Bash scripts

Douglas Scofield
Evolutionary Biology Centre and UPPMAX
douglas.scofield@ebc.uu.se

slides at
http://files.webb.uu.se/uploader/1372/Uppmax-Bash-Scripts-pptx.pdf

Bash scripts overview

- Why write a script?
- Bash variable substitution and variable names
- The first script
- Positional parameters
- Default values and checking values using ${...} constructs
- Making decisions with if statements
- File tests
- Tracing execution with –x
- Conditional execution with && and ||
- Looping with 'for'
- Looping with 'while read'
- Background processes and job control

What is a script, and why create one?

- A script is a file containing statements to be interpreted
- A Bash script contains statements for the Bash shell
  - familiar commands (grep, cat, etc.)
  - Bash syntax you are learning (< , > , | , $(...), ${...}, etc.)
  - Bash syntax for control flow (&& , || , if , for , & , wait , etc.)
  - Comments (lines that start with #)
- Python scripts, Perl scripts, etc.

- A script both describes and performs some process
  - it can be viewed without interpreting ("running") it
- A script's behaviour can be modified using parameters
- A script can be reused, by you or someone else

Bash variable substitution

- Normally, $VAR is replaced with the value of the variable
- This is also true within double quotes "..."
- This is **not** true within single quotes '...'

```
mbp: course $ VAR=fuzzy
mbp: course $ echo $VAR
fuzzy
mbp: course $ echo "$VAR"
fuzzy
mbp: course $ echo '$VAR'
$VAR
```

- Often, it is safest to enclose $VAR in double quotes, in case the value of VAR contains spaces
  - Bash could separate the value into space-delimited words otherwise

Bash variable names

- Bash variable names begin with a letter and contain letters, numbers and underscores '_'
- Proper substitution requires proper name recognition
- Use curly brackets ${VAR} to make the limits of the variable name explicit
- An underscore can also be preceded by a backslash to remove its 'part of a name' quality

```
mbp: course $ echo $VAR_file

mbp: course $ echo ${VAR}_file
fuzzy_file
mbp: course $ echo $VAR\_file
fuzzy_file
```

A first Bash script

- Go to the same directory where you created files ee, f, etc. yesterday

```
fb166: ~/course $ cat ee
this is a short file
this file is a little longer
```

- Create this, as 'script.sh', save it, and exit the editor

```
#!/bin/bash

cat ee
```

← this program will be used interpret the script when you run it **if** the shell finds '#!' at the beginning of the file. *to use PATH to find it*: `#!/usr/bin/env python3`

- The '.sh' is a convention meaning 'shell script' (Bash or Bourne)
  - Bash is an extension of Bourne shell, which is older and simpler
- Make it executable (`/bin/bash` will be used to interpret it)
  - `chmod +x script.sh`
- Run it!
  - `./script.sh`

```
fb166: ~/course $ ll
total 32
-rw-rw-r--@ 1 staff  21 Aug 21 12:36 e
-rw-rw-r--@ 1 staff  50 Aug 21 12:36 ee
-rw-rw-r--@ 1 staff  29 Aug 21 12:36 f
-rwxrwxr-x  1 staff  21 Aug 21 12:37 script.sh*
fb166: ~/course $ ./script.sh
this is a short file
this file is a little longer
```

Using a positional parameter

- Modify the script:

```
#!/bin/bash

FILE=$1
cat $FILE
```

You could also use `${1}` and `${FILE}`

- Run it with a parameter
  - `./script.sh ee` ← 'ee' is the first (only) positional parameter

```
fb166: ~/course $ ./script.sh ee
this is a short file
this file is a little longer
```

Try providing different parameters:

```
fb166: ~/course $ ./script.sh f
this file is a little longer
fb166: ~/course $ ./script.sh e
this is a short file
```

- Run it **without** a parameter
  - `./script.sh`
- Why does that happen?

```
fb166: ~/course $ ./script.sh
```

Optionally setting a parameter

- Modify the script:

```
#!/bin/bash

FILE=${1:-ee}
cat $FILE
```

`${1:-ee}` If $1 **is not set or is empty**, use 'ee' instead

It can be a variable: `${1:-$DEFAULT}`

- Run it with and without a parameter
  - `./script.sh f`
  - `./script.sh`
- We could also use `${1-ee}`, 'is not set' (without 'is empty')
  - a variable can be set but empty →
  - why do we not use this here?

```
fb166: ~/course $ cat test.sh
#!/bin/bash
TEST=
OUT1=${TEST:-out}
OUT2=${TEST-out}

fb166: ~/course $ bash -x ./test.sh
+ TEST=
+ OUT1=out
+ OUT2=
```

'bash –x' uses Bash to interpret the script, and instructs Bash to print lines as they are interpreted.

