Shell Functions -- A How To

Shell Functions

how to – create, save, & re-use

Marty McGowan

This book is for sale at http://leanpub.com/shellfunctions

This version was published on 2019-10-12

Tweet This Book!

Please help Marty McGowan by spreading the word about this book on Twitter!

The suggested hashtag for this book is #shellfunctions.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#shellfunctions

Martin McGowan Sr and Jr, who edited and published “The Appleton Press” (Appleton MN) for the first 50 yrs of the last 100.

Contents

Acknowledgments

- Bill Anderson, with whom I collaborated on “The Software Assembly Line” and has offered encouragement on this edition.
- my friends on the itToolbox, shell forum, for corrections to the “set --” idiom
- Nitin Chitniss, a Fidessa colleague, for the incentive to capture small shell functions
- my employers of the prior decade:
  - Benedictine Academy, Elizabeth NJ, who taught real challenges
  - Fidessa, who re-invigorated my love for Unix® and the shell
  - and Union Co College, Cranford NJ, who eased me into retirement

Preface

In this book you will learn how to write, use, save, and re-use shell functions. The examples have all been worked in the **bash** shell.

The shell concepts are introductory. You may be familiar with many of the concepts. But if you haven’t used shell functions, you will learn how simple and powerful they are to write, use, and save. Each chapter includes exercises to follow. Work them at your terminal to gain confidence in using shell functions.

By the time you complete the exercises in this book, you will now how to add functions to a shell library, how to recall and make them available in terminal commands or used by other functions.

Introduction

Each chapter is simple enough to require a half an hour of your time. Each is meant to be worked at a terminal window running a **bash** shell.

When you've completed these exercises, you will be comfortable with creating, using, saving and re-using shell functions.

You can explore the planned future topics for shell functions in the what's next section.

To get started, here are the few assumptions we make. That you:

- have access to an open terminal window
- can open simultaneous multiple terminal windows
- are running the **bash** shell

While it might be nice to have experience with one of the popular text editors: **notepad**, **textedit**, **vi(m)**, … or **emacs** to use for command line editing, the book includes a reference to keyboard command editing.

The text is sprinkled with links to more detailed treatment of fundamental topics. Each chapter has suggestions for experimenting Following the experiments will enable you to better grasp the shell concepts.

If you need help getting started or at any point in the material, you can [contact me here](mailto:martymcgowan@alum.mit.edu?subject=introduction). In any case I'd appreciate your feedback.

<link rel=”stylesheet” type=”text/css” href=”./mcgowan.css” />

1. Write a shell function

The simplest shell functions may be written on a single line at the command prompt.

1.1 New Functions

In this chapter, you will write and use two simple shell functions:

- hello - programmers birth announcement
- today - from the date command.

1.2 New Concepts

In this chapter you will learn these concepts, how to :

- write a demonstration function on the command line in different formats,
- call the function,
- show the function text.
- use features of the **date** command

1.3 Hello world

In this book, when you see a dollar sign, that stands in for your command prompt.

```
$ ...
```

Here is the Programmers Birth Announcement: **Hello World!**. Type this at your command prompt:

```
$ hello () { echo 'Hello World!'; }
```

On the above line, following the command prompt, type everything from **hello** thru the closing curly brace, followed by a carriage return.

You use the function by typing its name, “hello” at the command line. Here is the function definition (on line 1), followed by using it (line 2), and the shell’s response (line 3). The next shell prompt is on line 4.

```
1 $ hello () { echo 'Hello World!' ; }
2 $ hello
3 Hello World!
4 $
```

You can see the definition of a function with the **declare** bash built-in:

```
$ declare -f hello
```

Type that command. Notice your function has been slightly reformatted. More on that later. Here is the results of the command:

```
hello ()
{
    echo 'Hello World!'
}
```

1.4 Getting it right

The function syntax:

name () { command … ; }

has a **name** of your choosing, and a **command** or semi-colon-separated commands of your choosing. While there are other ways to define a function, I've found the parenthesis-pair simplest to identify the name. A pair of curly braces enclose the commands. And if the trailing curly brace is on the same line as a command, you need a semi-colon separator. You can separate commands on separate lines. The only mandatory space in the function definition is the space following the first curly brace.

For completenes, you will see two other formats to define a function. In this book, we are using the most concise. The **hello** function could have just as well been written as

```
$ function hello () { echo 'Hello World!' ; }
```

or

```
$ function hello { echo 'Hello World!' ; }
```

Experiment with both methods of defining the function and follow by using the **declare** example above.

1.5 More interesting

Arguments, like file names and options, make functions more useful. But before looking at how arguments are used, whet your appetite with this one, called **today**:

```
$ declare -f today
today ()
{
    date +%Y%m%d
}
$ today
20190517
$ ...
```

Type the definition and invoke your new function **today**. Since the **date** format specifcation takes almost any upper- or lower-case letter, we’ll experiment with the all the letter arguments to test another function.

1.6 Activity

- *What does the* ***declare*** *command tell you about the function syntax?*
- *how might you write a function to capture that idea?* a good answer requires you know how to use function arguments. feel free to experiment.
- *what would you name that function?*
- investigate the options to the **date** command: search for *date manual page*.

Mail me if you have questions[1]

<link rel="stylesheet" type="text/css" href="./mcgowan.css" />

[1]mailto:martymcgowan@alum.mit.edu?subject=writeAshellFunction

2. Use Function Arguments

In this chapter we add arguments when calling a function. When calling, the tokens after the function name are called the *arguments*. In the body of the function, the *positional parameters* ($1, $2, …) are assigned the respective argument values.

2.1 New Functions

- dateArg - display arg and date format
- date_arg - adorned dateArg

2.2 New Concepts

These are the new concepts:

- shell arguments - positional parameters
- for loop
- bash brace expansion { … }

The **for** loop iterates over a range of aguments. In the exercise, we'll also demonstrate a bash shorthand for argument expansion.

2.3 the date command

In the last exercise you used the **date** command in a very limited way. You are probably aware the date command has a number of upper- and lower-case options. Without going into great detail, an option to the **date** command is a plus sign (**+**) followed by any number of percent sign - single letter pairs. The format may contain any text; be sure to quote an argument which has embedded spaces. e.g.:

```
$ date "+%a%b... and some message"
```

Which in this case, produced

```
$ date "+%a%b... and some message"
FriJun... and some message
$
```

Since the command was run on a Friday in June. Most letters have some useful effect.

2.4 The first argument

In this exercise, you will write a helper function to remind you what each of the many options return. e.g. *what does "date +%a" return*, and so forth. Your function will display the letter argument, and the "date" result of using it.

Since the **date** command permits a formatting option, you might have done this:

```
$ date +a: %a
a: Sat
$ ...
```

Type that command. What's the purpose? You will see the value of a function argument, you can supply any letter to see its effect. Enter and use this function, by typing at your terminal:

```
$ dateArg () { date "+$s: %$1"; }
$ dateArg F
```

which results in:

```
$ dateArg F
F: 2019-05-19
$ ...
```

In the **dateArg** function, the *$1* takes on the value of the first **positional parameter** passed to the function. e.g. in the next line, the first positional parameter is the letter **F**. The formatted date option displays the argument, and with careful reading of the date manual command[1] for example[2], you see all the options displayed.

The **dateArg** function first displays the option letter and then the date format associated with that letter. Now type some arbitrary options, where some are likely to fail:

```
$ dateArg xxx
$ dateArg one
$ dateArg foo
$ dateArg f
$ dateArg n
$ dateArg x
$ dateArg X
```

What do you see? Can you explain each one?

[1]http://www.bing.com/search?q=unix+date+manual+command
[2]http://unixhelp.ed.ac.uk/CGI/man-cgi?date

2.5 Test them all

Now for the point of the exercise: testing every conceivable valid **date** option. To do that, you will need a new piece of shell syntax, the **for** loop, which looks like this:

for *var* in *list* ; do *command(s) using var …* ; done

This will work:

```
$ for opt in {a..z}; do dateArg $opt; done
```

Type that command now.

What do you see?

Now, some of your explanations come easier. Since the day, the hour or the minutes many be the same number, some letter options may give the same result, and therefore, still be ambiguous.

You've assigned the first positional parameter, and you can likely figure out how to deal with the second, third, … etc. Also, notice, if you haven't done this before, you've just assigned and used a **shell variable**, in this case the variable *opt*. You assign it just by the name, and (in the **for** loop) substitute the value with a leading dollar sign: **$opt**. In this case, **dateArg** is invoked with each of the lower case letters. The function is dressed up a bit to show off the formatting for newlines, tabs, and note that some of the arguments return themselvs.

Here is an example. The command, and it's result:

```
$ for arg in {a..z}; do date_arg $arg; done
$ ...
```

For extra credit: *How do you think you can test the upper-case letters as options?*

2.6 Questions

1. the **date_arg** function demonstrates a useful step in unit testing. How (and where) do you think it should be retained?

