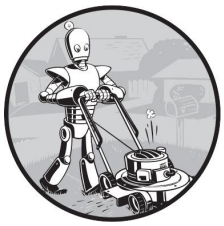
4

FUNCTIONS



A *function* is like a mini program within a program. Python provides several built-in functions, such as the `print()`, `input()`, and `len()` functions from the previous chapters, but you can also write your own. In this chapter, you'll create functions, explore the call stack used to determine the order in which functions in a program run, and learn about the scope of variables inside and outside functions.

Creating Functions

To better understand how functions work, let's create one. Enter this program into the file editor and save it as *helloFunc.py*:

```
def hello():
    # Prints three greetings
    print('Good morning!')
    print('Good afternoon!')
    print('Good evening!')

hello()
hello()
print('ONE MORE TIME!')
hello()
```

The first line is a `def` statement, which defines a function named `hello()`. The code in the block that follows the `def` statement is the body of the function. This code executes when the function is called, not when the function is first defined.

The `hello()` lines after the function are function calls. In code, a function call is just the function's name followed by parentheses, possibly with some number of arguments in between the parentheses. When the program execution reaches these calls, it will jump to the first line in the function and begin executing the code there. When it reaches the end of the function, the execution returns to the line that called the function and continues moving through the code as before.

Because this program calls `hello()` three times, the code in the `hello()` function is executed three times. When you run this program, the output looks like this:

```
Good morning!
Good afternoon!
Good evening!
Good morning!
Good afternoon!
Good evening!
ONE MORE TIME!
Good morning!
Good afternoon!
Good evening!
```

A major purpose of functions is to group code that gets executed multiple times. Without a function defined, you would have to copy and paste this code each time you wanted to run it, and the program would look like this:

```
print('Good morning!')
print('Good afternoon!')
print('Good evening!')
print('Good morning!')
print('Good afternoon!')
print('Good evening!')
```

```
print('ONE MORE TIME!')
print('Good morning!')
print('Good afternoon!')
print(' Good evening!')
```

In general, you always want to avoid duplicating code, because if you ever decide to update the code (for example, because you find a bug you need to fix), you'll have to remember to change the code in every place you copied it.

As you gain programming experience, you'll often find yourself *deduplicating*, which means getting rid of copied-and-pasted code. Deduplication makes your programs shorter, easier to read, and easier to update.

Arguments and Parameters

When you call the `print()` or `len()` function, you pass it values, called *arguments*, by entering them between the parentheses. You can also define your own functions that accept arguments. Enter this example into the file editor and save it as *helloFunc2.py*:

```
❶ def say_hello_to(name):
      # Prints three greetings to the name
provided
  ❷ print('Good morning, ' + name)
    print('Good afternoon, ' + name)
    print('Good evening, ' + name)

❸ say_hello_to('Alice')
say_hello_to('Bob')
```

When you run this program, the output looks like this:

```
Good morning, Alice
Good afternoon, Alice
Good evening, Alice
Good morning, Bob
```

```
Good afternoon, Bob
Good evening, Bob
```

The definition of the `say_hello_to()` function has a parameter called `name` ❶. *Parameters* are variables that contain arguments. When a function is called with arguments, the arguments are stored in the parameters. The first time the `say_hello_to()` function is called, it's passed the argument `'Alice'` ❸. The program execution enters the function, and the parameter `name` is automatically set to `'Alice'`, which gets printed by the `print()` statement ❷. You should use parameters in your function if you need it to follow slightly different instructions depending on the values you pass to the function call.

One special thing to note about parameters is that the value stored in a parameter is forgotten when the function returns. For example, if you added `print(name)` after `say_hello_to('Bob')` in the previous program, the program would give you an error because there is no variable named `name`. This variable gets destroyed after the function call `say_hello_to('Bob')` returns, so `print(name)` would refer to a `name` variable that does not exist.

The terms *define*, *call*, *pass*, *argument*, and *parameter* can be confusing. To review their meanings, consider a code example:

```
❶ def say_hello_to(name):
      # Prints three greetings to the name
provided
      print('Good morning, ' + name)
      print('Good afternoon, ' + name)
      print('Good evening, ' + name)
❷ say_hello_to('Al')
```

To *define* a function is to create it, just as an assignment statement like `spam = 42` creates the `spam` variable. The `def` statement defines the `say_hello_to()` function ❶. The `say_hello_to('Al')` line ❷ *calls* the now-created function, sending the execution to the top of the function's code. This function call is *passing* the string `'Al'` to the function. A value being passed in a function call is an *argument*. Arguments are assigned to local variables called *parameters*. The argument `'Al'` is assigned to the `name` parameter.

