CCP4 / DIAMOND WORKSHOP 2021

LINUX COMMAND LINE SCRIPTING

CCP4 WORKSHOP IMPORTANT COMMANDS

- Get the workshop computing environment set up:
  `module load ccp4-workshop`
  (this will `cd` you to a good place to work)

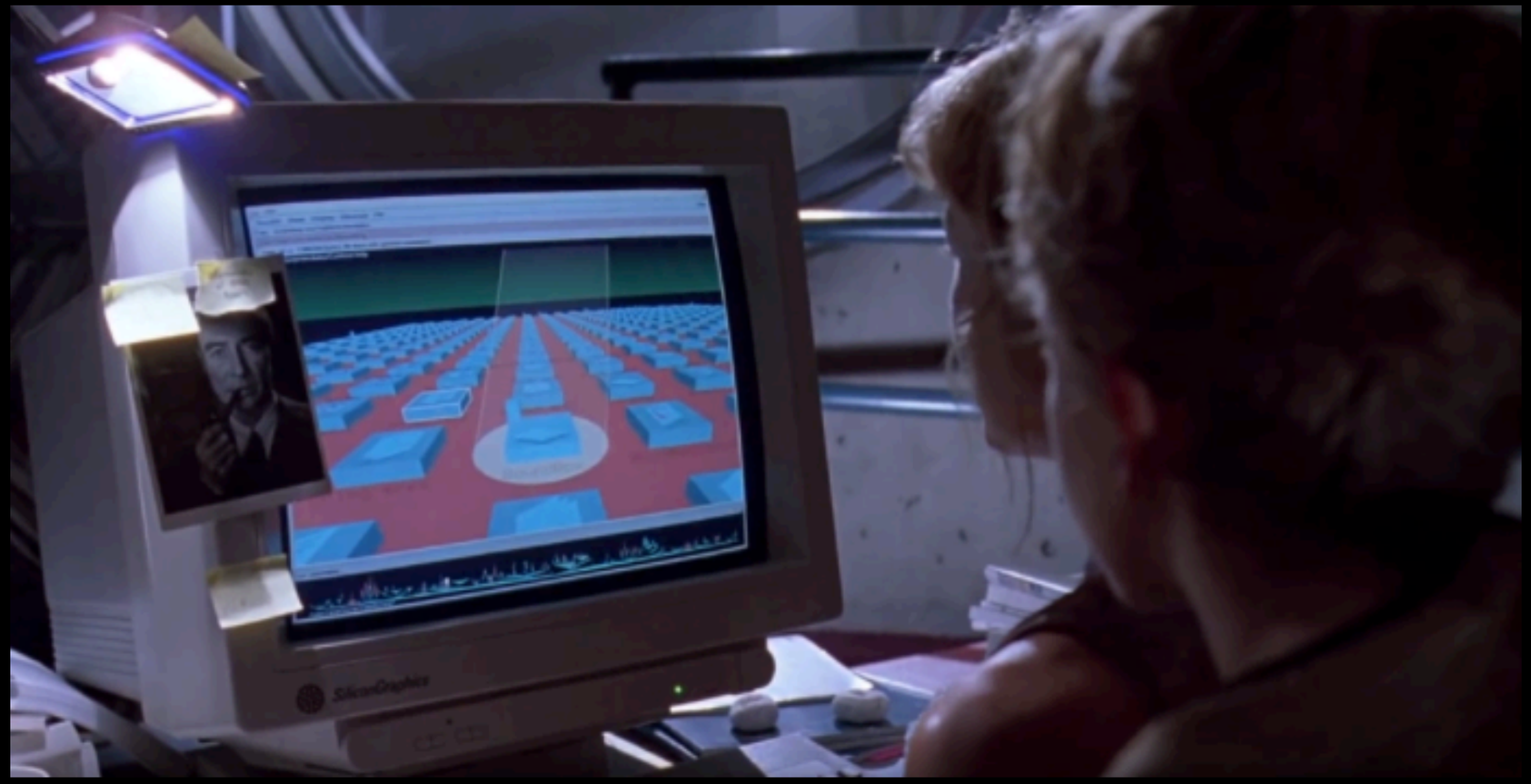
- Get to a processing node:
  `clusterme`
  which will put you on a processing node (rather than a login node) - good for “intensive processing”

WHAT IS LINUX / UNIX?

