Shell Functions

As programs get longer and more complex, they become more difficult to design, code, and maintain. As with any large endeavor, it is often useful to break a single, large task into a series of smaller tasks.

In this lesson, we will begin to break our single monolithic script into a number of separate functions.

To get familiar with this idea, let's consider the description of an everyday task -- going to the market to buy food. Imagine that we were going to describe the task to a man from Mars.

Our first top-level description might look like this:

1. Leave house
2. Drive to market
3. Park car
4. Enter market
5. Purchase food
6. Drive home
7. Park car
8. Enter house

This description covers the overall process of going to the market; however a man from Mars will probably require additional detail. For example, the "Park car" sub task could be described as follows:

1. Find parking space
2. Drive car into space
3. Turn off motor
4. Set parking brake
5. Exit car
6. Lock car

Of course the task "Turn off motor" has a number of steps such as "turn off ignition" and "remove key from ignition switch," and so on.

This process of identifying the top-level steps and developing increasingly detailed views of those steps is called *top-down design*. This technique allows you to break large complex tasks into many small, simple tasks.

As our script continues to grow, we will use top down design to help us plan and code our script.

If we look at our script's top-level tasks, we find the following list:

1. Open page
2. Open head section
3. Write title
4. Close head section
5. Open body section
6. Write title
7. Write time stamp

8. Close body section
9. Close page

All of these tasks are implemented, but we want to add more. Let's insert some additional tasks after task 7:

7. Write time stamp
8. Write system release info
9. Write up-time
10. Write drive space
11. Write home space
12. Close body section
13. Close page

It would be great if there were commands that performed these additional tasks. If there were, we could use command substitution to place them in our script like so:

```
#!/bin/bash

# sysinfo_page - A script to produce a system information HTML file

##### Constants

TITLE="System Information for $HOSTNAME"
RIGHT_NOW=$(date +"%x %r %Z")
TIME_STAMP="Updated on $RIGHT_NOW by $USER"

##### Main

cat <<- _EOF_
  <html>
  <head>
      <title>$TITLE</title>
  </head>

  <body>
      <h1>$TITLE</h1>
      <p>$TIME_STAMP</p>
      $(system_info)
      $(show_uptime)
      $(drive_space)
      $(home_space)
  </body>
  </html>
_EOF_
```

While there are no commands that do exactly what we need, we can create them using *shell functions*.

As we learned in lesson 2, shell functions act as "little programs within programs" and allow us to follow top-down design principles. To add the shell functions to our script, we change it so:

```
#!/bin/bash

# sysinfo_page - A script to produce an system information HTML file

##### Constants

TITLE="System Information for $HOSTNAME"
RIGHT_NOW=$(date +"%x %r %Z")
TIME_STAMP="Updated on $RIGHT_NOW by $USER"

##### Functions

system_info()
{

}


show_uptime()
{

}


drive_space()
{

}


home_space()
{

}

##### Main

cat <<- _EOF_
  <html>
  <head>
      <title>$TITLE</title>
  </head>

  <body>
      <h1>$TITLE</h1>
      <p>$TIME_STAMP</p>
      $(system_info)
      $(show_uptime)
      $(drive_space)
      $(home_space)
  </body>
  </html>
_EOF_
```

A couple of important points about functions: First, they must appear before you attempt to use them. Second, the function body (the portions of the function between the { and } characters) must contain at least one valid command. As written, the script will not execute without error, because the function bodies are empty. The simple way to fix this is to place a **`return`** statement in each function body. After you do this, our script will execute successfully again.

Keep Your Scripts Working

When you are developing a program, it is is often a good practice to add a small amount of code, run the script, add some more code, run the script, and so on. This way, if you introduce a mistake into your code, it will be easier to find and correct.

As you add functions to your script, you can also use a technique called *stubbing* to help watch the logic of your script develop. Stubbing works like this: imagine that we are going to create a function called "system_info" but we haven't figured out all of the details of its code yet. Rather than hold up the development of the script until we are finished with system_info, we just add an **`echo`** command like this:

```
system_info()
{
    # Temporary function stub
    echo "function system_info"
}
```

This way, our script will still execute successfully, even though we do not yet have a finished system_info function. We will later replace the temporary stubbing code with the complete working version.

The reason we use an **`echo`** command is so we get some feedback from the script to indicate that the functions are being executed.

Let's go ahead and write stubs for our new functions and keep the script working.

```
#!/bin/bash

# sysinfo_page - A script to produce an system information HTML file

##### Constants

TITLE="System Information for $HOSTNAME"
RIGHT_NOW=$(date +"%x %r %Z")
TIME_STAMP="Updated on $RIGHT_NOW by $USER"

##### Functions

system_info()
{
    # Temporary function stub
    echo "function system_info"
}
```

```
show_uptime()
{
    # Temporary function stub
    echo "function show_uptime"
}


drive_space()
{
    # Temporary function stub
    echo "function drive_space"
}


home_space()
{
    # Temporary function stub
    echo "function home_space"
}


##### Main

cat <<- _EOF_
  <html>
  <head>
      <title>$TITLE</title>
  </head>

  <body>
      <h1>$TITLE</h1>
      <p>$TIME_STAMP</p>
      $(system_info)
      $(show_uptime)
      $(drive_space)
      $(home_space)
  </body>
  </html>
_EOF_
```