It's easy to mix up these terms, but keeping them straight will ensure that you know precisely what the text in this chapter means.

Return Values and return Statements

When you call the `len()` function and pass it an argument such as `'Hello'`, the function call evaluates to the integer value `5`, which is the length of the string you passed it. In general, the value to which a function call evaluates is called the *return value* of the function.

When creating a function using the `def` statement, you can specify the return value with a `return` statement, which consists of the following:

- The `return` keyword
- The value or expression that the function should return

In the case of an expression, the return value is whatever this expression evaluates to. For example, the following program defines a function that returns a different string depending on the number it is passed as an argument. Enter this code into the file editor and save it as *magic8Ball.py*:

```
❶ import random

❷ def get_answer(answer_number):
      # Returns a fortune answer based on what
  int answer_number is, 1 to 9
    ❸ if answer_number == 1:
          return 'It is certain'
      elif answer_number == 2:
          return 'It is decidedly so'
      elif answer_number == 3:
          return 'Yes'
      elif answer_number == 4:
          return 'Reply hazy try again'
      elif answer_number == 5:
          return 'Ask again later'
      elif answer_number == 6:
          return 'Concentrate and ask again'
      elif answer_number == 7:
```

```
        return 'My reply is no'
    elif answer_number == 8:
        return 'Outlook not so good'
    elif answer_number == 9:
        return 'Very doubtful'

print('Ask a yes or no question:')
input('>')
❹ r = random.randint(1, 9)
❺ fortune = get_answer(r)
❻ print(fortune)
```

When the program starts, Python first imports the `random` module ❶. Then comes the definition of the `get_answer()` function ❷. Because the function isn't being called, the code inside it is not run. Next, the program calls the `random.randint()` function with two arguments: `1` and `9` ❹. This function evaluates a random integer between `1` and `9` (including `1` and `9` themselves), then stores it in a variable named `r`.

Now the program calls the `get_answer()` function with `r` as the argument ❺. The program execution moves to the top of that function ❸, storing the value `r` in a parameter named `answer_number`. Then, depending on the value in `answer_number`, the function returns one of many possible string values. The execution returns to the line at the bottom of the program that originally called `get_answer()` ❺ and assigns the returned string to a variable named `fortune`, which then gets passed to a `print()` call ❻ and printed to the screen.

Note that because you can pass return values as arguments to other function calls, you could shorten these three lines

```
r = random.randint(1, 9)
fortune = get_answer(r)
print(fortune)
```

to this single equivalent line:

```
print(get_answer(random.randint(1, 9)))
```

Remember that expressions consist of values and operators; you can use a function call in an expression because the call evaluates to its return value.

The None Value

In Python, a value called `None` represents the absence of a value. The `None` value is the only value of the `NoneType` data type. (Other programming languages might call this value `null`, `nil`, or `undefined`.) Just like the Boolean `True` and `False` values, you must always write `None` with a capital *N*.

This value-without-a-value can be helpful when you need to store something that shouldn't be confused for a real value in a variable. One place where `None` is used is as the return value of `print()`. The `print()` function displays text on the screen, and doesn't need to return anything in the same way `len()` or `input()` does. But since all function calls need to evaluate to a return value, `print()` returns `None`. To see this in action, enter the following into the interactive shell:

```
>>> spam = print('Hello!')
Hello!
>>> None == spam
True
```

Behind the scenes, Python adds `return None` to the end of any function definition with no `return` statement. This behavior resembles the way in which a `while` or `for` loop implicitly ends with a `continue` statement. Functions also return `None` if you use a `return` statement without a value (that is, just the `return` keyword by itself).

Named Parameters

Python identifies most arguments by their position in the function call. For example, `random.randint(1, 10)` is different from `random.randint(10, 1)`. The first call returns a random integer between `1` and `10` because the first argument determines the low end of

the range and the next argument determines its high end, while the second function call causes an error.

