Leanpub Sample Technical Book

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 2013-10-19

Tweet This Book!

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

The suggested hashtag for this book is #shell.

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

https://twitter.com/search?q=#shell

Contents

Acknowledgments

- my friends on the itToolbox, shell forum
- helpers from the StackOverflow
- Nitin Chitniss, for the incentive to capture small shell functions
- Bill Anderson, with whom I collaborated on “The Software Assembly Line” and has provided support over the decades
- Pat, for the first edit for “voice”
- my employers of the past 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 has eased me into retirement

Preface

In this book you will learn how to write, save, and re-use shell functions.

The shell concepts are introductory. You may be familiar with them all. But if you aren't familiar with shell functions, you will learn how simple and powerful they are to use. These exercises are for you if you are the least bit curious about how to write and use shell functions.

Introduction

Each chapter is simple enough to require a quarter to a half an hour of your time. Each is meant to be worked at a terminal window on a Unix®, Linux, or other *nix server.

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

As an introduction, this short book as is the first of similar books on the shell function. You can explore the planned topics 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 mulitple terminal windows
- are running the **bash** shell
- have experience with one of the popular editors: **vi**, vim, … or **emacs** for command line editing.

If you need help getting started, you can contact me here[1] now, and at relevant places in the exercises.

[1] mailto:mcgowan@alum.mit.edu?subject=introduction

1 Write a shell function

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

In this chapter, you will write and use two simple shell functions: **hello**, and **today**.

1.1 Hello world

In this book, you can assume your command prompt is the dollar sign:

```
  $ ...
```

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

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

So, on the above line, you type everything from **hello** thru the closing curly brace, followed by a carriage return.

You use the function by typing its *name* 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 line 4.

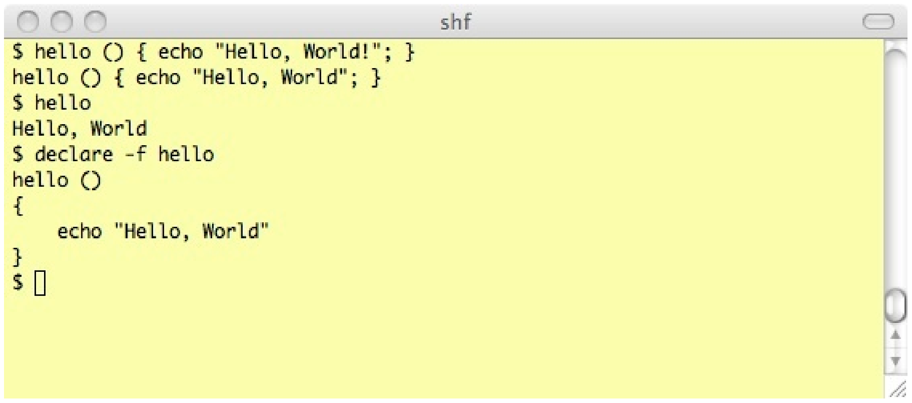
```
1  $ hello () { echo 'Hello World!' ; }
2  $ hello
3  Hello World!
4  $
```

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

```
$ declare -f hello
```

Type that command. Notice your function has been slightly reformatted. More on that later.

Here is a screenshot;



```
$ hello () { echo "Hello, World!"; }
hello () { echo "Hello, World"; }
$ hello
Hello, World
$ declare -f hello
hello ()
{
    echo "Hello, World"
}
$ 
```

1.2 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.

questions:

- *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?*

1.3 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**:

```
$ today () { date +%Y%m%d; }
$ today
20131008
$
```

Type the definition and invoke your new function **today**. Since **date** takes almost any upper- or lower-case letter, we'll deal with those later as arguments.

1.4 Activity

- investigate the options to the **date** command: search for *unix manual date*.

