ILLINOIS

Introduction to the Shell II

Lecture 2

Dirk Eddelbuettel

STAT 430: Data Science Programming Methods (Fall 2019)
Department of Statistics, University of Illinois

Last week

- General introduction to the Shell: why/how/where...
- We looked at ten key commands (plus a few more)
- Shell access via `terminal` window or via RStudio
- First look at the Gapminder data

Via the Website

- A compressed `tar` archive with files is available.
- Create new directory, change into it, download and unpack:

Via `git clone`

- Create a `git` repository on the command line
- Create new directory, change into and run `git clone`

Via a new RStudio

- Maybe the simplest way: start in your workspace
- Dropdown for new project allows 'New from repo'
- Paste repo URL, new project will be created

Last week we look at combining commands

```
# start in correct directory
cd gapminder/files_unfiltered
# compute line numbers per file
wc -l America/*csv
```

Try this!

And think about how we could sort the result.

Using the pipe

```
cd files_unfiltered
wc -l America/*csv | head -n -1
```

The `head -n -1` step shows everything but the last line.

This helps us to get rid of `total` as well.

This is one of those small differences where `bash` and related tools are slightly different from what you may see on macOS. We will use `bash` and the terminal in RStudio Cloud as the reference.

Using the pipe

```
cd files_unfiltered
wc -l America/*csv | head -n -1 | sort
```

Is this correct? If not, how can we make it correct?

Using the pipe

```
cd files_unfiltered
wc -l America/*csv | head -n -1 | sort -n
```

Numeric sorting. Also try `-r` for reverse.

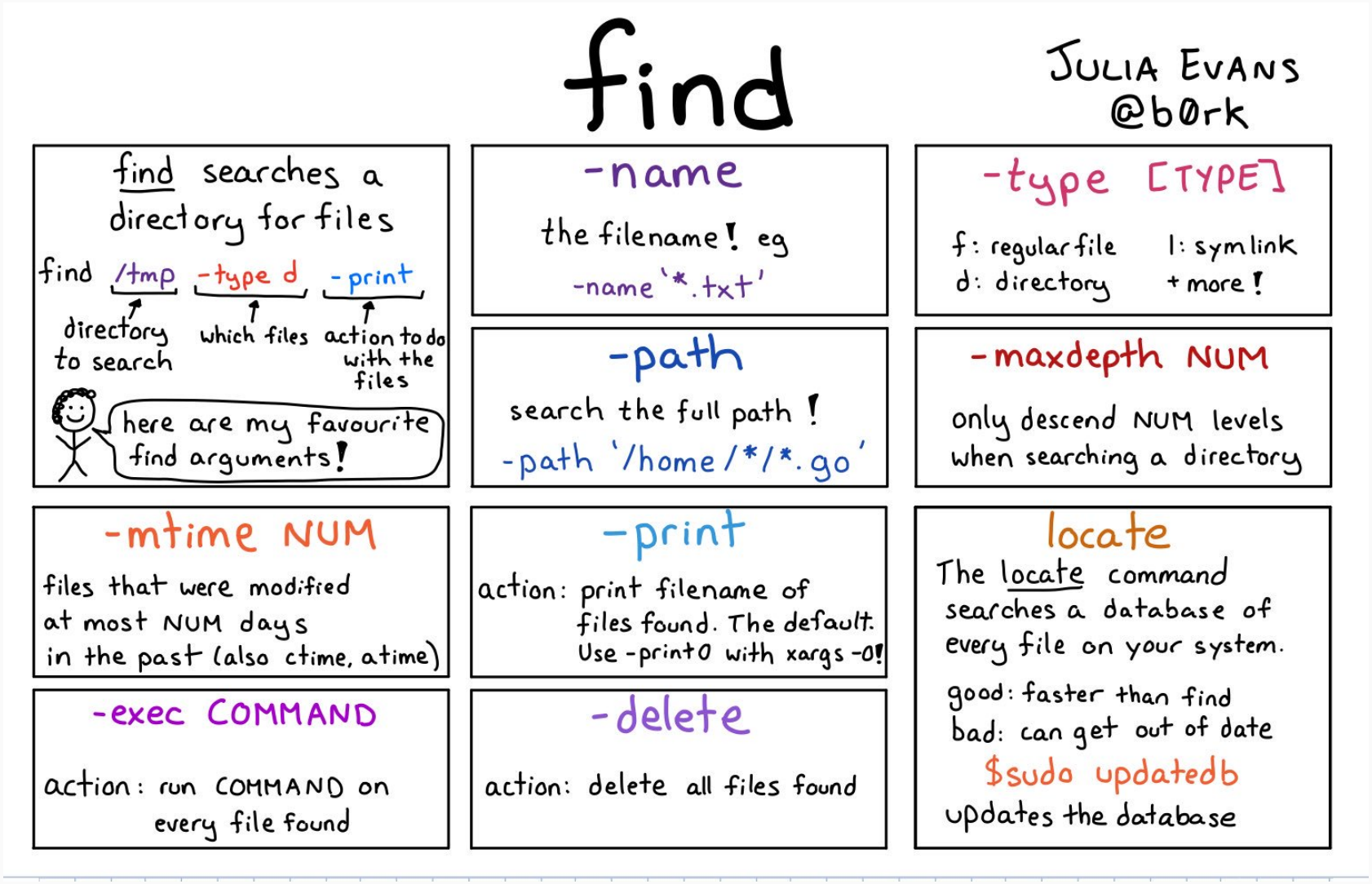
Using the pipe

```
cd files_unfiltered
wc -l America/*csv | head -n -1 | sort -n -r | head -20
```