On the other hand, Python identifies *named parameters* by the name placed before them in the function call. You'll also hear named parameters called keyword parameters or keyword arguments, though they have nothing to do with Python keywords. Programmers often use named parameters to provide optional arguments. For example, the `print()` function uses the optional parameters `end` and `sep` to specify separator characters to print at the end of its arguments and between its arguments, respectively. If you ran a program without these arguments

```
print('Hello')
print('World')
```

the output would look like this:

```
Hello
World
```

The two strings appear on separate lines because the `print()` function automatically adds a newline character to the end of the string it is passed. However, you can set the `end` named parameter to change the newline character to a different string. For example, if the code were this

```
print('Hello', end='')
print('World')
```

the output would look like this:

```
HelloWorld
```

The output appears on a single line because Python no longer prints a newline after `'Hello'`. Instead, it prints a blank string. This is useful if you need to disable the newline that gets added to the end of every `print()` function call. Say you wanted to print the heads-or-tails

results of a series of coin flips. Printing the output on a single line makes the output prettier, as in this *coinflip.py* program:

```
import random
for i in range(100):  # Perform 100 coin
flips.
    if random.randint(0, 1) == 0:
        print('H', end=' ')
    else:
        print('T', end=' ')
print()  # Print one newline at the end.
```

This program displays the H and T results on one compact line, instead of spreading them out with one H or T result per line:

```
T H T T T H H T T T T H H H H T H H T T T T T
H T T T T T H T T T T T H T H T
H H H T T H T T T T H T H H H T H H T H T T T
T T H T T H T T T T H T H H H T
T T T H T T T T H H H T H T H H H H T H H T
```

Similarly, when you pass multiple string values to `print()`, the function automatically separates them with a single space. To see this behavior, enter the following into the interactive shell:

```
>>> print('cats', 'dogs', 'mice')
cats dogs mice
```

You could replace the default separating string by passing the `sep` named parameter a different string. Enter the following into the interactive shell:

```
>>> print('cats', 'dogs', 'mice', sep=',')
cats,dogs,mice
```

You can add named parameters to the functions you write as well, but first, you'll have to learn about the list and dictionary data types in Chapters 6 and 7. For now, just know that some functions have optional named parameters you can specify when calling the function.

The Call Stack

Imagine that you had a meandering conversation with someone. You talked about your friend Alice, which then reminded you of a story about your co-worker Bob, but first you had to explain something about your cousin Carol. You finished your story about Carol and went back to talking about Bob, and when you finished your story about Bob, you went back to talking about Alice. But then you were reminded about your brother David, so you told a story about him, and then you got back to finishing your original story about Alice. Your conversation followed a *stack*-like structure, like in Figure 4-1. In a stack, items get added or removed from the top only, and the current topic is always at the top of the stack.

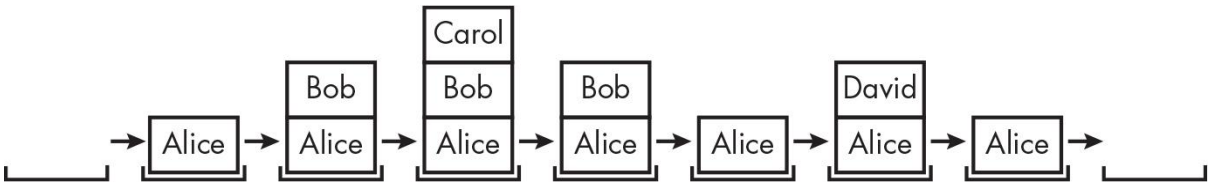


Figure 4-1: Your meandering conversation stack Description

Like your meandering conversation, calling a function doesn't send the execution on a one-way trip to the top of a function. Python will remember which line of code called the function so that the execution can return there when it encounters a `return` statement. If that original function called other functions, the execution would return to *those* function calls first, before returning from the original function call. The function call at the top of the stack is the execution's current location.

Open a file editor window and enter the following code, saving it as *abcdCallStack.py*:

```
def a():
    print('a() starts')
❶   b()
❷   d()
    print('a() returns')


def b():
```

```
    print('b() starts')
❸   c()
    print('b() returns')

def c():
❹   print('c() starts')
    print('c() returns')

def d():
    print('d() starts')
    print('d() returns')

❺ a()
```

If you run this program, the output will look like this:

```
a() starts
b() starts
c() starts
c() returns
b() returns
d() starts
d() returns
a() returns
```

When `a()` is called ❺, it calls `b()` ❶, which in turn calls `c()` ❸. The `c()` function doesn't call anything; it just displays `c() starts` ❹ and `c() returns` before returning to the line in `b()` that called it ❸. Once the execution returns to the code in `b()` that called `c()`, it returns to the line in `a()` that called `b()` ❶. The execution continues to the next line in the `a()` function ❷, which is a call to `d()`. Like the `c()` function, the `d()` function also doesn't call anything. It just displays `d() starts` and `d() returns` before returning to the line in `a()` that called it. Because `d()` contains no other code, the execution returns to the line in `a()` that called `d()` ❷. The last line in `a()`

displays `a() returns` before returning to the original `a()` call at the end of the program ❺.

