3
FUNCTIONS

You're already familiar with the `print()`, `input()`, and `len()` functions from the previous chapters. Python provides several built-in functions like these, but you can also write your own functions. A *function* is like a miniprogram within a program.

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

```
❶ def hello():
    ❷ print('Howdy!')
      print('Howdy!!!')
      print('Hello there.')


❸ hello()
  hello()
  hello()
```

You can view the execution of this program at *https://autbor.com/hellofunc/*. The first line is a `def` statement ❶, which defines a function named `hello()`. The code in the block that follows the `def` statement ❷ is the body of the function. This code is executed when the function is called, not when the function is first defined.

The `hello()` lines after the function ❸ 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 top 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.

Since 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:

```
Howdy!
Howdy!!!
Hello there.
Howdy!
Howdy!!!
Hello there.
Howdy!
Howdy!!!
Hello there.
```

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, and the program would look like this:

```
print('Howdy!')
print('Howdy!!!')
print('Hello there.')
print('Howdy!')
print('Howdy!!!')
print('Hello there.')
print('Howdy!')
print('Howdy!!!')
print('Hello there.')
```

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

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

def Statements with Parameters

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

```
❶ def hello(name):
    ❷ print('Hello, ' + name)

❸ hello('Alice')
  hello('Bob')
```

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

```
Hello, Alice
Hello, Bob
```

You can view the execution of this program at *https://autbor.com/hellofunc2/*. The definition of the `hello()` function in this program 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 `hello()` function is called, it is passed the argument `'Alice'` ❸. The program execution enters the function, and the parameter `name` is automatically set to `'Alice'`, which is what gets printed by the `print()` statement ❷.

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 `hello('Bob')` in the previous program, the program would give you a `NameError` because there is no variable named `name`. This variable is destroyed after the function call `hello('Bob')` returns, so `print(name)` would refer to a `name` variable that does not exist.

This is similar to how a program's variables are forgotten when the program terminates. I'll talk more about why that happens later in the chapter, when I discuss what a function's local scope is.

Define, Call, Pass, Argument, Parameter

The terms *define*, *call*, *pass*, *argument*, and *parameter* can be confusing. Let's look at a code example to review these terms:

```
❶ def sayHello(name):
      print('Hello, ' + name)
❷ sayHello('Al')
```

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

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 that a function call evaluates to is called the *return value* of the function.

When creating a function using the `def` statement, you can specify what the return value should be with a `return` statement. A `return` statement consists of the following:

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

When an expression is used with a `return` statement, the return value is what this expression evaluates to. For example, the following program defines a function that returns a different string depending on what 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 getAnswer(answerNumber):
    ❸ if answerNumber == 1:
        return 'It is certain'
    elif answerNumber == 2:
        return 'It is decidedly so'
    elif answerNumber == 3:
        return 'Yes'
    elif answerNumber == 4:
        return 'Reply hazy try again'
    elif answerNumber == 5:
        return 'Ask again later'
```

```
    elif answerNumber == 6:
        return 'Concentrate and ask again'
    elif answerNumber == 7:
        return 'My reply is no'
    elif answerNumber == 8:
        return 'Outlook not so good'
    elif answerNumber == 9:
        return 'Very doubtful'


❹ r = random.randint(1, 9)
❺ fortune = getAnswer(r)
❻ print(fortune)
```

You can view the execution of this program at *https://autbor.com/magic8ball/*. When this program starts, Python first imports the `random` module ❶. Then the `getAnswer()` function is defined ❷. Because the function is being defined (and not called), the execution skips over the code in it. Next, the `random.randint()` function is called with two arguments: `1` and `9` ❹. It evaluates to a random integer between `1` and `9` (including `1` and `9` themselves), and this value is stored in a variable named `r`.

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

Note that since you can pass return values as an argument to another function call, you could shorten these three lines:

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

to this single equivalent line:

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

Remember, expressions are composed of values and operators. A function call can be used in an expression because the call evaluates to its return value.

The None Value

In Python, there is a value called `None`, which 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, `None` must be typed with a capital *N*.

This value-without-a-value can be helpful when you need to store something that won'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, but it 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 is similar to how a `while` or `for` loop implicitly ends with a `continue` statement. Also, if you use a `return` statement without a value (that is, just the `return` keyword by itself), then `None` is returned.

Keyword Arguments and the print() Function

Most arguments are identified by their position in the function call. For example, `random.randint(1, 10)` is different from `random.randint(10, 1)`. The function call `random.randint(1, 10)` will return a random integer between `1` and `10` because the first argument is the low end of the range and the second argument is the high end (while `random.randint(10, 1)` causes an error).

However, rather than through their position, *keyword arguments* are identified by the keyword put before them in the function call. Keyword arguments are often used for *optional parameters*. For example, the `print()` function has the optional parameters `end` and `sep` to specify what should be printed at the end of its arguments and between its arguments (separating them), respectively.

If you ran a program with the following code:

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

the output would look like this:

```
Hello
World
```

The two outputted 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` keyword argument 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 is printed on a single line because there is no longer a newline printed after `'Hello'`. Instead, the blank string is printed. This is useful if you need to disable the newline that gets added to the end of every `print()` function call.

Similarly, when you pass multiple string values to `print()`, the function will automatically separate them with a single space. Enter the following into the interactive shell:

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

But you could replace the default separating string by passing the `sep` keyword argument a different string. Enter the following into the interactive shell:

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

You can add keyword arguments to the functions you write as well, but first you'll have to learn about the list and dictionary data types in the next two chapters. For now, just know that some functions have optional keyword arguments that can be specified when the function is called.

The Call Stack

Imagine that you have a meandering conversation with someone. You talk about your friend Alice, which then reminds you of a story about your coworker Bob, but first you

have to explain something about your cousin Carol. You finish you story about Carol and go back to talking about Bob, and when you finish your story about Bob, you go back to talking about Alice. But then you are reminded about your brother David, so you tell a story about him, and then get back to finishing your original story about Alice. Your conversation followed a *stack*-like structure, like in Figure 3-1. The conversation is stack-like because the current topic is always at the top of the stack.

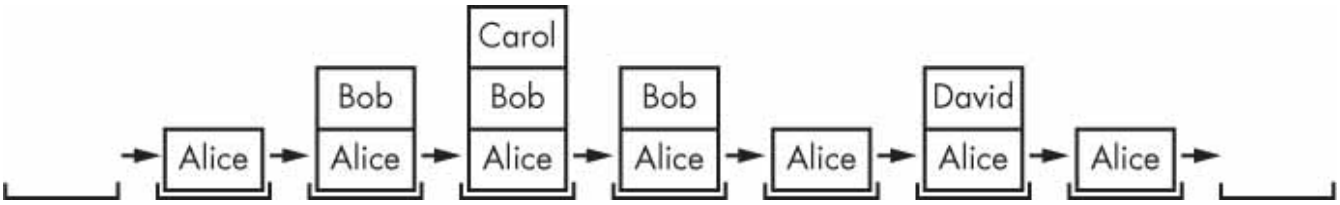

Figure 3-1: Your meandering conversation stack

Similar to our 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.

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
```

You can view the execution of this program at *https://autbor.com/abcdcallstack/*. 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 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 `b()` 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 `b()` that called it. Since `b()` contains no other code, the execution returns to the line in `a()` that called `b()` 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, Python handles it 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 another function call is made, Python puts another frame object on the call stack above the other one.

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 are always added and removed from the top of the stack and not from any other place. Figure 3-2 illustrates the state of the call stack in *abcdCallStack.py* as each function is called and returns.

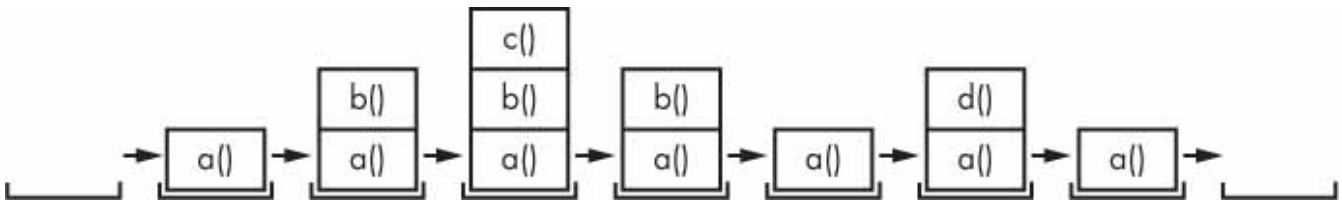

Figure 3-2: The frame objects of the call stack as abcdCallStack.py *calls and returns from functions*

The top of the call stack is which function the execution is currently in. When the call stack is empty, the execution is on a line outside of 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 Scope

Parameters and variables that are assigned in a called function are said to exist in that function's *local scope*. Variables that are assigned outside all functions 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 the 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. When a scope is destroyed, all the values stored in the scope's variables are forgotten. There is only one global scope, and it is created when your program begins. When your program terminates, the global scope is destroyed, and all its variables are forgotten. Otherwise, the next time you ran a program, the variables would remember their values from the last time you ran it.

