# An Introduction to Shell Scripting

Paul Brown

p.e.brown@warwick.ac.uk

# What is the shell?

- A command line user interface for Unix-like operating systems.
- Interactive and scripting modes

# What is the Bash Shell?

- Bourne Again SHell, replacing the older Bourne shell in 1989
- Default shell on most Linux systems and MacOS

# Features

- User customisable
- Wildcard matching
- Re-direction
- Command substitution
- Control structures

# When to use the shell

- As a wrapper for a workflow
- When launching other processes
- When doing lots of filesystem access
- When low level access to hardware is required

# When not to use the shell

Shell scripting is of much less use when any of the following are required

- Complex calculations
- A graphical user interface
- Any kind of debugging beyond very basic

# Starting up

- Often opened via the graphical desktop
- Startup files are read to provide user customisations, eg .bash_profile, .bashrc

# Some useful commands

```
paulbrosmacbook:var paulbrown$ cd $HOME
paulbrosmacbook:~ paulbrown$ pwd
/Users/paulbrown
paulbrosmacbook:~ paulbrown$ ls -l
total 104240
drwxr-xr-x    2 paulbrown  staff  64 19 Jun  2017 Anaconda
drwxr-xr-x   29 paulbrown  staff  928 30 Nov  2017 Android
drwx------+  93 paulbrown  staff  2976  1 Nov 12:34 Documents
-rw-r--r--    1 paulbrown  staff  8 22 Feb  2018 README.md
-rw-r--r--    1 paulbrown  staff  0 14 Jul  2015 mcmc.csv
lrwxr-xr-x    1 paulbrown  staff  25 20 Sep  2016 meme -> /
Users/paulbrown/meme4.11
paulbrosmacbook:~ paulbrown$ chmod 755 Documents
paulbrosmacbook:~ paulbrown$ cp -r Android Android.backup
paulbrosmacbook:~ paulbrown$ rm –r Android
```

# Environment Variables

```
Nero:~paulbrown$ echo $PATH
/bin:/usr/bin:/usr/sbin:/sbin:/usr/local/bin:~/
bin
Nero:~paulbrown$ meme
-bash: meme: command not found
Nero:~paulbrown$ export PATH=$PATH:/usr/local/
meme/bin
Nero:~paulbrown$ echo $PATH
/bin:/usr/bin:/usr/sbin:/sbin:/usr/local/bin:~/
bin:/usr/local/meme/bin
Nero:~paulbrown$ meme
 USAGE:
      meme  <dataset> [optional arguments]
```

# More variables

- There are no variable types
- VARNAME is a reference
- $VARNAME is the value held there

```
Nero:~paulbrown$ echo PATH
PATH
```

- Use ${…} to access substrings

```
paul-browns-macbook:~ paulbrown$ STR="Hello world"
paul-browns-macbook:~ paulbrown$ echo ${STR:6}
World
paul-browns-macbook:~ paulbrown$ echo ${STR/w/W}
Hello World
```

# Using quotation marks

- Important to know the difference between single and double quotes
- Expressions are evaluated inside “…”, but not inside ‘…’

```
paul-browns-macbook:~ paulbrown$ NAME="Paul"
paul-browns-macbook:~ paulbrown$ echo "Hello $NAME"
Hello PAUL
paul-browns-macbook:~ paulbrown$ echo 'Hello $NAME'
Hello $NAME
```

# Arrays

- Declaring an array

```
fruits=('Apple' 'Banana' 'Orange’)
```

- Accessing elements

```
echo ${fruits[0]}
```

# Reading files

cat, head, tail, more

```
nero:~ paulbrown$ grep “paulbrown” /var/log/secure
…
…
Nov  4 23:10:33 nero sshd[44146]:
pam_unix(sshd:session): session opened for user
paulbrown by (uid=0)
```

# Writing files

- A number of interactive text editors, eg vi, nano
- Also use re-direction >, >>
- echo “some content” >> script.sh

# Redirection

- Input to and output from command can be re-directed away from stdin and stdout
- Re-direct output to file

```
ls -l > dircontent.txt
```

Re-direct input from file

```
sort -k5 -n < dircontent.txt
```

# Redirection

Pipes are used to chain commands together so the output of one becomes the input of the next

```
tail –n 1000 logfile.log | sort | more
```

```
ls –l | sort –k5 –n
```

# Command substitution

- This allows the output of a command to be captured and used piped back to be used as an argument for something else, or to be captured in a variable
- Preferred way is to use `$(…)`

```
rm –f $(find . –name “*.txt”)
```

# Arithmetic expansion

## Use command substitution

```
paul-browns-macbook:~ paulbrown$ echo 2+3
2+3
paul-browns-macbook:~ paulbrown$ echo $((2+3))
5
paul-browns-macbook:~ paulbrown$ echo $(2+3)
-bash: 2+3: command not found
paul-browns-macbook:~ paulbrown$ let a=$((2+3))
paul-browns-macbook:~ paulbrown$ echo $a
5
```

Bash handles only integer types. Use bc to perform calculations with floating point types

```
paul-browns-macbook:~ paulbrown$ echo 'scale=3;4/3'  | bc
1.333
```

# Remote Shells

- rsh (remote shell). Do not use, insecure
- ssh (secure shell, port 22)

```
paul-browns-macbook:~ paulbrown$ ssh nero.wsbc.warwick.ac.uk
paulbrown@nero.wsbc.warwick.ac.uk's password:
Last login: Mon Nov  4 23:10:34 2019 from 95.149.133.253
-sh-4.1$ hostname
nero.wsbc.warwick.ac.uk
```

- Also sftp and scp

```
scp /local/stuff paulbrown@nero.wsbc.warwick.ac.uk:/home/paulbrown
```

# Shell scripting

- Conventionally, files have .sh extension
- Remember to set execute permission
- Script begins with

```
#!/bin/bash
```

# Input arguments

- Referred to as `$1`, `$2` etc..
- `$#` is the number of inputs
- Same applies to functions
- Use `read` to request user input

# Conditionals

- Surround an expression with [[ … ]]
- String operators : -z, -n, ==, !=, <, >, =~
- Numerical operators: -eq, -ne, -lt, -le, -gt, -ge
- File operators: -e, -f, -d, -r, -w, -x

# Conditionals

```
#!/bin/bash

if [[ $# -lt 3 ]] ; then
      echo “Not enough input arguments”
      exit 0
elif [[ $# -gt 5 ]] ; then
      echo “Too many input arguments”
      exit 0
else
      echo “OK”
fi
```

# Conditionals

- Can be chained together using logical operators &&, ||

```
#!/bin/bash

if [[ $# -lt 3 ]] || [[ $# -gt 5 ]]; then
        echo “Wrong number of input arguments”
        exit 0
else
        echo “OK”
fi
```

- These operators allow conditional execution

```
mkdir newdir || echo “Cannot create directory”
mkdir newdir && touch newdir/newfile
```

# While Loops

```
while read  line; do
    fields=(${line}) #expand to array
    …
done < infile
```

`break` and `continue` can be used within the loop body

# For loops

A for loop iterates a series of words in a string

```
for i in $(ls); do
        echo $i
done
```

A C-style for loop can be created using arithmetic expressions

```
for ((i = 0; i < 100; i++)); do
        echo $1
done
```

Range expression

```
for i in {1..10}; do
        echo $1
done
```

# Functions

```
myFunc() {
      local localVar=“Hello ”$1;
      echo localVar;
}
myFunc “Paul”
```

Return values can be captured by command substitution

# Getting help

- man pages for most commands
- Huge amount on online resources, eg a good cheat sheet at https://devhints.io/bash