The twenty longest data sets.

The `find` command

- `find` traverses directories applying tests
- Often used in pipes
- Useful arguments for
  - input types (files versus directories)
  - timestamps
  - modes
- Often used in conjunction with pipe
  - Either piped into `xargs`
  - Or using `exec`



Source: Julia Evans, https://twitter.com/b0rk/status/991880504805871616

find examples

Often used together with `xargs`:

```
find files_unfiltered/Asia -size +2k -type f | xargs wc
```

```
##    59    59 2544 files_unfiltered/Asia/Taiwan.csv
##    59    59 2563 files_unfiltered/Asia/Japan.csv
##   118   118 5107 total
```

This finds files with size of at least 2k bytes in or below the given directory.

find examples

One difficulty: file names with spaces! Small change:

```
cd files_unfiltered
find . -size +3k -type f -print0 | xargs --null wc
```

```
##    59   117 3131 ./Europe/Czech Republic.csv
##    59   117 3157 ./Europe/Slovak Republic.csv
##    58   115 3225 ./Americas/United States.csv
##   176   349 9513 total
```

We need `-print0` and `--null` to deal with space. (And we changed directories to make the displayed file names shorter.)

find examples

Another common task: files older / newer. (Cannot show this here as all example files have the same time stamp.)

```
# max and min days query
find /usr/local/lib/R/site-library/ \
         -maxdepth 1 -ctime +182 | wc -l
find /usr/local/lib/R/site-library/ \
         -maxdepth 1 -ctime -7 | wc -l
```

find examples

Previous slide showed, respectively, how many directories were

- created over a six months ago: `-ctime +DAYS`
- created within the last week: `-ctime -DAYS`
- where DAYS is the number of days

There are other operators

- `mtime` and `atime` for modified and accessed (but the latter is not always updated)
- all `*time` variants have `*min` equivalents for minutes

find examples

A useful idiom: Find old files matching a particular name:

```
# locate files 365 days or older matching *.backup
find dir -type f -ctime +365 -name \*.backup
```

We can use this 'as is' to have the output listed. Once assured that these are the files we want to operate on:

```
# locate files 365 days or older matching *.backup
find dir -type f -ctime +365 -name \*.backup | \
    xargs --null tar cvfz /tmp/archive.tar.gz
```

This command would archive them in a compressed tarfile in `/tmp`.

find examples

Continuing from the previous slide: once the archive is created to our content (so that we have copies should we need them)

```
# locate files 365 days or older matching *.backup
find dir -type f -ctime +365 -name \*.backup | \
    xargs rm -v
```

This command would delete them. Always be careful with `rm` and check twice.