The *call stack* is how Python remembers where to return the execution after each function call. The call stack isn't stored in a variable in your program; rather, it's a section of your computer's memory that Python handles automatically behind the scenes. When your program calls a function, Python creates a *frame object* on the top of the call stack. Frame objects store the line number of the original function call so that Python can remember where to return. If the program makes another function call, Python adds another frame object above the other one on the call stack.

When a function call returns, Python removes a frame object from the top of the stack and moves the execution to the line number stored in it. Note that frame objects always get added and removed from the top of the stack, and not from any other place. Figure 4-2 illustrates the state of the call stack in *abcdCallStack.py* as each function is called and returns.

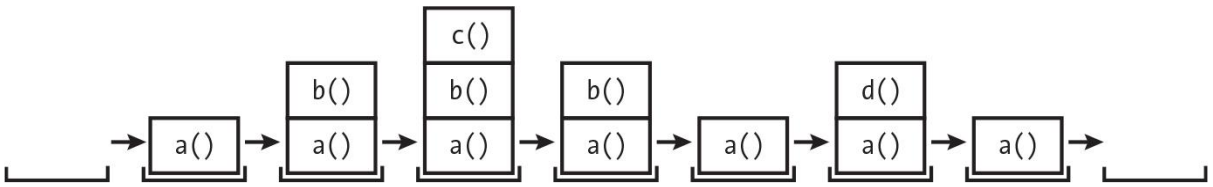


Figure 4-2: The frame objects of the call stack as abcdCallStack.py *calls and returns from functions* Description

The top of the call stack is the currently executing function. When the call stack is empty, the execution is on a line outside all functions.

The call stack is a technical detail that you don't strictly need to know about to write programs. It's enough to understand that function calls return to the line number they were called from. However, understanding call stacks makes it easier to understand local and global scopes, described in the next section.

Local and Global Scopes

Only code within a called function can access the parameters and variables assigned in that function. These variables are said to exist in that function's *local scope*. By contrast, code anywhere in a program can access variables that are assigned outside all functions. These variables are said to exist in the *global scope*. A variable that exists in a local scope is called a *local variable*, while a variable that exists in a global scope is called a *global variable*. A variable must be one or the other; it cannot be both local and global.

Think of a *scope* as a container for variables. There is only one global scope, created when your program begins. When your program

terminates, it destroys the global scope, and all of its variables get forgotten. A new local scope gets created whenever a program calls a function. Any variables assigned in the function exist within the function's local scope. When the function returns, the local scope gets destroyed, along with these variables.

Python uses scoping because it enables a function to modify its variables, yet interact with the rest of the program through its parameters and its return value only. This narrows down the number of lines of code that might be causing a bug. If your program contained nothing but global variables, and contained a bug caused by a variable set to a bad value, you might struggle to track down the location of this bad value. It could have been set from anywhere in the program, which could be hundreds or thousands of lines long! But if the bug occurred in a local variable, you can restrict your search to a single function.

For this reason, while using global variables in small programs is fine, it's a bad habit to rely on global variables as your programs get larger and larger.

Scope Rules

When working with local and global variables, keep the following rules in mind:

- Code that is in the global scope, outside all functions, can't use local variables.
- Code that is in one function's local scope can't use variables in any other local scope.
- Code in a local scope can access global variables.
- You can use the same name for different variables if they are in different scopes. That is, there can be a local variable named `spam` and a global variable also named `spam`.

Let's review these rules with examples.

Code That Is in the Global Scope Can't Use Local Variables

Consider the following code, which will cause an error when you run it:

```
def spam():
  ❶ eggs = 'sss'
spam()
print(eggs)
```

The program's output will look like this:

```
Traceback (most recent call last):
  File "C:/test1.py", line 4, in <module>
    print(eggs)
NameError: name 'eggs' is not defined
```

The error happens because the `eggs` variable exists only in the local scope created when `spam()` is called ❶. Once the program execution returns from `spam()`, that local scope gets destroyed, and there is no longer a variable named `eggs`. So when your program tries to run `print(eggs)`, Python gives you an error saying that `eggs` is not defined. This makes sense if you think about it; when the program execution is in the global scope, no local scopes exist, so there can't be any local variables. This is why you can only reference global variables in the global scope.