Mail me if you have questions[1]

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

2 Use function arguments

In the last exercise you used the **date** command in a function – in a very limited way. You are probably aware the date command has a number of upper- and lower-case options. Without going in great detail, a **date** option 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"
```

2.1 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 "F: +%F";
```

20251218-CORRECTION: date "+F: %F";

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

```
$ dateArg () { date "+$1: %$1"; }
$ dateArg c
c: Mon Oct  7 16:47:05 2013
$ dateArg F
F: 2013-10-07
$
```

In the **dateArg** shell 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. So, 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
```

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

2.2 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:

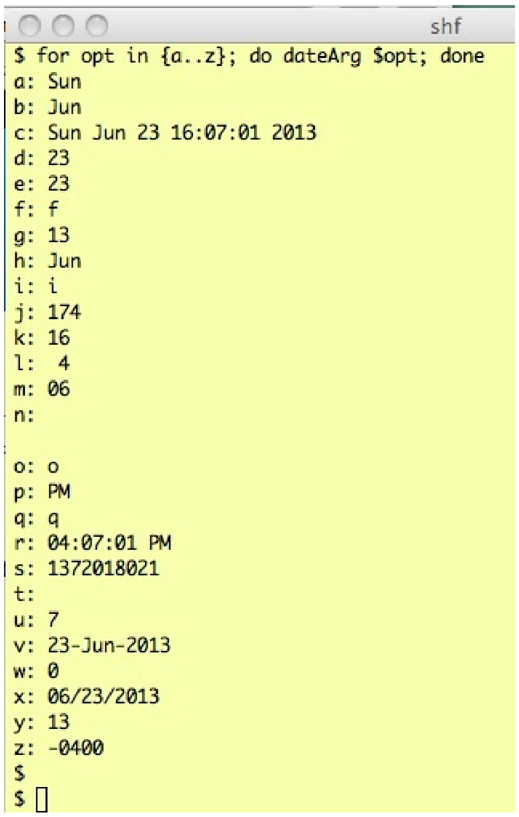
```
$ for opt in {a..z}; do dateArg $opt; done
```

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**. So, in this case, **dateArg** is invoked with each of the lower case letters.

Here is an example: 

```
$ for opt in {a..z}; do dateArg $opt; done
a: Sun
b: Jun
c: Sun Jun 23 16:07:01 2013
d: 23
e: 23
f: f
g: 13
h: Jun
i: i
j: 174
k: 16
l:  4
m: 06
n:

o: o
p: PM
q: q
r: 04:07:01 PM
s: 1372018021
t:
u: 7
v: 23-Jun-2013
w: 0
x: 06/23/2013
y: 13
z: -0400
$
$ 
```

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

2.3 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}
```

Mail me if you have questions[3]

[3] mailto:mcgowan@alum.mit.edu?subject=useFunctionArguments

2025-12-18: this exercise seems to be too soon.
Also, “foreach” is a defined function in “zsh”; so this name cannot be used.
“map” is a better name for this function, and it can be defined in both “zsh” and “bash”.

3 Inspect a function body

In this exercise, you will:

- use wild-card parameters
- use default parameters
- set postional parameters
- create a function to display functions

Recall, in a previous exercise you learned how to inspect a function body. And you were asked, *what would you name a function to display a Function **BoDY**.*

Did I give it away. That's what I called it: **fbdy**.

To review:

```
$ declare -f hello     # shows the hello function
```

By the way, 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 future exercise, you will learn a simple means to recover your work from day-to-day, session-to-session.

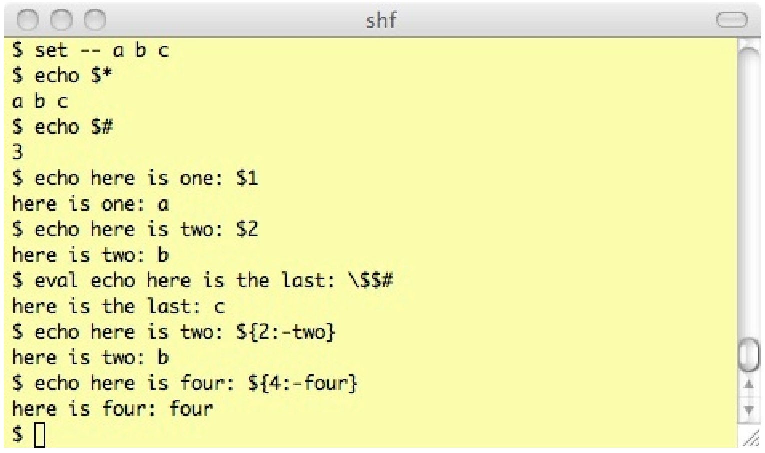
3.1 More about arguments

In addition the **positional parameters**: 1, 2, 3, ... you will learn of other parameter features. Two most important at this time are how to express all positional parameters, and how to assignn a **default value** to a positional parmeter. But, to start, here's how to **set** positional parameters. Enter these commands at your terminal window:

```
$ set -- a b c        # "set minus minus ..."
$ echo $*
$ echo $#
$ echo here is one: $1
$ echo here is two: $2
$ eval echo here is the last: \$$#
$ echo here is two: ${2:-two}
$ echo here is four: ${4:-four}
```

