



Part III

Advanced Shell Scripting

IN THIS PART

Chapter 17

Creating Functions

Chapter 18

Writing Scripts for Graphical Desktops

Chapter 19

Introducing sed and gawk

Chapter 20

Regular Expressions

Chapter 21

Advanced sed

Chapter 22

Advanced gawk

Chapter 23

Working with Alternative Shells

Creating Functions

IN THIS CHAPTER

Basic script functions

Returning a value

Using variables in functions

Array and variable functions

Function recursion

Creating a library

Using functions on the command line

Often while writing shell scripts, you'll find yourself using the same code in multiple locations. If it's just a small code snippet, it's usually not that big of a deal. However, rewriting large chunks of code multiple times in your shell script can get tiring. The bash shell provides a way to help you out by supporting user-defined functions. You can encapsulate your shell script code into a function and use it as many times as you want anywhere in your script. This chapter walks you through the process of creating your own shell script functions and demonstrates how to use them in other shell script applications.

Basic Script Functions

As you start writing more complex shell scripts, you'll find yourself reusing parts of code that perform specific tasks. Sometimes, it's something simple, such as displaying a text message and retrieving an answer from the script users. Other times, it's a complicated calculation that's used multiple times in your script as part of a larger process.

In each of these situations, it can get tiresome writing the same blocks of code over and over in your script. It would be nice to just write the block of code once and be able to refer to that block of code anywhere in your script without having to rewrite it.

The bash shell provides a feature allowing you to do just that. *Functions* are blocks of script code that you assign a name to and reuse anywhere in your code. Anytime you need to use that block of

code in your script, you simply use the function name you assigned it (referred to as *calling* the function). This section describes how to create and use functions in your shell scripts.

Creating a function

There are two formats you can use to create functions in bash shell scripts. The first format uses the keyword `function`, along with the function name you assign to the block of code:

```
function name {  
    commands  
}
```

The *name* attribute defines a unique name assigned to the function. Each function you define in your script must be assigned a unique name.

The *commands* are one or more bash shell commands that make up your function. When you call the function, the bash shell executes each of the commands in the order they appear in the function, just as in a normal script.

The second format for defining a function in a bash shell script more closely follows how functions are defined in other programming languages:

```
name() {  
    commands  
}
```

The empty parentheses after the function name indicate that you're defining a function. The same naming rules apply in this format as in the original shell script function format.

Using functions

To use a function in your script, specify the function name on a line, just as you would any other shell command:

```
$ cat test1  
#!/bin/bash  
# using a function in a script  
  
function func1 {  
    echo "This is an example of a function"  
}  
  
count=1  
while [ $count -le 5 ]  
do  
    func1  
    count=$(( $count + 1 )  
done
```

```

done

echo "This is the end of the loop"
func1
echo "Now this is the end of the script"
$
$ ./test1
This is an example of a function
This is an example of a function
This is an example of a function
This is an example of a function
This is an example of a function
This is the end of the loop
This is an example of a function
Now this is the end of the script
$

```

Each time you reference the `func1` function name, the bash shell returns to the `func1` function definition and executes any commands you defined there.

The function definition doesn't have to be the first thing in your shell script, but be careful. If you attempt to use a function before it's defined, you'll get an error message:

```

$ cat test2
#!/bin/bash
# using a function located in the middle of a script

count=1
echo "This line comes before the function definition"

function func1 {
    echo "This is an example of a function"
}

while [ $count -le 5 ]
do
    func1
    count=$(( $count + 1 ))
done
echo "This is the end of the loop"
func2
echo "Now this is the end of the script"

function func2 {
    echo "This is an example of a function"
}
$
$ ./test2
This line comes before the function definition
This is an example of a function

```

```
This is an example of a function
This is an example of a function
This is an example of a function
This is an example of a function
This is the end of the loop
./test2: func2: command not found
Now this is the end of the script
$
```

The first function, `func1`, was defined after a couple of statements in the script, which is perfectly fine. When the `func1` function was used in the script, the shell knew where to find it.

However, the script attempted to use the `func2` function before it was defined. Because the `func2` function wasn't defined, when the script reached the place where we used it, it produced an error message.