Code That Is in a Local Scope Can't Use Variables in Other Local Scopes

Python creates a new local scope whenever a program calls a function, even when the function is called from another function. Consider this program:

```
def spam():
    eggs = 'SPAMSPAM'
  ❶ bacon()
  ❷ print(eggs)  # Prints 'SPAMSPAM'

def bacon():
    ham = 'hamham'
    eggs = 'BACONBACON'

❸ spam()
```

When the program starts, it calls the `spam()` function ❸, creating a local scope. The `spam()` function sets the local variable `eggs` to `'SPAMSPAM'`, then calls the `bacon()` function ❶, creating a second local scope. Multiple local scopes can exist at the same time. In this new local scope, the local variable `ham` gets set to `'hamham'`, and a local variable `eggs` (which differs from the one in `spam()`'s local scope)

gets created and set to `'BACONBACON'`. At this point, the program has two local variables named `eggs` that exist simultaneously: one that is local to `spam()` and one that is local to `bacon()`.

When `bacon()` returns, Python destroys the local scope for that call, including its `eggs` variable. The program execution continues in the `spam()` function, printing the value of `eggs` ❷. Because the local scope for the call to `spam()` still exists, the only `eggs` variable is the `spam()` function's `eggs` variable, which was set to `'SPAMSPAM'`. This is what the program prints.

Code That Is in a Local Scope Can Use Global Variables

So far, I've demonstrated that code in the global scope can't access variables in a local scope; nor can code in a different local scope. Now consider the following program:

```
def spam():
    print(eggs)  # Prints 'GLOBALGLOBAL'
eggs = 'GLOBALGLOBAL'
spam()
print(eggs)
```

Because the `spam()` function has no parameter named `eggs`, nor any code that assigns `eggs` a value, Python considers the function's use of `eggs` a reference to the global variable `eggs`. This is why the program prints `'GLOBALGLOBAL'` when it's run.

Local and Global Variables Can Have the Same Name

Technically, it's perfectly acceptable to use the same variable name for a global variable and local variables in different scopes. But, to simplify your life, avoid doing this. To see what could happen, enter the following code into the file editor and save it as *localGlobalSameName.py*:

```
def spam():
  ❶ eggs = 'spam local'
    print(eggs)  # Prints 'spam local'
```

```
def bacon():
  ❷ eggs = 'bacon local'
    print(eggs)  # Prints 'bacon local'
    spam()
    print(eggs)  # Prints 'bacon local'

❸ eggs = 'global'
bacon()
print(eggs)  # Prints 'global'
```

When you run this program, it outputs the following:

```
bacon local
spam local
bacon local
global
```

This program actually contains three different variables, but confusingly, they're all named `eggs`. The variables are as follows:

- A variable named `eggs` that exists in a local scope when `spam()` is called ❶
- A variable named `eggs` that exists in a local scope when `bacon()` is called ❷
- A variable named `eggs` that exists in the global scope ❸

Because these three separate variables all have the same name, it can be hard to keep track of the one in use at any given time. Instead, give all variables unique names, even when they appear in different scopes.

The global Statement

If you need to modify a global variable from within a function, use the `global` statement. Including a line such as `global eggs` at the top of a function tells Python, "In this function, `eggs` refers to the global variable, so don't create a local variable with this name." For example, enter the following code into the file editor and save it as *globalStatement.py*:

```
def spam():
  ❶ global eggs
  ❷ eggs = 'spam'

eggs = 'global'
spam()
print(eggs)  # Prints 'spam'
```

When you run this program, the final `print()` call will output this:

```
spam
```

Because `eggs` is declared `global` at the top of `spam()` ❶, setting `eggs` to `'spam'` ❷ changes the value of the globally scoped `eggs`. No local `eggs` variable is ever created.

Scope Identification

Use these four rules to tell whether a variable belongs to a local scope or the global scope:

1. A variable in the global scope (that is, outside all functions) is always a global variable.
2. A variable in a function with a `global` statement is always a global variable in that function.
3. Otherwise, if a function uses a variable in an assignment statement, it is a local variable.
4. However, if the function uses a variable but never in an assignment statement, it is a global variable.

