Putting It Together - Some Useful Examples

In this lesson, you're going to see many of the ideas you've learned put together in more realistic contexts, so you can get a flavour for what bash can do for you on a day-to-day basis.

WE'LL COVER THE FOLLOWING

How Important is this Lesson?

You can easily skip this lesson if you want as nothing here is new, but following this can embed some concepts and keep your motivation going before the final part!

Output With Time

Frequently, I want to get the output of a command along with the time. Let's say I'm running vmstat while a server is having problems, and I want to ensure that I've logged the time that vmstat relates to. Type this in:

```
function dateit() {
    while read line
    do
        echo "$line $(date '+ %m-%d-%Y %H:%M:%S')"
    done
}
vmstat 1 | dateit
```

Type the above code into the terminal in this lesson.

Terminal

> Note: `vmstat` is a program available on most Linux flavours that gives you a view of what the system resource usage looks like. It is not available on Mac OSes. If vmstat does not exist for you, then replace with `vm_stat`, which should be available.

You should be able to follow what's happening there based on what you've learned so far in

- Functions
- Loops
- Command Substitution
- Pipes and Redirects

The one exception is the `while read line` line. `read` is a shell builtin that takes input from standard input until a newline is seen (or an 'end of file' byte on its own line).

```
read
some input
```

Type the above code into the terminal in this lesson.

It returns a true exit code if input was seen:

```
echo $?
```

Type the above code into the terminal in this lesson.

and false if an end of file character is seen (with `\C-d`):

```
read
^D
echo $?
```

Type the above code into the terminal in this lesson.

If an argument is given to `read`, then a variable is populated with the input:

```
read myinput
some input
echo $myinput
```

Type the above code into the terminal in this lesson.

So in the `dateit` function we input above, the while loop keeps taking input from standard input and acts on each line as it's read in, echoing each line to standard output until is gets an end-of-file character, at which point the while loop terminates as read returned a false exit code.

You will see the date appended to each line in the output. Experiment with the function to place the date before the line, or even on a separate line. See also the exercises below.

Where Am I?

You may be familiar with the `pwd` builtin, which gives you your current working directory (*cwd*). Similarly, there is an environment variable (`PWD`) that bash normally sets that stores the cwd.

```
pwd
echo $PWD
```

Type the above code into the terminal in this lesson.

Very often in scripts, you will want to know where the current working directory of the process is.

But also (very often) you will want to know where *the script you are running* is located *from within the script*.

For example, if you are running a script that is found in your `PATH` (ie not in your local folder), and you want to refer to another file relative to that script from within that script, then you will need to know where that script is located.

```
cat > /tmp/pwdscript.sh << 'EOF'
echo My pwd is: $PWD
echo I am running in: $(dirname ${BASH_SOURCE[0]})
```

```
chmod +x /tmp/pwdscript.sh
/tmp/pwdscript.sh
```

Type the above code into the terminal in this lesson.

Have a play with this command and read the code over to see what it does. If you can't work something out try the hint below.

Show Hint

Was the output of history what you expected? `HISTCONTROL` can determine what gets stored in your history. The directives are separated by colons. Here we use `ignoredups` to tell history to ignore commands that are repeats of the last-recorded command. In the above input, the two consecutive `ls` commands are combined into one in the history. If you want to be really severe about your history, you can also use `erasedups`, which adds your latest command to the history, but then wipes all previous examples of the same command out of the history. What would this have done to the history output above?

`ignorespace` tells bash to not record commands that begin with a space, like the `pwd` in the listing above.

`CTRL-r`

Bash offers you another means to use your history.

Hit `CTRL` and hold it down. Then hit the `r` key. You should see this on your terminal:

```
(reverse-i-search)`':
```

Let go. Now type `grep`. You should see a previous grep command. If you keep hitting `CTRL+r` you will cycle through all commands that had grep in them, most recent first.

If you want to cycle forward (if you hit `CTRL+r` too many times and go past the one you want (I do this a lot)), hit `CTRL+s`.

```
function extract() {
```

```
function extract() {
    if [ -z "$1" ]
    then
        echo "Usage: extract <file_name>.<zip|rar|bz2|gz|tar|tbz2|tgz|Z|7z|xz|ex|tar.bz2|tar.
    else
        if [ -f $1 ] ; then
           case $1 in
             *.7z)        7z x $1        ;;
             *.bz2)       bunzip2 $1     ;;
             *.exe)       cabextract $1  ;;
             *.gz)        gunzip $1      ;;
             *.tar.bz2)   tar xvjf $1    ;;
             *.tar.gz)    tar xvzf $1    ;;
             *.tar.xz)    tar xvJf $1    ;;
             *.tar)       tar xvf $1     ;;
             *.tbz2)      tar xvjf $1    ;;
             *.tgz)       tar xvzf $1    ;;
             *.Z)         uncompress $1  ;;
             *.xz)        unxz $1        ;;
             *.lzma)      unlzma $1      ;;
             *.rar)       unrar x -ad $1 ;;
             *.zip)       unzip $1       ;;
             *)           echo "extract: '$1' - unknown archive method" ;;
           esac
        else
            echo "$1 - file does not exist"
        fi
    fi
}
mkdir lbthw_misc
cd lbthw_misc
touch a b c
tar cvfz test.tgz a b c
rm a b c
extract test.tgz
```

Type the above code into the terminal in this lesson.

Challenge: explain what each line does.

You should be able to do this based on the lessons on:

- Tests
- Functions
- Loops

Output Absolute File Path

Quite often I want to give co-workers an absolute reference on a server to a file that I am looking at. One way to do this is to cut and paste the output of `pwd`, add a `/` to it, and then add the filename I want to share.

This takes a few seconds to achieve, and since it happens regularly, it’s a great candidate for a time-saving function:

```
function sharefiles() {
    for file in $(ls "$@"); do
        echo -n $(pwd)
        [[ $(pwd) != "/" ]] && echo -n /
        echo $file
    done
}
sharefiles
```

Type the above code into the terminal in this lesson.

You should be able to work out what is going on in the above function based on the lessons:

- Functions
- Loops
- Variables

The only novel element is the special parameter `$@`. This (very useful) variable gives you all the arguments passed to the function (or program, if called outside a function).

Saving time by writing a function is often a great idea, but deciding what is worth automating is a non-trivial task.

Here are some things you want to think about when deciding what to automate:

- How often do you perform the task?
- How much effort is it to automate (it's easy to under-estimate this!)
- Will the automation require effort to maintain?
- Do you always perform the task on the same machine?
- Do you control that machine?

My experience is that the effects of automation can be very powerful, but the above factors can also limit the return on investment.

Think about what you could automate today!

Think about what you could automate today!

What Next?

Well done! You’ve completed this part, and taken your bash knowledge from theoretical to practical. The next section - ‘Advanced Bash’ - covers the more sophisticated features of bash that will up your bash game to the next level.