Loops

Simple Task: How many files for continent?

```
for c in files/*; do
    echo -n "${c} has "
    ls -1 ${c} | wc -l
done
```

```
## files/Africa has 52
## files/Americas has 25
## files/Asia has 33
## files/Europe has 30
## files/Oceania has 2
```

Simple Task: How many files for continent?

```
for c in files/*; do
    echo -n "${c} has "
    ls -1 ${c} | wc -l
done
```

Analysis

- `for` loop with variable `c` over each entry in `files/*`
- `echo` displays to output, `-n` suppresses newline
- in a quoted string `${c}` refers to current value of `c`
- same as arguments to `ls` – here we count

Basic structure:

```
for var in expression; do
    command1
    command2
    ...
    commandN
done
```

Here the expression can be as simple as a fully emurated set – here three constant words

```
for var in tic tac toc; do
    echo "Var is ${var}"
done

## Var is tic
## Var is tac
## Var is toc
```

One key insight is that a simple shell wildcard expression *also creates such a list*. As `ls A*` returns the three continents we can do the same:

```
cd files
for var in A*; do
    echo "Var is ${var}"
done

## Var is Africa
## Var is Americas
## Var is Asia
```

An alternative form uses a sub-shell which issues an explicit command:

```
cd files
for var in $(ls -d A*); do
    echo "Var is ${var}"
done
```

```
## Var is Africa
## Var is Americas
## Var is Asia
```

Note that we tell `ls` to only list directories as opposed to listing inside them too.

Control Flow

Conditional code

- The `if` clause is the most common conditional
- There can be an optional `else` branch
- Each `if` in shell is terminated by a matching `fi`
- The conditional expression is in square brackets
- Comparison operators for numeric values
  - `-eq` and `-ne` for equal / not equal
  - `-lt` and `-gt` for strict less / greater than
  - `-le` and `-ge` for less equal / greater equal
- An example follows in a moment

Scripts

A very powerful (if misunderstood) notion

- In essence, if you can execute two or more commands ...

  ... then you can also create a shell script
- Shell scripts can be come "commands" you use
- A key way to automate processes

Minimal tutorial

- Use an editor and create a text file
- In RStudio: File -> New File -> Text File
- Add some commands and save under a new name, say `myscript.sh`
- Then do two important things:
  - In the first line add `#!/bin/bash` which adds the location of the `bash` shell program in a standardized ("shebang") format
  - Use the `chmod` command to set executable permissions: if the file is saved as `myscript.sh` use `chmod a+x myscript.sh`

Minimal example

```
#!/bin/bash
currentDir=$(pwd)
echo "We are currently in directory ${currentDir}"
```

Good idioms

- In the first few lines you can use
  - `set -e` to exit immediately on error
  - `set -u` to exit if an undeclared variable is referenced (typo ...)
  - `set -x` to see step by step flow
- Example follows

Good idioms

Try the following

```
#!/bin/bash
set -eux
pwd
today=$(date +%Y%m%d)
echo "Today is ${todate}"  # typo
echo "Never reached"
```

Run it, then correct the typo and run it again.

Conditional code (cont'ed)

- A simple example follows
- We assign the result of a shell command to a variable
- This uses the sub-shell command `$(...)` which is very useful
- Bases on the value we branch
- An `else if` is contracted to `elif` in shell
- We extract the first field delimited by space