Produce an error if a parameter is missing

- Modify the script:

```
#!/bin/bash

FILE=${1:?Please provide a parameter}
cat $FILE
```

- `${VAR:?msg}` means exit with *msg* as an error if VAR is not set or is empty
- Run it with and without a parameter
  - `./script.sh f`
  - `./script.sh`

```
fb166: ~/course $ ./script.sh f
this file is a little longer
fb166: ~/course $ ./script.sh
./script.sh: line 3: 1: Please provide a parameter
```

- We could also leave off the colon, `${1?...}`, 'is not set'

There are many other ${...} features

- Yesterday I covered these for removing suffixes and prefixes
  - `${VAR%suff}`, `${VAR%%suff}`, `${VAR#pref}`, `${VAR##pref}`
- Assign a value to VAR if it is missing with `${VAR:=value}`

```
fb166: ~/course $ cat assign.sh
#!/bin/bash
DIR=
echo "The directory to use is ${DIR:-/home/douglas}"
echo $DIR
echo "The directory to use is ${DIR:=/home/douglas}"
echo $DIR
fb166: ~/course $ ./assign.sh
The directory to use is /home/douglas

The directory to use is /home/douglas
/home/douglas
```

- Many more
- This is called parameter expansion or parameter substitution
  - http://wiki.bash-hackers.org/syntax/pe
  - http://www.tldp.org/LDP/abs/html/parameter-substitution.html

Make a decision: if-then-else-fi

```
#!/bin/bash

FILE=${1:?Please provide a parameter}
if [[ "$FILE" == "f" ]]
then
    echo "Thank you, catting now..."
else
    echo "Parameter must be 'f'"
    exit 1
fi
cat $FILE
```

Double brackets
Space separation

Quoted in case of spaces

then, else, fi on separate lines

'exit 1' is failure
'exit 0' is success (default)

- Run it
  - ./script.sh f
  - ./script.sh ee
  - ./script.sh

```
fb166: ~/course $ ./script.sh f
Thank you, catting now...
this file is a little longer
fb166: ~/course $ ./script.sh ee
Parameter must be 'f'
fb166: ~/course $ ./script.sh
./script.sh: line 3: 1: Please provide a parameter
```

- http://www.tldp.org/LDP/abs/html/comparison-ops.html

Make a decision: if-then-fi (simplified)

```
#!/bin/bash

FILE=${1:?Please provide a parameter}
if [[ "$FILE" != "f" ]]
then
    echo "Parameter must be 'f'"
    exit 1
fi
echo "Thank you, catting now..."
cat $FILE
```

Sense of test reversed

Failed test 'falls through'

- Run it
  - ./script.sh f
  - ./script.sh ee
  - ./script.sh

```
fb166: ~/course $ ./script.sh f
Thank you, catting now...
this file is a little longer
fb166: ~/course $ ./script.sh ee
Parameter must be 'f'
fb166: ~/course $ ./script.sh
./script.sh: line 3: 1: Please provide a parameter
```

Testing for file conditions

```
#!/bin/bash

FILE=${1:?Please provide a parameter}
if [[ ! -e "$FILE" ]] ; then
    echo "$FILE does not exist" ; exit 1
elif [[ -d "$FILE" ]] ; then
    echo "$FILE is a directory" ; exit 1
else
    echo "$FILE might be ok..."
fi
cat $FILE
```

Double brackets (use spaces!)

-e exists
! not
-d is a directory
-f is a regular file

Use ; to stack commands on one line, including if and then

elif combines else and if

- ./script.sh z
- mkdir thisdir
- ./script.sh thisdir
- ./script.sh ee

```
fb166: ~/course $ ./script.sh z
z does not exist
fb166: ~/course $ mkdir thisdir
fb166: ~/course $ ./script.sh thisdir
thisdir is a directory
fb166: ~/course $ ./script.sh ee
ee might be ok...
this is a short file
this file is a little longer
```

- Many others: http://tldp.org/LDP/Bash-Beginners-Guide/html/sect_07_01.html

Tracing what is happening: -x

- Use 'bash –x' to run the script
  - lines prefixed with '+' are statements as they are interpreted

```
#!/bin/bash

FILE=${1:?Please provide a parameter}
if [[ ! -e "$FILE" ]] ; then
    echo "$FILE does not exist" ; exit 1
elif [[ -d "$FILE" ]] ; then
    echo "$FILE is a directory" ; exit 1
else
    echo "$FILE might be ok..."
fi
cat $FILE
```

```
rackham3: ~/course $ bash -x ./script.sh thisdir
+ '[' -z '' ']'
+ case "$-" in
+ __lmod_vx=x
```

some lines cut out

```
+ FILE=thisdir
+ [[ ! -e thisdir ]]
+ [[ -d thisdir ]]
+ echo 'thisdir is a directory'
thisdir is a directory
+ exit 1
```

- Use 'set –x' inside a script to enable it, 'set +x' to disable
  - focus on particular parts of a script

Run a command if another succeeded or failed

- Create the script 'success.sh':

```
#!/bin/bash
# comment: these are like mini if-then
cat ee f > zz && cat zz
cat zzz || echo "something went wrong with zzz"
```

&& perform the next command if the first **succeeded**
|| perform the next command if the first **failed**