why is "eval" introduced here?;
`echo here is the last: $#` works

Check your results here:



```
$ set -- a b c
$ echo $*
a b c
$ echo $#
3
$ echo here is one: $1
here is one: a
$ echo here is two: $2
here is two: b
$ eval echo here is the last: \$$#
here is the last: c
$ echo here is two: ${2:-two}
here is two: b
$ echo here is four: ${4:-four}
here is four: four
$ 
```

use features of positional argument

Experiment with other arguments to the **set** command. By the way, you used the *minus* – option to **set**. (The first minus means an "option character follows", so the second minus is the "minus" option, which sets the **positional parameters**.)

questions

What does the

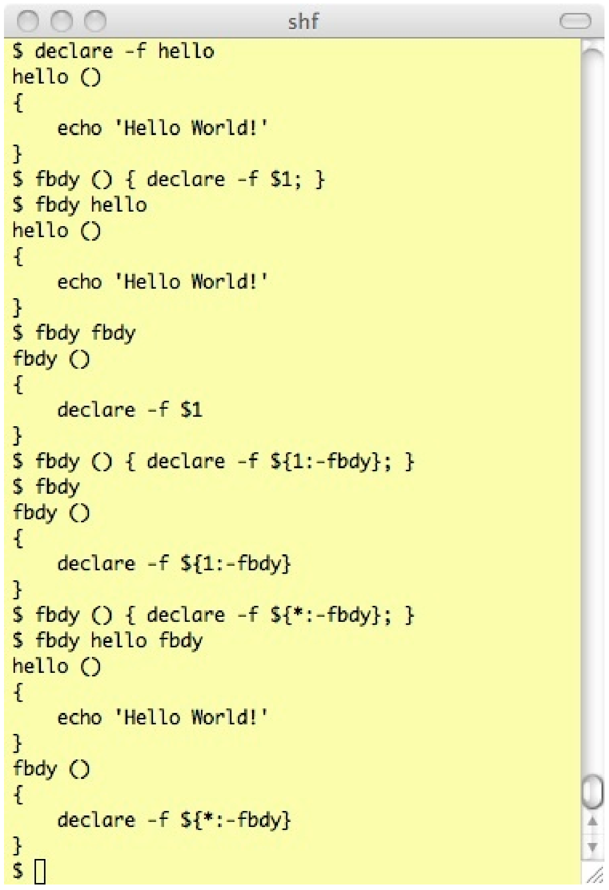
- *asterisk (*) mean?*
- *sharp/hash (#) mean in this context?*
- ***eval** command do?* hint: omit it from the command line.

3.2 The function body

You can take advantage of this information to write a function that returns a function body.
As you write this function, think of *what is a good **default** argument?*. Enter these, a line at a time:

```
$ declare -f hello              # as before
$ fbdy () { declare -f $1; }    # the simple version
$ fbdy hello                    # same as before
$ fbdy fbdy                     # no kidding!, so why not ...
$ fbdy () { declare -f ${1:-fbdy}; }    # so ...
$ fbdy                          # shows how to do it!, and
$ fbdy () { declare -f ${*:-fbdy}; }       # permits ...
$ fbdy hello fbdy               # whew, let's take a break
```

Here are the results for the function body:



```
$ declare -f hello
hello ()
{
    echo 'Hello World!'
}
$ fbdy () { declare -f $1; }
$ fbdy hello
hello ()
{
    echo 'Hello World!'
}
$ fbdy fbdy
fbdy ()
{
    declare -f $1
}
$ fbdy () { declare -f ${1:-fbdy}; }
$ fbdy
fbdy ()
{
    declare -f ${1:-fbdy}
}
$ fbdy () { declare -f ${*:-fbdy}; }
$ fbdy hello fbdy
hello ()
{
    echo 'Hello World!'
}
fbdy ()
{
    declare -f ${*:-fbdy}
}
$ 
```

define and use fbdy

questions

- *were you able to follow exercise?*
- *did you understand each step?* if not, Contact Me[1]
- *what does the final **fbdy** do with no arguments?*

