Introduction to Bash scripting

Video lecture

What is a (Bash) script?

- A text file with Bash commands, (including functions)
- Anything you can do in the command line, you can do in a script (and vice versa)
  - Except, you can give arguments to scripts - like you can to commands
- You can save commands in a text file, so that you can run them in the same order again
- You can give arguments to scripts, when you run them
- You can change the permissions of the text file, to make it “executable”
- Scripts start with a special line:
  `#!/bin/bash`
  which helps your operating system to understand that Bash should run it

Lemme see the code, already!

```
#!/bin/bash

echo "Welcome $USER"
echo -n "Time and date is "
date
echo -n "Your IP address is "
hostname -I
uname -n
```

Lemme see the code, already!

```
#!/bin/bash                          ← The “shebang”

echo "Welcome $USER"                 ← echo text and value of env. variable
echo -n "Time and date is "          ← echo text without newline
date                                 ← run a command
echo -n "Your IP address is "        ← echo text without newline
hostname -I                          ← run a command
uname -n                             ← run a command
```

Some useful techniques

- `echo -n` - print without newline - next command goes to the same line
- Command substitution - `$(`*`command`*`)` - creates the text from *`command`*
  `echo "Today is $(date +%A)"` (prints e.g. Today is Monday)
- Always use quotes around text
- Semicolon allows for two commands on the same line

Some useful techniques

- `echo -n` - print without newline - next command goes to the same line

```
$ echo -n "No newline after this: "
No newline after this: rikard@newdelli:~$
      ← notice the prompt's position
```

Some useful techniques

- `echo -n` - print without newline - next command goes to the same line
- Semicolon allows for two commands on the same line

```
$ echo -n "No newline after this: "; date
No newline after this: tis 17 sep 2019 11:05:57 CEST
$
```

Some useful techniques

- Command substitution - `$(`*command*`)` - creates the text from *command*
  `echo "Today is $(date +%A)"` (prints e.g. Today is Monday)

```
$ echo "These are logged into your computer: $(users)"
These are logged into your computer: rikard

$ echo "First line of script is: $(head -1 script.sh)"
First line of script is: #!/bin/bash
```

Some useful techniques

- Always use quotes around text

```
$ lwp-request -m HEAD http://wiki.juneday.se/mediawiki/index.php?search=script&title=Hello
[1] 3967
rikard@newdelli:~/itid/bash$
200 OK
Date: Tue, 17 Sep 2019 09:13:46 GMT
Server: Apache/2.4.34 (Red Hat) OpenSSL/1.0.2k-fips mod_auth_kerb/5.4 PHP/5.5.21
Content-Type: text/html; charset=UTF-8

[1]+  Done                    lwp-request -m HEAD
http://wiki.juneday.se/mediawiki/index.php?search=script

$ echo $title
Hello
$
```

Some useful techniques

- Always use quotes around text
  - Run the following command (without quotes)
  - You may need to press enter once
  - Explain what happened

```
$ wget http://wiki.juneday.se/mediawiki/index.php/File:FTP_license_plate.jpg&PWD=/tmp
```

Let’s make a short script

(Live - we’re showing how to create a text file with a script, saving it, changing permissions, running it)

script name will be hello.sh

Creating variables

- It’s good practice to create variables for constants
- Typically early in the script
- Allows you to change a value in one place, use it in many places

```
BACKUP_DIR="/home/rikard/backup/"
IP="130.241.135.117"
HOSTNAME="$(uname -n)"
BACKUP_FILE="backup_$(date +%Y-%m-%d).bak"
```

Using variables

- You have `NAME="Bengt"`
- To use it, put a `$` before the variable name:
  ```
  echo "The name is: $NAME"
  ```
- You can use curly braces around the name to distinguish it from text:
  ```
  prefix="pre"
  echo "${prefix}historical findings"
  prehistorical findings
  ```
- Without the curlies, bash would assume a variable called prefixhistorical:
  ```
  echo "$prefixhistorical"
  ```

Simplistic backup script

```
$ cat do_backup.sh
#!/bin/bash

FILE="welcome.sh"
DATE=$(date +%Y-%m-%d)
BACKUP="$FILE.bak.$DATE"

cp "$FILE" "$BACKUP"
gzip -9 "$BACKUP"
```

Change permissions of the file

```
$ chmod u+x do_backup.sh
```

Result

```
$ ls -l
total 12
-rwxrw-r-- 1 rikard rikard 119 aug  2 12:28 do_backup.sh
-rwxrw-r-- 1 rikard rikard 186 aug  2 12:08 welcome.sh
-rwxrw-r-- 1 rikard rikard 181 aug  2 12:29 welcome.sh.bak.2019-08-02.gz

# Yey! We saved five bytes from compression!
```

Why is the script not perfect?

