Bash scripts

Marcus Holm

slides courtesy of:
Douglas Scofield
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
- Basics of programming
- Making decisions with if statements
- File tests
- Tracing execution with –x
- Conditional execution with && and ||
- Looping with 'for'
- Looping with 'while read’
- A script of your own
- 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 #)

• You can also write 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
```
Hands-on starts now

• Copy the contents of /proj/g2017016/labs/bash_scripts to an appropriate place in your home directory
  – e.g. ~/uppmax-intro/bash_scripts
• In this directory, you’ll be writing a new script from scratch
• Open a new file called ‘script.sh’
  – nano script.sh
  – vi script.sh
  – gedit script.sh &
A first Bash script

• Write the text below in ‘script.sh’, save it, and exit the editor

```bash
#!/bin/bash

cat file1
```

this program will be used interpret the script when you run it if the shell finds ‘#!’ at the beginning of the file, e.g., #!/usr/bin/python

• 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
Using a command-line parameter

• Modify the script:

```bash
#!bin/bash
FILE=$1
cat $FILE
```

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

• Run it with a parameter

  - `.script.sh file1`  'file1' is the first (only) parameter

• Run it with a few different parameters

• Run it **without** a parameter

  - `.script.sh`

• Why does that happen?
Optionally setting a parameter

- Modify the script:

```bash
#!/bin/bash
if [ $1 ]
then
  FILE=$1
else
  FILE="file1"
fi
cat $FILE
```

“If $1” i.e. if $1 is set and not empty, use $1, otherwise use “file1”

- Run it with and without a parameter
  - ./script.sh file2
  - ./script.sh

- This is a very common task, must we dedicate 6 lines to this “boilerplate”?
Optionally setting a parameter

• Modify the script:

```
#!/bin/bash
FILE=${1:-file1}
cat $FILE
```

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

• Run it with and without a parameter
  – `./script.sh file2`
  – `./script.sh`

• We could also use `${1-file1}`

• ‘is not set’ (without 'is empty')
  – a variable can be set but empty
  – why do we not use this here?

‘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:

```bash
#!/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 file2
  – ./script.sh

• We could also leave off the colon, `${1?...}`, ‘is not set’
There are many other `${...}` features

- Yesterday we covered these for removing suffixes and prefixes
  - `${VAR%suff}`, `${VAR%%suff}`, `${VAR#pref}`, `${VAR##pref}`
- Many more exist
- E.g. assign a value to VAR if it is missing with `${VAR:=value}`

- This is called parameter expansion or parameter substitution
Theory time: Basic Programming Constructs

• Scripts (and all programs) are built using a small number of building blocks.

• Execution control structures
  – Do stuff only in some cases (if-then)
  – Do stuff many times (loops)

• Variable manipulation
  – String operations
  – Arithmetic
  – Logical operations

• Input, output, and other system functions
How is programming done?

• There are many methods, but they all have a few things in common.
  – Start small
  – Work incrementally
  – Test your work as often as possible

• My script doesn’t work, how do I fix it?
  • You’re probably wondering “why doesn’t it work?”
  • The key to debugging is to ask, “what is it actually doing?”
  • Get up, grab a coffee in the break room, sit back down, and explain your script to your rubber ducky.
End of “lecture”

- Keep working through these slides at your own pace
- Ask for help when you get stuck or just have a question
#!/bin/bash

FILE=${1:?Please provide a parameter}
if [[ "$FILE" == "file2" ]]
then
    echo "Thank you, catting now..."
else
    echo "Parameter must be 'file2'"
    exit 1
fi
cat $FILE

- Run it
  - ./script.sh file2
  - ./script.sh file1
  - ./script.sh

- Double brackets [[..]] are flexible syntax and better in most circumstances, but in this case single brackets [..] also work.
Make a decision: if-then-fi (simplified)

#!/bin/bash

FILE=${1:?Please provide a parameter}
if [[ "FILE" != "file2" ]]
then
    echo "Parameter must be 'file2'"
    exit 1
fi

echo "Thank you, catting now..."
cat $FILE

• Run it
  – ./script.sh file2
  – ./script.sh file1
  – ./script.sh
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

- ./script.sh z
- mkdir thisdir
- ./script.sh thisdir
- ./script.sh file2

• Many others:
Tracing what is happening: -x

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

```bash
#!/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
```

- 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':

```bash
#!/bin/bash
# comment: these are like mini if-then
# this is called "boolean short-circuiting"
cat file1 file2 > zz && cat zz
cat zzz || echo "something went wrong with zzz"
```

```bash
&& perform the next command if the first succeeded
|| perform the next command if the first failed
```

• Run it
  – chmod +x success.sh
  – ./success.sh

• Even on the command line, separate multiple 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':

```bash
#!/bin/bash

for FILE in file1 file2 thisdir
do
  if [ -d "$FILE" ] ; then
    echo "$FILE is a directory"
  fi
done
```

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

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

mbp: course $ ./loop.sh
thisdir is a directory
For loops can use wildcards for the list

- Write the script 'loop.sh':
  ```bash
  #!/bin/bash
  for FILE in * ; do
    test -d "$FILE" || echo "$FILE is not a directory"
  done
  ```

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

- 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
```

• Run it
  - ./loop.sh file1 file2
  - ./loop.sh thisdir zz
  - ./loop.sh *

Loop while a condition holds: while loops

• Create the script 'while.sh'

```bash
#!/bin/bash

MAX = 10000
NUM = 1
while [[ NUM -lt MAX ]]
  do
    echo $NUM
    NUM=$(( NUM + NUM ))
  done
```

• Run it
  – chmod +x while.sh
  – ./while.sh

• Experiment with more/less spaces around "$(" and "[["
Loop over lines in a file with “while read”

• Create the script 'while2.sh'

```bash
#!/bin/bash

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

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

• Run it
  – ls *.sh > files
  – chmod +x while2.sh
  – ./while2.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 ...
Putting the pieces together

• Now it’s time for you to write your own script.
• Below is a suggestion for this task, but if you have an idea of your own then go ahead and try to do it now!

• The task:
  – First, create a file containing a list of words (just make them up)
  – Then, write a script...
  – that takes a file name as a parameter and ...
  – reads the file and ...
  – creates a new file for each word in the file

• When you’re done with this, you can continue to practice by making modifications, e.g. handling errors or writing content into the files.
More useful Bash knowledge: 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

```bash
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 $ 
```

There is **much** more to learn about Bash

- Simple maths can be done within `(( ... ))` (without `$`)
  - Tru rewriting “while.sh”

```bash
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://ryanstutorials.net/bash-scripting-tutorial/](http://ryanstutorials.net/bash-scripting-tutorial/)
[http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html](http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html)