[1]mailto:mcgowan@alum.mit.edu?subject=function_body

3.3 assesment

Review, if necessary, your understanding of:

- *wild-card parameters*
- *default parameters*
- *setting positional parameters*
- *how the function body is displayed. you might search* ***unix shell declare***

Mail me if you have questions[2]

[2]mailto:mcgowan@alum.mit.edu?subject=inspecteAfunctionBody

4 Loop with foreach

Now you will work with the **for** syntax to produce a **foreach** function that handles many loop requirements.
You will use the built-in **shift** command to control our positional parameters, and the **local** command to define a variable.

The shell has other useful *looping* constructs, namely the **while** loop. Our focus here, the **for** loop. The while loop is useful in situtations where the loop test may be more complicated than: *is there another argument to deal with?* Since here you will examine a known list of arguments, you want a **for** loop.

You will use the **for** loop to write a function I suggested in your exercise with function arguments:

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

Recall the dateArg function:

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

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

4.1 The for syntax

Recall the earlier example with **dateArg**:

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

specifically:

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

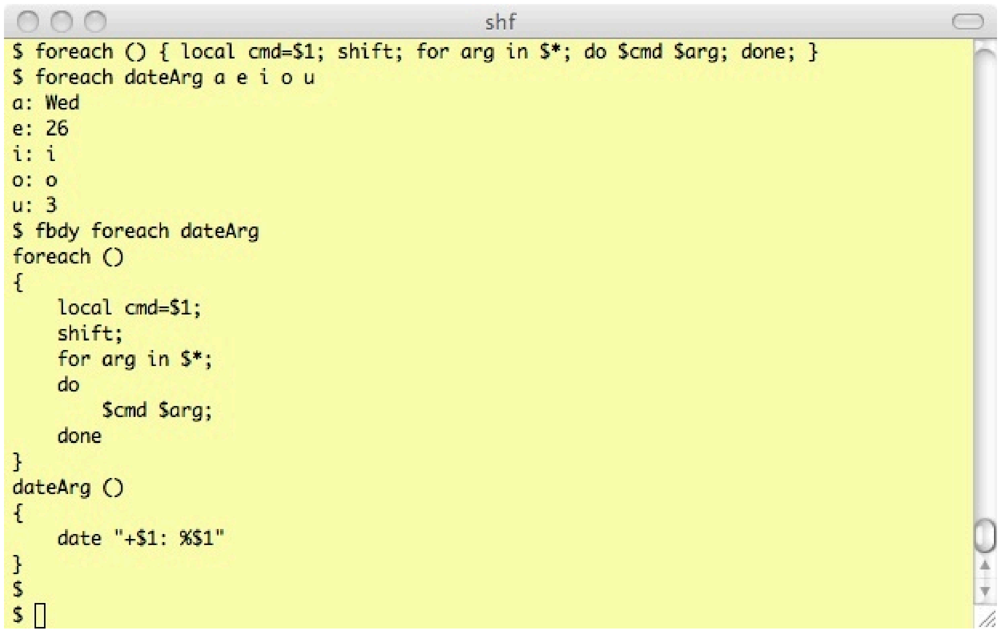
If you don’t have your **dateArg** function handy, re-enter it now. Then type the above command to execute it. Notice the generic *opt* argument could be any relevant name. Also, notice the postion of the dateArg function; it is called once per lower-case letter.