2.7 Next Steps

Think about this using the latest example. How would you write a **foreach** function to simplify the whole command to this:

```
$ foreach dateArg {a..z}
```

This last bit of syntax introduces bash Brace Expansion[3]

In the example, it expands to the lower case alphabet. Another examples are quite common.

```
$ set {a,e,i,o,u}; echo $# $*
$ set -- file.{c,o); echo $# $*
```

Mail me if you have questions[4]

<link rel=”stylesheet” type=”text/css” href=”./mcgowan.css” /> <p><link rel=”stylesheet” type=”text/css” href=”./mcgowan.css” /></p>

<h1>Inspect a function body</h1>

<p>In this chapter, we’ll look at specific details of functions, particularly dealing with arguments, and write a function to display functions.</p>

<h2>New Concepts</h2>

<p>In this exercise, you will use these concepts:</p>

<ul> <li>set positional parameters</li> <li>use default parameters</li> <li>use <em>eval</em> to evaluate deferred expression</li> <li>use wild-card parameters</li> <li>use a numeric parameter</li> <li>set and use an alias</li> </ul>

<p>For the time being, think of an <code>alias</code> as a short-hand for a function.</p>

<h2>New Functions</h2>

<p>Recall, in a <a href=”#writeAshellFunction”>previous exercise</a> you learned how to inspect a function body. And you were asked, <em>what would you name a function to display a <strong>F</strong>unction <strong>B</strong>o<strong>DY</strong>.</em></p>

<p>Did I give it away. That’s what I called it: <strong>fbdy</strong>.</p>

<p>To review:</p>

<pre><code>$ declare -f hello # shows the hello function </code></pre>

[3]https://www.gnu.org/software/bash/manual/html_node/Shell-Expansions.html#Shell-Expansions
[4]mailto:martymcgowan@alum.mit.edu?subject=useFunctionArguments

<p>If you've logged off and logged back in since the previous exercise, you've lost the function definition and will have to re-enter it. In a <a href="#collectSaveandReuseFunctions">future exercise</a>, you will learn a simple means to recover your work from day-to-day, from one command session to the next.</p>

<ul> <li>fbdy - display a function</li> </ul>

<h2>More about arguments</h2>

<p>In addition to the numbered <strong>positional parameters</strong>: 1, 2, 3, … you will learn of other parameter features: </p>

<ul> <li>how to express positional parameters</li> <li>report the number of positional parameters, and </li> <li>assign a <strong>default value</strong> to a positional parameter. </li> </ul>

<p>Here's how to <strong>set</strong> positional parameters.</p>

<p><strong>Enter these commands</strong> at your terminal window:</p>

<pre><code> set a b c # set positional parameters to "a b c" echo $* # show them echo $# # how many? echo here is One: $1 eval echo here is the Last: $$# echo here is Two: $2 echo here is Two: ${2:-two} echo here is Four: ${4:-four} alias ea="echo $# $*" ea </code></pre>

<p>In the "here is the Last" example two things happened. In order to inspect the contents of the last positional parameter. you could have typed</p>

<pre><code> $ echo here is the Last: $3 </code></pre>

<p>In general, without knowing the number of positional parameters, you need to find the number of the last parameter. Having found it you can then evaluate it. The shell idiom:</p>

<pre><code> $ eval …. $ … </code></pre>

<p>where the leading backslash defers evaluation of the positional parameter ${ … } until the command has been parsed, the dollar-sign is escaped, or deferred for the subsequent evaluation, triggered by the preceending <em>eval</em> built-in. In this case</p>

<pre><code> $ eval echo … $$# becomes $ echo … $3 </code></pre>

<p>Check your results here:</p>

<pre><code>a b c 3 here is One: a here is the Last: c here is Two: b here is Two: b here is Four: four 3 a b c </code></pre>

<h3>experiments</h3>

<p>Experiment with other arguments to the <strong>set</strong> command. Single-letter options also treat following arguments as <strong>positional parameters</strong>, for example:</p>

<pre><code> $ set -x # turns on command tracing $ … # execute some commands, before you $ set +x # turn it off $ set +x a b c # also sets three parameters </code></pre>

<p>Use the <em>set</em> builtin to set the positional parameters. Experiment with these examples</p>

<pre><code> $ set one Two seven # replaces any previous setting, as does $ set $3 four five # but keep one, add others $ set $* six seven # keep all $ set one $* # in a different order </code></pre>

<p>Future chapters explore the situations where the "–" option flag is required.</p>

<p>In these examples the trailing <em>sharp (#)</em> is the shell comment syntax. And more about shell comments later as well.</p>

<h3>questions</h3>

<p>In the commands you entered at the terminal, what does the: </p>

<ul> <li><em>asterisk ($*) mean?</em></li> <li><em>sharp/hash ($#) mean in this context?</em></li> <li><em>eval command do?</em> hint: omit it from the command line.</li> </ul>

<p>In the next section we will use the positional parameters within a function.</p>

<h2>The function body</h2>

<p>You can take advantage of this information to write a function that returns a function. As you write this function, think of <em>what is a good <strong>default</strong> argument?</em>. Enter these, a line at a time where the <em>echo === N ===</em> are inserted as benchmarks, and may be omitted:</p>

<pre><code>declare -f hello # as before echo === 1 === fbdy () { declare -f $1; } # the simple version fbdy hello # same as before echo === 2 ===
fbdy fbdy # no kidding!, so why not ... echo === 3 ===
fbdy () { declare -f ${1:-fbdy}; } # so ... fbdy # shows how to do it!, and echo === 4 ===
fbdy () { declare -f ${*:-fbdy}; } # permits ... fbdy hello fbdy # whew, let's take a break </code></pre>

<p>Here are the results for the function body:</p>

<pre><code>hello () { echo 'Hello World!' } === 1 === hello () { echo 'Hello World!' } === 2 === fbdy () { declare -f $1 } === 3 === fbdy () { declare -f ${1:-fbdy} } === 4 === hello () { echo 'Hello World!' } fbdy () { declare -f ${*:-fbdy} } </code></pre>

<p>Take note of the use of a default parameter in the highlighted example. After using a single argument, the following example takes advantage of the <em>wild-card ($</em>)*, which expands into all the arguments, or if none, then the default.</p>

<p>You may want to search <em>bash parameter expansion</em></p>

<h3>questions</h3>

<ul> <li><p><em>were you able to follow exercise?</em></p></li> <li><p><em>did you understand each step?</em> if not, <a href="mailto:martymcgowan@alum.mit.edu?subject=function_-body">Contact Me</a></p></li> <li><em>what does the final <strong>fbdy</strong> do with no arguments?</em></li> </ul>

<h2>assessment</h2>

<p>Review, if necessary, your understanding of:</p>

<ul> <li><em>wild-card parameters</em></li> <li><em>default parameters</em></li> <li><em>setting positional parameters</em></li> <li>how the function body is displayed. you might search <em>bash shell declare</em>.</li> </ul>

<p><link rel=”stylesheet” type=”text/css” href=”./mcgowan.css” /> <link rel=”stylesheet” type=”text/css” href=”./mcgowan.css” /></p> <p><link rel=”stylesheet” type=”text/css” href=”./mcgowan.css” /> <link rel=”stylesheet” type=”text/css” href=”./mcgowan.css” /></p>

<h1>Loop with foreach</h1>

<p>In the last chapter you used the <em>for ... do; ... done</em>; syntax to iterate over a list.</p>

<h2>New Concepts</h2>

<ul> <li>avoid using a local variable in a function</li> <li>re-use a command from the command history</li> <li>use expanded positional parameter features</li> </ul>

<h2>New Functions</h2>

<ul> <li>foreach - execute function on args ...</li> </ul>

<h2>The for syntax</h2>

<p>Now you will work with the <strong>for</strong> syntax to produce the <strong>foreach</strong> function that handles many loop requirements.</p>

<p>The shell has other useful <em>looping</em> constructs, namely the <strong>while</strong> loop, which executes while a conditional expression is true. The <strong>for</strong> loop is our focus here. It executes for each of its arguments.</p>

<p>You will use the <strong>for</strong> loop to write a function suggested in your <a href=”#useFunctionArgumen
with function arguments</a>:</p>

<pre><code>$ foreach dateArg {a..z} </code></pre>

<p>Recall the dateArg function:</p>

<p>dateArg () { date “+$1: %$1”; } </p>

<p>So, the reason for the <strong>foreach</strong> function should now be clear: <em>execute the first argument, a function or command “for each” of the remaining arguments</em> The <strong>bash</strong> shell has added the syntactic sugar <em>{a..z}</em> to produce the lower case letters as separate arguments in any command. Long before that feature became available, I used functions named <em>letters, Letters</em>, and <em>LETTERS</em> to produce the lower- and upper-case alphabets.</p>

<p>Recall the earlier example with <strong>dateArg</strong>: </p>

<p><em>for var in list... ; do command(s) using $var ... ; done</em></p>

<p>specifically:</p>

<pre><code>$ for opt in {a..z}; do dateArg $opt; done </code></pre>

<p>If you don’t have your <strong>dateArg</strong> function handy, re-enter it now. Then type the above command to execute it. Notice the generic <em>opt</em> argument could be any relevant name. Also, notice the position of the dateArg function; it is called once per lower-case letter.</p>

<h2>The foreach function</h2>

<p>Enter this text to create your function, and test it:</p>

<pre><code>foreach () { for arg in ${@:2}; do $1 $arg; done; } echo “# foreach dateArg ...” foreach dateArg a e i o u # a purposely shorter list echo “# fbdy foreach dateArg” fbdy foreach dateArg </code></pre>