- Run it
  - chmod +x success.sh
  - ./success.sh

```
mbp: course $ ./success.sh
this is a short file
this file is a little longer
this file is a little longer
cat: zzz: No such file or directory
something went wrong with zzz
```

- On the command line, separate commands with && instead of ; for safety, for example if results are required for following commands

Do something to multiple items: for loops

- Create the script 'loop.sh':

```
#!/bin/bash

for FILE in ee f thisdir
do
    if [[ -d "$FILE" ]] ; then
        echo "$FILE is a directory"
    fi
done
```

Items in this list are assigned to FILE one after the other, and the statements between do ... done are interpreted for each

- Run it
  - chmod +x loop.sh
  - ./loop.sh

```
mbp: course $ ./loop.sh
thisdir is a directory
```

For loops can use wildcards for the list

- Modify the script 'loop.sh':

```
#!/bin/bash

for FILE in * ; do
    test -d "$FILE" || echo "$FILE is not a directory"
done
```

`test -d FILE` is successful when `if [[ -d FILE ]] ; then ... fi` would be true

- * matches all files in the current directory
  - `./loop.sh`

```
mbp: course $ ./loop.sh
assign.sh is not a directory
e is not a directory
ee is not a directory
f is not a directory
loop.sh is not a directory
script.sh is not a directory
success.sh is not a directory
test.sh is not a directory
zz is not a directory
```

- Any wildcard expression can be used
- This can be very useful on the command line:
  - `for F in *.txt ; do mv "$F" "00_$F" ; done`

Loop over all parameters

- Modify to use "$@" for the list, which means all parameters

```
#!/bin/bash

echo "The name of this script is $0"
echo "There are $# parameters"

for FILE in "$@" ; do
    test -d "$FILE" && echo "$FILE is a directory"
done
```

Use "$@" and not $@ to wrap each parameter with " "

- Run it
  - `./loop.sh ee f`
  - `./loop.sh thisdir zz`
  - `./loop.sh *`

```
mbp: course $ ./loop.sh ee f
The name of this script is ./loop.sh
There are 2 parameters
mbp: course $ ./loop.sh thisdir zz
The name of this script is ./loop.sh
There are 2 parameters
thisdir is a directory
mbp: course $ ./loop.sh *
The name of this script is ./loop.sh
There are 10 parameters
thisdir is a directory
```

Loop while a condition holds: while loops

- Create the script 'while.sh'

```
#!/bin/bash

FILE=${1:?Please provide a file to read}
cat "$FILE" | while read -r LINE
do
    if [[ -f "$LINE" ]] ; then
        echo "$LINE is a file, working on $LINE ..."
        # other commands could go here
    fi
done
```

While there are lines left in $FILE, read each into LINE

- Run it
  - `ls *.sh > files`
  - `chmod +x while.sh`
  - `./while.sh files`

```
mbp: course $ ./while.sh files
Working on assign.sh ...
Working on loop.sh ...
Working on script.sh ...
Working on success.sh ...
Working on test.sh ...
Working on while.sh ...
```

Multiple things at once: background processes

- Typically a command is running in the **foreground**
  - the shell waits for it to complete before returning a prompt
- Commands can be run in the **background** using '&'
  - useful if the command might take a while to complete

```
mbp: course $ find . -name "*.sh" > output &
[1] 18503
mbp: course $
[1]+  Done                    find . -name "*.sh" > output
```

- Multiple commands can be run in the background
- Useful within a script, too
- Use '**wait**' to wait until all background processes are done
  - e.g., if background processes are creating files needed for a next step
  - without 'wait', a script can finish before its background processes
  - with SLURM on Uppmax, this will kill all user processes run by the job

Use job control to manipulate running processes

- Ctrl-c Kill the foreground process
- Ctrl-z Stop the foreground process
- bg Continue running stopped process but in background
- & Put new process in the background immediately
- jobs List background processes
- fg Move background process to foreground

```
mbp: course $ find / -name "*.sh" >allscripts 2>/dev/null
^Z
[1]+  Stopped                 find / -name "*.sh" > allscripts 2> /dev/null
mbp: course $ bg
[1]+ find / -name "*.sh" > allscripts 2> /dev/null &
mbp: course $ jobs
[1]+  Running                 find / -name "*.sh" > allscripts 2> /dev/null &
mbp: course $ fg
find / -name "*.sh" > allscripts 2> /dev/null
^C
mbp: course $ jobs
mbp: course $
```

https://www.gnu.org/software/bash/manual/html_node/Job-Control-Builtins.html

There is <u>much</u> more to learn about Bash

- Simple maths can be done within `(( ... ))` (without $)

```
mbp: course $ X=10
mbp: course $ (( X = X + 5 ))
mbp: course $ echo $X
15
```

- File dates: `if [[ "$FILE1" -nt "$FILE2" ]] ; then ... fi`
- A separate subshell can be created with `( ... )`
  - put it in the background: `( command1; command2 ) &`

- These slides contain enough to do many useful things
  - I rarely use more than this

http://linuxconfig.org/bash-scripting-tutorial
http://ryanstutorials.net/bash-scripting-tutorial/
http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html