Shellscripting: Absolute Basics

Joseph Hallett

January 12, 2023

Whats all this about?

I've written a *lot* of code over the years:

Assembly, C and Java	as an engineer
Commonlisp	for my own projects
Haskell	to build compilers
PostScript	to draw really efficient diagrams
LaTeX	to publish books
	...several a dozen other things too

Which language have I written the most code in?

Which language do I use to solve most tasks?

Which language do I like the least?

Shellscripting!

Normally we type commands for the terminal on a commandline...

- But we can automate them and stick them into scripts

Anything you have to do more than once...

Write a script for it!

- Saves a tonne of time
- Often easier than writing a full program

For example...

```
#! /bin/sh
GREP=grep
if [ $(uname) = "OpenBSD" ]; then
    # Use GNU Grep on OpenBSD
    GREP=ggrep
fi

${GREP} -Pi "^${1}$" /usr/share/dict/words
```

Sometimes I cheat at *Wordle*:

- I want to know a word that matches a regex exactly
- I can search the system dictionary file at `/usr/share/dist/words`
- `grep` can do the search, but I need to explicitly specify GNU Grep on systems where it isn't the default

  ```
  knotwords 'st[^aeo]pid'
  ```

  ```
  - stupid
  ```

Or for example...

```
#! /usr/bin/env bash
if [ $1 = "should" -a $2 = "also" -a $3 = "run" ]; then
  shift 3
  gum confirm "Run 'doas $*'?" && doas $*
elif [ $1 = "should" -a $2 = "also" -a $3 = "remove" ]; then
  gum confirm "Delete '$4'?" && doas rm -fr "${4}"
else
  2>&1 printf " WARNING  You should read the commands you"
  2>&1 printf "paste more carefully\n"
fi
```

Sometimes when I upgrade my computer it tells me to delete some files or run some commands:
`You should also run rcctl restart pf`
Copying and pasting the precise text is a pain...

- Can I just copy the whole line and run that?

(Of course I can... should I though?)

Or for a further example...

```
#! /usr/bin/env bash
# Fix kitty
/usr/local/opt/bin/fix-kitty

# Update sources
cd /usr/src && cvs -q up -Pd -A
cd /usr/ports && cvs -q up -Pd -A
cd /usr/xenocara && cvs -q up -Pd -A
```

After I upgrade my computer I need to run a couple of standard commands.

- I can never remember them
- Batch them up!

So whats this really about?

Shellscripting is about automating all those tedious little jobs

- Byzantine syntax (based on shell commands)
- Awful for debugging
- Requires magical knowledge
- Probably the most useful thing you'll ever learn

Luckilly we have help

Shell scripting is somewhat magical, and there are *lots* of gotchas...

https://www.shellcheck.net

Wonderful tool to spot unportable/dangerous things in shell scripts

- Commandline tool available
- Run it on *everything* you ever write
- `shellcheck` is great

```
shellcheck `command -v knotwords`
In /home/joseph/.local/bin/knotwords line 2:
GREP=grep
^--^ SC2209 (warning): Use var=$(command) to assign output (or quote to assign string).


In /home/joseph/.local/bin/knotwords line 3:
if [ $(uname) = "OpenBSD" ]; then
     ^------^ SC2046 (warning): Quote this to prevent word splitting.

For more information:
  https://www.shellcheck.net/wiki/SC2046 -- Quote this to prevent word splitt...
  https://www.shellcheck.net/wiki/SC2209 -- Use var=$(command) to assign outp...
```

So how do you write one?

Start the file with the *shebang* `#!` then the path to the interpreter of the script plus any arguments:

For portable POSIX shellscripts `#! /bin/sh/`

For less portable BASH scripts `#! /usr/bin/env bash`

Then

- `chmod +x my-script.sh`
- `./my-script.sh`

The rest of the file will be run by the interpretter you specified

- or `sh my-script.sh` if you don't want to/can't mark it executable.

(Hey this is also why Python scripts start `#! /usr/bin/env python3`)

Why `env`?

Hang on, you might be saying, I know that bash is always in `/bin/bash`… can I just put that as my interpretter path?

Yes, but…

In the beginning `/bin` was reserved for just *system* programs

- and `/usr/bin` for admin installed programs
- and `/usr/local/bin` for locally installed programs
- and `/opt/bin` for optional installed programs
- and `/opt/local/bin` for optional locally installed programs
- and `~/.local/bin` for a *users* programs
- …oh and sometimes they're even mounted on different disks!

This is *kinda* madness.

- So *must* Linux systems said look we'll just stick everything in `/bin` and stop using multiple partitions
- But some said no it should be `/usr/bin`, one said `/Applications/`, and others stuck them in `/usr/bin` but symlinked them to `/bin`
- And on some systems users grew fed up of the outdated system `bash` and compiled their own and installed it in `~/.local/bin` …
- …and ever tried using Python `venv`?

env

```
ENV(1)                  General Commands Manual                  ENV(1)

NAME
     env – set and print environment

SYNOPSIS
     env [-i] [name=value ...] [utility [argument ...]]

DESCRIPTION
     env executes utility after modifying the environment as specified on the
     command line.  The option name=value specifies an environment variable,
     name, with a value of value.
```

What `env` does is look through the `PATH` and tries to find the program specified and runs it.

...Path?

There is an environment variable called `PATH` that tells the system where all the programs are:

- Colon separated list of paths

If you want to alter it you can add a line like to your shell's config

```
export PATH="${PATH}:/extra/directory/to/search"
```

Your shells config is *possibly* in `~/.profile` but it often varies... check the `man` page for your `${SHELL}`
Also some shells have different syntax (e.g. `fish`)...

```
$ tr ':' $'\n' <<< $PATH
/home/joseph/.local/share/python/bin
/bin
/usr/bin
/sbin
/usr/sbin
/usr/X11R6/bin
/usr/local/bin
/usr/local/sbin
/home/joseph/.local/bin
/usr/local/opt/bin
/usr/games
/usr/local/games
/usr/local/jdk-17/bin
/home/joseph/.local/share/go/bin
```

Basic Syntax

Shell scripts are written by chaining commands together

`A; B`	run A then run B
`A | B`	run A and feed its output as the input to B
`A && B`	run A and if successful run B
`A || B`	run A and if not successful run B

How does it know if its successful?

Programs *return* a 1 byte exit value (e.g. C main ends with `return 0;`)

- This gets stored into the variable `${?}` after every command runs.
- `0` indicates success (usually)
- `>0` indicates failure (usually)

This can then be used with commands like `test`:

```
do_long_running_command
test $? -eq 0 && printf "Command succeeded\n"
```

Or the slightly shorter:

```
do_long_running_command
[ $? -eq 0 ] && printf "Command succeeded\n"
```

Wrap up

Thats the basics of shell scripting,

- Include a `#!`
- Always use `env`
- `$?` contains the exit code

Next time

Control flow and more advanced shell scripting for shellscripts.

Bonus puzzle

Why is this the case?

```
[ $? -eq 0 ] # works
[$? -eq 0]   # doesn't work
```

Different shells

(Just use bash unless you care about *extreme* portability in which case use POSIX sh)

Typical Shells

sh	POSIX shell
bash	Bourne Again shell (default on Linux)
zsh	Z Shell (default on Macs), like bash but with more features
ksh	Korne shell (default on BSD)

Other shells

dash	simplified faster bash, used for booting on Linux
Busybox sh	simplified bash you find on embedded systems

Weird shells

fish	More usable shell (but different incompatible syntax)
elvish	Nicer syntax for scripting (but incompatible with POSIX)
nushell	Nicer output (but incompatible, and weird)