You also need to be careful about your function names. Remember, each function name must be unique, or you'll have a problem. If you redefine a function, the new definition overrides the original function definition, without producing any error messages:

```
$ cat test3
#!/bin/bash
# testing using a duplicate function name

function func1 {
echo "This is the first definition of the function name"
}

func1

function func1 {
    echo "This is a repeat of the same function name"
}

func1
echo "This is the end of the script"
$
$ ./test3
This is the first definition of the function name
This is a repeat of the same function name
This is the end of the script
$
```

The original definition of the `func1` function works fine, but after the second definition of the `func1` function, any subsequent uses of the function use the second definition.

Returning a Value

The bash shell treats functions like mini-scripts, complete with an exit status (see Chapter 11). There are three different ways you can generate an exit status for your functions.

The default exit status

By default, the exit status of a function is the exit status returned by the last command in the function. After the function executes, you use the standard `$?` variable to determine the exit status of the function:

```
$ cat test4
#!/bin/bash
# testing the exit status of a function

func1() {
    echo "trying to display a non-existent file"
    ls -l badfile
}

echo "testing the function: "
func1
echo "The exit status is: $?"
$
$ ./test4
testing the function:
trying to display a non-existent file
ls: badfile: No such file or directory
The exit status is: 1
$
```

The exit status of the function is 1 because the last command in the function failed. However, you have no way of knowing if any of the other commands in the function completed successfully or not. Look at this example:

```
$ cat test4b
#!/bin/bash
# testing the exit status of a function

func1() {
    ls -l badfile
    echo "This was a test of a bad command"
```

```
}

echo "testing the function:"
func1
echo "The exit status is: $?"
$
$ ./test4b
testing the function:
ls: badfile: No such file or directory
This was a test of a bad command
The exit status is: 0
$
```

This time, because the function ended with an `echo` statement that completed successfully, the exit status of the function is 0, even though one of the commands in the function failed. Using the default exit status of a function can be a dangerous practice. Fortunately, we have a couple of other solutions.

Using the return command

The bash shell uses the `return` command to exit a function with a specific exit status. The `return` command allows you to specify a single integer value to define the function exit status, providing an easy way for you to programmatically set the exit status of your function:

```
$ cat test5
#!/bin/bash
# using the return command in a function

function dbl {
    read -p "Enter a value: " value
    echo "doubling the value"
    return ${value * 2}
}

dbl
echo "The new value is $?"
$
```

The `dbl` function doubles the integer value contained in the `$value` variable provided by the user input. It then returns the result using the `return` command, which the script displays using the `$?` variable.

You must be careful, however, when using this technique to return a value from a function. Keep the following two tips in mind to avoid problems:

- Remember to retrieve the return value as soon as the function completes.
- Remember that an exit status must be in the range of 0 to 255.

If you execute any other commands before retrieving the value of the function, using the `$?` variable, the return value from the function is lost. Remember that the `$?` variable returns the exit status of the last executed command.

The second problem defines a limitation for using this return value technique. Because an exit status must be less than 256, the result of your function must produce an integer value less than 256. Any value over that returns an error value:

```
$ ./test5
Enter a value: 200
doubling the value
The new value is 1
$
```

You cannot use this return value technique if you need to return either larger integer values or a string value. Instead, you need to use another method, demonstrated in the next section.

Using function output

Just as you can capture the output of a command to a shell variable, you can also capture the output of a function to a shell variable. You can use this technique to retrieve any type of output from a function to assign to a variable:

```
result='dbl'
```

This command assigns the output of the `dbl` function to the `$result` shell variable. Here's an example of using this method in a script:

```
$ cat test5b
#!/bin/bash
# using the echo to return a value

function dbl {
    read -p "Enter a value: " value
    echo ${value * 2}
}

result=$(dbl)
echo "The new value is $result"
$
$ ./test5b
Enter a value: 200
The new value is 400
$
$ ./test5b
```



```
Enter a value: 1000
The new value is 2000
$
```

The new function now uses an `echo` statement to display the result of the calculation. The script just captures the output of the `dbl` function instead of looking at the exit status for the answer.

There's a subtle trick that this example demonstrates. You'll notice that the `dbl` function really outputs two messages. The `read` command outputs a short message querying the user for the value. The bash shell script is smart enough to not consider this as part of the `STDOUT` output and ignores it. If you had used an `echo` statement to produce this query message to the user, it would have been captured by the shell variable as well as the output value.