<p>Here are the <em>foreach dateArg</em> results:</p>

<pre><code># foreach dateArg ... a: Sat e: 18 i: i o: o u: 6 # fbdy foreach dateArg foreach () { for arg in ${@:2}; do $1 $arg; done } dateArg () { date “+$1: %$1” } </code></pre>

<p>The first <strong>foreach</strong> definition treats the first argument as the command and the second and subsequent arguments to that command, which may be a function.</p>

<p>Foreach may handle an indefinite number of arguments, the first is always executed “for each” following argument. It uses the highlighted parameter expansion syntax, in this case <em>${@:2}</em> which selects from the second argument through the remainder. This allows the use of <em>$1</em> in it’s position, saving the use of a local variable name. Another reason for concise functions: you are not juggling too many names or concepts to require local variable names. Find your own comfort level with local variables. And always use the <em>local</em> keyword. Shell variables are <strong>global</strong> unless declared to be local.</p>

<p>Notice the use of <strong>fbdy</strong> it encourages writing concise functions.</p>

<p>The inspiration to write the <strong>foreach</strong> function came from teaching a course in <a href=”https://en.wikipedia.org/wiki/Tcl” title=”Tcl in Wikipedia”>Tcl</a>, which has a similar function. It seems useful to have available in the shell.</p>

<h2>Questions</h2>

<ul> <li><em>did you compare the foreach function to the for command?</em></li> <li><em>did you notice the additional syntax in the function?</em></li> <li><em>did you notice how the function body was displayed?</em></li> <li><em>what does ${@:2} mean?</em></li> <li><em>is it possible to nest calls to foreach</em></li> </ul>

<link rel=”stylesheet” type=”text/css” href=”./mcgowan.css” /> <p><link rel=”stylesheet” type=”text/css” href=”./mcgowan.css” /></p>

<p><link rel=”stylesheet” type=”text/css” href=”./mcgowan.css” /></p>

<h1>A Brief History of Shell Commands</h1>

<p>In the <a href=”#writeAshellFunction”>first chapter</a>, we composed one-line functions on the command line. And promised more information on how to <em>compose simple functions on the command line</em>. This exercise fulfills the promise. You will see how using the command history makes it easy to create a shell function from an often-used command.</p>

<p>A subsidiary objective of this chapter is to postone the use of a text editor on a file. If you are comfortable with any of the text editors, by all means use your favorite. You will still gain something from facility on the command line.</p>

<p>There is a short collection of composing functions on the command line on my <a href="https://www.youtube.co
4038hJH6JTCJKoqunFuQ" title="Marty's YouTube Channel">YouTube channel</a>. It should help to compose and edit functions on the command line.</p>

<p>No new functions are introduced in this chapter.</p>

<h2>New Concepts</h2>

<p>The objective of this chapter is to define a function by using the command history. </p>

<ul> <li>use shell command history</li> <li>learn and apply keyboard commands to find and view bash history</li> <li>discover a useful option of the <code>set</code> built-in</li> <li>identify two editor modes to edit command history</li> <li>use command history numbers to recall a previous command</li> <li>define a function by adding the function syntax, either by editing a previously recovered command, or cutting and pasting a piece of history with a mouse.</li> </ul>

<p>To do this, you will walk thru some command history, navigating forward and backward, finding a line and edit the line in your history to turn it into a function.</p>

<h2>Shell Command History</h2>

<p>Here is an <a href="https://www.youtube.com/watch?v=MbXofShhMv8" title="Joe James on Command History">excellent tutorial</a> on using the shell command history.</p>

<p>The basic command is:</p>

<pre><code>$ history [nn] </code></pre>

<p>where the optional <code>nn</code> shows the last nn lines of history. Here's an example:</p>

<pre><code>$ history 24 1857 cd Mobile\ Documents/ 1858 cd com~apple~CloudDocs/ 1859 cd .Trash/ 1860 llrt 1861 rm -f MyIdea* 1862 commonplace 1863 dirs 1864 ftp run 1865 daily_open 1866 ftp user 1867 . ftp_user 1868 ftp run 1869 manuscript 1870 manuscript 1871 shellfunctions 1872 ta shellfunctions 1873 alias | grep -i dropbox 1874 ft 1875 miw 1876 backup MyIdeaWarehouse.html 1877 backup MyIdeaWarehouse.html 1878 ftp run 1879 history 1880 provenance history 24 $ … </code></pre>

<p>Note the numbered entries. By this time, I'd entered 1880 commands in this terminal session. Any command may be re-executed by selecting it's number:</p>

<pre><code>$!1870 </code></pre>

<p>will re-execute the <code>manuscript</code> command, which may have itself been executed by using the re-execute the prior command:</p>

<pre><code>$!! # which is a short-hand for $!-1 # to execute the N-th (1) prior command </code></pre>

<p>The above tutorial introduces a comprehensive treatment of command-history editing. </p>

<p>In addition to the keyboard editing approach above, there are two other means of command-line editing, <em>vi</em> and <em>emacs</em> editor-based commands. These use the respective text editors for shell history, you navigate with the editor's line and inter-line movement mechanisms.</p>

<p>You might reasonably ask at this point, "So, what's with this editor business? I thought we wouldn't have to learn an editor." Be assured, it's easier to navigate thru command history if you can handle the few cursor movement commands the editor offers. Then editing a single line is simply moving the cursor back and forth to insert or delete characters.</p>

<p>Since the keyboard technique is pervasive, you will likely not need to explore the editor history modes. If you are running a shell in an <code>emacs</code> window is one reason to use an editor command mode. In emacs, the editor keyboard functions will conflict with command-history keyboard functions.</p>

<h2>Editor Modes</h2>

<p>This section introduces the text-editor capabilities. If you feel comfortable with the keyboard editing described above, you may skip to the next section. To see all the available shell options use a <strong>set -o</strong> command. Try that at your terminal window. Execute this command to see which your current editor mode is set at:</p>

<pre><code>$ set -o # shows all the options and their state $ set -o | grep -E '^(vi|emacs)' # collect the edit modes emacs on vi off $ </code></pre>

<p>This shows I am using the <em>emacs</em> mode, which I prefer to <em>vi</em> mode or keyboard navigation. The editor modes toggle: if one is on, the other is off. To set your preference, you might:</p>

<pre><code>$ set -o vi </code></pre>

<p>to switch to the <em>vi</em> mode. You may want to consult <a href="http://www.catonmat.net/download/bas
history-cheat-sheet.txt" title="Cheat sheet">this cheat sheet of editor modes</a>. If this is your first encounter with command history, after trying the keyboard editing above, I recommend vi mode. Vi is a moded editor. After switching to vi mode, as above, and inspecting the cheat sheet, the vi shortcuts are entered by preceding the first command with the <em>escape</em> "[esc]" on my keyboard, followed by the letter from the table. Therefore to go back in history <em>[esc] k</em>. The escape key toggles navigation while in vi mode. To continue back, continue to strike the <em>k</em> key; to go forward, the <em>j</em>. To stop navigation, the <em>[esc]</em> key.</p>

<h2>Turn a Command into a Shell Function</h2>

<p>Recall your <a href="#useFunctionArguments">experiment with the dateArg function</a>. Your first attempt was a simple date command:</p>

<pre><code>$ date +%F </code></pre>

<p>Here's a piece of my recent shell history which shows how the command may be turned into a function:</p>

<pre><code>$ history | awk '$1 > 616'

617 date +%F 618 date "+F: %F" 619 set F 620 date "+$1: %$1" 621 set c 622 date "+$1: %$1" 623 history | awk '$2 ~ /date/' 624 history | awk '$1 > 590' 625 history | awk '$1 > 616' $ </code></pre>

<p>How did I discover the line number to begin?</p>

<p>I'm looking for the date command, so the first attempt (my # 623) was to identify all the uses of <em>date</em>. I thought I'd start at the reported # 590, that proved to be more than I needed, so I settled on #614, which was the latest bit of practice</p>

<p>Type the first three commands (617, 618, 619) at your terminal window now. And do <strong>not</strong> type the line numbers. Line number 619 sets the first <strong>positional parameter</strong> to the letter <em>F</em>, just as it would be when passed to a function. You might try</p>

<pre><code>$ echo $1 </code></pre>

<p>or, if you still have the <em>ea</em> alias()[]:</p>

<pre><code>$ ea </code></pre>

<p>to verify that.</p>

<p>Then, on line 620, instead of typing the whole command, you may use the history to recover your previous typing. In this case, the command is quite brief; you could have typed it again from scratch. If you need practice in your history, instead of typing 620 literally, do this:</p>

<ul> <li><p>go back two commands, on my terminal, and maybe yours, the up-arrow also works. In emacs mode, hit <strong>[ctrl]P</strong> to go back one, in vi mode, hit <strong>[esc]</strong> to enter command mode, where <strong>K</strong> goes back in history and <strong>J</strong> goes forward.</p></li> <li><p>use line-editing commands, again the <strong>delete</strong> or backspace keys may work. If you need more help, consult the cheat sheet above.</p></li> <li><p>change the literal <em>F</em>. to the shell variable <em>$1</em> in both instances, and then either <em>Enter</em> or <em>Return</em> to execute the command.</p></li> </ul>