To get a better feel for these rules, here's an example program. Enter the following code into the file editor and save it as *sameNameLocalGlobal.py*:

```
def spam():
  ❶ global eggs
     eggs = 'spam'  # This is the global
variable.
```

```
def bacon():
  ❷ eggs = 'bacon'  # This is a local
variable.

def ham():
  ❸ print(eggs)  # This is the global
variable.

eggs = 'global'  # This is the global
variable.
spam()
print(eggs)
```

In the `spam()` function, `eggs` refers to the global `eggs` variable because the function includes a `global` statement for it ❶. In `bacon()`, `eggs` is a local variable because the function includes an assignment statement for it ❷. In `ham()` ❸, `eggs` is the global variable because the function contains no assignment statement or `global` statement for the variable. If you run *sameNameLocalGlobal.py*, the output will look like this:

```
spam
```

If you try to use a local variable in a function before you assign a value to it, as in the following program, Python will give you an error. To see this, enter the following into the file editor and save it as *sameNameError.py*:

```
def spam():
    print(eggs)  # ERROR!
  ❶ eggs = 'spam local'

❷ eggs = 'global'
spam()
```

If you run the previous program, it produces an error message:

```
Traceback (most recent call last):
  File "C:/sameNameError.py", line 6, in
<module>
    spam()
  File "C:/sameNameError.py", line 2, in spam
    print(eggs)  # ERROR!
UnboundLocalError: local variable 'eggs'
referenced before assignment
```

This error happens because Python sees that there is an assignment statement for `eggs` in the `spam()` function ❶ and, therefore, considers any mention of an `eggs` variable in `spam()` to be a local variable. But because `print(eggs)` is executed before `eggs` is assigned anything, the local variable `eggs` doesn't exist. Python won't fall back to using the global `eggs` variable ❷.

The name of the error, `UnboundLocalError`, can be somewhat confusing. In Python, *binding* is another way of saying *assigning*, so this error indicates that the program used a local variable before it was assigned a value.

FUNCTIONS AS "BLACK BOXES"

Often, all you need to know about a function are its inputs (the parameters) and its output value; you don't always have to burden yourself with how the function's code actually works. When you think about functions in this high-level way, it's common to say that you're treating a function as a "black box."

This idea is fundamental to modern programming. Later chapters in this book will show you several modules with functions that were written by other people. While you can take a peek at the source code if you're curious, you don't need to know how these functions work in order to use them. And because writing functions without global variables is encouraged, you usually don't have to worry about the function's code interacting with the rest of your program.

Exception Handling

Right now, getting an error, or *exception*, in your Python program means the entire program will crash. You don't want this to happen in real-world programs. Instead, you want the program to detect errors, handle them, and then continue to run.

For example, consider the following program, which has a divide-by-zero error. Open a file editor window and enter the following code, saving it as *zeroDivide.py*:

```
def spam(divide_by):
    return 42 / divide_by

print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

We've defined a function called `spam`, given it a parameter, and then printed the value of that function with various parameters to see what happens. This is the output you get when you run the previous code:

```
21.0
3.5
Traceback (most recent call last):
  File "C:/zeroDivide.py", line 6, in
<module>
    print(spam(0))
  File "C:/zeroDivide.py", line 2, in spam
    return 42 / divide_by
ZeroDivisionError: division by zero
```

A `ZeroDivisionError` happens whenever you try to divide a number by zero. From the line number given in the error message, you know that the `return` statement in `spam()` is causing an error.

Most of the time, exceptions indicate a bug in your code that you need to fix. But sometimes exceptions can be expected and recovered

from. For example, in Chapter 10 you'll learn how to read text from files. If you specify a filename for a file that doesn't exist, Python raises a `FileNotFoundError` exception. You might want to handle this exception by asking the user to enter the filename again rather than having this unhandled exception immediately crash your program.

Errors can be handled with `try` and `except` statements. The code that could potentially have an error is put in a `try` clause. The program execution moves to the start of a following `except` clause if an error happens.

You can put the previous divide-by-zero code in a `try` clause and have an `except` clause contain code to handle what happens when this error occurs:

```
def spam(divide_by):
    try:
        # Any code in this block that causes
ZeroDivisionError won't crash the program:
        return 42 / divide_by
    except ZeroDivisionError:
        # If ZeroDivisionError happened, the
code in this block runs:
        print('Error: Invalid argument.')

print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

When code in a `try` clause causes an error, the program execution immediately moves to the code in the `except` clause. After running that code, the execution continues as normal. If the program doesn't raise an exception in the `try` clause, the program skips the code in the `except` clause. The output of the previous program is as follows:

```
21.0
3.5
Error: Invalid argument.
```

```
None
42.0
```

Note that any errors that occur in function calls in a `try` block will also be caught. Consider the following program, which instead has the `spam()` calls in the `try` block:

```
def spam(divide_by):
    return 42 / divide_by

try:
    print(spam(2))
    print(spam(12))
    print(spam(0))
    print(spam(1))
except ZeroDivisionError:
    print('Error: Invalid argument.')
```

When this program is run, the output looks like this:

```
21.0
3.5
Error: Invalid argument.
```

The reason `print(spam(1))` is never executed is because once the execution jumps to the code in the `except` clause, it does not return to the `try` clause. Instead, it just continues moving down the program as normal.