- Computer OS - started life in the 60's - developed by bearded 1960's hacker types
- Still alive today because it is flexible, useful and powerful, not because it is easy, friendly or fun
- Is what makes macOS, iPhone etc. work under the hood as well as Android

☝ original UNIX developers
(beards were mandatory)

THIS IS NOT HOW WE USE A UNIX SYSTEM (SORRY)



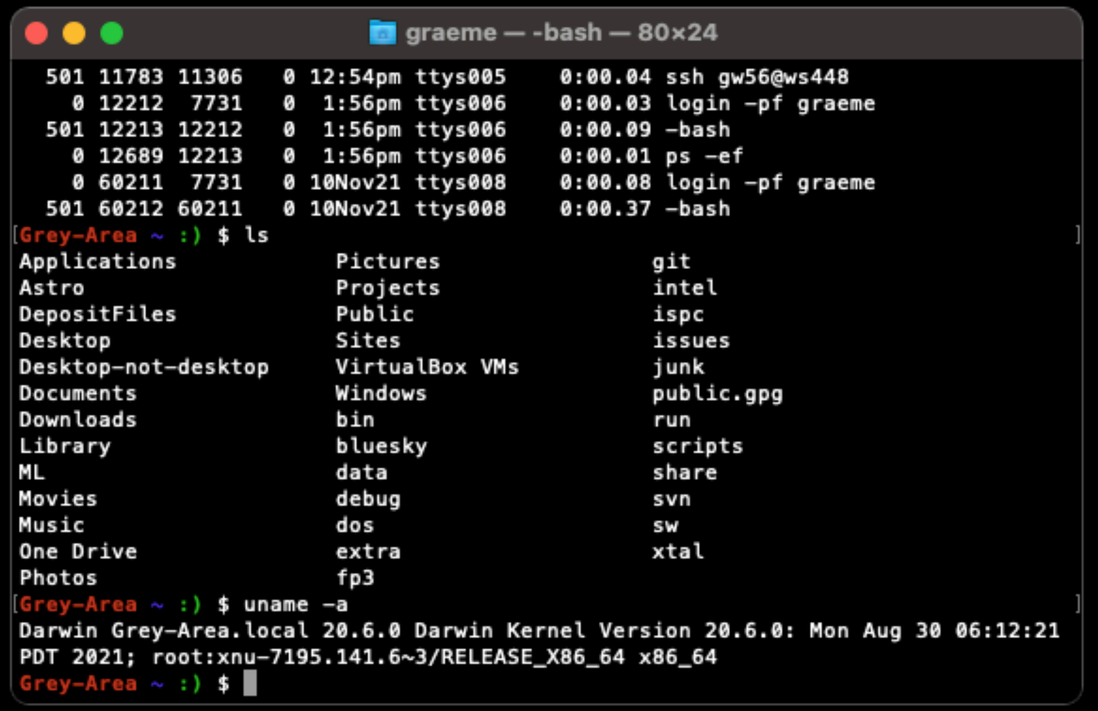
(at this point speaker realises this movie is like 30 years old)

THE UNIX PHILOSOPHY

- You don’t have bespoke programs to do stuff
- You have little tools you can *combine* to get the job done - often in multiple ways
- We usually combine tools with pipes | - these connect output of one program to input of another - I will talk more about these here
- Alternatively you can *replace* something with the output of something else
- Everything is a string but can sometimes be treated as a number

THE UNIX SHELL

- Program to allow you to “talk to” the computer
- Usually called something like bash or csh
- On Windows you have powershell and command prompt
- Looks like simple command interpreter but you can write programs
- Because this comes from ’60’s expect obscure nomenclature



IF YOU REMEMBER NOTHING ELSE...

Google: “How do I do {this thing} on the Linux command line”

KEY CONCEPTS

Programs have

- command line “arguments”
- “standard input” (i.e. input from the command line) - `stdin`
- “standard output” (i.e. output to the terminal) - `stdout`
- “standard error” i.e. output which is not “standard output” like error messages - `stderr`

KEY CONCEPTS

- *variables* - placeholder words - ${thing}
- *echo* - print something to the screen - echo “thing is ${thing}”
- *for / do* - do something multiple times - for thing in things; do echo ${thing}; done
- *mkdir / cd* - make directory / change directory - mkdir work; cd work
- *find* - look for files / directories - find . -name ‘thing’
- *pipes* - send what comes out of a program into another - echo thing | cat
- *sed / grep* - string processing tools - echo ${thing} | sed ’s/thing/th1ng/‘
- *sub shell* - $(command) - replace this token with the output of command