NOTE

Using this technique, you can also return floating point and string values, making this an extremely versatile method for returning values from functions.

Using Variables in Functions

You might have noticed in the `test5` example in the previous section that we used a variable called `$value` within the function to hold the value that it processed. When you use variables in your functions, you need to be somewhat careful about how you define and handle them. This is a common cause of problems in shell scripts. This section goes over a few techniques for handling variables both inside and outside your shell script functions.

Passing parameters to a function

As mentioned earlier in the “Returning a Value” section, the bash shell treats functions just like mini-scripts. This means that you can pass parameters to a function just like a regular script (see Chapter 14).

Functions can use the standard parameter environment variables to represent any parameters passed to the function on the command line. For example, the name of the function is defined in the `$0` variable, and any parameters on the function command line are defined using the variables `$1`, `$2`, and so on. You can also use the special variable `$#` to determine the number of parameters passed to the function.

When specifying the function in your script, you must provide the parameters on the same command line as the function, like this:

```
func1 $value1 10
```

The function can then retrieve the parameter values using the parameter environment variables. Here's an example of using this method to pass values to a function:

```
$ cat test6
#!/bin/bash
# passing parameters to a function

function addem {
    if [ $# -eq 0 ] || [ $# -gt 2 ]
    then
        echo -1
    elif [ $# -eq 1 ]
    then
        echo ${ $1 + $1 }
    else
        echo ${ $1 + $2 }
    fi
}

echo -n "Adding 10 and 15: "
value=$(addem 10 15)
echo $value
echo -n "Let's try adding just one number: "
value=$(addem 10)
echo $value
echo -n "Now trying adding no numbers: "
value=$(addem)
echo $value
echo -n "Finally, try adding three numbers: "
value=$(addem 10 15 20)
echo $value
$
$ ./test6
Adding 10 and 15: 25
Let's try adding just one number: 20
Now trying adding no numbers: -1
Finally, try adding three numbers: -1
$
```

The `addem` function in the `test6` script first checks the number of parameters passed to it by the script. If there aren't any parameters, or if there are more than two parameters, `addem` returns a value of `-1`. If there's just one parameter, `addem` adds the parameter to itself for the result. If there are two parameters, `addem` adds them together for the result.

Because the function uses the special parameter environment variables for its own parameter values, it can't directly access the script parameter values from the command line of the script. The following example fails:

```
$ cat badtest1
#!/bin/bash
```

```
# trying to access script parameters inside a function

function badfunc1 {
    echo $[ $1 * $2 ]
}

if [ $# -eq 2 ]
then
    value=$(badfunc1)
    echo "The result is $value"
else
    echo "Usage: badtest1 a b"
fi
$
$ ./badtest1
Usage: badtest1 a b
$ ./badtest1 10 15
./badtest1: * : syntax error: operand expected (error token is "*"
)
The result is
$
```

Even though the function uses the `$1` and `$2` variables, they aren't the same `$1` and `$2` variables available in the main part of the script. Instead, if you want to use those values in your function, you have to manually pass them when you call the function:

```
$ cat test7
#!/bin/bash
# trying to access script parameters inside a function

function func7 {
    echo $[ $1 * $2 ]
}

if [ $# -eq 2 ]
then
    value=$(func7 $1 $2)
    echo "The result is $value"
else
    echo "Usage: badtest1 a b"
fi
$
$ ./test7
Usage: badtest1 a b
$ ./test7 10 15
The result is 150
$
```

By passing the `$1` and `$2` variables to the function, they become available for the function to use, just like any other parameter.

Handling variables in a function

One thing that causes problems for shell script programmers is the *scope* of a variable. The scope is where the variable is visible. Variables defined in functions can have a different scope than regular variables. That is, they can be hidden from the rest of the script.

Functions use two types of variables:

- Global
- Local

The following sections describe how to use both types of variables in your functions.

Global variables

Global variables are variables that are valid anywhere within the shell script. If you define a global variable in the main section of a script, you can retrieve its value inside a function. Likewise, if you define a global variable inside a function, you can retrieve its value in the main section of the script.