4.2 The foreach function

Enter this text to create your function, and test it:

```
$ foreach () { local cmd="$1"; shift; for arg in "$@"; do $cmd $arg; done; }
$ foreach dateArg a e i o u      # a purposely shorter list
$ fbdy foreach dateArg
```

Here are the *foreach dateArg* results:



```
$ foreach () { local cmd=$1; shift; for arg in $*; do $cmd $arg; done; }
$ foreach dateArg a e i o u
a: Wed
e: 26
i: i
o: o
u: 3
$ fbdy foreach dateArg
foreach ()
{
    local cmd=$1;
    shift;
    for arg in $*;
    do
        $cmd $arg;
    done
}
dateArg ()
{
    date "+$1: %$1"
}
$
$ 
```

using foreach to show the meaning and value of date options

4.3 Questions

- *did you compare the foreach function to the for command?*
- *did you notice the additional syntax in the function?*
- *explain what you think the **shift** keyword does.*
- *did you notice how the function body was displayed?*
- *what do you suppose is the meaning of the **local** keyword.*

Mail me if you have questions[1]

[1] mailto:mcgowan@alum.mit.edu?subject=loopWithForeach

5 A brief history of shell commands

In the introduction to the first chapter, I claimed *composing simple functions takes place on the command line*. This exercise delivers on that claim. You will see how using the command history makes it easy to create a shell function from an often-used command.

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.

5.1 Shell history modes

The shell has two editor-based history modes, *vi* and *emacs* as well as keyboard commands to navigate the shell history. To use the text editors for shell history, you navigate with the editor's line and inter-line movement mechanisms.

To see all the available shell options use a **set -o** command. Try that at your terminal window. Execute this command to see which your current editor mode is set at:

```
$ set -o | grep -E '^(vi|emacs)'
emacs           on
vi              off
$
```

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

```
$ set -o vi
```

to switch to the *vi* mode. You may want to consult this cheatsheet of editor modes[1].

[1] http://www.catonmat.net/download/bash-history-cheat-sheet.txt

5.2 How is a function invented

Recall your experiment with the dateArg function. Your first attempt was a simple date command:

```
$ date +%F
```

Here’s a piece of my recent shell history which shows how the command may be turned into a function:

```
$ history | awk '$1 > 803 && $1 < 810'
  804  date +%F
  805  date "+F: %F"
  806  set -- F
  807  date "+$1: %$1"
  808  set -- c
  809  date "+$1: %$1"
$
```

Type the first three commands (804 – 806) at your terminal window now. *do **not** type the line numbers*. Line number 806 sets the first **positional parameter** to the letter *F*, just as it would be when passed to a function. You might try

```
 echo $1
```

to verify that.

Then, on line 807, 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 807 literally, do this:

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

On line 808, change the first positional parameter to the lower-case *c*, and re-execute the command by stepping back two lines in history to line 807.

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.

- Assessment
  - do you have a preferred editor mode, or do you prefer keyboard navigation?
  - did you experiment with several options to the **date** command?
  - did you refresh your experiment with the *set* – command?
- Questions
  - do you understand how the formatted **date** options behave?
  - … how the *set* – command works?

5.3 The simple step – the function

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:

```
$ !807
date "+$1: %$1"
c: Mon Jul  1 09:58:20 2013
$ dateArg () { date "+$1: %$1"; }
$ fbdy  dateArg
dateArg ()
{
    date "+$1: %$1"
}
$
```

Where my history line number was 807. 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 *dateArg () { … }*, simply retrieve the just-executed command by going back one line in the history (to the *date* … command), but not re-execute it.

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

5.4 Review

- *Did you understand the **history** command?*. The pipe to **awk** is a little bonus?
- *Did you succeed in creating the **dateArg** function?*
- *If not, can you assess where you got hung up?*
- *Are you able to recall all the functions you have written?* You will learn this shortly.

Here's a sneak peak at one of the next facets:

```
$ gh () { history | grep -i ${*:-gh}; }
```

What might you use it for?

You may Contact Me[2]

[2] mailto:mcgowan@alum.mit.edu?subject=aBriefHistoryOfShell

6 Semantic comments

In this section, you will solve a problem introduced by our recently introduced fbdy function which seems to have an udesirable side-effect: it strips comments.

To turn that to our advantage, just replace the syntactic comment with a semantic version, which you will create in this exercise.

6.1 The problem

You saw how the **fbdy** functions displays the full text of the function body, including the name, and importantly, in a fashion which is suitable for use as the function definition. Since fbdy takes multiple arguments, it can collect the text of any functions on your output. Also, since the builtin **declare** omits any comments you may have included, I found it necessary to invent two *semantic* features to keep comments in the resulting function body.

