Software Carpentry

@ Uni Trento

Marianne Corvellec Lachlan Deer

November 2018

Figure 1: Hello

Preliminaries

Introductions

- Instructors:
  - Marianne
  - Lachlan (me)
- Organizers:
  - Rafaello
  - (Others. . .)
- Helpers:
  - Gianfranco Abrusci, Mirko Bagnarol, Matteo Finazzer, Simone Orioli, Michele Turelli
- Learners:
  - You

Learning Objectives

- **Goal:** gain familiarity with a 'reproducible' computational workflow
  - How?: *hands-on* workshop
  - Coverage:
    - the Unix Shell: text based instructions
    - Git: Version control
    - Python
    - Reproducible Workflows: Make

Learning Objectives

We don't expect you to have any computational experience before we start

- By the end we want you to be able to:
  1. work with data in Python (summary/group statistics, plotting) ...
  2. ... in a way that is reproducible by yourself and others
- My number one coding tip:
  - Don't focus on the syntax, but on the way we think
    - you can google the syntax, hard to google how we think

Learning Environment

- I like learning environments that are *relaxed*, but *focussed*
  - Communicate as peers, there's no hierachy in this room
- Feel free to:
  - Interupt and ask questions
  - Share relevant experiences
  - Let us know when something isn't working for you
  - Talk to us in breaks
- Why have we given you sticky notes?
  - Letting us know how you are progressing
  - Providing us with feedback
- There is a code of conduct here
  - Summary: *Let's be nice to each other*

Structure

- Day One:
  - Morning (now-ish): Unix Shell
  - Afternoon: Python
- Day Two:
  - Morning: Git
  - Afternoon: Make
- Session Times:
  - Morning: 09.00 - 12.00
  - Lunch: 12.00 - 13.00
  - Afternoon: 13.00 - 14.00
- There are coffee breaks. . . roughly half way through a session
- Join us for lunch
- Open to arrange a drink this evening

All the links in one place:

- Course Website is here
- A community document (Etherpad) is here
- These slides are available here
- Pre-course survey is here
- Post-workshop survey is here