PRESENTATION STYLE

People words look like this

```
computer words look like this
```

TOY EXAMPLE

With whom am I sharing this computer? (not that we care)

```
$ who
eck56356 pts/0        2021-10-23 13:34 (x.x.x.x)
jgv29813 pts/1        2021-11-16 12:49 (x.x.x.x)
src77879 pts/3        2021-11-20 12:29 (x.x.x.x)
twi18192 pts/4        2021-11-20 11:39 (x.x.x.x)
src77879 pts/10       2021-11-20 09:19 (x.x.x.x)
eby15606 pts/12       2021-11-02 08:19 (x.x.x.x)
voo82358 pts/17       2021-11-19 18:54 (x.x.x.x)
gw56     pts/19       2021-11-20 15:49 (x.x.x.x)
igx79944 pts/29       2021-10-31 21:03 (x.x.x.x)
hqp77592 pts/30       2021-11-19 17:40 (x.x.x.x)
jfb65573 pts/34       2021-11-18 16:03 (x.x.x.x)
gw56     pts/14       2021-11-20 15:38 (x.x.x.x)
xmk93728 pts/15       2021-11-19 12:48 (x.x.x.x)
gw56     pts/42       2021-11-19 16:24 (x.x.x.x)
```

TOY EXAMPLE

With whom am I sharing this computer?

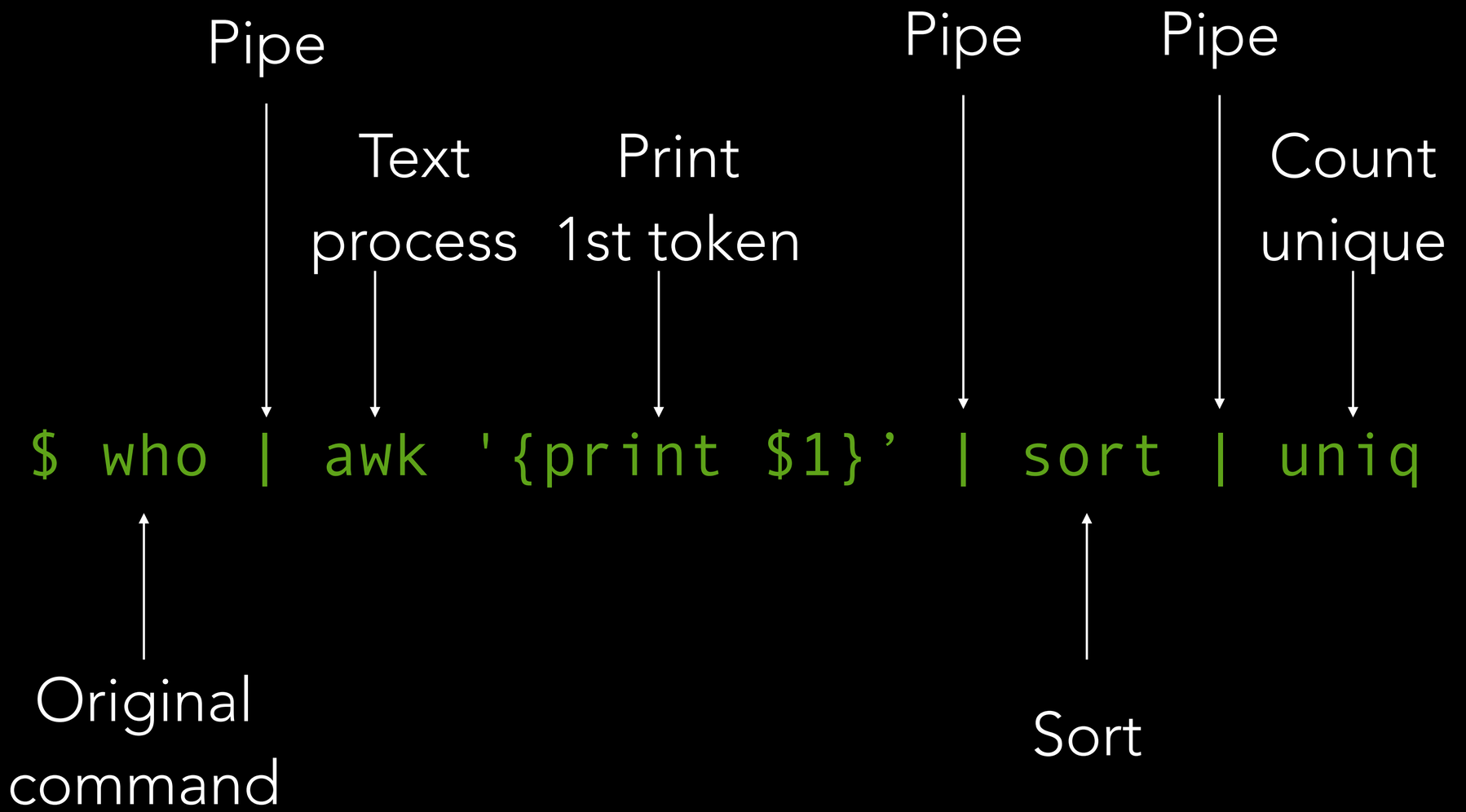
```
$ who | awk '{print $1}'
eck56356
jgv29813
src77879
twi18192
src77879
eby15606
voo82358
gw56
igx79944
hqp77592
jfb65573
gw56
xmk93728
gw56
```