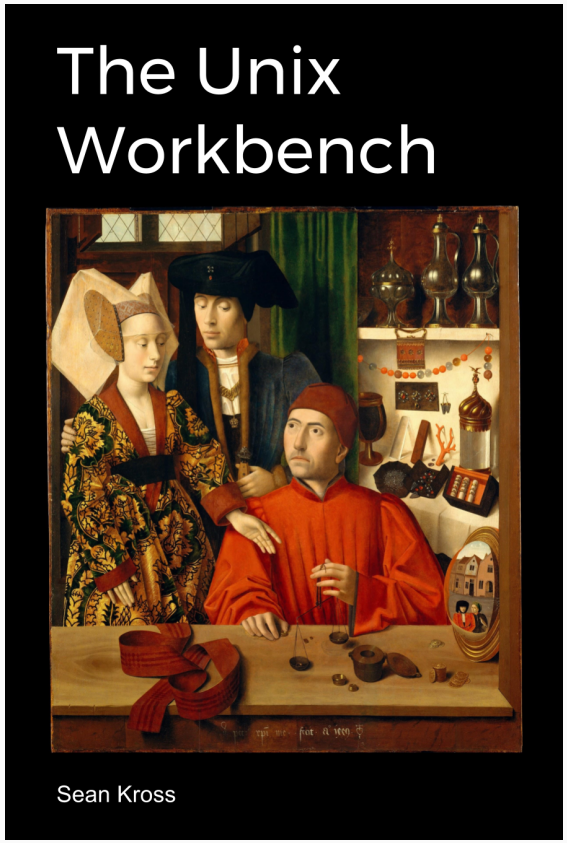
If / else Example

```
#!/bin/bash
set -eu
cd examples/gapminder/files_unfiltered

for file in Europe/*.csv; do
    length=$(wc -l "${file}" | cut -f 1 -d " ")
    if [ ${length} -lt 20 ]; then
        echo "Small file ${file}"
    elif [ ${length} -lt 40 ]; then
        echo "Medium file ${file}"
    else
        echo "Large file ${file}"
    fi
done
```

Write a shell script

- Include the 'shebang' in line 1: `#!/bin/bash`
- Run a command a two
- Maybe even add a loop
- Set the mode via `chmod`
- Run it



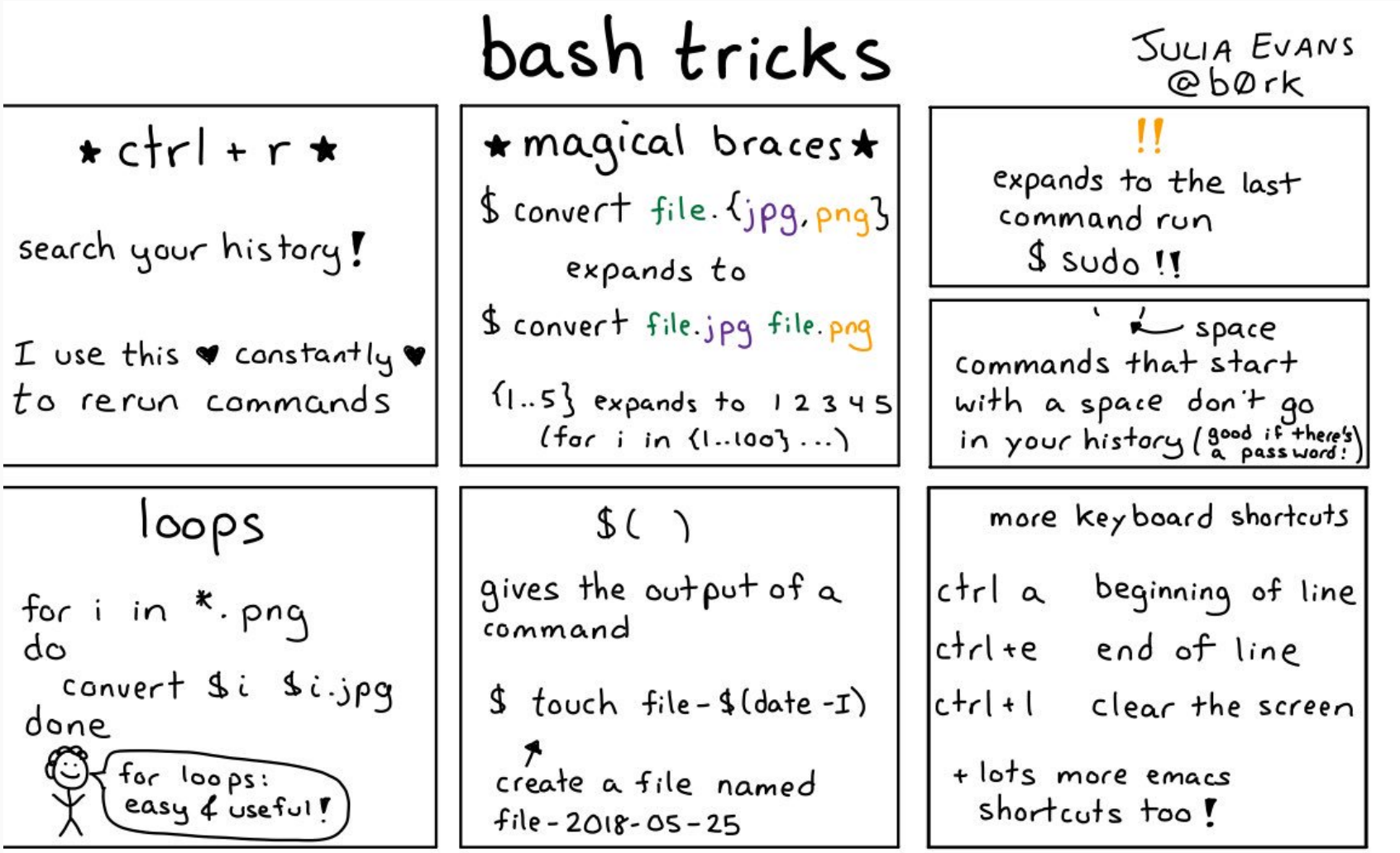
Please read (at least parts of) Chapter 5 on "Bash Programming" at https://seankross.com/the-unix-workbench/

