Table of Contents

Based on https://missing.csail.mit.edu/2020/shell-tools/

- shellcheck: handy tool to check your `bash` scripts (source)
  - install via `sudo apt-get install shellcheck`

Assignment and string delimiters

- =: assignment; n.b. no spaces

```
$ foo=bar
$ echo $foo
bar
```

- : space character performs argument splitting

```
$ foo = bar
Command 'foo' not found
```

- `$foo`: access value of variable `foo`
- `'` is for literal strings; bash performs no interpretation
- `"`: bash will substitute variable values

```
$ foo=bar
$ echo "Value is $foo"
Value is bar
$ echo 'Value is $foo'
$ Value is $foo
```

- `;`: separate multiple commands on a single line

Functions

bash allows you to define functions

```
mcd () {
    mkdir -p "$1"
    cd "$1"
}
```

- `$1`: first argument to script/function

Special Variables

bash has many special variables to refer to arguments, error codes etc. Reference list. - `$0`: script name - `$1` to `$9`: script arguments. `$1` is the first argument, … - `$@`: all arguments - `$#`: number of arguments - `$?`: return code of the previous command - `$$`: process id for the current script - `!!`: entire last command with arguments - when execution fails due to lack of permissions, quickly execute last command with sudo by doing `sudo !!` - `$_`: last argument from last command. If you are in an interactive shell, you can also quickly get this value by typing `Esc` followed by `.`

Streams

- `STDIN`: standard input stream
- `STDOUT`: standard output stream
- `STDERR`: standard error output stream

Stream redirection

- `1> foo`: redirect `STDOUT` to `foo`
- `2> foo`: redirect `STDERR` to `foo`
- `&> foo`: redirect both `STDOUT` and `STDERR` to `foo`

Return codes and conditional execution

- `0`: everything executed correctly
- otherwise: an error occurred
- **`true`** program always has 0 return code
- **`false`** programs always has 1 return code
- Exit codes can be used to conditionally execute commands:
  - `&&`: and
  - `||`: or

```
1 $ false || echo "Oops, fail"
2 Oops, fail
```

As false returns 1, `echo` is executed

```
1 $ true || echo "Will not be printed"
2 $
```

As true returns 0, echo will not be executed

```
1 $ true && echo "Things went well"
2  Things went well
```

The second command executes as the first runs without errors

```
1 $ false && echo "Will not be printed"
2 $
```

The second command doesn't execute as the first returns an error state

```
1 $ false ; echo "This will always run"
2 This will always run
```

Command and process substitution

```
1 $ foo=$(pwd)
2 $ echo $foo
```

```
/home/user/
```

Here we are getting the output of command `pwd` and storing it as a variable.

- **Command substitution**: `$(CMD)` executes `CMD` and substitutes output of command in-place.
- `for file in $(ls)`: shell calls `ls`, then iterates over those values
- **process substitution**: `<(CMD)` executes `CMD` and places output in a temporary file, and substitutes with that file name. This is useful if command expects values to be passed by file instead of *STDIN*
- `diff <(ls foo)<(ls bar)`: show differences between files in dirs `foo` and `bar` by first creating files listing foo and bar, and then `diff`ing them

Here's an example that iterates through provided arguments `grep`s for string `foobar` and appends to the file as a comment if it isn't found

```
#!/bin/bash

echo "Starting program at $(date)" # Date will be substituted

echo "Running program $0 with $# arguments with pid $$"

for file in $@; do
    grep foobar $file > /dev/null 2> /dev/null
    # When pattern is not found, grep has exit status 1
    # We redirect STDOUT and STDERR to a null register since we do not care about them
    if [[ $? -ne 0 ]]; then     # if return code of last command is not equal to 0
        echo "File $file does not have any foobar, adding one"
        echo "# foobar" >> "$file"
    fi
done
```

