("Hello", "World")
Hello World
>>> # Explicitly set separator to string "-*-".
>>> print("Hello", "World", sep="-*-")
Hello-*-World
>>> # Issue two separate print() statements.  (Multiple statements can
>>> # appear on a line if they are separated by semicolons.)
```

```
>>> # By default, print() terminates its output with a newline.
>>> print("Why,", "Hello"); print("World!")
Why, Hello
World!
>>> # Override the default separator and line terminator with the
>>> # optional arguments of sep and end.
>>> print("Why,", "Hello", sep="-*-", end="^v^"); print("World!")
Why,-*-Hello^v^World!
```

In line 2 `print()` is called with two arguments. Since no optional arguments are given, the subsequent output has a blank space separating the arguments and the output is terminated with a newline. In line 5 the optional argument is set to `-*-` which then appears between the arguments in the output, as shown in line 6. In line 10 two `print()` statements are given (recall that multiple statements can appear on a single line if they are separated by semicolons). The output from each of these statements is terminated with the default newline characters as shown implicitly in the subsequent output in lines 11 and 12. Line 15 again contains two `print()` statements but the first `print()` statement uses the optional parameters to set the separator and line terminator to the strings `-*-` and `^v^`, respectively.

As explained earlier in this chapter, we create functions in Python using a `def` statement. In the header, enclosed in parentheses, we include the list of formal parameters for the function. Python provides several different constructs for specifying how the parameters of a function are handled. We can, in fact, define functions of our own which accept an arbitrary number of arguments and employ optional arguments. Exploring all the different constructs can be a lengthy endeavor and the use of multiple arguments isn't currently of interest. However, creating functions with optional arguments is both simple and useful. Thus, let's consider how one defines a function with optional parameters.

First, as shown in Listing 4.16, let's define a function without optional parameters which squares its argument. The function is defined in lines 1 and 2 and then, in line 3, invoked with an argument of `10`.

Listing 4.16 Simple function to square its argument. Here the function is invoked by passing it the actual parameter which is the literal `10`.

```
>>> def square(x):
...     return x * x
...
>>> square(10)
100
```

Recall that when we write `square(10)`, the *actual* parameter is `10`. The actual parameter is assigned to the formal parameter and then the body of the function is executed. So, in this example, where we have `square(10)`, it is as if we had issued the statement `x=10` and then executed the body of the `square()` function.

If we so choose, we can explicitly establish the connection between the formal and actual parameters when we call a function. Consider the code in Listing 4.17 where the `square()`

function is defined exactly as above. Now, however, in line 4, when the function is invoked, the formal parameter `x` is explicitly set equal to the actual parameter `10`.

Listing 4.17 The same square function is defined as in Listing 4.16. However, in line 4, when the function is invoked, the formal parameter `x` is explicitly assigned the value of the actual parameter `10`.

```
1 >>> def square(x):
2 ...     return x * x
3 ...
4 >>> square(x=10)
5 100
```

Note carefully what appears in the argument when the `square()` function is called in line 4. We say that `x` is assigned `10`. This assignment is performed, the body of the function is then executed, and, finally, the returned value is `100` which is shown on line 5. In practice, for this simple function, there is really no reason to explicitly assign `10` to `x` in the invocation of the function. However, some functions have many arguments, some of which are optional and some of which are not. For these functions, explicitly assigning values to the parameters can aid readability (even when the parameters are required).

What happens if we use a different variable name (other than `x`) when we invoke the `square()` function? Listing 4.18 provides the answer.

Listing 4.18 The `square()` function is unchanged from the previous two listings. When the function is invoked in line 4, an attempt is made to assign a value to something that is not a formal parameter of the function. This produces an error.

```
1 >>> def square(x):
2 ...     return x * x
3 ...
4 >>> square(y=10)
5 Traceback (most recent call last):
6   File "<stdin>", line 1, in <module>
7 TypeError: square() got an unexpected keyword argument 'y'
```

In line 4 we tried to assign the value `10` to the variable `y`. But, the `square()` function doesn't have any formal parameter named `y`, so this produces an error (as shown in lines 5 through 7). We have to use the formal parameter name that was used when the function was defined (in this case `x`).

Now, with this background out of the way: To create a function with one or more optional parameters, we simply assign a default value to the corresponding formal parameter(s) in the header of the function definition. An example helps to illustrate this.

Let's create a function called `power()` that raises a given number to some exponent. The user can call this function with either one or two arguments. The second argument is optional and corresponds to the exponent. If the exponent is not given explicitly, it is assumed to be `2`, i.e.,

the function will square the given value. The first argument is required (i.e., not optional) and represents the number that should be raised to the given exponent. The code in Listing 4.19 shows how to implement the `power()` function. Notice that in the header of the function definition (line 1), we simply assign a default value of `2` to the formal parameter `exponent`.

Listing 4.19 Function to raise a given number to an exponent. The exponent is an optional parameter. If the exponent is not explicitly given when the function is invoked, it defaults to `2` (i.e., the function returns the square of the number).