<p>On line 621, change the first positional parameter to the lower-case <em>c</em>, and re-execute the command by stepping back two lines in history to line 807.</p>

<p>You can repeat that for as many letter options as you'd like. Also, if you want to do any "out-of-bounds" testing, this is a good time to try.</p>

<h2>The simple step – the function</h2>

<p>With command history, it is now quite simple to turn your recent efforts into a function. All that is necessary is to wrap the latest command instance with the function syntax, the most important part of which is a proper name. As it's possible to recall a history command by number, here is how you might proceed:</p>

<pre><code>$!622 date "+$1: %$1" c: Mon Jul 1 09:58:20 2013 $ dateArg () { date "+$1: %$1"; } $ fbdy dateArg dateArg () { date "+$1: %$1" } $ </code></pre>

<p>Where my history line number was 622. Inspect your history and recover your command with the exclamation using the line-number or your command. Now, rather than type the whole function

definition <em>dateArg () { ... }</em>, simply retrieve the just-executed command by going back one line in the history (to the <em>date ...</em> command), but not re-execute it.</p>

<ul> <li><p>Move to the beginning of the just-retrieved line in your history buffer (i.e., do not hit the Enter key),</p></li> <li><p>then enter the function name, parenthesis, and opening brace, <em>dateArg () {</em> In front of the function text,</p></li> <li><p>and finally, go to the end of the line, and close with the semi-colon and closing brace, <em>; }</em>.</p></li> </ul>

<h2>Terminal Cut and Paste</h2>

<p>Should using editor modes (vi or emacs) be a problem, your terminal should also offer cut and paste from past history on the command line.</p>

<p>If that's your preferred mode, then</p>

<ul> <li><p>start the command by entering the leading text:</p>

<p>$ dateArg () {</p></li> <li><p>then follow by cutting and pasting the command:</p>

<p>$ date "+$1: %$1"</p></li> <li><p>which will appear: </p>

<p>$ dateArg () { date "+$1: %$1"</p></li> <li><p>and close by entering the closing syntax:</p>

<p>$; }</p></li> <li><p>yielding:</p>

<p>$ dateArg () { date "+$1: %$1"; }</p></li> </ul>

<p>and you have your function. I recommend the editor modes. The time investment will be repaid.</p>

<h2>Review</h2>

<ul> <li><p>Assessment</p>

<ul> <li><p>do you have a preferred editor mode, or do you prefer keyboard navigation?</p></li> <li><p>did you experiment with several options to the <strong>date</strong> command?</p></li> <li><p>did you <a href="#inspectAfunctionBody">refresh your experiment</a> with the <em>set −</em> command?</p></li> </ul></li> <li><p>Questions</p>

<ul> <li><p>do you understand how the formatted <strong>date</strong> options behave?</p></li> <li><p>... how the <em>set −</em> command works?</p></li> </ul></li> <li><p><em>Did you understand the <strong>history</strong> command?</em>. The pipe to <strong>awk</strong> is a little bonus?</p></li> <li><p><em>Did you succeed in creating the <strong>dateArg</strong> function?</em></p></li> <li><p><em>If not, can you assess where you got hung up?</em></p></li> <li><p><em>Are you able to recall all the functions you have written?</em> You will learn this shortly.</p></li> </ul>

<p>Here's a sneak peak at one of the next facets:</p>

<pre><code>$ gh () { history | grep -i ${*:-()}; } </code></pre>

<p>What might you use it for?</p>

<p>You may <a href="mailto:martymcgowan@alum.mit.edu?subject=aBriefHistoryOfShell">Contact Me</a></p>

<link rel="stylesheet" type="text/css" href="./mcgowan.css" /> <p><link rel="stylesheet" type="text/css" href="./mcgowan.css" /></p>

<p><link rel="stylesheet" type="text/css" href="./mcgowan.css" /></p>

<h1>Shdoc – SHell DOC comments</h1>

<p>A few of the scripting and system programming languages have comment conventions for their functions. <a href="http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html" title="Oracle JavaDoc main page">JavaDoc</a> was possibly the earliest example. Others instances are <a href="https://en.wikipedia.org/wiki/Pydoc" title="Wikipedia Pydoc">pydoc</a>, <a href="http://perldoc.perl.org">perldoc</a>, <a href="https://en.wikipedia.org/wiki/YARD*soft*
<a href="https://en.wikipedia.org/wiki/RDoc" title="RubyDoc Wikipedia page">RDOC</a>, <a href="https://en.wiki
title="mkd Wikipedia page">mkd</a> , ... and, in general, [asample][comparsion].</p>

<p>As yet, there is no similar feature for the shell. Let's begin.</p>

<h2>New Concepts</h2>

<p>This exercise sets a direction in documenting shell function behavior.</p>

<ul> <li>document a function, lessons learned from Java, Perl, PYthon ...</li> <li>shell array FUNCNAME</li> <li>which function called this function</li> <li><strong>awk</strong>, simple usage</li> </ul>

<h2>New Functions</h2>

<ul> <li>myname - what is the name of my calling function</li> <li>report_notfunction - report if not called by it's argument</li> <li>report_notpipe - report if standard input is not on a pipe</li> <li>report_usage - used by reporting functions</li> <li>shdoc - _shdoc over a list of functions</li> <li>_shdoc – supplies output function format for colon-comments</li> <li>shd_justcolon – returns leading colon comments</li> </ul>

<h2>the convention</h2>

<p>Since the <code>fbdy</code> function, using the <em>declare</em> built-in, supplies the function body in a standard format, it's safe to use that as the conventional layout of a function. With the standard layout, let's adopt the convention as these others do, that <em>the first comment lines after the function definition are special, and regarded as the function API interface</em>.</p>

<h2>the first functions</h2>

<p>The main function is <code>shdoc</code>, using a few local utilities, and others which will be described below.</p>

<pre><code>_shdoc () { : date: 2018-02-16; report_notfunction $1 && return 1; echo "function ${1}_doc {"; declare -f $1 | shd_justcolon; echo "}" } shdoc () { : this is a shell doclib "shdoc" comment; : an shdoc comment is the first ":"-origin lines; : in the shell function, the rest being the executable.; :

2019-01-28 use local _shdoc and produce declare -f format; foreach _shdoc ${*:-$(myname)} } shd_-justcolon () { : returns leading colon-comments from a SINGLE function; report_notpipe && return 1; awk ‘ NR > 2 { if ($1 !∼ /^:/) exit else print } ‘ } myname () { : ∼ [n]; : returns name of caller OR callers caller ...; : date: 2018-02-16; echo ${FUNCNAME[${1:-1}]} } </code></pre>

<p>So, <code>shdoc</code> takes an indefinite number of arguments, defaulting to itself, <code>myname</code>, calling <code>shd_each</code> for each of them. While it’s tempting to use the wild-card properties of the <em>declare</em> built-in, this keeps the <em>awk</em> code simpler.</p>

<p>The <code>report_notfunction</code> is part of the <strong>report</strong> family of functions. These functions are built to report on <em>assertion failures</em>.</p>

<p>And, lastly, the <code>myname</code> function, using the <em>bash</em> built-in array, FUNCNAME, returns the name of the calling function N levels up the stack. It defaults to 1, returning the name of the immediate caller.</p>

<p>Here is the result of running shdoc on the same functions.</p>

<pre><code>$ shdoc ...

function _shdoc_doc { : date: 2018-02-16; } function shdoc_doc { : this is a shell doclib “shdoc” comment; : an shdoc comment is the first “:”-origin lines; : in the shell function, the rest being the executable.; : 2019-01-28 use local _shdoc and produce declare -f format; } function shd_justcolon_doc { : returns leading colon-comments from a SINGLE function; } function myname_doc { : ∼ [n]; : returns name of caller OR callers caller ...; : date: 2018-02-16; } </code></pre>

<p>Note, <code>shdoc</code> produces its comment in a function body by the same name with the <em>_doc</em> ending.</p>

<h2>utility functions</h2>

<p>Here are the <strong>report<em></strong> functions, and a simple <strong>trace</em>call</strong>, and the rest of their sub-functions.</p>

<pre><code>report_notfunction () { : returns: TRUE when 1st arg “ISN’T” a function,; : ... FALSE if it IS a function; : date: 2018-02-16; declare -f $1 > /dev/null && return 1; report_usage $1 “ISN’T” a function } report_notpipe () { : returns: TRUE when STDIN “ISN’T” a pipe; : ... FALSE if stdin is a pipe; : date: 2018-02-16; [[-p /dev/stdin]] && return 1; report_usage “ISN’T” reading a pipe } report_usage () { : writes: usage message of report_... caller FAILURE to STDERR; : date: 2018-03-30; echo USAGE $(myname 3): $* 1>&2 } </code></pre>

<p>Using <strong>isfunction</strong> is debatable. My preference is to use meaningful function names, even where the built-in or command is so simple.</p>

<p>And the result of <code>shdoc</code> on those functions:</p>

<pre><code>function report_notfunction_doc { : returns: TRUE when 1st arg “ISN’T” a function,; : ... FALSE if it IS a function; : date: 2018-02-16; } function report_notpipe_doc { : returns: TRUE when STDIN “ISN’T” a pipe; : ... FALSE if stdin is a pipe; : date: 2018-02-16; } function report_usage_doc { : writes: usage message of report_... caller FAILURE to STDERR; : date: 2018-03-30; } </code></pre>