Unix shell

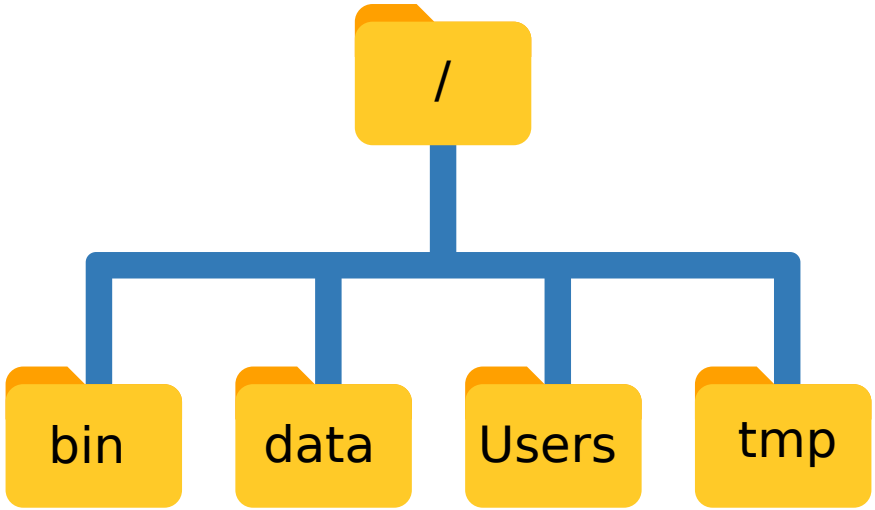

Figure 2: Your Filesystem

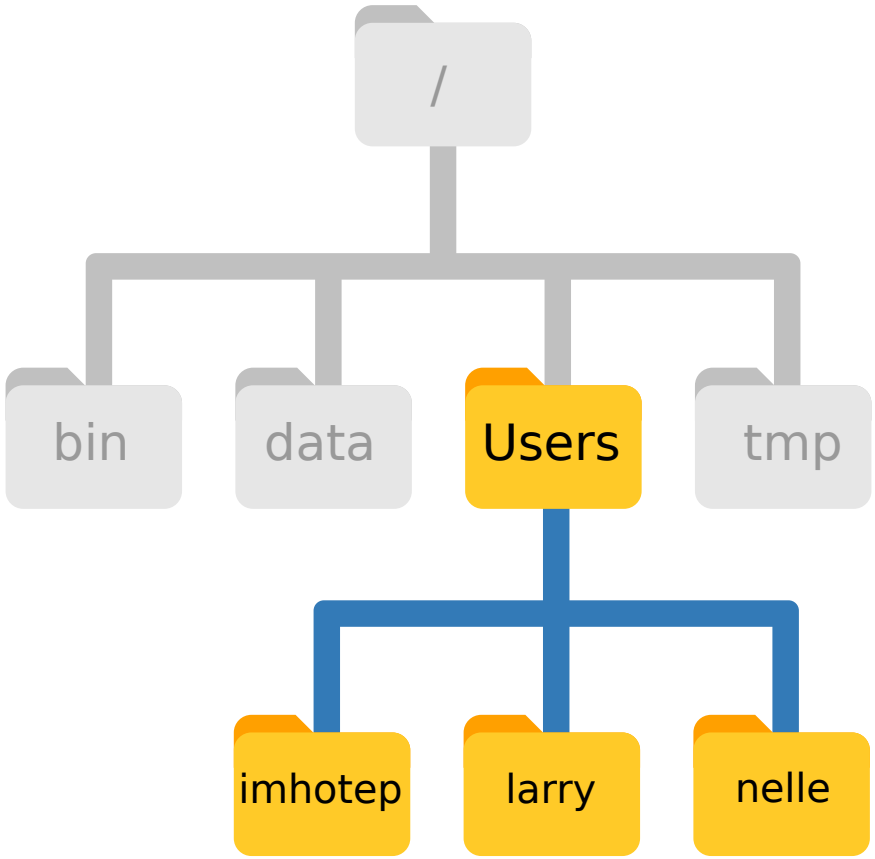

Figure 3: Home Directory Structure

Exercise: relative path resolution

Using the filesystem diagram below, if `pwd` displays `/Users/thing`, what is the output of `ls -F ../backup`?

1. `../backup: No such file or directory`
2. `2012-12-01 2013-01-08 2013-01-27`
3. `2012-12-01/ 2013-01-08/ 2013-01-27/`
4. `original/ pnas_final/ pnas_sub/`