Here is the original function:

```
fbdy ()
{
    declare -f ${*:-fbdy}
}
```

For your *first exercise* in this section, enter this as your function:

```
$ fbdy () {
    # fbdy uses declare builtin with variable # of args, default fbdy
    declare -f ${*:-fbdy}
}
$ ...      # then, run it
$ fbdy     # what do you see?
```

Since you have written a few functions to this point, you may want to experiment with them by placing comments in the function and satisfying yourself the **declare** builtin omits them on it's output.

So, how to preserve a meaningful comment? I've taken a two-step process. First, a comment should never show up on the standard output, and second, since some *comments* may show up as ERROR, WARNING, or TRACE, ... messages, a semantic way can create a *quiet* comment. This makes it useful in our code. A "quiet comment" serves just as a syntactic common would: a visual comment for the code reader or developer.

6.2 Ignore quietly comment

To handle the first criteria you will write a simple function, “comment”. which puts the comment in a useful place, the *standard error*, usually referred to as **stderr**. The second criteria is equally simple, *quietly* ignore the stderr.

Here are two functions to do those jobs, with a third thrown in for completeness:

```
comment        () { echo "$@" 1>&2; }
quietly        () { "$@" 2> /dev/null; }
ignore        () { "$@" > /dev/null; }
```

For your *second exercise*, enter them on the command line, just as you see there, and test that you have entered them properly:

```
$ fbdy comment quietly ignore
```

You will exercise them more fully in a moment.

There are a few facets worth noting here. Presumably, you are familiar with **echo** by now. It copies its arguments to the *standard output*, or **stdout**. The new constructs are:

- the ”**$@**” standing for “quote all the arguments”: *command “$1” “$2” …*
- the uses of the number **2**, in the comment function, where it follows the output redirection (>) symbol and in the **quietly** function, where it preceeds it. More in a moment,
- the output redirection to **/dev/null**.

In the **comment**, the meaning of *1>&2* is: *duplicate (the “&” token) the stdout (1) on the stderr (2)*

and in the **quietly** function, “2> /dev/null” means: *write the stderr (2) on the* ***/dev/null*** file which is the Unix bit-bucket, or trash-bin

I’ve included the **ignore** function which treats the *stdout* similarly to the **quietly** function’s treatment of *stderr*.

6.3 Using semantic comments

For your *third exercise*, enter these commands

```
echo this is a test
comment this too, is a test

ignore echo this is a test
ignore comment this is a test

quietly echo this is a test
quietly comment this too, is a test
```

*Can you reconcile the differences between **echo** and **comment***

Mail me if you have questions[1]

[1] mailto:mcgowan@alum.mit.edu?subject=semanticComments

7 Collect, save, and re-use functions

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.

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.

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

7.1 Collecting

Now you will collect the functions you have created in this book. With the exeception of **hello**, they will all prove to be generally useful:

```
hello
today
dateArg
fbdy
foreach
gh
quietly
ignore
comment
```

To collect these funcitons, use the **fbdy** 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 *Why*? 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:

```
$ set | grep '()'
```

To collect all these functions, execute this command: for the moment, output will just be to your *stdout*, or terminal:

```
$ fbdy hello today dateArg fbdy foreach gh quietly ignore comment

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 ()
{
    "$@" > /dev/null
}
comment ()
{
    echo "$@" 1>&2
}
```

Examine your work

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

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

7.2 Saving

As one extra feature, to document your work, I've added a useful function, **prov** standing for the word provenance[1],

First, enter the **prov** function, and just for fun, **book_history**:

```
prov ()
{
      printf "prov_doc ()\n{\n    comment $*\n}\n"; "$@"
}
book_history ()
{
    echo hello today dateArg fbdy foreach gh quietly ignore comment
}
```

So, now do this to save your work:

```
 $ prov fbdy $(book_history) prov book_history | tee booklib
 $ chmod +x booklib
 $ cat booklib
```

Produces this, and by virtue of the **tee** command, you've just stored in **booklib**.

```
prov_doc ()
{
    comment fbdy hello today dateArg fbdy foreach gh quietly ignore comment prov \
book_history
}
today ()
{
    date +%Y%m%d
}
dateArg ()
{
    date "+$1: %$1"
}
fbdy ()
{
    declare -f ${*:-fbdy}
```