By default, any variables you define in the script are global variables. Variables defined outside of a function can be accessed within the function just fine:

```
$ cat test8
#!/bin/bash
# using a global variable to pass a value

function dbl {
    value=$(( $value * 2 ))
}

read -p "Enter a value: " value
dbl
echo "The new value is: $value"
$
$ ./test8
Enter a value: 450
The new value is: 900
$
```

The `$value` variable is defined outside of the function and assigned a value outside of the function. When the `dbl` function is called, the variable and its value are still valid inside

the function. When the variable is assigned a new value inside the function, that new value is still valid when the script references the variable.

This can be a dangerous practice, however, especially if you intend to use your functions in different shell scripts. It requires that you know exactly what variables are used in the function, including any variables used to calculate values not returned to the script. Here's an example of how things can go bad:

```
$ cat badtest2
#!/bin/bash
# demonstrating a bad use of variables

function func1 {
    temp=$(( $value + 5 ))
    result=$(( $temp * 2 ))
}

temp=4
value=6

func1
echo "The result is $result"
if [ $temp -gt $value ]
then
    echo "temp is larger"
else
    echo "temp is smaller"
fi
$
$ ./badtest2
The result is 22
temp is larger
$
```

Because the `$temp` variable was used in the function, its value is compromised in the script, producing a result that you may not have intended. There's an easy way to solve this problem in your functions, as shown in the next section.

Local variables

Instead of using global variables in functions, any variables that the function uses internally can be declared as local variables. To do that, just use the `local` keyword in front of the variable declaration:

```
local temp
```

You can also use the `local` keyword in an assignment statement while assigning a value to the variable:

```
local temp=$(( $value + 5 ))
```

The `local` keyword ensures that the variable is limited to only within the function. If a variable with the same name appears outside the function in the script, the shell keeps the two variable values separate. Now you can easily keep your function variables separate from your script variables and share only the ones you want to share:

```
$ cat test9
#!/bin/bash
# demonstrating the local keyword

function func1 {
    local temp=$(( $value + 5 ))
    result=$(( $temp * 2 ))
}

temp=4
value=6

func1
echo "The result is $result"
if [ $temp -gt $value ]
then
    echo "temp is larger"
else
    echo "temp is smaller"
fi
$
$ ./test9
The result is 22
temp is smaller
$
```

Now when you use the `$temp` variable within the `func1` function, it doesn't affect the value assigned to the `$temp` variable in the main script.

Array Variables and Functions

Chapter 6 discussed an advanced way of allowing a single variable to hold multiple values by using arrays. Using array variable values with functions is a little tricky, and there are some special considerations. This section describes a technique that allows you to do that.

Passing arrays to functions

The art of passing an array variable to a script function can be confusing. If you try to pass the array variable as a single parameter, it doesn't work:

```
$ cat badtest3
#!/bin/bash
```

```
# trying to pass an array variable

function testit {
    echo "The parameters are: $@"
    thisarray=$1
    echo "The received array is ${thisarray[*]}"
}

myarray=(1 2 3 4 5)
echo "The original array is: ${myarray[*]}"
testit $myarray
$
$ ./badtest3
The original array is: 1 2 3 4 5
The parameters are: 1
The received array is 1
$
```

If you try using the array variable as a function parameter, the function only picks up the first value of the array variable.

To solve this problem, you must disassemble the array variable into its individual values and use the values as function parameters. Inside the function, you can reassemble all the parameters into a new array variable. Here's an example of doing this:

```
$ cat test10
#!/bin/bash
# array variable to function test

function testit {
    local newarray
    newarray=($(echo "$@"))
    echo "The new array value is: ${newarray[*]}"
}

myarray=(1 2 3 4 5)
echo "The original array is ${myarray[*]}"
testit ${myarray[*]}
$
$ ./test10
The original array is 1 2 3 4 5
The new array value is: 1 2 3 4 5
$
```

The script uses the `$myarray` variable to hold all the individual array values to place them all on the command line for the function. The function then rebuilds the array variable from the command line parameters. Once inside the function, the array can be used just like any other array:

```
$ cat test11
#!/bin/bash
```

```
# adding values in an array

function addarray {
    local sum=0
    local newarray
    newarray=$(echo "$@")
    for value in ${newarray[*]}
    do
        sum=$(( sum + $value ))
    done
    echo $sum
}

myarray=(1 2 3 4 5)
echo "The original array is: ${myarray[*]}"
arg1=$(echo ${myarray[*]})
result=$(addarray $arg1)
echo "The result is $result"
$
$ ./test11
The original array is: 1 2 3 4 5
The result is 15
$
```

The `addarray` function iterates through the array values, adding them together. You can put any number of values in the `myarray` array variable, and the `addarray` function adds them.