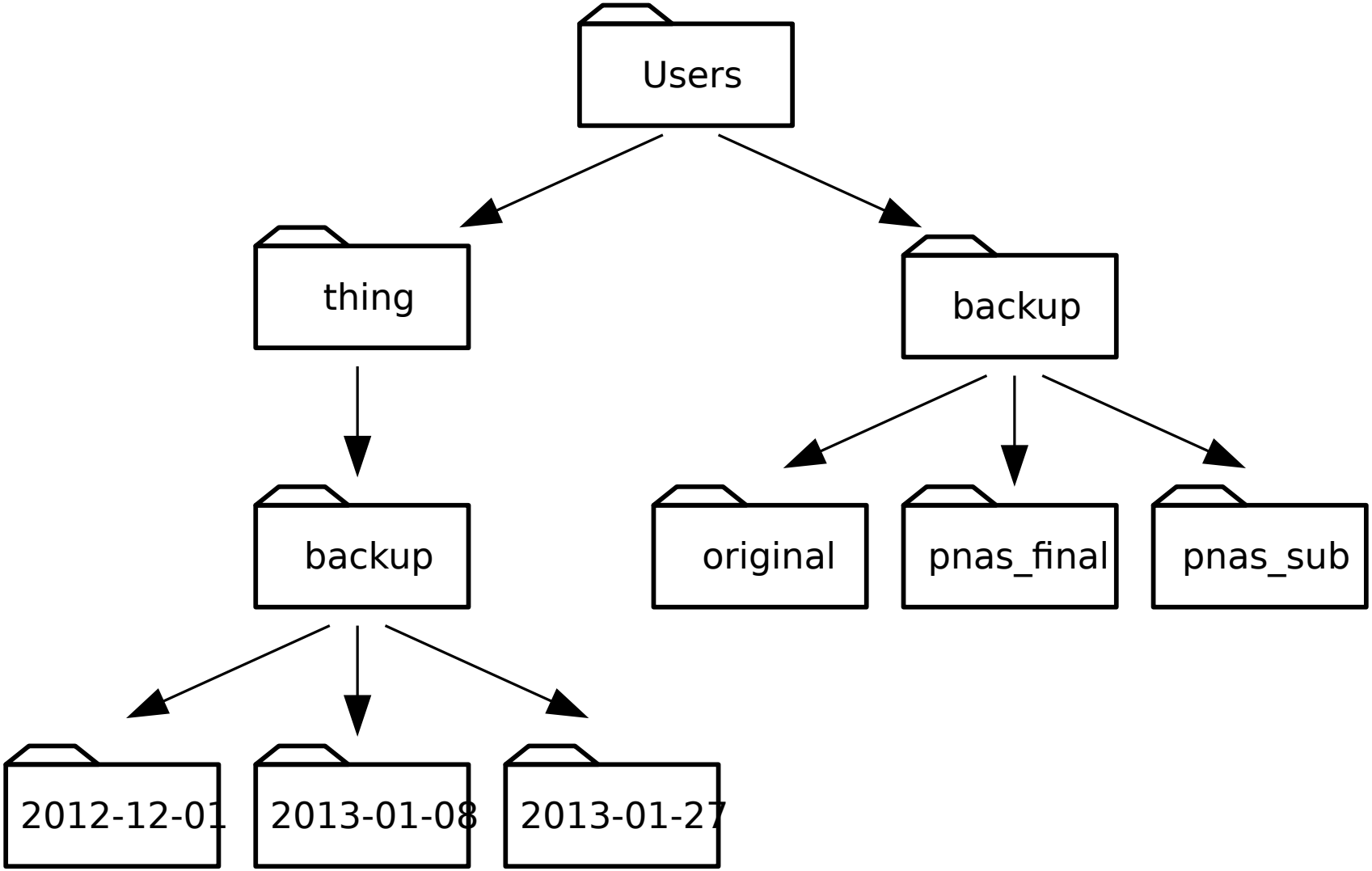

Figure 4: Example Filesystem

Solution

1. No: there is a directory backup in /Users.
2. No: this is the content of Users/thing/backup, but with .. we asked for one level further up.
3. No: see previous explanation.
4. Yes: ../backup/ refers to /Users/backup/.

Exercise: Renaming Files

Suppose that you created a .txt file in your current directory to contain a list of the statistical tests you will need to do to analyze your data, and named it: statstics.txt
After creating and saving this file you realize you misspelled the filename! You want to correct the mistake, which of the following commands could you use to do so?

1. `cp statstics.txt statistics.txt`
2. `mv statstics.txt statistics.txt`
3. `mv statstics.txt .`
4. `cp statstics.txt .`

Solution

1. No. While this would create a file with the correct name, the incorrectly named file still exists in the directory and would need to be deleted.
2. Yes, this would work to rename the file.
3. No, the period(.) indicates where to move the file, but does not provide a new file name; identical file names cannot be created.
4. No, the period(.) indicates where to copy the file, but does not provide a new file name; identical file names cannot be created.

Exercise: Moving and Copying

What is the output of the closing `ls` command in the sequence shown below?

```
$ pwd
/Users/jamie/data
$ ls
proteins.dat
$ mkdir recombine
$ mv proteins.dat recombine
$ cp recombine/proteins.dat ../proteins-saved.dat
$ ls
```

1. `proteins-saved.dat recombine`
2. `recombine`
3. `proteins.dat recombine`
4. `proteins-saved.dat`

Solution

1. No. proteins-saved.dat is located at `/Users/jamie`
2. Yes
3. No. proteins.dat is located at `/Users/jamie/data/recombine`
4. No. proteins-saved.dat is located at ‘`/Users/jamie`

Exercise: Copying Structure, without files

You're starting a new experiment, and would like to duplicate the file structure from your previous experiment without the data files so you can add new data.

Assume that the file structure is in a folder called '2016-05-18-data', which contains a data folder that in turn contains folders named `raw` and `processed` that contain data files. The goal is to copy the file structure of the `2016-05-18-data` folder into a folder called `2016-05-20-data` and remove the data files from the directory you just created.

Which of the following set of commands would achieve this objective? What would the other commands do?

Exercise: Copying Structure, without files (cont.)