<h2>awk digression</h2>

<p>This is our first encounter with the <strong>awk</strong> programming language. It’s too useful in shell environments to not have brief mention in an application such as this.</p>

<p><strong>Do this:</strong> Read the <a href=”https://en.wikipedia.org/wiki/AWK” title=”Awk programming language”>main article</a>.</p>

<p>For our purpose, an awk script is made up of <em>pattern { action }</em> pairs.</p>

<p>In its one use here, in the function <code>_shdoc</code>, the built-in <em>declare -f</em> puts the function body on the standard output to <code>shd_justcolon</code> using <a href=”https://en.wikipedia.org/wi
title=”Awk programming language”>awk</a>, with its only pattern <code>NR > 2</code>, works on every line after the second, printing those lines which begin a colon (:). The script exits when the first non-leading colon is encountered.</p>

<p>The <code>_shdoc</code> function adds the leading and trailing context. It may all have been done in awk.</p>

<p>The purpose of this exercise is to show the complementary nature of awk to the shell. It is much easier to read and understand the awk code than the equivalent shell expressions for the task. There’s no clear standard, but you will find a line between data-processing using <strong>awk</strong> and manipulating system objects, files, directories, and their relationships using the <strong>shell</strong> .</p>

<h2>Activity</h2>

<ol> <li><p>write a shell function, using awk to print function names from function definitions on the either the standard input or from named files. (hint: use the <code>cat ${*:–}</code> idiom and think, “what syntax identifies a function name?” in the text)</p></li> <li><p>add sufficient <strong>shdoc</strong> to the function. does it need much more than the requirements in the activity request?</p></li> </ol>

<h2>References</h2>

<ol> <li><a href=”https://en.wikipedia.org/wiki/AWK” title=”Awk programming language”>awk</a> – pattern matching, execution language</li> <li><a href=”https://en.wikipedia.org/wiki/Comparison_-of_documentation_generators”>comparison</a> – comparison of documentation generators</li> <li><a href=”http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html” title=”Oracle JavaDoc main page”>javadoc</a> – how to document java function definitions</li> <li><a href=”https://en.wikipedia.org/wiki/Mkd*(software)” title=”mkd Wikipedia page”>mkd</a> </li> <li><a href=”http://perldoc.perl.org”>perldoc</a> – how do document perl functions</li> <li><a href=”https://en.wikipedia.org/wiki/Pydoc” title=”Wikipedia Pydoc”>pydoc</a> – how do document python functions</li> <li><a href=”https://en.wikipedia.org/wiki/RDoc” title=”RubyDoc Wikipedia page”>RDOC</a></li> <li><a href=”https://en.wikipedia.org/wiki/YARD*(software)”>YARD</a> –</li> </ol>

<link rel=”stylesheet” type=”text/css” href=”./mcgowan.css” /> <p><link rel=”stylesheet” type=”text/css” href=”./mcgowan.css” /></p>

<h1>A Brief History of Shell Commands</h1>

<p>In the <a href="#writeAshellFunction">first chapter</a>, we composed one-line functions on the command line. And promised more information on how to <em>compose simple functions on the command line</em>. This exercise fulfills the promise. You will see how using the command history makes it easy to create a shell function from an often-used command.</p>

<p>A subsidiary objective of this chapter is to postone the use of a text editor on a file. If you are comfortable with any of the text editors, by all means use your favorite. You will still gain something from facility on the command line.</p>

<p>There is a short collection of composing functions on the command line on my <a href="https://www.youtube.co
4038hJH6JTCJKoqunFuQ" title="Marty's YouTube Channel">YouTube channel</a>. It should help to compose and edit functions on the command line.</p>

<p>No new functions are introduced in this chapter.</p>

<h2>New Concepts</h2>

<p>The objective of this chapter is to define a function by using the command history. </p>

<ul> <li>use shell command history</li> <li>learn and apply keyboard commands to find and view bash history</li> <li>discover a useful option of the <code>set</code> built-in</li> <li>identify two editor modes to edit command history</li> <li>use command history numbers to recall a previous command</li> <li>define a function by adding the function syntax, either by editing a previously recovered command, or cutting and pasting a piece of history with a mouse.</li> </ul>

<p>To do this, you will walk thru some command history, navigating forward and backward, finding a line and edit the line in your history to turn it into a function.</p>

<h2>Shell Command History</h2>

<p>Here is an <a href="https://www.youtube.com/watch?v=MbXofShhMv8" title="Joe James on Command History">excellent tutorial</a> on using the shell command history.</p>

<p>The basic command is:</p>

<pre><code>$ history [nn] </code></pre>

<p>where the optional <code>nn</code> shows the last nn lines of history. Here's an example:</p>

<pre><code>$ history 24 1857 cd Mobile\ Documents/ 1858 cd com~apple~CloudDocs/ 1859 cd .Trash/ 1860 llrt 1861 rm -f MyIdea* 1862 commonplace 1863 dirs 1864 ftp run 1865 daily_open 1866 ftp user 1867 . ftp_user 1868 ftp run 1869 manuscript 1870 manuscript 1871 shellfunctions 1872 ta shellfunctions 1873 alias | grep -i dropbox 1874 ft 1875 miw 1876 backup MyIdeaWarehouse.html 1877 backup MyIdeaWarehouse.html 1878 ftp run 1879 history 1880 provenance history 24 $ … </code></pre>

<p>Note the numbered entries. By this time, I'd entered 1880 commands in this terminal session. Any command may be re-executed by selecting it's number:</p>

<pre><code>$!1870 </code></pre>

<p>will re-execute the <code>manuscript</code> command, which may have itself been executed by using the re-execute the prior command:</p>

<pre><code>$!! # which is a short-hand for $!-1 # to execute the N-th (1) prior command </code></pre>

<p>The above tutorial introduces a comprehensive treatment of command-history editing. </p>

<p>In addition to the keyboard editing approach above, there are two other means of command-line editing, <em>vi</em> and <em>emacs</em> editor-based commands. These use the respective text editors for shell history, you navigate with the editor's line and inter-line movement mechanisms.</p>

<p>You might reasonably ask at this point, "So, what's with this editor business? I thought we wouldn't have to learn an editor." Be assured, it's easier to navigate thru command history if you can handle the few cursor movement commands the editor offers. Then editing a single line is simply moving the cursor back and forth to insert or delete characters.</p>

<p>Since the keyboard technique is pervasive, you will likely not need to explore the editor history modes. If you are running a shell in an <code>emacs</code> window is one reason to use an editor command mode. In emacs, the editor keyboard functions will conflict with command-history keyboard functions.</p>

<h2>Editor Modes</h2>

<p>This section introduces the text-editor capabilities. If you feel comfortable with the keyboard editing described above, you may skip to the next section. To see all the available shell options use a <strong>set -o</strong> command. Try that at your terminal window. Execute this command to see which your current editor mode is set at:</p>

<pre><code>$ set -o # shows all the options and their state $ set -o | grep -E '^(vi|emacs)' # collect the edit modes emacs on vi off $ </code></pre>

<p>This shows I am using the <em>emacs</em> mode, which I prefer to <em>vi</em> mode or keyboard navigation. The editor modes toggle: if one is on, the other is off. To set your preference, you might:</p>

<pre><code>$ set -o vi </code></pre>

<p>to switch to the <em>vi</em> mode. You may want to consult <a href="http://www.catonmat.net/download/bas
history-cheat-sheet.txt" title="Cheat sheet">this cheat sheet of editor modes</a>. If this is your first encounter with command history, after trying the keyboard editing above, I recommend vi mode. Vi is a moded editor. After switching to vi mode, as above, and inspecting the cheat sheet, the vi shortcuts are entered by preceding the first command with the <em>escape</em> "[esc]" on my keyboard, followed by the letter from the table. Therefore to go back in history <em>[esc] k</em>. The escape key toggles navigation while in vi mode. To continue back, continue to strike the <em>k</em> key; to go forward, the <em>j</em>. To stop navigation, the <em>[esc]</em> key.</p>

<h2>Turn a Command into a Shell Function</h2>

<p>Recall your <a href="#useFunctionArguments">experiment with the dateArg function</a>. Your first attempt was a simple date command:</p>

<pre><code>$ date +%F </code></pre>

<p>Here's a piece of my recent shell history which shows how the command may be turned into a function:</p>

<pre><code>$ history | awk '$1 > 616'

617 date +%F 618 date "+F: %F" 619 set F 620 date "+$1: %$1" 621 set c 622 date "+$1: %$1" 623 history | awk '$2 ~ /date/' 624 history | awk '$1 > 590' 625 history | awk '$1 > 616' $ </code></pre>

<p>How did I discover the line number to begin?</p>

<p>I'm looking for the date command, so the first attempt (my # 623) was to identify all the uses of <em>date</em>. I thought I'd start at the reported # 590, that proved to be more than I needed, so I settled on #614, which was the latest bit of practice</p>

<p>Type the first three commands (617, 618, 619) at your terminal window now. And do <strong>not</strong> type the line numbers. Line number 619 sets the first <strong>positional parameter</strong> to the letter <em>F</em>, just as it would be when passed to a function. You might try</p>