```
1  >>> def power(x, exponent=2):
2  ...     return x ** exponent
3  ...
4  >>> power(10)      # 10 squared, i.e., 10 ** 2.
5  100
6  >>> power(3)       # 3 squared, i.e., 3 ** 2.
7  9
8  >>> power(3, 0.5) # Square root of 3, i.e., 3 ** 0.5.
9  1.7320508075688772
10 >>> power(3, 4)    # 3 ** 4
11 81
12 >>> power(2, exponent=3)    # 2 ** 3
13 8
14 >>> power(x=3, exponent=3) # 3 ** 3
15 27
16 >>> power(exponent=3, x=5) # 5 ** 3
17 125
18 >>> power(exponent=3, 5)    # Error!
19   File "<stdin>", line 1
20 SyntaxError: non-keyword arg after keyword arg
```

Lines 4 through 7 show the argument is squared when `power()` is called with a single argument, i.e., the value of the argument is assigned to the formal parameter `x` and the default value of `2` is used for `exponent`. Lines 8 through 11 show what happens when `power()` is called with two arguments. The first argument is assigned to `x` and the second is assigned to `exponent`, thus overriding `exponent`'s default value of `2`. In lines 8 and 10, the assignment of actual parameters to formal parameters is based on position—the first actual parameter is assigned to the first formal parameter and the second actual parameter is assigned to the second formal parameter (this is no different than what you have previously observed with multi-parameter functions, such as `calc_bmi()` in Listing 4.7). So, for example, based on the call in line 8, `3` is assigned to `x` and `0.5` is assigned to `exponent` (which yields $\sqrt{3}$).

In line 12 we see that the function can be called with the optional parameter explicitly "named" in an assignment statement (thus optional parameters are sometimes called *named parameters*). Line 14 shows that, in fact, both parameters can be named when the function is called. Keep in mind, however, that `x` is *not* an optional parameter—a value must be provided for `x`. If one names all the parameters, then the order in which the parameters appear is not important. This is illustrated in line 16 where the optional parameter appears first and the required parameter appears

second (here the function calculates 5^3). Finally, line 18 shows that we cannot put an optional parameter before an unnamed required parameter.

Let us consider one more example in which a function calculates the y value of a straight line. Recall that the general equation for a line is

$$y = mx + b$$

where x is the independent variable, y is the dependent variable, m is the slope, and b is the intercept (i.e., the value at which the line crosses the y axis). Let's write a function called `line()` that, in general, has three arguments corresponding to x, m, and b. The slope and intercept will be optional parameters, and the slope will have a default value of 1 and the intercept a default value of 0. Listing 4.20 illustrates both the construction and use of this function.

Listing 4.20 Function to calculate the y value for a straight line given by $y = mx + b$. The value of x must be given. However, m and b are optional with default values of 1 and 0, respectively.

```
1  >>> def line(x, m=1, b=0):
2  ...     return m * x + b
3  ...
4  >>> line(10)          # x=10 and defaults of m=1 and b=0.
5  10
6  >>> line(10, 3)       # x=10, m=3, and default of b=0.
7  30
8  >>> line(10, 3, 4)    # x=10, m=3, b=4.
9  34
10 >>> line(10, b=4)     # x=10, b=4, and default of m=1.
11 14
12 >>> line(10, m=7)     # x=10, m=7, and default of b=0.
13 70
```

In lines 4, 6, and 8, the function is called with one, two, and three arguments, respectively. In these three calls the arguments are not named; hence the assignment to the formal parameters is based solely on position. When there is a single (actual) argument, this is assigned to the formal argument `x` while `m` and `b` take on the defaults values of `1` and `0`, respectively. When there are two unnamed (actual) arguments, as in line 6, they are assigned to the first two formal parameters, i.e., `x` and `m`. However, if one of the arguments is named, as in lines 10 and 12, the assignment of actual parameters to formal parameters is no longer dictated by order.

In line 10, the first argument (`10`) is assigned to `x`. The second argument dictates that the formal parameter `b` is assigned a value of `4`. Since nothing has been specified for the value of `m`, it is assigned the default value.

4.9 Chapter Summary

The template for defining a function is:

```
def <function_name>(<params>):
    <body>
```

where the function name is a valid identifier, the *formal parameters* are a comma-separated list of variables, and the body consists of an arbitrary number of statements that are indented to the same level.

A function is called/invoked by writing the function name followed by parentheses that enclose the *actual parameters* which are also known as the *arguments*.

A function that does not explicitly return a value is said to be a *void function*. Void functions return `None`.

A variable defined as a formal parameter or defined in the body of the function is not defined outside the function, i.e., the variables only have *local scope*. Variables accessible throughout a program are said to have *global scope*.

Generally, a function should obtain data via its parameters and return data via a **`return`** statement.

`print()` does not return anything (it generates output) and the `return` statement does not print anything (it serves to return a value).

If comma-separated expressions are given as part of the `return` statement, the values of these expressions are returned as a collection of values that can be used with simultaneous assignment. The values are in a `tuple` as described in Chap. 6.

Function definitions may be nested inside other functions. When this is done, the inner function is only usable within the body of the function in which it is defined. Typically such nesting is not used.

The scoping rules for functions are the same as for variables. Anything defined inside a function, including other functions, is local to that function. Variables and functions defined external to functions have global scope and are visible "everywhere."

Often programs are organized completely in terms of functions. A function named **`main()`** is, by convention, often the first function called at the start of a program (but after defining all the functions). The statements in `main()` provide the other function calls that are necessary to complete the program. Thus, the program consists of a number of function definitions and the last line of the program file is a call to `main()`.

An optional parameter it created by assigning a default value to the formal parameter in the header of the function definition.

4.10 Review Questions

1. The following code is executed

```
def f(x):
    return x + 2, x * 2

x, y = f(5)
print(x + y)
```

What is the output produced by the `print()` statement?

(a) `7 10`

(b) `17`

(c) `x + y`

(d) This produces an error.

(e) None of the above.

2. True or False: Names that are valid for variables are also valid for functions.

3. What output is produced by the `print()` statement when the following code is executed?

```
def calc_q1(x):
    q = 4 * x + 1
    return q

calc_q1(5)
print(q)
```

(a) `24`

(b) `21`

(c) `q`

(d) This produces an error.

(e) None of the above.

4. What is the value of `q` after the following code has been executed?

```
def calc_q2(x):
    q = 4 * x + 1
    print(q)

q = calc_q2(5)
```

(a) `24`

(b) `21`

(c) This produces an error.

(d) None of the above.

5. What is the value of `q` after the following code has been executed?

```
q = 20
def calc_q3(x):
    q = 4 * x + 1
    return q

q = calc_q3(5)
```

(a) `24`

(b) `21`

(c) This produces an error.

(d) None of the above.

6. What is the output produced by the `print()` statement in the following code?

```
def calc_q4(x):
    q = 4 * x + 1

print(calc_q4(5))
```

(a) `24`

(b) `21`

(c) `q`

(d) This produces an error.

(e) None of the above.

7. What is the output of the `print()` statement in the following code?

```
abc = 5 + 6 // 12
print(abc)
```

(a) This produces an error.

(b) `5 + 6 // 12`

(c) `5`

(d) `5.5`

(e) `6`

8. What is the output of the `print()` statement in the following code?

```
def = 5 + 6 % 7
print(def)
```

(a) This produces an error.

(b) `5 + 6 % 7`

(c) `11`

(d) `4`

9. The following code is executed:

```
def get_input():
    x = float(input("Enter a number: "))
    return x

def main():
    get_input()
    print(x ** 2)

main()
```

At the prompt the user enters `2`. What is the output of this program?

(a) `x ** 2`

(b) `4`

(c) `4.0`

(d) This produces an error.

(e) None of the above.

10. The following code is executed:

```
def get_input():
    x = float(input("Enter a number: "))
    return x

def main():
    print(get_input() ** 2)

main()
```

At the prompt the user enters `2`. What is the output of this program?

(a) `get_input() ** 2`

(b) `4`

(c) `4.0`

(d) This produces an error.

(e) None of the above.

11. What is the value of `z` after the following code is executed?