Option One:

```
$ cp -r 2016-05-18-data/ 2016-05-20-data/
$ rm 2016-05-20-data/raw/*
$ rm 2016-05-20-data/processed/*
```

Option 2:

```
$ rm 2016-05-20-data/raw/*
$ rm 2016-05-20-data/processed/*
$ cp -r 2016-05-18-data/ 2016-5-20-data/
```

Option 3:

```
$ cp -r 2016-05-18-data/ 2016-05-20-data/
$ rm -r -i 2016-05-20-data/
```

Solution

1. Achieves this objective. First we have a recursive copy of a data folder. Then two rm commands which remove all files in the specified directories. The shell expands the '*' wild card to match all files and subdirectories.
2. The second set of commands have the wrong order: attempting to delete files which haven't yet been copied, followed by the recursive copy command which would copy them.
3. Would achieve the objective, but in a time-consuming way. The first command copies the directory recursively, but the second command deletes interactively, prompting for confirmation for each file and directory

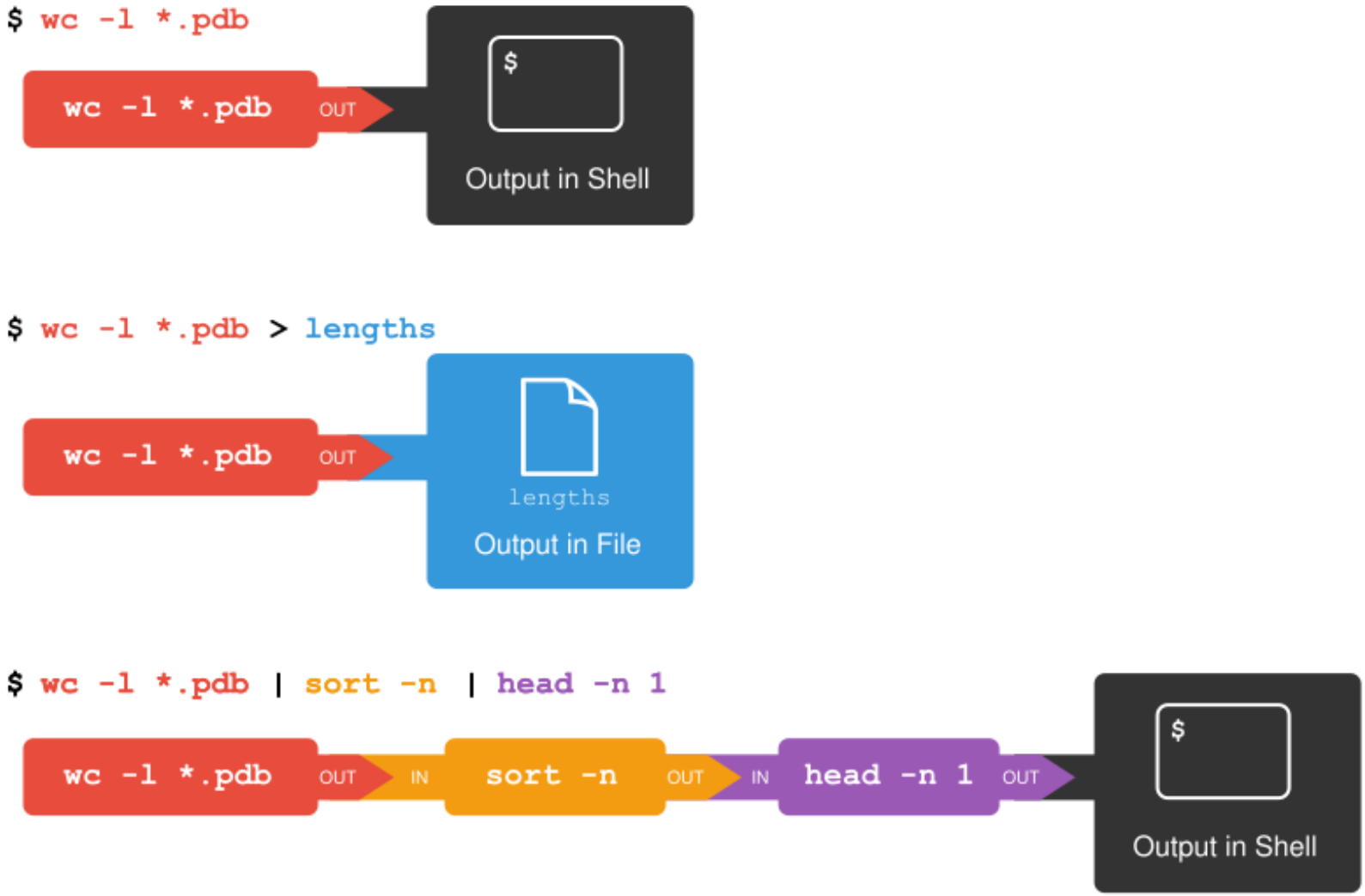

Figure 5: Pipe & Filter Logic

Exercise: Piping Commands Together

In our current directory, we want to find the 3 files which have the least number of lines. Which command listed below would work?

```
wc -l * > sort -n > head -n 3
wc -l * | sort -n | head -n 1-3
wc -l * | head -n 3 | sort -n
wc -l * | sort -n | head -n 3
```

Solution

Option 4 is the solution.