Returning arrays from functions

Passing an array variable from a function back to the shell script uses a similar technique. The function uses an `echo` statement to output the individual array values in the proper order, and the script must reassemble them into a new array variable:

```
$ cat test12
#!/bin/bash
# returning an array value

function arraydbl {
    local origarray
    local newarray
    local elements
    local i
    origarray=$(echo "$@")
    newarray=$(echo "$@")
    elements=$(( $# - 1 ))
    for (( i = 0; i <= $elements; i++ ))
    {
        newarray[$i]=$[ ${origarray[$i]} * 2 ]
    }
}
```



```
    echo ${newarray[*]}
}

myarray=(1 2 3 4 5)
echo "The original array is: ${myarray[*]}"
arg1=$(echo ${myarray[*]})
result=$(arraydblr $arg1)
echo "The new array is: ${result[*]}"
$
$ ./test12
The original array is: 1 2 3 4 5
The new array is: 2 4 6 8 10
```

The script passes the array value, using the *\$arg1* variable to the *arraydblr* function. The *arraydblr* function reassembles the array into a new array variable, and it makes a copy for the output array variable. It then iterates through the individual array variable values, doubles each value, and places it into the copy of the array variable in the function.

The *arraydblr* function then uses the *echo* statement to output the individual values of the array variable values. The script uses the output of the *arraydblr* function to reassemble a new array variable with the values.

Function Recursion

One feature that local function variables provide is *self-containment*. A self-contained function doesn't use any resources outside of the function, other than whatever variables the script passes to it in the command line.

This feature enables the function to be called *recursively*, which means that the function calls itself to reach an answer. Usually, a recursive function has a base value that it eventually iterates down to. Many advanced mathematical algorithms use recursion to reduce a complex equation down one level repeatedly, until they get to the level defined by the base value.

The classic example of a recursive algorithm is calculating factorials. A factorial of a number is the value of the preceding numbers multiplied with the number. Thus, to find the factorial of 5, you'd perform the following equation:

$$5! = 1 * 2 * 3 * 4 * 5 = 120$$

Using recursion, the equation is reduced down to the following format:

$$x! = x * (x-1)!$$

or in English, the factorial of *x* is equal to *x* times the factorial of *x-1*. This can be expressed in a simple recursive script:

```
function factorial {
    if [ $1 -eq 1 ]
```

```

    then
        echo 1
    else
        local temp=$(( $1 - 1 ))
        local result='factorial $temp'
        echo $[ $result * $1 ]
    fi
}

```

The factorial function uses itself to calculate the value for the factorial:

```

$ cat test13
#!/bin/bash
# using recursion

function factorial {
    if [ $1 -eq 1 ]
    then
        echo 1
    else
        local temp=$(( $1 - 1 ))
        local result=$(factorial $temp)
        echo $[ $result * $1 ]
    fi
}

read -p "Enter value: " value
result=$(factorial $value)
echo "The factorial of $value is: $result"
$
$ ./test13
Enter value: 5
The factorial of 5 is: 120
$

```

Using the factorial function is easy. Having created a function like this, you may want to use it in other scripts. Next, we look at how to do that efficiently.

Creating a Library

It's easy to see how functions can help save typing in a single script, but what if you just happen to use the same single code block between scripts? It's obviously challenging if you have to define the same function in each script, only to use it one time in each script.

There's a solution for that problem! The bash shell allows you to create a *library file* for your functions and then reference that single library file in as many scripts as you need to.