A local scope is created whenever a function is called. Any variables assigned in the function exist within the function's local scope. When the function returns, the local scope is destroyed, and these variables are forgotten. The next time you call the function, the local variables will not remember the values stored in them from the last time the function was called. Local variables are also stored in frame objects on the call stack.

Scopes matter for several reasons:

- Code in the global scope, outside of all functions, cannot use any local variables.
- However, code in a local scope can access global variables.
- Code in a function's local scope cannot use variables in any other local scope.
- 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`.

The reason Python has different scopes instead of just making everything a global variable is so that when variables are modified by the code in a particular call to a function, the function interacts with the rest of the program only through its parameters and the return value. This narrows down the number of lines of code that may be causing

a bug. If your program contained nothing but global variables and had a bug because of a variable being set to a bad value, then it would be hard to track down where this bad value was set. It could have been set from anywhere in the program, and your program could be hundreds or thousands of lines long! But if the bug is caused by a local variable with a bad value, you know that only the code in that one function could have set it incorrectly.

While using global variables in small programs is fine, it is a bad habit to rely on global variables as your programs get larger and larger.

Local Variables Cannot Be Used in the Global Scope

Consider this program, which will cause an error when you run it:

```
def spam():
❶ eggs = 31337
spam()
print(eggs)
```

If you run this program, the 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 is 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 only global variables can be used in the global scope.

Local Scopes Cannot Use Variables in Other Local Scopes

A new local scope is created whenever a function is called, including when a function is called from another function. Consider this program:

```
def spam():
    ❶ eggs = 99
    ❷ bacon()
```

```
    ❸ print(eggs)


def bacon():
    ham = 101
    ❹ eggs = 0


❺ spam()
```

You can view the execution of this program at *https://autbor.com/otherlocalscopes/*. When the program starts, the `spam()` function is called ❺, and a local scope is created. The local variable `eggs` ❶ is set to `99`. Then the `bacon()` function is called ❷, and a second local scope is created. Multiple local scopes can exist at the same time. In this new local scope, the local variable `ham` is set to `101`, and a local variable `eggs`—which is different from the one in `spam()`'s local scope—is also created ❹ and set to `0`.

When `bacon()` returns, the local scope for that call is destroyed, including its `eggs` variable. The program execution continues in the `spam()` function to print the value of `eggs` ❸. Since 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 `99`. This is what the program prints.

The upshot is that local variables in one function are completely separate from the local variables in another function.

Global Variables Can Be Read from a Local Scope

Consider the following program:

```
def spam():
    print(eggs)
eggs = 42
spam()
print(eggs)
```

You can view the execution of this program at *https://autbor.com/readglobal/*. Since there is no parameter named `eggs` or any code that assigns `eggs` a value in the `spam()` function, when `eggs` is used in `spam()`, Python considers it a reference to the global variable `eggs`. This is why `42` is printed when the previous program is run.

Local and Global Variables with the Same Name

Technically, it's perfectly acceptable to use the same variable name for a global variable and local variables in different scopes in Python. But, to simplify your life, avoid doing

this. To see what happens, 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
```

You can view the execution of this program at *https://autbor.com/localglobalsamename/*. There are actually three different variables in this program, but confusingly they are all named `eggs`. The variables are as follows:

❶ A variable named `eggs` that exists in a local scope when `spam()` is called.

❷ A variable named `eggs` that exists in a local scope when `bacon()` is called.

❸ A variable named `eggs` that exists in the global scope.

Since these three separate variables all have the same name, it can be confusing to keep track of which one is being used at any given time. This is why you should avoid using the same variable name in different scopes.

The global Statement

If you need to modify a global variable from within a function, use the `global` statement. If you have a line such as `global eggs` at the top of a function, it 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)
```

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

```
spam
```

You can view the execution of this program at *https://autbor.com/globalstatement/*. Because `eggs` is declared `global` at the top of `spam()` ❶, when `eggs` is set to `'spam'` ❷, this assignment is done to the globally scoped `eggs`. No local `eggs` variable is created.