TOY EXAMPLE

With whom am I sharing this computer?

```
$ who | awk '{print $1}' | sort | uniq -c
      1 eby15606
      1 eck56356
      3 gw56
      1 hqp77592
      1 igx79944
      1 jfb65573
      1 jgv29813
      2 src77879
      1 twi18192
      1 voo82358
      1 xmk93728
```

UNPACKING THIS COMMAND



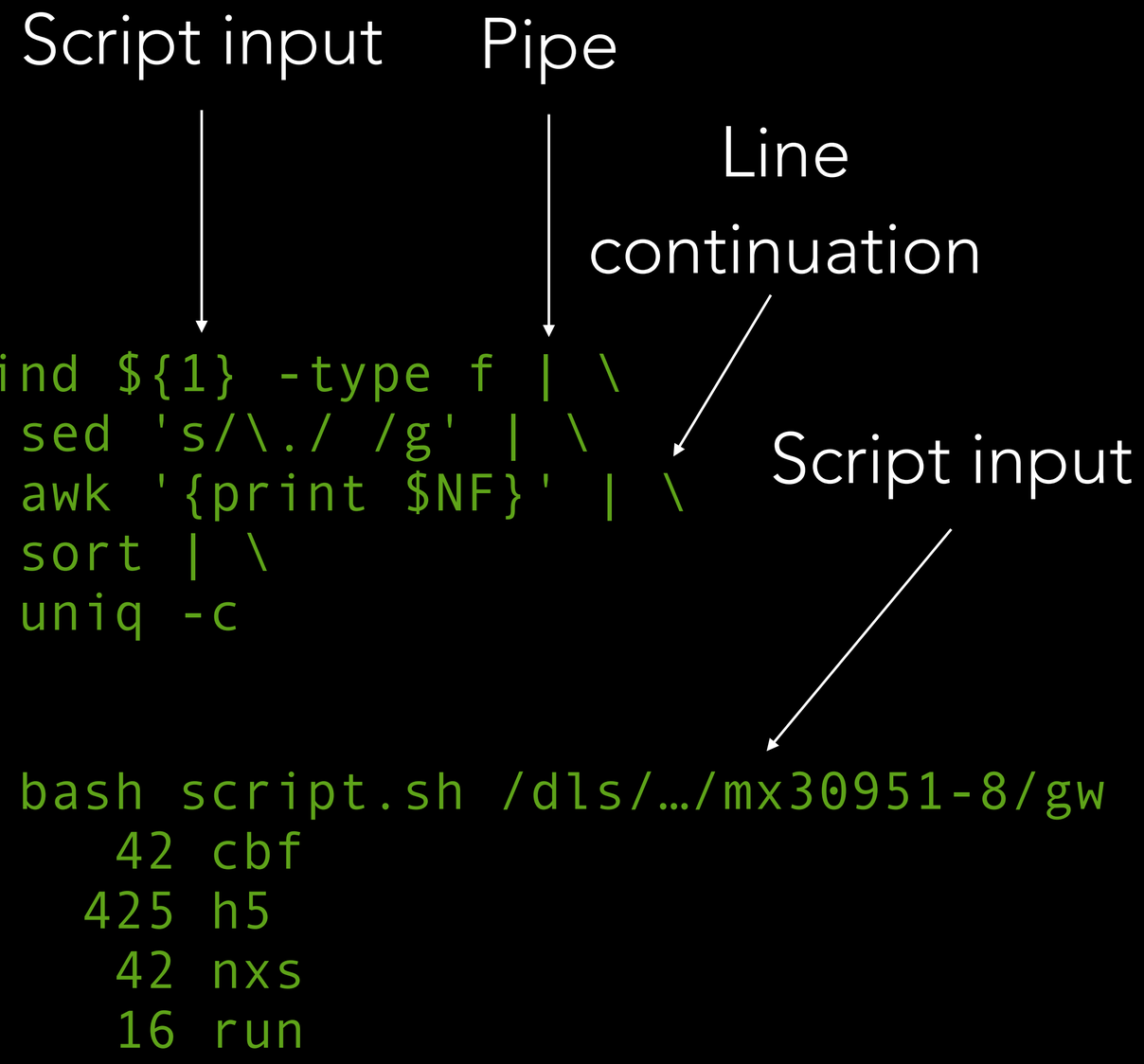
SHELL COMMANDS

```
$ who | awk '{print $1}' | sort | uniq -c
```