The first step in the process is to create a common library file that contains the functions you need in your scripts. Here's a simple library file called `myfuncs` that defines three simple functions:

```
$ cat myfuncs
# my script functions

function addem {
    echo $[ $1 + $2 ]
}

function multem {
    echo $[ $1 * $2 ]
}

function divem {
    if [ $2 -ne 0 ]
    then
        echo $[ $1 / $2 ]
    else
        echo -1
    fi
}
$
```

The next step is to include the `myfuncs` library file in your script files that want to use any of the functions. This is where things get tricky.

The problem is with the scope of shell functions. As with environment variables, shell functions are valid only for the shell session in which you define them. If you run the `myfuncs` shell script from your shell command line interface prompt, the shell creates a new shell and runs the script in that new shell. This defines the three functions for that shell, but when you try to run another script that uses those functions, they aren't available.

This applies to scripts as well. If you try to just run the library file as a regular script file, the functions don't appear in your script:

```
$ cat badtest4
#!/bin/bash
# using a library file the wrong way
./myfuncs

result=$(addem 10 15)
echo "The result is $result"
$
$ ./badtest4
./badtest4: addem: command not found
The result is
$
```

The key to using function libraries is the `source` command. The `source` command executes commands within the current shell context instead of creating a new shell to execute them. You use the `source` command to run the library file script inside of your shell script. This makes the functions available to the script.

The `source` command has a shortcut alias, called the *dot operator*. To source the `myfuncs` library file in a shell script, you just need to add the following line:

```
. ./myfuncs
```

This example assumes that the `myfuncs` library file is located in the same directory as the shell script. If not, you need to use the appropriate path to access the file. Here's an example of creating a script that uses the `myfuncs` library file:

```
$ cat test14
#!/bin/bash
# using functions defined in a library file
. ./myfuncs

value1=10
value2=5
result1=$((addem $value1 $value2))
result2=$((multem $value1 $value2))
result3=$((divem $value1 $value2))
echo "The result of adding them is: $result1"
echo "The result of multiplying them is: $result2"
echo "The result of dividing them is: $result3"
$
$ ./test14
The result of adding them is: 15
The result of multiplying them is: 50
The result of dividing them is: 2
$
```

The script successfully uses the functions defined in the `myfuncs` library file.

Using Functions on the Command Line

You can use script functions to create some pretty complex operations. Sometimes, it would be nice to be able to use these functions directly on the command line interface prompt.

Just as you can use a script function as a command in a shell script, you can also use a script function as a command in the command line interface. This is a nice feature because after you define the function in the shell, you can use it from any directory on the system; you don't have to worry about a script being in your `PATH` environment variable. The trick is to get the shell to recognize the function. You can do that in a couple of ways.

Creating functions on the command line

Because the shell interprets commands as you type them, you can define a function directly on the command line. You can do that in two ways.

The first method defines the function all on one line:

```
$ function divem { echo $[ $1 / $2 ]; }
$ divem 100 5
20
$
```

When you define the function on the command line, you must remember to include a semicolon at the end of each command, so the shell knows where to separate commands:

```
$ function doubleit { read -p "Enter value: " value; echo $[
$value * 2 ]; }
$
$ doubleit
Enter value: 20
40
$
```

The other method is to use multiple lines to define the function. When you do that, the bash shell uses the secondary prompt to prompt you for more commands. Using this method, you don't need to place a semicolon at the end of each command; just press the Enter key:

```
$ function multem {
> echo $[ $1 * $2 ]
> }
$ multem 2 5
10
$
```

When you use the brace at the end of the function, the shell knows that you're finished defining the function.

CAUTION

Be extremely careful when creating functions on the command line. If you use a function with the same name as a built-in command or another command, the function overrides the original command.

Defining functions in the .bashrc file

The obvious downside to defining shell functions directly on the command line is that when you exit the shell, your function disappears. For complex functions, this can become a problem.

A much simpler method is to define the function in a place where it is reloaded by the shell each time you start a new shell.

The best place to do that is the `.bashrc` file. The bash shell looks for this file in your home directory each time it starts, whether interactively or as the result of starting a new shell from within an existing shell.

Directly defining functions

You can define the functions directly in the `.bashrc` file in your home directory. Most Linux distributions already define some things in the `.bashrc` file, so be careful not to remove those items. Just add your functions to the bottom of the existing file. Here's an example of doing that:

```
$ cat .bashrc
# .bashrc

# Source global definitions
if [ -r /etc/bashrc ]; then
    . /etc/bashrc
fi

function addem {
    echo $[ $1 + $2 ]
}
$
```

The function doesn't take effect until the next time you start a new bash shell. After you do that, you can use the function anywhere on the system.