- We now have a script for backing up exactly one, file (always the same)
- We have to write one such script for every file we want to backup...
- Solution?
- Make the script generic

Arguments to scripts

- Instead of “hard coding” the file name, we can require that the user running the script provides the name as an argument
- Arguments tell the script *what to do*
- Arguments to scripts end up in $1 $2 $3 ... ${10}, ${11} ... etc
- We can give descriptive names to the arguments inside the script:

```
FILE="$1"
BACKUP_DIR="$2"
```

Allowing any number of arguments

- Let’s say that we want our backup script to follow the following synopsis:
  `./do_backup`**`s`**`.sh` *`file1 [file2... fileN]`*
- We don’t know how many files the user will provide as arguments
- How do we know what to call each argument? $1, sure, but then?
- Solution: Loop through *all arguments* (as shown on the next slide)

Looping over all arguments

```
$ cat arguments.sh
#!/bin/bash

for ARG in "$@"
do
    echo "$ARG"
done

$ ./arguments.sh a b c d
a
b
c
d
```

General for loop syntax

```
for VAR in LIST; do COMMAND(S); done
```

```
# example:
$ for color in orange purple green; do echo "Color: $color";
done
Color: orange
Color: purple
Color: green
```

In a script, we use indentation instead of ;

```
for color in orange purple green
do
  echo "Color: $color"
done
# works also in the command line - you'll get the secondary
# prompt:
$ for color in orange purple green
> do
> echo "Color: $color"
> done
Color: orange
Color: purple
Color: green
```

Improving the backup script

- Let’s let the user provide the list of files to back up
- We need to have separate backup file names, why not use the original name for each file as part of the backup file name?
- We’ll allow one or more arguments
- What happens if the user provides no arguments? What should the script do then?
- What happens if the user provides file names that are not actually files?

When you have a list of “What ifs”...

- Then you need to use the if-statement
- Example: What if the user provides no arguments?
- Translates to (in pseudo code - not actual code)

```
if number of arguments is zero
then
  Tell the user that arguments are neede
  exit with an exit code != 0
fi
```

IF-statement general syntax

```
if command
then
 command(s)
elif command
then
 command(s)
else
 command(s)
fi
```

IF-statement general syntax

```
if [[ some_test ]]
then
 command(s)
elif [[ some_other_test ]]
then
 command(s)
else
 command(s)
fi
```

IF-statement general syntax shortest version

```
if [[ some_test ]]
then
 command(s)
fi
```

Number of arguments

- $# contains the number of arguments to the script, so we can use:

```
if [[ $# == 0 ]]
then
  echo "You need to provide at least one argument"
  exit 1
fi
```

Number of arguments - error messages to stderr

- Send error messages to stderr (file descriptor 2)

```
if [[ $# == 0 ]]
then
  echo "You need to provide at least one argument" >&2
  exit 1
fi
```

Improving the backup script - use file names

- We need to have separate backup file names, why not use the original name for each file as part of the backup file name?

```
DATE=$(date +%Y-%m-%d)
SUFFIX=".bak.$DATE"

# now we can do e.g.:
cp "$FILE" "$FILE$SUFFIX"
# to create a backup of, say,
# file.txt to file.txt.bak.2019-01-24
```

Improving the backup - what about missing files?

- What happens if the user provides file names that are not actually files?

Check if the file exists before copying it to the backup file name:

```
if [[ -e "$FILE" ]]
then
  cp "$FILE" "$FILE$SUFFIX"
else
  # Do something here, the file doesn’t exist!
fi
```

File tests (can be used e.g. in if-statements)

- `-e` file exists
- `-f` file is a *regular* file (not a directory or device file)
- `-s` file is not zero size
- `-d` file is a directory
- `-p` file is a pipe
- `-h` or `-L` file is a symbolic link
- `-S` file is a socket
- `-t` file (descriptor) is associated with a terminal device
- `-r` file has read permission (for the user running the test)
- `-w` file has write permission (for the user running the test)
- `-x` file has execute permission (for the user running the test)
- and more...

Source: https://www.tldp.org/LDP/abs/html/fto.html

Strategy for missing files

- The user can provide many files
- If many files are missing, you can create a list and print it at the end

```
    if [[ -e "$FILE" ]]
    then
        cp "$FILE" "$FILE$SUFFIX" && gzip -9 "$FILE$SUFFIX"
    else
        BAD_FILES="$BAD_FILES $FILE"
    fi
# Then you check if the BAD_FILES variable is empty
```

Checking if a variable is empty

- You can use the -z flag to check if a variable is emtpy:

```
if [[ -z "$BAD_FILES" ]]
then
   exit 0
else
    echo "These files were not found: $BAD_FILES"
    exit 1
fi
```

Putting things together

```
#!/bin/bash

DATE=$(date +%Y-%m-%d)
SUFFIX=".bak.$DATE"
BAD_FILES=""
BACKUP_DIR="${HOME}/backups"

for FILE in "$@"
do
    if [[ -e "$FILE" ]]
    then
        cp "$FILE" "${BACKUP_DIR}/${FILE}${SUFFIX}"
    else
        BAD_FILES="$BAD_FILES $FILE"
    fi
done
if [[ -z "$BAD_FILES" ]]
then
   exit 0
else
    echo "These files were not found: $BAD_FILES"
    exit 1
fi
```

Challenge for you

- Rewind (or use the video slides) to the previous slide with the full script
- What happens if the backup directory doesn’t exist?
- Write a solution which
  - Checks that the directory exists
  - Creates the directory if it doesn’t exist

What else is there?

- Functions - we have written about functions in Bash on the wiki
- traps - you can e.g. detect that the script was terminated
- reading from stdin - you can write a script so that it can be used as part of a pipeline
- while loops
- switch - an alternative to if
- arrays, associative arrays
- declare - you can add some properties to variables using declare
- and much more - most of the above is covered on the wiki (use the search function) or a search engine to find resources online

Useful tips

- Install `shellcheck` and use it to check your scripts - lots of useful feedback from that command:

```
$ shellcheck ./script.sh
In ./script.sh line 4:
echo $day
     ^-- SC2086: Double quote to prevent globbing and word splitting.
```