The pipe character | is used to feed the standard output from one process to the standard input of another. > is used to redirect standard output to a file.

Try it in the `data-shell/molecules` directory!

Exercise: Which pipe?

The file `animals.txt` contains 586 lines of data formatted as follows:

```
2012-11-05,deer
2012-11-05,rabbit
2012-11-05,raccoon
2012-11-06,rabbit
...
```

Assuming your current directory is `data-shell/data/`, what command would you use to produce a table that shows the total count of each type of animal in the file?

1. `grep {deer, rabbit, raccoon, deer, fox, bear} animals.txt | wc -l`
2. `sort animals.txt | uniq -c`
3. `sort -t, -k2,2 animals.txt | uniq -c`
4. `cut -d, -f 2 animals.txt | uniq -c`
5. `cut -d, -f 2 animals.txt | sort | uniq -c`
6. `cut -d, -f 2 animals.txt | sort | uniq -c | wc -l`

Solution

Option 5. is the correct answer.

If you have difficulty understanding why, try running the commands, or sub-sections of the pipelines (make sure you are in the `data-shell/data` directory).

Exercise: Saving Files in a loop

In the same directory, what is the effect of this loop?

```
for alkanes in *.pdb
do
    echo $alkanes
    cat $alkanes > alkanes.pdb
done
```

What if we replace > with >>?

Solution

Option 1. The text from each file in turn gets written to the alkanes.pdb file. However, the file gets overwritten on each loop interation, so the final content of alkanes.pdb is the text from the propane.pdb file

Replacing with >> leads to all output being printed into alkanes.pdb

Exercise: Doing a Dry Run

A loop is a way to do many things at once — or to make many mistakes at once if it does the wrong thing. One way to check what a loop would do is to echo the commands it would run instead of actually running them.
Suppose we want to preview the commands the following loop will execute without actually running those commands:

```
for file in *.pdb
do
  analyze $file > analyzed-$file
done
```

What is the difference between the two loops below, and which one would we want to run?

Exercise: Doing a Dry Run (cont.)

```
# Version 1
for file in *.pdb
do
  echo analyze $file > analyzed-$file
done
```

```
# Version 2
for file in *.pdb
do
  echo "analyze $file > analyzed-$file"
done
```

Solution

The second version is the one we want to run. This prints to screen everything enclosed in the quote marks, expanding the loop variable name because we have prefixed it with a dollar sign.

The first version redirects the output from the command echo analyze *filetoafile*, *analyzed*−file. A series of files is generated: analyzed-cubane.pdb, analyzed-ethane.pdb etc.

Try both versions for yourself to see the output! Be sure to open the analyzed-*.pdb files to view their contents.

Exercise: Script to List Unique Species

Leah has several hundred data files, each of which is formatted like this:

```
2013-11-05,deer,5
2013-11-05,rabbit,22
2013-11-05,raccoon,7
2013-11-06,rabbit,19
2013-11-06,deer,2
2013-11-06,fox,1
2013-11-07,rabbit,18
2013-11-07,bear,1
```

An example of this type of file is given in `data-shell/data/animal-counts/animals.txt`.

Write a shell script called `species.sh` that takes any number of filenames as command-line arguments, and uses `cut`, `sort`, and `uniq` to print a list of the unique species appearing in each of those files separately.

Solution

```
# Script to find unique species in csv files where species
# This script accepts any number of file names as command l

# Loop over all files
for file in $@
do
    echo "Unique species in $file:"
    # Extract species names
    cut -d , -f 2 $file | sort | uniq
done
```

Exercise: Debugging Scripts

Suppose you have saved the following script in a file called `do-errors.sh` in Nelle's north-pacific-gyre/2012-07-03 directory:

```
# Calculate stats for data files.
for datafile in "$@"
do
    echo $datfile
    bash goostats $datafile stats-$datafile
done
```

When you run it:

```
$ bash do-errors.sh NENE*[AB].txt
```

the output is blank. To figure out why, re-run the script using the -x option:

```
bash -x do-errors.sh NENE*[AB].txt
```

Solution

The -x flag causes bash to run in debug mode. This prints out each command as it is run, which will help you to locate errors. In this example, we can see that echo isn't printing anything. We have made a typo in the loop variable name, and the variable datfile doesn't exist, hence returning an empty string.