Sourcing function files

Just as in a shell script, you can use the `source` command (or its alias the dot operator) to add functions from an existing library file to your `.bashrc` script:

```
$ cat .bashrc
# .bashrc

# Source global definitions
if [ -r /etc/bashrc ]; then
    . /etc/bashrc
fi

. /home/rich/libraries/myfuncs
$
```

Make sure that you include the proper pathname to reference the library file for the bash shell to find. The next time you start a shell, all the functions in your library are available at the command line interface:

```
$ addem 10 5
15
$ multem 10 5
50
$ divem 10 5
2
$
```

Even better, the shell also passes any defined functions to child shell processes so your functions are automatically available for any shell scripts you run from your shell session. You can test this by writing a script that uses the functions without defining or sourcing them:

```
$ cat test15
#!/bin/bash
# using a function defined in the .bashrc file

value1=10
value2=5
result1=$(addem $value1 $value2)
result2=$(multem $value1 $value2)
result3=$(divem $value1 $value2)
echo "The result of adding them is: $result1"
echo "The result of multiplying them is: $result2"
echo "The result of dividing them is: $result3"
$
$ ./test15
The result of adding them is: 15
The result of multiplying them is: 50
The result of dividing them is: 2
$
```

Even without sourcing the library file, the functions worked perfectly in the shell script.

Following a Practical Example

There's much more to using functions than just creating your own functions to work with. In the open source world, code sharing is key, and that also applies to shell script functions. Quite a few different shell script functions are available for you to download and use in your own applications.

This section walks through downloading, installing, and using the GNU shtool shell script function library. The shtool library provides some simple shell script functions for performing everyday shell functions, such as working with temporary files and folders or formatting output to display.

Downloading and installing

The first step in the process is to download and install the GNU shtool library to your system so you can use the library functions in your own shell scripts. To do that, you need to use an FTP client program or a browser in a graphical desktop. Use this URL to download the shtool package:

```
ftp://ftp.gnu.org/gnu/shtool/shtool-2.0.8.tar.gz
```

This downloads the file `shtool-2.0.8.tar.gz` to the download folder. From there, you can use the `cp` command line tool or the graphical file manager tool in your Linux distribution (such as Nautilus in Ubuntu) to copy the file to your Home folder.

After you copy the file to your Home folder, you can extract it using the `tar` command:

```
tar -zxvf shtool-2.0.8.tar.gz
```

This extracts the package files into a folder named `shtool-2.0.8`. Now you're ready to build the shell script library file.

Building the library

The shtool distribution file must be configured for your specific Linux environment. To do that, it uses standard `configure` and `make` commands, commonly used in the C programming environment. To build the library file, you just need to run two commands:

```
$ ./configure
$ make
```

The `configure` command checks the software necessary to build the shtool library file. As it finds the tools it needs, it modifies the configuration file with the proper paths to the tools.

The `make` command runs through the steps to build the shtool library file. The resulting file (`shtool`) is the full library package file. You can test the library file using the `make` command as well:

```
$ make test
Running test suite:
echo.....ok
mdate.....ok
table.....ok
prop.....ok
move.....ok
install.....ok
mkdir.....ok
mkln.....ok
```



```
mkshadow.....ok
fixperm.....ok
rotate.....ok
tarball.....ok
subst.....ok
platform.....ok
arx.....ok
slo.....ok
scpp.....ok
version.....ok
path.....ok
OK: passed: 19/19
$
```

The test mode tests all the functions available in the shtool library. If all pass, then you're ready to install the library into a common location on your Linux system so all your scripts can use it. To do that, you can use the `install` option of the `make` command. However, you need to be logged in as the root user account to run it:

```
$ su
Password:
# make install
./shtool mkdir -f -p -m 755 /usr/local
./shtool mkdir -f -p -m 755 /usr/local/bin
./shtool mkdir -f -p -m 755 /usr/local/share/man/man1
./shtool mkdir -f -p -m 755 /usr/local/share/aclocal
./shtool mkdir -f -p -m 755 /usr/local/share/shtool
...
./shtool install -c -m 644 sh.version /usr/local/share/shtool/sh.version
./shtool install -c -m 644 sh.path /usr/local/share/shtool/sh.path
#
```

Now you're ready to start using the functions in your own shell scripts!