There are four rules to tell whether a variable is in a local scope or global scope:

- If a variable is being used in the global scope (that is, outside of all functions), then it is always a global variable.
- If there is a `global` statement for that variable in a function, it is a global variable.
- Otherwise, if the variable is used in an assignment statement in the function, it is a local variable.
- But if the variable is not used 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

def bacon():
  ❷ eggs = 'bacon' # this is a local

def ham():
  ❸ print(eggs) # this is the global
```

```
eggs = 42 # this is the global

spam()

print(eggs)
```

In the `spam()` function, `eggs` is the global `eggs` variable because there's a `global` statement for `eggs` at the beginning of the function ❶. In `bacon()`, `eggs` is a local variable because there's an assignment statement for it in that function ❷. In `ham()` ❸, `eggs` is the global variable because there is no assignment statement or `global` statement for it in that function. If you run *sameNameLocalGlobal.py*, the output will look like this:

```
spam
```

You can view the execution of this program at *https://autbor.com/sameNameLocalGlobal/*. In a function, a variable will either always be global or always be local. The code in a function can't use a local variable named `eggs` and then use the global `eggs` variable later in that same function.

> **NOTE**
>
> *If you ever want to modify the value stored in a global variable from in a function, you must use a `global` statement on that variable.*

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
```

You can view the execution of this program at *https://autbor.com/sameNameError/*. This error happens because Python sees that there is an assignment statement for `eggs` in the `spam()` function ❶ and, therefore, considers `eggs` to be local. But because `print(eggs)` is executed before `eggs` is assigned anything, the local variable `eggs` doesn't exist. Python will *not* fall back to using the global `eggs` variable ❷.

FUNCTIONS AS "BLACK BOXES"

Often, all you need to know about a function are its inputs (the parameters) and 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(divideBy):
    return 42 / divideBy


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 / divideBy
ZeroDivisionError: division by zero
```

You can view the execution of this program at *https://autbor.com/zerodivide/*. 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.

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(divideBy):
    try:
        return 42 / divideBy
    except ZeroDivisionError:
        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. The output of the previous program is as follows:

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

```
None
42.0
```

You can view the execution of this program at *https://autbor.com/tryexceptzerodivide/*. 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(divideBy):
    return 42 / divideBy


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.
```

You can view the execution of this program at *https://autbor.com/spamintry/*. 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:

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

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

Type 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.

indentIncreasing = 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 1/10 of a second.


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


        else:
            # Decrease the number of spaces:
            indent = indent - 1
            if indent == 0:
                # Change direction:
                indentIncreasing = 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.

indentIncreasing = 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 are before the band of eight asterisks and `indentIncreasing` contains a Boolean value to determine if 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, for our program, we want it to cleanly handle the `KeyboardInterrupt` exception by calling `sys.exit()`. (The code for this is in the `except` statement at the end of the program.)

The `while True:` infinite loop will repeat the instructions in our program forever. This involves using `' ' * indent` to print the correct amount of spaces of 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. The `time.sleep()` function hasn't been covered yet, but suffice it to say that it introduces a one-tenth-second pause in our program at this point.

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

Next, we want to adjust the amount of indentation for the next time we print asterisks. If `indentIncreasing` is `True`, then we want to add one to `indent`. But once indent reaches `20`, we want the indentation to decrease.

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

Meanwhile, if `indentIncreasing` was `False`, we want to subtract one from `indent`. Once indent reaches `0`, we 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.

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

If the user presses CTRL-C at any point that the program execution is in the `try` block, the `KeyboardInterrrupt` exception is raised and handled by this `except` statement. 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 an infinite loop, the user has a way to shut down the program.

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 variables in other functions. This limits what code could be changing the values of your variables, which can be helpful when it comes to debugging your code.

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?
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?

Practice Projects

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 type in 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.

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

The output of this program could look something like this:

```
Enter number:
3
10
5
16
8
```

```
4
2
1
```

Input Validation

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