[1]http://www.merriam-webster.com/dictionary/provenance

```
}
foreach ()
{
    cmd="$1";
    shift;
    for arg in "$@";
    do
        $cmd "$arg";
    done
}
gh ()
{
    history | grep -i $*
}
quietly ()
{
    "$@" 2> /dev/null
}
ignore ()
{
    "$@" > /dev/null
}
comment ()
{
    echo "$@" 1>&2
}
prov ()
{
    printf "prov_doc ()\n{\n    comment $*\n}\n";
    "$@"
}
book_history ()
{
    echo hello today dateArg fbdy foreach gh quietly ignore comment
}
```

Examine your work

- There are three functions in the library, not listed in the **book_history**. How did they get there?
- Particularly, the **prov_doc** function, where did that come from?
- How might you modify the **book_history** to include itself and **prov**?

- Do you need to add **prov_doc** to the book history?

Your **booklib** is ready for re-use.

7.3 Reuse

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

test in a new session

- Keep your current terminal session.
- record your current directory: echo $PWD
- Open another and change directory to the same location as your original session.
- *source* your new library:

```
$ source booklib        # or  ". booklib"; that's a period
$ fbdy $(book_history)  # shows the function bodies
```

If everything is in order, you're ready for the next step.

restore your library when logging in

Use the shell's login sequence to enable your new library on each terminal session. The most straightforward is to use your **.profile**, routinely executed by the *login* sequence. Add this line to the end of your .profile:

```
source {/the/directory/for/your/}booklib
comment $(book_history)
```

You have enabled the functions in the booklib and then written the function names on the *stderr* as a reminder. The **book_history** and **comment** functions supplied the list, and then put them on the standard error.

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 **.myrc** for these personal features. So, after logging in, often, my first command is just:

```
$ . ~/.myrc
```

which "sources" *.myrc*. The "rc" part is a bit of Unix history, standing for **R**untime **C**onfiguration. So, it's *MY Runtime Configuration.*

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

7.4 Future direction

These commands show you the direction you'll take:

```
$ set | grep '()'     # then
$ set | awk '$2 ~ /\(\)/ { print $1 }'
```

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

Mail me if you have questions[2]

[2] mailto:mcgowan@alum.mit.edu?subject=collecSaveandReuseFunctions

8 Whats next?

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:

```
echo
date
declare
for
do
done
set
eval
local
shift
history
grep
awk
source
tee
chmod
cat
printf
```

If you have any questions about them, it's quite simple to search for **bash shell** *command-name*.

You now possess the skills to begin crafting your own function library and make it available on 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.

As you do that, you'll recognize other challenges:

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

8.1 Looking ahead

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

- capture a daily log of when functions were created.
- quickly update a function library with additions or changes
- deleting a function or moving a function from one library to another,

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

- building a Quick-Reference, which may be part of
- more extensive documentation
- what you may,may not, and must do when *source*ing a library.

This is all supported by what I've called: *Then only backup system you'll ever need*. That said, in the last two years, I've been an avid user of github[1]. And not yet that experienced with **git**, 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 **cp** to retrieve a backed-up version, which may be more than one edition old.

And to control versions, the subject of the *cloud* appears. At this moment, that includes git, and Dropbox[2], 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 **HOME** directory.

At some point I will discuss the distinction between using shell function libraries and the more common parlance of the *shell script*. 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 appliction suitable for a cron job[3]

[1]http://github.com/applemcg

[2]http://Dropbox.com

[3]http://en.wikipedia.org/wiki/Cron

8.2 Further ahead

I have shell functions for the major application areas of:

- make – the utility conventionally used to build programs, I've long held that **make** offers much wider use than in application development, particualrly documentation, testing, and management information.
- database – I've been carrying around a personal copy of the too-little-used RDB[4].
  Originally built on awk[5] there are now *perl*-based implementations. I've stayed close to the author's original idea that *the shell is the only 4GL you'll ever need.*
- documentation – I'm a more recent convert to **markdown**. This offers the prospect of sharpening your tools to produce a properly indexed and referenced document. This is one of my goals here.

Mail me if you have questions[6]

[4]http://www.amazon.com/Relational-Database-Management-Prentice-Hall-Software/dp/013938622X
[5]http://www.grymoire.com/Unix/Awk.html
[6]mailto:mcgowan@alum.mit.edu?subject=whatsNext