```
def f1(x, y):
    print((x + 1) / (y - 1))

z = f1(3, 3) + 1
```

(a) 3

(b) 3.0

(c) 2

(d) This produces an error.

12. What is the value of z after the following code is executed?

```
def f2(x, y):
    return (x + 1) / (y - 1)

z = f2(3, 3) + 1
```

(a) 3

(b) 3.0

(c) 2

(d) This produces an error.

(e) None of the above.

13. What is the value of z after the following code is executed?

```
def f3(x, y = 2):
    return (x + 1) / (y - 1)

z = f3(3, 3) + 1
```

(a) 3

(b) 3.0

(c) 2

(d) This produces an error.

(e) None of the above.

14. What is the value of z after the following code is executed?

```
def f3(x, y = 2):
    return (x + 1) / (y - 1)

z = f3(3) + 1
```

(a) 3

(b) 3.0

(c) 2

(d) This produces an error.

(e) None of the above.

15. The following code is executed.

```
def inc_by_two(x):
    x = x + 2
    return x

x = 10
inc_by_two(x)
print("x = ", x)
```

What is the output produced by the `print()` statement?

ANSWERS: 1) b; 2) True; 3) d, `q` is not defined outside of the function; 4) d, the function is a void function and hence `q` is `None`; 5) b; 6) e, the output is `None` since this is a void function; 7) c; 8) a, `def` is a keyword and we cannot assign a value to it; 9) d, the variable `x` is not defined in `main()`; 10) c; 11) d, `f1()` is a void function; 12) b; 13) b; 14) e, `z` would be assigned `5.0`; 15) `x = 10`.

4.11 Exercises

1. Write a function called `convert_to_days()` that takes no parameters. Have your function prompt the user to input numbers of hours, minutes, and seconds. Write a helper function called `get_days()` that uses these values and converts them to days in `float` form (fractions of a day are allowed). `get_days()` should return the number of days. Use this helper function within the `convert_to_days()` function to display the numbers of days to the user. The built-in function `round()` takes two arguments: a number and an integer indicating the desired precision (i.e., the desired number of digits beyond the decimal point). Use this function to round the number of days four digits after the decimal point.

   The following demonstrates the proper behavior of `convert_to_days()`:

```
>>> convert_to_days()
Enter number of hours: 97
Enter number of minutes: 54
Enter number of seconds: 45

The number of days is: 4.0797
```

2. Write a function called `calc_weight_on_planet()` that calculates your equivalent weight on another planet. This function takes two arguments: your weight on Earth in pounds and the surface gravity of the planet of interest with units m/s^2. Make the second argument optional and supply a default value of 23.1 m/s^2 which is the approximate surface gravity of Jupiter (Earth's suface gravity is approximately 9.8 m/s^2). To perform the conversion, use

the equation: weight is equal to mass times suface gravity. Since your weight on Earth is given and you know the Earth's surface gravity, have your function use this information to calculate your mass (it is fine if, at this point, the units of mass are a mix of Imperial and the MKS system). Then, use your mass and the given surface gravity to calculate your effective weight on the other planet.

The following demonstrates the proper behavior of this function:

```
>>> calc_weight_on_planet(120, 9.8)
120.0
>>> calc_weight_on_planet(120)
282.85714285714283
>>> calc_weight_on_planet(120, 23.1)
282.85714285714283
```

3. Write a function called `num_atoms()` that calculates how many atoms are in n grams of an element given its atomic weight. This function should take two parameters: the amount of the element in grams and an optional argument representing the atomic weight of the element. The atomic weight of any particular element can be found on a periodic table but make the default value for the optional argument the atomic weight of gold (Au) 196.97 with units in grams/mole. A mole is a unit of measurement that is commonly used in chemistry to express an amount of a substance. Avogadro's number is a constant, 6.022×10^{23} atoms/mole, that can be used to find the number of atoms in a given sample. Use Avogadro's number, the atomic weight, and the amount of the element in grams to find the number of atoms present in the sample. Your function should return this value.

   The following demonstrates the proper behavior of this function using 10 grams and the atomic weight of gold (default), carbon, and hydrogen:

```
>>> num_atoms(10)
3.0573183733563486e+22
>>> num_atoms(10, 12.001)
5.017915173735522e+23
>>> num_atoms(10, 1.008)
5.97420634920635e+24
```

4. The aspect ratio of an image describes the relationship between the width and height. Aspect ratios are usually expressed as two numbers separated by a colon that represent width and height respectively. Common aspect ratios in photography are 4:3, 3:2, and 16:9. An image that has an aspect ratio of $x : y$ means that for every x inches of width you will have y inches of height no matter the size of the image (and, of course, you can use any unit of length, not just inches, and even abstract units such as pixels). Suppose you are writing a blog and you have an image that is m units wide and n units high but your blog only has space for an image that is z units wide (where z is less than m). Write a function called `calc_new_height()` that returns the height the image must be in order to preserve the aspect ratio (i.e., a height that will not distort the image). This function takes no arguments and prompts the user for the current width, the current height, and the desired new width. In addition to *returning* the

new height, this function also prints the value. The new height is a `float` regardless of the types of the values the user entered.

The following demonstrates the proper behavior of this function:

```
>>> calc_new_height()
Enter the current width: 800
Enter the current height: 560
Enter the desired width: 370

The corresponding height is: 259.0
259.0
```

5. Write a function called `convert_temp()` that takes no arguments. This function obtains a temperature in Fahrenheit from the user and uses two helper functions to convert this temperature to Celsius and Kelvin. Write a helper function called `convert_to_celsius()` that takes a single argument in Fahrenheit and returns the temperature in Celsius using the formula

$$T_c = \frac{5}{9}(T_f - 32).$$

where T_c is the temperature in Celsius and T_f is the temperature in Fahrenheit. Write another helper function called `convert_to_kelvin()` that takes a single argument in degrees Celsius and returns degrees Kelvin using the formula

$$T_k = T_c + 273.15.$$

where T_k is the temperature in Kelvin. Use these two functions within your `convert_temp()` function to display (i.e., print) the temperatures for the user. The `convert_temp()` does not return anything.

The following demonstrates the proper behavior of this function:

```
>>> convert_temp()
Enter a temperature in Fahrenheit: 32

The temperature in Fahrenheit is: 32
The temperature in Celsius is: 0.0
The temperature in Kelvin is: 273.15
>>> convert_temp()
Enter a temperature in Fahrenheit: 80

The temperature in Fahrenheit is: 80
The temperature in Celsius is: 26.666666666666668
The temperature in Kelvin is: 299.81666666666666
```

Chapter 5

Introduction to Objects

In the coming chapters, we want to unleash some of the power of Python. To access this power we have to introduce something that is syntactically different from what we have seen so far. In order to understand the reason for this change in syntax, it is best to have a simple understanding of *objects* and *object oriented programming*.

Computer languages, considered as a whole, differ widely in how they allow programmers to construct a program. There are some languages that permit object oriented programming, often abbreviated OOP. Python is one such language (C++, Java, and C# are other popular OOP languages). As will be described, OOP provides a way of organizing programs that is more similar to the way people think (i.e., more so than if one uses a non-OOP approach). Thus, in many instances an OOP approach provides the clearest and simplest route to a solution. However, as you might imagine, when something attempts to reflect the way humans think, there can be many subtleties and several layers of complexity. We do *not* want to explore any of these subtleties and complexities here.

At the conclusion of this chapter you should have a sufficiently clear understanding of a couple of aspects of OOP—to the point where you should be comfortable with the material in chapters that follow. But, there is no expectation that this chapter will enable you to write your own programs that fully exploit the power of OOP. (So, if some questions about OOP linger at the end of this chapter, it is to be expected and should not be a concern. At the end of the chapter we will revisit the "take away" message of this material.)

5.1 Overview of Objects and Classes

In previous chapters we considered data such as strings, integers, and `floats`. We also considered functions that can accept data via a parameter (or parameters) and can return data via a `return` statement. Functions are created to accomplish particular tasks and are invariably restricted in the type of data they can accept. For example, if we write a function to capitalize the characters in a string, it doesn't make sense to pass numeric data to this function. As another example, if a function calculates the absolute value, it doesn't make sense to pass this function a string.

When you pause to think about data and functions, you quickly realize they are related in some way. The fact that only certain forms of data make sense with certain functions ties data and functions together conceptually. In OOP there is still the ability to work with traditional data and

From the file: `object-basics.tex`