<pre><code>$ echo $1 </code></pre>

<p>or, if you still have the <em>ea</em> alias()[]:</p>

<pre><code>$ ea </code></pre>

<p>to verify that.</p>

<p>Then, on line 620, instead of typing the whole command, you may use the history to recover your previous typing. In this case, the command is quite brief; you could have typed it again from scratch. If you need practice in your history, instead of typing 620 literally, do this:</p>

<ul> <li><p>go back two commands, on my terminal, and maybe yours, the up-arrow also works. In emacs mode, hit <strong>[ctrl]P</strong> to go back one, in vi mode, hit <strong>[esc]</strong> to enter command mode, where <strong>K</strong> goes back in history and <strong>J</strong> goes forward.</p></li> <li><p>use line-editing commands, again the <strong>delete</strong> or backspace keys may work. If you need more help, consult the cheat sheet above.</p></li> <li><p>change the literal <em>F</em>. to the shell variable <em>$1</em> in both instances, and then either <em>Enter</em> or <em>Return</em> to execute the command.</p></li> </ul>

<p>On line 621, change the first positional parameter to the lower-case <em>c</em>, and re-execute the command by stepping back two lines in history to line 807.</p>

<p>You can repeat that for as many letter options as you'd like. Also, if you want to do any "out-of-bounds" testing, this is a good time to try.</p>

<h2>The simple step – the function</h2>

<p>With command history, it is now quite simple to turn your recent efforts into a function. All that is necessary is to wrap the latest command instance with the function syntax, the most important

part of which is a proper name. As it’s possible to recall a history command by number, here is how you might proceed:</p>

<pre><code>$!622 date “+$1: %$1” c: Mon Jul 1 09:58:20 2013 $ dateArg () { date “+$1: %$1”; } $ fbdy dateArg dateArg () { date “+$1: %$1” } $ </code></pre>

<p>Where my history line number was 622. Inspect your history and recover your command with the exclamation using the line-number or your command. Now, rather than type the whole function definition <em>dateArg () { … }</em>, simply retrieve the just-executed command by going back one line in the history (to the <em>date …</em> command), but not re-execute it.</p>

<ul> <li><p>Move to the beginning of the just-retrieved line in your history buffer (i.e., do not hit the Enter key),</p></li> <li><p>then enter the function name, parenthesis, and opening brace, <em>dateArg () {</em> In front of the function text,</p></li> <li><p>and finally, go to the end of the line, and close with the semi-colon and closing brace, <em>; }</em>.</p></li> </ul>

<h2>Terminal Cut and Paste</h2>

<p>Should using editor modes (vi or emacs) be a problem, your terminal should also offer cut and paste from past history on the command line.</p>

<p>If that’s your preferred mode, then</p>

<ul> <li><p>start the command by entering the leading text:</p>

<p>$ dateArg () {</p></li> <li><p>then follow by cutting and pasting the command:</p>

<p>$ date “+$1: %$1”</p></li> <li><p>which will appear: </p>

<p>$ dateArg () { date “+$1: %$1”</p></li> <li><p>and close by entering the closing syntax:</p>

<p>$; }</p></li> <li><p>yielding:</p>

<p>$ dateArg () { date “+$1: %$1”; }</p></li> </ul>

<p>and you have your function. I recommend the editor modes. The time investment will be repaid.</p>

<h2>Review</h2>

<ul> <li><p>Assessment</p>

<ul> <li><p>do you have a preferred editor mode, or do you prefer keyboard navigation?</p></li> <li><p>did you experiment with several options to the <strong>date</strong> command?</p></li> <li><p>did you <a href=”#inspectAfunctionBody”>refresh your experiment</a> with the <em>set –</em> command?</p></li> </ul></li> <li><p>Questions</p>

<ul> <li><p>do you understand how the formatted <strong>date</strong> options behave?</p></li> <li><p>… how the <em>set –</em> command works?</p></li> </ul></li> <li><p><em>Did you understand the <strong>history</strong> command?</em>. The pipe to <strong>awk</strong> is a little bonus?</p></li> <li><p><em>Did you succeed in creating the <strong>dateArg</strong> function?</em></p></li> <li><p><em>If not, can you assess where you got hung up?</em></p></li>

<li><p><em>Are you able to recall all the functions you have written?</em> You will learn this shortly.</p></li> </ul>

<p>Here's a sneak peak at one of the next facets:</p>

<pre><code>$ gh () { history | grep -i ${*:-()}; } </code></pre>

<p>What might you use it for?</p>

<p>You may <a href="mailto:martymcgowan@alum.mit.edu?subject=aBriefHistoryOfShell">Contact Me</a></p>

<link rel="stylesheet" type="text/css" href="./mcgowan.css" /> <p><link rel="stylesheet" type="text/css" href="./mcgowan.css" /></p>

<p><link rel="stylesheet" type="text/css" href="./mcgowan.css" /></p>

<p><link rel="stylesheet" type="text/css" href="./mcgowan.css" /></p>

<h1>Shdoc – SHell DOC comments</h1>

<p>A few of the scripting and system programming languages have comment conventions for their functions. <a href="http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html" title="Oracle JavaDoc main page">JavaDoc</a> was possibly the earliest example. Others instances are <a href="https://en.wikipedia.org/wiki/Pydoc" title="Wikipedia Pydoc">pydoc</a>, <a href="http://perldoc.perl.org">perldoc</a>, <a href="https://en.wikipedia.org/wiki/YARD*(softv*
<a href="https://en.wikipedia.org/wiki/RDoc" title="RubyDoc Wikipedia page">RDOC</a>, <a href="https://en.wikip
title="mkd Wikipedia page">mkd</a> , … and, in general, [asample][comparsion].</p>

<p>As yet, there is no similar feature for the shell. Let's begin.</p>

<h2>New Concepts</h2>

<p>This exercise sets a direction in documenting shell function behavior.</p>

<ul> <li>document a function, lessons learned from Java, Perl, PYthon …</li> <li>shell array FUNCNAME</li> <li>which function called this function</li> <li><strong>awk</strong>, simple usage</li> </ul>

<h2>New Functions</h2>

<ul> <li>myname - what is the name of my calling function</li> <li>report_notfunction - report if not called by it's argument</li> <li>report_notpipe - report if standard input is not on a pipe</li> <li>report_usage - used by reporting functions</li> <li>shdoc - _shdoc over a list of functions</li> <li>_shdoc – supplies output function format for colon-comments</li> <li>shd_justcolon – returns leading colon comments</li> </ul>

<h2>the convention</h2>

<p>Since the <code>fbdy</code> function, using the <em>declare</em> built-in, supplies the function body in a standard format, it's safe to use that as the conventional layout of a function. With the standard layout, let's adopt the convention as these others do, that <em>the first comment lines after the function definition are special, and regarded as the function API interface</em>.</p>

<h2>the first functions</h2>

<p>The main function is <code>shdoc</code>, using a few local utilities, and others which will be described below.</p>

<pre><code>_shdoc () { : date: 2018-02-16; report_notfunction $1 && return 1; echo "function ${1}_-doc {"; declare -f $1 | shd_justcolon; echo "}" } shdoc () { : this is a shell doclib "shdoc" comment; : an shdoc comment is the first ":"-origin lines; : in the shell function, the rest being the executable.; : 2019-01-28 use local _shdoc and produce declare -f format; foreach _shdoc ${*:-$(myname)} } shd_-justcolon () { : returns leading colon-comments from a SINGLE function; report_notpipe && return 1; awk ' NR > 2 { if ($1 !~ /^:/) exit else print } ' } myname () { : ~ [n]; : returns name of caller OR callers caller …; : date: 2018-02-16; echo ${FUNCNAME[${1:-1}]} } </code></pre>

<p>So, <code>shdoc</code> takes an indefinite number of arguments, defaulting to itself, <code>myname</code>, calling <code>shd_each</code> for each of them. While it's tempting to use the wild-card properties of the <em>declare</em> built-in, this keeps the <em>awk</em> code simpler.</p>

<p>The <code>report_notfunction</code> is part of the <strong>report</strong> family of functions. These functions are built to report on <em>assertion failures</em>.</p>

<p>And, lastly, the <code>myname</code> function, using the <em>bash</em> built-in array, FUNCNAME, returns the name of the calling function N levels up the stack. It defaults to 1, returning the name of the immediate caller.</p>

<p>Here is the result of running shdoc on the same functions.</p>

<pre><code>$ shdoc …

function _shdoc_doc { : date: 2018-02-16; } function shdoc_doc { : this is a shell doclib "shdoc" comment; : an shdoc comment is the first ":"-origin lines; : in the shell function, the rest being the executable.; : 2019-01-28 use local _shdoc and produce declare -f format; } function shd_just-colon_doc { : returns leading colon-comments from a SINGLE function; } function myname_doc { : ~ [n]; : returns name of caller OR callers caller …; : date: 2018-02-16; } </code></pre>

<p>Note, <code>shdoc</code> produces its comment in a function body by the same name with the <em>_doc</em> ending.</p>

<h2>utility functions</h2>

<p>Here are the <strong>report<em></strong> functions, and a simple <strong>trace</em>call</strong>, and the rest of their sub-functions.</p>