A Short Program: Zigzag

Let's use the programming concepts you've learned so far to create a small animation program. This program will create a back-and-forth zigzag pattern until the user stops it by pressing the Mu editor's Stop button or by pressing CTRL-C. When you run this program, the output will look something like this:

```
    ********
   ********
  ********
 ********
********
 ********
  ********
   ********
    ********
```

Enter the following source code into the file editor, and save the file as *zigzag.py*:

```
import time, sys
indent = 0  # How many spaces to indent
indent_increasing = True  # Whether the
indentation is increasing or not

try:
    while True:  # The main program loop
        print(' ' * indent, end='')
        print('********')
        time.sleep(0.1) # Pause for 1/10th of
a second.

        if indent_increasing:
            # Increase the number of spaces:
            indent = indent + 1
            if indent == 20:
                # Change direction:
                indent_increasing = False
        else:
            # Decrease the number of spaces:
            indent = indent - 1
```

```
                if indent == 0:
                    # Change direction:
                    indent_increasing = True
except KeyboardInterrupt:
    sys.exit()
```

Let's look at this code line by line, starting at the top:

```
import time, sys
indent = 0 # How many spaces to indent
indent_increasing = True  # Whether the
indentation is increasing or not
```

First, we'll import the `time` and `sys` modules. Our program uses two variables. The `indent` variable keeps track of how many spaces of indentation occur before the band of eight asterisks, and the `indent_increasing` variable contains a Boolean value to determine whether the amount of indentation is increasing or decreasing:

```
try:
    while True:  # The main program loop
        print(' ' * indent, end='')
        print('********')
        time.sleep(0.1) # Pause for 1/10 of a
second.
```

Next, we place the rest of the program inside a `try` statement. When the user presses CTRL-C while a Python program is running, Python raises the `KeyboardInterrupt` exception. If there is no `try-except` statement to catch this exception, the program crashes with an ugly error message. However, we want our program to cleanly handle the `KeyboardInterrupt` exception by calling `sys.exit()`. (You can find the code that accomplishes this in the `except` statement at the end of the program.)

The `while True:` infinite loop will repeat the instructions in the program forever. This involves using `' ' * indent` to print the correct number of spaces for the indentation. We don't want to automatically print a newline after these spaces, so we also pass `end=''` to the first `print()` call. A second `print()` call prints the band of asterisks. We haven't discussed the `time.sleep()` function yet; suffice it to say that it introduces a one-tenth-of-a-second pause in our program:

```
        if indent_increasing:
            # Increase the number of spaces:
            indent = indent + 1
            if indent == 20:
                indent_increasing = False  #
Change direction
```

Next, we want to adjust the amount of indentation used the next time we print asterisks. If `indent_increasing` is `True`, we'll add `1` to `indent`, but once indent reaches `20`, we'll decrease the indentation:

```
        else:
            # Decrease the number of spaces:
            indent = indent - 1
            if indent == 0:
                indent_increasing = True  #
Change direction
```

If `indent_increasing` is `False`, we'll want to subtract one from `indent`. Once indent reaches `0`, we'll want the indentation to increase once again. Either way, the program execution will jump back to the start of the main program loop to print the asterisks again.

If the user presses CTRL-C at any point that the program execution is in the `try` block, this `except` statement raises and handles the `KeyboardInterrupt` exception:

```
except KeyboardInterrupt:
    sys.exit()
```

The program execution moves inside the `except` block, which runs `sys.exit()` and quits the program. This way, even though the main program loop is infinite, the user has a way to shut down the program.

A Short Program: Spike

Let's create another scrolling animation program. This program uses string replication and nested loops to draw spikes. Open a new file in your code editor, enter the following code, and save it as *spike.py*:

```
import time, sys

try:
    while True:  # The main program loop
        # Draw lines with increasing length:
        for i in range(1, 9):
            print('-' * (i * i))
            time.sleep(0.1)

        # Draw lines with decreasing length:
        for i in range(7, 1, -1):
            print('-' * (i * i))
            time.sleep(0.1)
except KeyboardInterrupt:
    sys.exit()
```

When you run this program, it repeatedly draws the following spike:

```
-
----
---------
```

```
---------------
-----------------------
-----------------------------------
-----------------------------------------
----
-----------------------------------------
------------------
-----------------------------------------
----
---------------------------------
------------------------
---------------
---------
----
```

Like the previous zigzag program, the spike program needs to call the `time.sleep()` and `sys.exit()` functions. The first line imports the `time` and `sys` modules. A `try` block will catch the `KeyboardInterrupt` exception raised when the user presses CTRL-C, and an infinite loop continues drawing spikes forever.