- This is a shell command - or a “one liner” - these are powerful
- The pipe `|` passes the output of `who` into `awk`, `awk` into `sort` etc.
- Usually I would write them one “pipe” at a time e.g. as demo just now worked
- You can use these in scripts or straight on the command line
- Learning how to use these means you carry the tools in your head

SHELL SCRIPTS

- Literally: a text file containing commands that the shell should perform
- Precisely the same as typing the commands on the shell, just saved in a text file so you can use them again
- Easier to work with longer command sequences



WHY SCRIPTS?

- So you can try something, modify it, and try it again
- So it is still there “next time”
- A useful record of what you actually did
- Most importantly: so you can perform complex operations over and over again without wearing out your fingers / mouse / sanity or making mistakes

REAL EXAMPLE

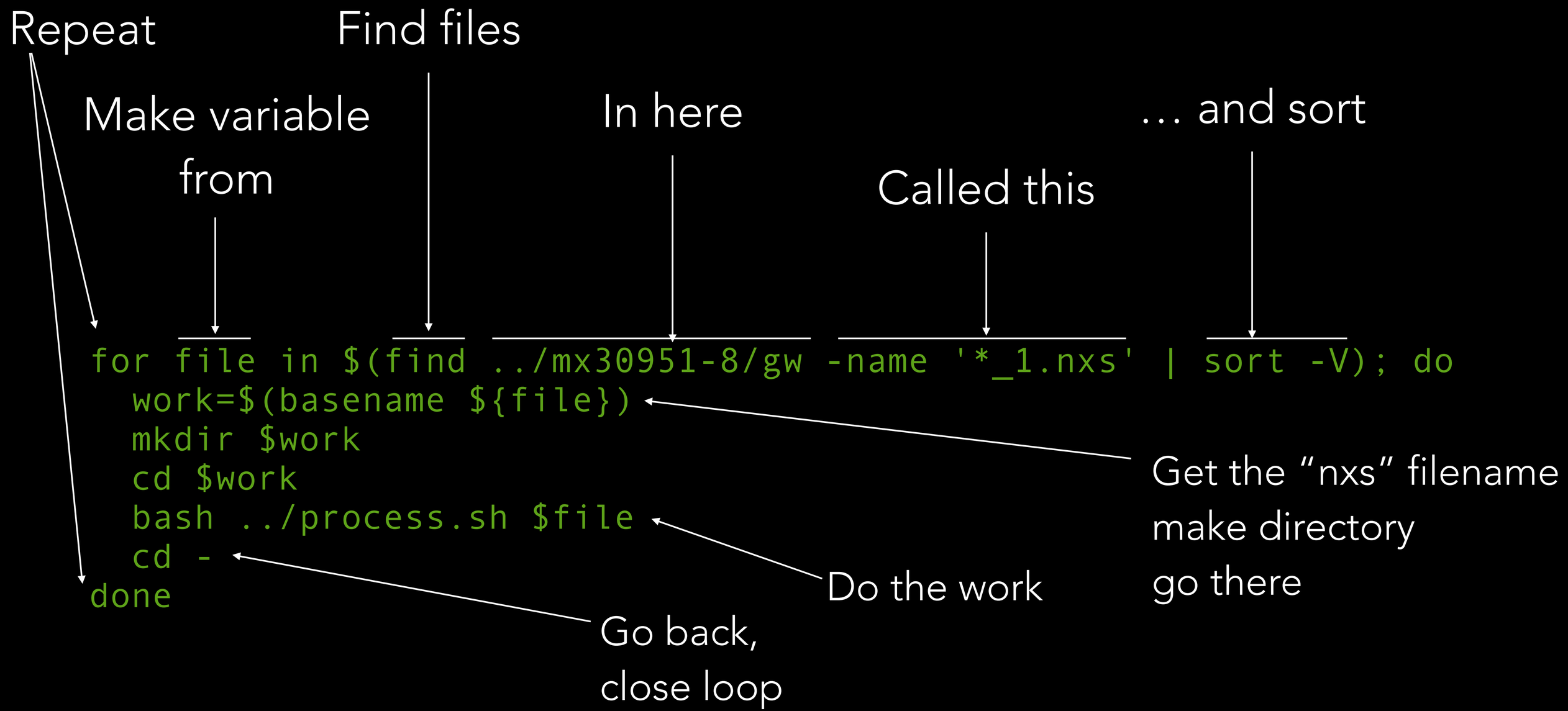
I want to process every data the same way (data processing next week!)

```
$ cat process.sh
dials.import ${1}
dials.find_spots imported.expt
dials.index imported.expt strong.refl
dials.refine indexed.expt indexed.refl
dials.integrate refined.expt refined.refl
dials.symmetry integrated.expt integrated.refl
dials.scale symmetrized.expt symmetrized.refl anomalous=true
dials.export scaled.expt scaled.refl
```