- `/dev/null` is a special device you can write to and input will be discarded
- Comparisons reference: manpage for `test`
  - `-f`: if a file exists
- Use `[[ ]]` over `[ ]` as chances of mistakes are reduced, but this is not portable to `sh` (Explanation)

Shell globbing

Shell *globbing*: expanding expression e.g. filename expansion

- wildcards
  - `?`: match one character
  - `*`: match any number of chars
- `{}`: common substring expansion
  - e.g. `convert image.{png,jpg}` expands to `convert image.png image.jpg`

More examples: * `cp /path/to/project/{foo,bar,baz}.sh /newpath` expands to `cp /path/to/project/foo.sh /path/to/project/bar.sh /path/to/project/baz.sh /newpath` * Combinations: `mv *{.py,.sh} folder` moves all `*.py` and `*.sh` files to folder

This creates files foo/a, foo/b, … foo/h, bar/a, bar/b, … bar/h:

```
$ mkdir foo bar
$ touch {foo,bar}/{a..j}
$ touch foo/x bar/y
```

Show differences between files in foo and bar

```
$ diff <(ls foo) <(ls bar)
# Outputs
# < x
# ---
# > y
```

- `|`: pipes are also import in scripting; putting output of one program as input to next program
- `bash` scripting

Shebang

e.g. at start of script you will see line `#!/bin/bash` * `#!`: shebang/hashbang indicates this is to be interpreted as an executable file * `/bin/bash`: indicates the program loader should run this file, with the path to the current script as the first argument * best practice: write shebang lines using (`env`)[http://man7.org/linux/man-pages/man1/env.1.html] command to resolve to wherever the command lives in the system, maximising script portability. This will use `PATH` environment variable * e.g. `#!/usr/bin/env python`

Shell functions and scripts: the differences

- functions need to be in shell language; scripts can be in any language, hence the shebang
- functions are loaded when the definition is read; scripts are loaded at time of execution

- functions are executed in current shell environment; scripts execute in their own process
  - functions can modify environment variables e.g. change current directory, while scripts cannot
  - scripts are passed by value environment variables that have been exported using `export`

Shell Tools

Finding how to use commands

Given a command how do you go about finding out what it does and its different options? 1. Run with `-h` or `--help` flags 2. Look at `man` page for more detail 3. Interactive tools (e.g. ncurses): `:help` or `?` 4. TLDR pages focuses on giving example use cases of a command so you can quickly figure out which options to use. * e.g. tldr pages for `tar` and `ffmpeg`

Finding files

`find`

- `find` recursively searches for files matching some criteria. Some examples:

```
# Find all directories named src
find . -name src -type d
# Find all python files that have a folder named test in their path
find . -path '**/test/**/*.py' -type f
# Find all files modified in the last day
find . -mtime -1
# Find all zip files with size in range 500k to 10M
find . -size +500k -size -10M -name '*.tar.gz'
```

Find can also perform actions on the stuff it finds, which helps simplify monotonous tasks.

```
# Delete all files with .tmp extension
find . -name '*.tmp' -exec rm {} \;
# Find all PNG files and convert them to JPG
find . -name '*.png' -exec convert {} {.}.jpg \;
```

`fd`

`find` has tricky syntax: e.g. to find files that match some pattern `PATTERN` you have to execute `find -name '*PATTERN*'` (or `-iname` if you want the pattern matching to be case insensitive). * you can

start building aliases for those scenarios but as part of the shell philosophy is good to explore using alternatives. * you can find (or even write yourself) replacements for some. e.g. `fd` is a simple, fast and user-friendly alternative to `find`. * colorized output, default regex matching, Unicode support, more intuitive syntax * syntax to find a pattern `PATTERN` is `fd PATTERN`.