The shtool library functions

The shtool library provides quite a few functions that can come in handy when working with shell scripts. Table 17.1 shows the functions available in the library.

TABLE 17.1 The shtool Library Functions

Function	Description
Arx	Creates an archive with extended features
Echo	Displays the string value with construct expansion
fixperm	Changes file permissions inside a folder tree

install	Installs a script or file
mdate	Displays modification time of a file or directory
mkdir	Creates one or more directories
Mkln	Creates a link using relative paths
mkshadow	Creates a shadow tree
move	Moves files with substitution
Path	Works with program paths
platform	Displays the platform identity
Prop	Displays an animated progress propeller
rotate	Rotates logfiles
Scpp	The sharing C pre-processor
Slo	Separates linker options by library class
Subst	Uses sed substitution operations
Table	Displays field-separated data in a table format
tarball	Creates tar files from files and folders
version	Creates a version information file

Each of the `shtool` functions has lots of options and arguments that you can use to modify how it works. Here's the format to use a `shtool` function:

```
shtool [options] [function [options] [args]]
```

Using the library

You can use the `shtool` functions directly from the command line or from within your shell scripts. Here's an example of using the `platform` function inside a shell script:

```
$ cat test16
#!/bin/bash

shtool platform
$ ./test16
Ubuntu 14.04 (iX86)
$
```

The `platform` function returns the Linux distribution and the CPU hardware that the host system is using. One of my favorites is the `prop` function. It creates a spinning propeller from alternating the `\`, `|`, `/`, and `-` characters while something is processing. That's a great tool to help show your shell script users that something is happening in the background while the script is running.

To use the `prop` function, you just pipe the output of the function you want to monitor to the `shtool` script:

```
$ ls -al /usr/bin | shtool prop -p "waiting..."
waiting...
$
```

The `prop` function alternates between the propeller characters to indicate that something is happening. In this case, it's the output from the `ls` command. How much of that you see depends on how fast your CPU can list out all the files in the `/usr/bin` folder! The `-p` option allows you to customize the output text that appears before the propeller characters. Now that's getting fancy!

Summary

Shell script functions allow you to place script code that's repeated throughout the script in a single place. Instead of having to rewrite blocks of code, you can create a function containing the code block and then just reference the function name in your script. The bash shell jumps to the function code block whenever it sees the function name used in the script.

You can even create script functions that return values. This allows you to create functions that interact with the script, returning both numeric and character data. Script functions can return numeric data by using the exit status of the last command in the function or using the `return` command. The `return` command allows you to programmatically set the exit status of your function to a specific value based on the results of the function.

Functions can also return values using the standard `echo` statement. You can capture the output data using the backtick character as you would any other shell command. This enables you to return any type of data from a function, including strings and floating-point numbers.

You can use shell variables within your functions, assigning values to variables and retrieving values from existing variables. This allows you to pass any type of data both into and out of a script function from the main script program. Functions also allow you to define local variables, which are accessible only from within the function code block. Local variables allow you to create self-contained functions, which don't interfere with any variables or processes used in the main shell script.

Functions can also call other functions, including themselves. When a function calls itself, it is called recursion. A recursive function often has a base value that is the terminal value of the function. The function continues to call itself with a decreasing parameter value until the base value is reached.

If you use lots of functions in your shell scripts, you can create library files of script functions. The library files can be included in any shell script file by using the `source` command, or its alias, the dot operator. This is called sourcing the library file. The shell doesn't run the library file but makes the functions available within the shell that runs the script. You can use this same technique to create functions that you can use on the normal shell command line. You can either define functions directly on the command line or you can add them to your `.bashrc` file so they are available for each new shell session you start. This is a handy way to create utilities that can be used no matter what your `PATH` environment variable is set to.

The next chapter discusses the use of text graphics in your scripts. In this day of modern graphical interfaces, sometimes a plain text interface just doesn't cut it. The bash shell provides some easy ways for you to incorporate simple graphics features in your scripts to help spice things up.