REAL EXAMPLE

I want to process every data the same way (data processing next week!)

```
$ cat runall.sh
for file in $(find ../mx30951-8/gw -name '*_1.nxs' | sort -V); do
  work=$(basename ${file})
  mkdir $work
  cd $work
  bash ../process.sh $file
  cd -
done
```


Repeat
Find files
Make variable
from
In here
Called this
… and sort
for file in $(find ../mx30951-8/gw -name '*_1.nxs' | sort -V); do
work=$(basename ${file})
mkdir $work
cd $work
bash ../process.sh $file
cd -
done
Get the “nxs” filename
make directory
go there
Do the work
Go back,
close loop

BUT WHY IS THIS USEFUL?

- The stuff in the loop can do a lot of work, in a consistent manner
- You can run all this work on every data set, missing none
- You can get the computer to do a lot of work for you, with you doing very little
- This does depend on you being systematic in how you work

SCRIPT VARIABLES

- Scripts have special variables -
  ```
  ${1} ${2} …
  ${@}
  ```

- These are the script *arguments* i.e. what followed the script on the command line

- You may find it useful to “alias” these in your script:
  ```
  IMAGE=${1}
  DIRECTORY=${2}
  ```

SCRIPT VARIABLES

process.sh

```
IMAGE=${1}
DIRECTORY=${2}
echo "Processing ${DIRECTORY}/${IMAGE}"
```

```
$ bash process.sh data_0001.cbf image/directory
Processing image/directory/data_0001.cbf
```

ENVIRONMENT VARIABLES

- Standard variables which come from “the environment” not your script
- See all of the current variables with `env`
- Contains words like USER, PWD and OLDPWD - you can use these in your scripts - for example:

```
echo ${USER}
mkdir ${PWD}/work
cd ${OLDPWD}
```

FILES

- (Text) file input and output can use “redirection” operators `>` and `<`
- Allows saving results to a file - can also use `tee` to save stuff to a file and *also* pass on

```
who | awk ‘{print $1}’ | sort | uniq -c | tee users.dat
```

- Alternatively can just pipe to a file with

```
who | awk ‘{print $1}’ | sort | uniq -c > users.dat
```

AN EXERCISE

- We will make a file containing the real names of everyone we are sharing the computer with, and save this in the visit processing area as our own user name (e.g. xxxNNNNN)
- We will do this only with the command line
- Instructions - https://tinyurl.com/4ck37kdz

- Run `module load ccp4-workshop` first so the result goes to the right place - if you get to the end you will find some more work to do

SUMMARY

- Shell scripting is a powerful way of getting repetitive work done
- It may feel arcane to start with but there are rules
- There is no need to memorise the how, remember the ideas and google the rest

AN EXERCISE

https://tinyurl.com/4ck37kdz