<pre><code>report_notfunction () { : returns: TRUE when 1st arg "ISN'T" a function,; : … FALSE if it IS a function; : date: 2018-02-16; declare -f $1 > /dev/null && return 1; report_usage $1 "ISN'T" a function } report_notpipe () { : returns: TRUE when STDIN "ISN'T" a pipe; : … FALSE if stdin is a pipe; : date: 2018-02-16; [[-p /dev/stdin]] && return 1; report_usage "ISN'T" reading a pipe } report_usage () { : writes: usage message of report_… caller FAILURE to STDERR; : date: 2018-03-30; echo USAGE $(myname 3): $* 1>&2 } </code></pre>

<p>Using <strong>isfunction</strong> is debatable. My preference is to use meaningful function names, even where the built-in or command is so simple.</p>

<p>And the result of <code>shdoc</code> on those functions:</p>

<pre><code>function report_notfunction_doc { : returns: TRUE when 1st arg “ISN’T” a function,; : … FALSE if it IS a function; : date: 2018-02-16; } function report_notpipe_doc { : returns: TRUE when STDIN “ISN’T” a pipe; : … FALSE if stdin is a pipe; : date: 2018-02-16; } function report_usage_doc { : writes: usage message of report_… caller FAILURE to STDERR; : date: 2018-03-30; } </code></pre>

<h2>awk digression</h2>

<p>This is our first encounter with the <strong>awk</strong> programming language. It’s too useful in shell environments to not have brief mention in an application such as this.</p>

<p><strong>Do this:</strong> Read the <a href=”https://en.wikipedia.org/wiki/AWK” title=”Awk programming language”>main article</a>.</p>

<p>For our purpose, an awk script is made up of <em>pattern { action }</em> pairs.</p>

<p>In its one use here, in the function <code>_shdoc</code>, the built-in <em>declare -f</em> puts the function body on the standard output to <code>shd_justcolon</code> using <a href=”https://en.wikipedia.org/wi
title=”Awk programming language”>awk</a>, with its only pattern <code>NR > 2</code>, works on every line after the second, printing those lines which begin a colon (:). The script exits when the first non-leading colon is encountered.</p>

<p>The <code>_shdoc</code> function adds the leading and trailing context. It may all have been done in awk.</p>

<p>The purpose of this exercise is to show the complementary nature of awk to the shell. It is much easier to read and understand the awk code than the equivalent shell expressions for the task. There’s no clear standard, but you will find a line between data-processing using <strong>awk</strong> and manipulating system objects, files, directories, and their relationships using the <strong>shell</strong> .</p>

<h2>Activity</h2>

<ol> <li><p>write a shell function, using awk to print function names from function definitions on the either the standard input or from named files. (hint: use the <code>cat ${*:–}</code> idiom and think, “what syntax identifies a function name?” in the text)</p></li> <li><p>add sufficient <strong>shdoc</strong> to the function. does it need much more than the requirements in the activity request?</p></li> </ol>

<h2>References</h2>

<ol> <li><a href=”https://en.wikipedia.org/wiki/AWK” title=”Awk programming language”>awk</a> – pattern matching, execution language</li> <li><a href=”https://en.wikipedia.org/wiki/Comparison_-of_documentation_generators”>comparison</a> – comparison of documentation generators</li> <li><a href=”http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html” title=”Oracle JavaDoc main page”>javadoc</a> – how to document java function definitions</li> <li><a href=”https://en.wikipedia.org/wiki/Mkd*(software)” title=”mkd Wikipedia page”>mkd</a> </li> <li><a href=”http://perldoc.perl.org”>perldoc</a> – how do document perl functions</li> <li><a href=”https://en.wikipedia.org/wiki/Pydoc” title=”Wikipedia Pydoc”>pydoc</a> – how do document*

python functions</li> <li><a href="https://en.wikipedia.org/wiki/RDoc" title="RubyDoc Wikipedia page">RDOC</a></li> <li><a href="https://en.wikipedia.org/wiki/YARD(software)">YARD</a> –</li> </ol>

<link rel="stylesheet" type="text/css" href="./mcgowan.css" /> <p><link rel="stylesheet" type="text/css" href="./mcgowan.css" /></p>

<h1>Collect, save, and re-use functions</h1>

<p>All that remains for your newly-gained facility with shell functions is to collect a group of them, save them in a useful way, and make them available for reuse: to make them part of your working vocabulary.</p>

<p>To collect the functions, and save in a useful way means when you return to a later terminal session, you don't have to re-enter them, but think of then as commands, as if they were separate shell scripts.</p>

<p>In this exercise, you'll collect, save, and make the functions you've created in this book available for routine re-use.</p>

<h2>New Concepts</h2>

<ul> <li>collect new commandline functions</li> <li>save the functions into a library</li> <li>re-use the function library</li> <li>record command usage and results</li> <li>save a list of functions</li> <li>redirecting stderr and stdout</li> </ul>

<h2>New Functions</h2>

<ul> <li>gh – Grep the History</li> <li>quietly – quiet the noise, e.g. stderr</li> <li>ignore – the stdout. there are times to ~</li> <li>prov – command provenance and results</li> <li>function_list – list of functions to this point.</li> </ul>

<h2>History, and Quietly</h2>

<p>First, three simple little function to add peace and quiet to our commands. Two more will follow later in after saving the functions. Adding these functions demonstrates how maintain a library of short functions withour recourse to a text editor</p>

<p>The <strong>history</strong> builtin gets us started. It's often a command, or more likely, part of a command we remember:</p>

<pre><code>$ history </code></pre>

<p>lists our command number, a one-up sequence from our last login followed by the command. Here is a <a href="http://www.catonmat.net/download/bash-history-cheat-sheet.txt" title="Cheat sheet">cheat sheet of editor modes</a> for the many command options.</p>

<p>So to make our life simpler, this function Grep's the History:</p>

<pre><code>gh () { history | grep -i ${*:-.}; } </code></pre>

<p>When developing functions it may be useful to put diagnostics on the standard error: <code>stderr</code>. The first opportunity is what I call the <em>semantic comment</em>. The shell offers the sharp sign as a leading comment delimiter:</p>

<pre><code>$ command with args ... # this is a comment </code></pre>

<p>which must be used with care:</p>

<pre><code>some_function () { : this is likely a comment ... } </code></pre>

<p>Here we say “likely” since it is possible to execute things on such a line. For the moment, the semantic comment:</p>

<pre><code>comment () { echo $* 1>&2; } </code></pre>

<p>used anywhere, frequently in a function</p>

<pre><code>some_function () { comment this is certainly a comment ... } </code></pre>

<p>as the text <em>this is certainly a comment</em> is sent to the stderr. The noisy syntax in the <strong>comment</strong> function, “1>&2”, says to “redirect the standard output (1) onto the standard error (2)”. Once learned and put in very few functions, it may be forgotten until you have to explain it to one more novice than yourself.</p>

<p>Two similar functions will further reduce the need for more noise. These are <strong>quietly</strong> and <strong>ignore</strong>. We have a choice here. My thought was “quietly” is for the <em>stderr</em>, and “ignore” is for the <em>stdout</em>. Occasionally, you can ignore your output; often you will want the error messages to go quietly.</p>

<pre><code>quietly () { “$@” 2>/dev/null; } ignore () { “$@” >/dev/null; } </code></pre>

<p>Here <em>*/dev/null</em> is the proverbial bit-bucket. The “2>” syntax says direct to the <em>stderr</em>, while the simple “>” directs the <em>stdout</em>.</p>

<p>So, for example:</p>

<pre><code>$ ignore ls file # makes no sense, i.e. throws the output $ quietly ls file # suppresses “file not found” message </code></pre>

<h2>An Experiment in Comments</h2>

<p>Enter the following function on the command line, first entering the function definition, then executing each command in turn.</p>

<p>First, note since the function definition includes a trailing comment, the closing <code>}</code> curly brace must be entered on a separate line.</p>

<p>Then execute the <code>comment_demo</code> function, then execute it as the argument to <code>quietly</code> and then to <code>ignore</code>. </p>

<pre><code>$ comment_demo () { comment this is a comment; # this is NOT a comment } $ comment_demo $ quietly comment_demo $ ignore comment_demo $ fbdy comment_demo </code></pre>

<p>Here are the results of executing those commands</p>

<pre><code>comment_demo this is a comment

quietly comment_demo

ignore comment_demo this is a comment

fbdy comment_demo comment_demo () { comment this is a comment } </code></pre>

<p>The surprise with the last test: <code>fbdy collect_demo</code> is the disappearance of the sharp comment. To demonstrate, edit sharp (<code>#</code>) comments into any of your functions. Then either with <code>fbdy</code> or declare -f notice all those nicely placed comments disappear. This then is why the <code>comment</code> function is useful in itself. Fortunately, bash offers yet another means to comment shell functions. This will be taken up in a later chapter.</p>

<h2>Collecting</h2>

<p>Now you will collect the functions you have created in this book. With the exception of <strong>hello</strong>, they will all prove to be generally useful:</p>

<pre><code>hello today dateArg fbdy foreach gh quietly ignore </code></pre>

<p>To collect these functions, use the <strong>fbdy</strong> function. If you've used more than one terminal session by this point, you will have to go back through the chapters and re-enter them. Use a single terminal session to do this. A fair question is <em>Why</em>? Because the shell functions are in your shell environment. To see all your current functions, use this basic command. It will also show a few other shell features:</p>