Not everything in there is needed (*i.e.* Sections 5.1, 5.3, 5.5) are less relevant for us, but it does not hurt to skim them.

http://www.bashoneliners.com/ (medium to advanced)

https://opensource.com/article/18/5/bash-tricks

https://jvns.ca/blog/2017/03/26/bash-quirks/



Source: Julia Evans, https://twitter.com/b0rk/status/1000208860060307456

`diff` AND `patch`

ILLINOIS

- Two important tools for text comparison and updates:
  - `diff`
  - `patch`
- `diff` computes difference between (text, not binary) files
- `patch` takes the input of `diff` and alters files accordingly
- Together they give us means to programmatically alter files

Two files:

```
cat textOne.txt
```

```
## Red house
## Green car
## Blue sky
## Orange fruit
```

```
cat textTwo.txt
```

```
## Red house
## Grey cat
## Blue sky
## Black dog
## Orange fruit
```

diff AND patch EXAMPLE

Running `diff -u textOne.txt textTwo.txt` creates the following "unified" diff output:

```
--- textOne.txt 2018-11-11 16:13:11.325501900 -0600
+++ textTwo.txt 2018-11-11 16:27:14.820875044 -0600
@@ -1,4 +1,5 @@
 Red house
-Green car
+Grey cat
 Blue sky
+Black dog
 Orange fruit
```

The `diff` output denotes

- the two files and their timestamps,
- the range of lines 1 to 4 and 1 to 5,
- two additions and one removal

With `diff -u textOne.txt textTwo.txt > one2two.patch` we create a patch file.

With `patch --verbose < one2two.patch` we would alter file `textOne.txt` and make it identical to file `textTwo.txt`.

Why? Because we

- first computed the difference, or *delta* between one and two
- and then applies the delta – so the files will be identical

- This will seem trivial on these five lines files.
- But it recurses over directories
- It is not limited in the number of files.
- And the keys are the
  - *computable difference* and
  - *reliable application* of deltas, or "patches"
- This is at the heart of version control systems such as `git`.

Summary

The shell

- piping / composing commands
- the powerful `find` command
- shell loops: iterating over items
- control structure: `if` / `else` / `fi`
- shell scripts as a way to automate running shell commands
- `diff` to compute difference between files
- `patch` to apply these differences to alter files