Shellscripting: The Basics (1)

normally you type commands for the terminal on the commandline...

BUT, you are able to automate this process by sticking them in a file which the computer automatically runs

- anything you have to do more than once, write a shell script for it

So what is shell scripting?

- it allows you to batch up a series of commands into one file
- and run them all as though they were a single command

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
```

another 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?)

a final 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!

Shellscripting basically lets you batch up a series of commands, and then run them as if they were a single command:

Shellscripting drawbacks

- the syntax is based on shell commands, so it is very weird
- hard to debug
- requires magical knowledge

However shellscripting is super helpful if you can learn it

Help with writing Shellscripts

- **linters** check source code has been written correctly - and safetly

https://www.shellcheck.net/

- this is one for shell scripting
- will check everything you write, and will spot problems ahead of time
- there is also a commandline tool available
- run this on everything you ever write

Setting ShellCheck up on commandline

1. install it

```
sudo apt-get install shellcheck  // LINUX

brew install shellcheck         // on mac
```

2. Run it on your script

```
shellcheck script.sh   // (replace script.sh with path to file)
```

3. **Review the Output**: ShellCheck will analyze your script and print out any warnings, suggestions, or errors it finds. These messages can help you identify potential issues or areas for improvement in your script.
4. **Optional Flags**: ShellCheck has several optional flags that you can use to customize its behavior. For example:
    - `f FORMAT` to specify the output format (`json`, `checkstyle`, `gcc`, `tty`, etc.).
    - `x` to allow following sourced files (if they're accessible).
    - `a` or `-enable=all` to enable all optional checks.

You can see all options by running `shellcheck --help`.

example command line w flags:

```
shellcheck -f gcc script.sh
```

```
insert example shellscript and shellcheck messages...
```

Writing a Shellscript

start with a shebang: **#!** and then the path to the interpreter of the script plus any arguments e.g.

- **for portable POSIX shellscripts:**

```
#! /bin/sh
```

- f**or less portable BASH scripts:**

```
 #! /usr/bin/env bash
```

then you need to give executable permission:

- chmod +x my -script.sh

then you can run it as normal:

- ./my-script.sh
- **the rest of the file will be passed into the specified interpreter**

What is env?

e.g. in:

```
#! /usr/bin/env bash
```

so if we always know bash is in bin bash why use env???

- to do with shellscripting being an old UNIX technology
  - in the beginning /bin was reserved for **JUST system programs**
  - and usr/local/bin for locally installed programs
  - /opt/bin for **optionally installed programs**
  - and MANY MORE

This was madness!

- so most LINUX systems (around the year 2000) said they will stick everything in /bin and stopped using numerous partitions
- but some said it should be /usr/bin, someone else said /Applications/, and others stuck them in /usr/bin but linked them to /bin

So the env command lets you manipulate the environment a program runs in

```
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.
```

- importantly when you use env, it will look through the PATH and try to find the program you specified and run it

- PATH variable just says where all your programs are installed

PATH variable

- PATH is an **environment variable**
- it tells the system where all the programs are

Inside PATH is just a colon seperated list of paths

- when you ask your system to run a program (e.g. bash), it will look through each of the directories **specified in the PATH** in order, and try and find the program in there
- if i finds it, it will run it, if i cant find the program it will try the next directory until it has gone through every directory in the path

if you want to alter the path you can edit it by adding a line to your shells config such as:

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

Basic Shellscript Syntax

Shellscripts are written by chaining commands together:

```
A; B //run A then run B

A | B // run A, feed its output as the input to B

A && B // run A, and if it is successful run B

A || B // run A and if not succesful run B
```

How do you know if the program is successful??

- programs **return a 1 byte exit value** (e.g. return 0 in end of main in C)
- this return value gets stored inside a variable called **${?}** after every command runs
- 0 usually indicates success
    - non-zero indicate some sort of failure

These values can then be queried with commands such as: **test**

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

shorter syntax for the test command, use square brackets:

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

Conclusion

- include #!
- always use env
- $? contains the exit code

Review Section:

`PATH` is an environment variable in Unix-like operating systems that specifies a colon-separated list of directories where the system should look for executable files. It is used to locate executable files without specifying their full paths.
On the other hand,
`env` is a command-line utility that allows for the modification or printing of the environment variables for a command, enabling the execution of a command in a modified environment. While `PATH` is specifically concerned with locating executable files, `env` is more versatile, allowing for broader environment variable manipulation during command execution.