The first `for` loop draws spikes of increasing sizes:

```
# Draw lines with increasing length:
for i in range(1, 9):
    print('-' * (i * i))
    time.sleep(0.1)
```

Because the `i` variable is set to `1`, then `2`, then `3`, and so on, up to but not including `9`, the following `print()` call replicates the `'-'` strings by `1 * 1` (that is, `1`), then `2 * 2` (that is, `4`), then `3 * 3` (that is, `9`), and so on. This code is what creates the strings that are 1, 4, 9, 16, 25, 36, 49, and then 64 dashes long. By having exponentially longer strings, we create the top half of the spike.

To draw the bottom half of the spike, we need another `for` loop that causes `i` to start at `7` and then decrease down to `1`, not including `1`:

```
    # Draw lines with decreasing length:
    for i in range(7, 1, -1):
        print('-' * (i * i))
        time.sleep(0.1)
```

You can modify the `9` and the `7` values in the two `for` loops if you want to change how wide the spike becomes. The rest of the code will continue to work just fine with these new values.

Summary

Functions are the primary way to compartmentalize your code into logical groups. Since the variables in functions exist in their own local scopes, the code in one function cannot directly affect the values of local variables in other functions. This limits the sections of code able to change the values of your variables, which can be helpful when it comes to debugging.

Functions are a great tool to help you organize your code. You can think of them as black boxes: they have inputs in the form of parameters and outputs in the form of return values, and the code in them doesn't affect variables in other functions.

In previous chapters, a single error could cause your programs to crash. In this chapter, you learned about `try` and `except` statements, which can run code when an error has been detected. This can make your programs more resilient to common error cases.

Practice Questions

1. Why are functions advantageous to have in your programs?
2. When does the code in a function execute: when the function is defined or when the function is called?
3. What statement creates a function?
4. What is the difference between a function and a function call?
5. How many global scopes are there in a Python program? How many local scopes are there?
6. What happens to variables in a local scope when the function call returns?
7. What is a return value? Can a return value be part of an expression?
8. If a function does not have a `return` statement, what is the return value of a call to that function?
9. How can you force a variable in a function to refer to the global variable?

10. What is the data type of `None`?
11. What does the `import areallyourpetsnamederic` statement do?
12. If you had a function named `bacon()` in a module named `spam`, how would you call it after importing `spam`?
13. How can you prevent a program from crashing when it gets an error?
14. What goes in the `try` clause? What goes in the `except` clause?
15. Write the following program in a file named *notrandomdice.py* and run it. Why does each function call return the same number?

```
import random
random_number = random.randint(1, 6)

def get_random_dice_roll():
    # Returns a random integer from 1 to 6
    return random_number

print(get_random_dice_roll())
print(get_random_dice_roll())
print(get_random_dice_roll())
print(get_random_dice_roll())
```

Practice Programs

For practice, write programs to do the following tasks.

The Collatz Sequence

Write a function named `collatz()` that has one parameter named `number`. If `number` is even, then `collatz()` should print `number // 2` and return this value. If `number` is odd, then `collatz()` should print and return `3 * number + 1`.

Then, write a program that lets the user enter an integer and that keeps calling `collatz()` on that number until the function returns the value `1`. (Amazingly enough, this sequence actually works for any integer; sooner or later, using this sequence, you'll arrive at 1! Even mathematicians aren't sure why. Your program is exploring what's called the *Collatz sequence*, sometimes called "the simplest impossible math problem.")

Remember to convert the return value from `input()` to an integer with the `int()` function; otherwise, it will be a string value. To make the output more compact, the `print()` calls that print the numbers should have a `sep=' '` named parameter to print all values on one line.

The output of this program could look something like this:

```
Enter number:
3
3 10 5 16 8 4 2 1
```

Hint: An integer `number` is even if `number % 2 == 0`, and it's odd if `number % 2 == 1`.

Input Validation

Add `try` and `except` statements to the previous project to detect whether the user entered a non-integer string. Normally, the `int()` function will raise a `ValueError` error if it is passed a non-integer string, as in `int('puppy')`. In the `except` clause, print a message to the user saying they must enter an integer.