`locate`

`locate` * uses a compiled index/database for quickly searching * database that is updated using `updatedb`. * in most systems `updatedb` is updated daily via `cron` * trade-off compared to find/fd vs locate is between speed vs freshness. * `find` etc. can files using attributes (e.g. file size, modification time or file permissions) while `locate` just uses the name. * in depth comparison

Finding code

- You want to search for all files that contain some pattern, along with where pattern occurs in those files.
- `grep`: generic tool for matching patterns from input text
  - many flags, very versatile
  - `-C` for getting **C**ontext around the matching line. e.g. `grep -C 5` will print 5 lines before and after the match.
  - `-v` for in**v**erting the match, i.e. print all lines that do **not** match the pattern
  - `-R` **R**ecursively goes into directories and look for text files for the matching string

But `grep -R` can be improved in many ways, such as ignoring `.git` folders, using multi CPU support, etc. Various alternatives, but pretty much cover same use case: * ack * ag and rg. * Recommended: ripgrep (`rg`) as fast and intuitive

Some examples:

```
# Find all python files where I used the requests library
rg -t py 'import requests'
# Find all files (including hidden files) without a shebang line
rg -u --files-without-match "^#!"
# Find all matches of foo and print the following 5 lines
rg foo -A 5
# Print statistics of matches (# of matched lines and files )
rg --stats PATTERN
```

History: Finding shell commands

- Problem: you want to find specific commands you typed at some point.
- up arrow: gives you back your last command, slowly goes through your shell history
- `history`: lets you access your shell history programmatically; print your shell history to the standard output
  - we can pipe that output to `grep` and search for patterns
    * e.g. `history | grep find` will print commands with the substring "find".
- if you start a command with a leading space it won't be added to you shell history. useful when you are typing commands with passwords or other bits of sensitive information.

- If you make the mistake of not adding the leading space you can always manually remove the entry by editing your `.bash_history` or `.zhistory`.
- `Ctrl+R`: backwards search through your history.
  - After pressing `Ctrl+R` you can type a substring you want to match for commands in your history.
  - As you keep pressing it you will cycle through the matches in your history.
  - This can also be enabled with the UP/DOWN arrows in zsh.
  - A nice addition: fzf bindings. `fzf` is a general purpose fuzzy finder that can used with many commands. used to fuzzily match through your history and present results in a convenient and visually pleasing manner.

History-based autosuggestions

- First introduced in fish shell, this feature dynamically autocompletes your current shell command with the most recent command that you typed that shares a common prefix with it
- can be enabled in zsh and it is a great quality of life trick for your shell

Directory Navigation

- Problem: how to quickly navigate directories
- writing shell aliases
- creating symlinks with ln -s
- `fasd`: Find frequent and/or recent files and directories

  - Fasd ranks files and directories by *frecency* (both *frequency* and *recency*)
  - most straightforward use is *autojump* which adds a `z` command that you can use to quickly `cd` using a substring of a *frecent* directory. E.g. if you often go to `/home/user/files/cool_project` you can simply `z cool` to jump there.

More complex tools to get an overview of a directory structure * `tree` * `broot` * full fledged file managers: * `nnn` * `ranger`

Alternative Shells

`zsh`: Z shell is an extended version of the Bourne Shell (`sh`), based on bash zsh cheatsheet Pros: ref * automatic `cd`: just type name of directory * recursive path expansion: e.g. `/u/lo/b` expands to `/user/local/bin` * spelling correction and approx. completion * plugin and theme

Installation guide Install Oh My Zsh To get fonts working correctly using Ubuntu via WSL, had to install powerline fonts on Windows by cloning the repo and installing (ref):

```
>git clone https://github.com/powerline/fonts.git
>cd fonts
>.\install.ps1
```

Manually select a powerline font on the Ubuntu window for special characters to work. Change directory colours (select from here)

```
# using dircolors.ansi-dark
curl https://raw.githubusercontent.com/seebi/dircolors-solarized/master
    /dircolors.ansi-dark --output ~/.dircolors
## set colors for LS_COLORS
eval `dircolors ~/.dircolors`
```