<pre><code>$ set | grep '()' </code></pre>

<p>To collect all these functions, execute this command: for the moment, output will just be to your <em>stdout</em>, or terminal:</p>

<pre><code>$ fbdy hello today dateArg fbdy foreach gh quietly ignore </code></pre>

<p>produces:</p>

<pre><code>today () { date +%Y%m%d } dateArg () { date "+$1: %$1" } fbdy () { declare -f ${*:-fbdy} } foreach () { cmd="$1"; shift; for arg in "$@"; do $cmd "$arg"; done } gh () { history | grep -i $* } comment () { echo $* 1>&2 } quietly () { "$@" 2> /dev/null } ignore () { "$@" > /dev/null } </code></pre>

<h3>Examine your work</h3>

<ul> <li>did you see all the functions?</li> <li>if not, you can use the results here to re-enter the missing.</li> <li>have you recently used the <strong>gh</strong> function? it should help to run down the missing.</li> </ul>

<p>You're now ready to save, so you can easily re-use your functions.</p>

<h2>Saving</h2>

<p>As one extra feature, to document your work, I've added a useful function, <strong>prov</strong> standing for the word <a href="http://www.merriam-webster.com/dictionary/provenance">provenance</a>,</p>

<p>First, enter the <strong>prov</strong> function, and just for fun, <strong>function_list</strong>: </p>

<pre><code>prov () { printf "prov_doc ()\n{\n : $*\n}\n"; "$@" } function_list () { echo hello today dateArg fbdy foreach gh quietly ignore } </code></pre>

<p>So, now do this to save your work:</p>

<pre><code>$ prov fbdy $(function_list) prov function_list | tee firstlib $ chmod +x firstlib $ cat firstlib </code></pre>

<p>Produces this, and by virtue of the <strong>tee</strong> command, you've just stored in <strong>firstlib</strong>.</p>

<pre><code>prov_doc () { : fbdy hello today dateArg fbdy foreach gh quietly ignore prov function_list } hello () { echo 'Hello World!' } today () { date +%Y%m%d } dateArg () { date "+$1: %$1" } fbdy () { declare -f ${*:-fbdy} } foreach () { cmd="$1"; shift; for arg in "$@"; do $cmd "$arg"; done } gh () { history | grep -i $* } quietly () { "$@" 2> /dev/null } ignore () { "$@" > /dev/null } prov () { printf "prov_doc ()\n{\n : $*\n}\n"; "$@" } function_list () { echo hello today dateArg fbdy foreach gh quietly ignore } </code></pre>

<h3>Examine your work</h3>

<ul> <li>There are three functions in the library, not listed in the <strong>function_list</strong>. How did they get there?</li> <li>Particularly, the <strong>prov_doc</strong> function, where did that come from?</li> <li>How might you modify the <strong>function_list</strong> to include itself and <strong>prov</strong>?</li> <li>Do you need to add <strong>prov_doc</strong> to the book history?</li> </ul>

<p>Your <strong>firstlib</strong> is ready for re-use. </p>

<h2>Reuse</h2>

<p>Do this in two steps, first to show the underlying mechanism, then how to make sure it's routinely available for each terminal session.</p>

<h3>test in a new session</h3>

<ul> <li>Keep your current terminal session. </li> <li>record your current directory: echo $PWD</li>
<li>Open another and change directory to the same location as your original session.</li> <li><p><em>source</em> your new library:</p>

<p>$ source firstlib # or ". firstlib"; that's a period $ fbdy $(function_list) # shows the function bodies</p></li> </ul>

<p>If everything is in order, you're ready for the next step.</p>

<h3>restore your library when logging in</h3>

<p>Use the shell's login sequence to enable your new library on each terminal session. The most straight-forward is to use your <strong>.profile</strong>, routinely executed by the <em>login</em> sequence. Add this line to the end of your .profile:</p>

<pre><code>source {/the/directory/for/your/}firstlib echo $(function_list) 1>$2 </code></pre>

<p>You have enabled the functions in the firstlib and then written the function names on the <em>stderr</em> as a reminder. The <strong>function_list</strong> and <strong>comment</strong> functions supplied the list, and then put them on the standard error.</p>

<p>We will see, in a later book, my own personal practice is a bit different. I like to keep my profile clean of frequent changes, so I use yet another file, which I call <strong>.myrc</strong> for these personal features. So, after logging in, often, my first command is just:</p>

<pre><code>$. ~/.myrc </code></pre>

<p>which “sources” <em>.myrc</em>. The “rc” part is a bit of Unix history, standing for <strong>R</strong>untime <strong>C</strong>onfiguration. So, it’s <em>MY Runtime Configuration</em>.</p>

<p>Choose your preferred way to source the library, edit the file, and login yet one more time. Notice that you saw the comment list of functions</p>

<h2>References</h2>

<h2>Future direction</h2>

<p>These commands show you the direction you’ll take:</p>

<pre><code>$ set | grep ‘()’ # then $ set | awk ‘$2 ~ /()/ { print $1 }’ </code></pre>

<p>The latter, while it may not be perfect, is the basis for function-maintenance commands which will be a topic in a later book.</p>

<p>Mail me if <a href=”mailto:martymartymcgowan@alum.mit.edu?subject=collecSaveandReuseFunctions”>you have questions</a></p>

<link rel=”stylesheet” type=”text/css” href=”./mcgowan.css” /> <p><link rel=”stylesheet” type=”text/css” href=”./mcgowan.css” /></p>

<p><link rel=”stylesheet” type=”text/css” href=”./mcgowan.css” /></p>

<h1>Whats next? {#whatsNext}</h1>

<p>Before taking a peek at what’s next, it’s time to assess where you are. First, you’ve used these commands and shell built-ins, in more or less the order encountered in the text:</p>

<pre><code>echo date declare for do done set eval local shift history grep awk source tee chmod cat printf </code></pre>

<p>If you have any questions about them, it’s quite simple to search for <strong>bash shell</strong> <em>command-name</em> in your favorite search engine.</p>

<p>You now possess the skills to begin crafting your own function library and make it available for later terminal sessions. You are now able to collect and re-use functions as you come to need them. You’ll collect them, store and re-use them as multiple instances of the one case you have worked in the book.</p>

<p>As you do that, you’ll recognize other challenges:</p>

<ul> <li><p>how do I use these functions as part of another library?</p></li> <li><p>can I repair a function quickly as the need arises and easily restore it to its proper library?</p></li> <li><p>it seems I'm getting a large collection of functions: how do I keep them straight?</p></li> </ul>

<h2>Looking ahead {#whats-next-looking}</h2>

<p>The very next subject I'm planning is that of library management. I've functions on hand to:</p>

<ul> <li><p>capture a daily log of when functions were created.</p></li> <li><p>quickly update a function library with additions or changes</p></li> <li><p>delete a function or move a function from one library to another, </p></li> </ul>

<p>This all belongs to a practice I've established on the content and form of a function library:</p>

<ul> <li><p>building a Quick-Reference, which may be part of </p></li> <li><p>more extensive documentation</p></li> <li><p>what you may, may not, and must do when <em>source</em>ing a library.</p></li> </ul>

<p>This is all supported by what I've called: <em>Then only backup system you'll ever need</em>. That said, in the last two years, I've been an avid user of <a href="http://github.com/applemcg">github</a>. Since I'm not yet that experienced with <strong>git</strong>, so what you'll see in the backup system I offer might be called a crutch, but I use it as the routine check-point, preserving files in a more accessible state than git offers: i.e., using ordinary commands, such as <strong>cp</strong> to retrieve a backed-up version, which may be more than one edition old.</p>

<p>And to control versions, the subject of the <em>cloud</em> appears. At this moment, that includes git, and <a href="http://Dropbox.com">Dropbox</a>, so a practice and a function library to make that usage as concise as you need. In the last month (Sept-Oct 2013) I've started using Dropbox as my virtual <strong>HOME</strong> directory.</p>

<p>At some point I will discuss the distinction between using shell function libraries and the more common parlance of the <em>shell script</em>. If you haven't noticed this yet, I hadn't use the term before. That's conscious, since much of my practice is devoted to the command line. We will need to make the connection between this view of the shell, and connecting to the business needs. A good place to do that is constructing an application suitable for a <a href="http://en.wikipedia.org/wiki/Cron">cron_job</a></p>

<h2>Further ahead</h2>

<p>I have shell functions for the major application areas of:</p>

<ul> <li><p>file backup – "The Only Backup You'll Ever Need"</p></li> <li><p>database – I've been carrying around a personal copy of the too-little-used <a href="http://www.amazon.com/Relational-Database-Management-Prentice-Hall-Software/dp/013938622X">RDB</a>. <br /> Originally built on <a href="http://www.grymoire.com/Unix/Awk.html">awk</a> there are now <em>perl</em>-based implementations. I've stayed close to the author's original idea that <em>the shell is the only 4GL you'll ever need</em>.</p></li> <li><p>documentation – quite honestly, I'd worked on a nice auxiliary library for Markdown, the technology used in this book. However, I'm doing most of my personal writing in Emacs' OrgMode. Which brings up another question: what is the better way to leave text for the ages?</p></li> </ul>

<p>Mail me if <a href=”mailto:martymcgowan@alum.mit.edu?subject=whatsNext”>you have questions</a></p>