NB configuration menu accessed by

```
$ autoload -Uz zsh-newuser-install
$ zsh-newuser-install -f
```

Exercises

1. Read `man ls` and write an `ls` command that lists files in the following manner

    - Includes all files, including hidden files
    - Sizes are listed in human readable format (e.g. 454M instead of 454279954)
    - Files are ordered by recency
    - Output is colorized

A sample output would look like this

```
-rw-r--r--   1 user group 1.1M Jan 14 09:53 baz
drwxr-xr-x   5 user group  160 Jan 14 09:53 .
-rw-r--r--   1 user group  514 Jan 14 06:42 bar
-rw-r--r--   1 user group 106M Jan 13 12:12 foo
drwx------+ 47 user group 1.5K Jan 12 18:08 ..
```

{% comment %} ls -lath –color=auto {% endcomment %}

2. Write bash functions `marco` and `polo` that do the following. Whenever you execute `marco` the current working directory should be saved in some manner, then when you execute `polo`, no matter what directory you are in, `polo` should `cd` you back to the directory where you executed `marco`. For ease of debugging you can write the code in a file `marco.sh` and (re)load the definitions to your shell by executing `source marco.sh`.

{% comment %} marco() { export MARCO=$(pwd) }

polo() { cd “$MARCO” } {% endcomment %}

1. Say you have a command that fails rarely. In order to debug it you need to capture its output but it can be time consuming to get a failure run. Write a bash script that runs the following script until it fails and captures its standard output and error streams to files and prints everything at the end. Bonus points if you can also report how many runs it took for the script to fail.

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

n=$(( RANDOM % 100 ))

if [[ n -eq 42 ]]; then
   echo "Something went wrong"
   >&2 echo "The error was using magic numbers"
   exit 1
fi

echo "Everything went according to plan"
```

{% comment %} #!/usr/bin/env bash

count=0 until [[“$?” -ne 0]]; do count=$((count+1)) ./random.sh &> out.txt done

echo “found error after $count runs” cat out.txt {% endcomment %}

1. As we covered in lecture `find`’s `-exec` can be very powerful for performing operations over the files we are searching for. However, what if we want to do something with **all** the files, like creating a zip file? As you have seen so far commands will take input from both arguments and STDIN. When piping commands, we are connecting STDOUT to STDIN, but some commands like

`tar` take inputs from arguments. To bridge this disconnect there's the `xargs` command which will execute a command using STDIN as arguments. For example `ls | xargs rm` will delete the files in the current directory.

Your task is to write a command that recursively finds all HTML files in the folder and makes a zip with them. Note that your command should work even if the files have spaces (hint: check `-d` flag for `xargs`) `bash find . -type f -name "*.html"| xargs -d '\n'tar -cvzf archive.tar.gz`

2. (Advanced) Write a command or script to recursively find the most recently modified file in a directory. More generally, can you list all files by recency?

Solutions

1. `$ ls -laht --color`

2.

```
marco() {
    export MARCO=$(pwd)
}

polo() {
    cd "$MARCO"
}
```

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

n=$(( RANDOM % 100 ))

if [[ n -eq 42 ]]; then
  echo "Something went wrong"
  >&2 echo "The error was using magic numbers"
  exit 1
fi

echo "Everything went according to plan"
```

3. First create some html files `bash $ mkdir foo bar $ touch {foo,bar}/{a..g}.html $ touch {a..g}.html $ find . -path "*.html"./a.html ./b.html ./bar/a.html ./bar/b.html ./bar/c.html ./bar/d.html ./bar/e.html ./bar/f.html ./bar/g.html ./c.html ./d.html ./e.html ./f.html ./foo/a.html ./foo/b.html ./foo/c.html ./foo/d.html ./foo/e.html ./foo/f.html ./foo/g.html ./g.html $ find . -name "*.html"| xargs -d '\n'tar -cvzf test.tar.gz`