Shell Scripting

Dr. Ross Dickson, Computational Research Consultant

Imagine ...

...you have a program you're tweaking, and 200 input files to run every time it changes

```
$ numol <methane.in >methane.out
$ numol <ethane.in >ethane.out
$ numol <propane.in >propane.out
$ ...
```

How many kinds of wrong is this?

Get some files

```
$ mkdir Scripting
$ cd Scripting
$ pwd
~/Scripting
$ cp /home/rdickson/demo/Scripting/* .
$ ls
ethane.in    numol        propane.in
methane.in   numol-job    qstat
$
```

Submit jobs

```
$ cat numol_job
#$ -cwd
#$ -l h_rt=24:0:0
#$ -j yes
./numol < methane.in > methane.out
$ qsub numol_job
Your job 6254997 (“numol_job”) has been
submitted.
```

Only need one job script

```
$ cat numol_job
#$ -cwd
#$ -l h_rt=24:0:0
#$ -j yes
./numol < $1.in > $1.out
$ bash numol_job methane
$ qsub numol_job methane
```

`$1` is a **positional argument**,
a special case of a shell **variable**

Shell variables

```
$ pet=fido
$ legs=4
$ echo $pet, $legs
fido, 4
```

Set **variables** with '='.
(Notice no spaces around '=')
Get value with '**$**'.
'**echo**' is the analogue of 'print'.

Use script over and over...

```
$ for molecule in ethane methane propane
> do
>     qsub numol-job $molecule
> done
Your job...has been submitted...
$
```

Keywords **for ... in ... do ... done**.
('>' here is the *secondary prompt*,
not the redirection operator.)

Make a list of molecules

```
$ ls *.in >test-set
$ vim test-set
ethane.in
methane.in
propane.in
~
:1.$s/.in$//
```

There are advantages to learning a full-featured editor like vim or emacs. Global search-and-replace is one.

Use the list

```
$ for molecule in $(echo test-set)
> do
>     qsub numol-job $molecule
> done
Your job...has been submitted...
$
```

`$(command)` substitutes the output of the embedded *command* into that place in the outer command.

Seen so far

`<input >output`	redirection
`$1`	positional arguments
`var=value`	variables: setting
`echo $var`	printing

`for var in list; do`
`  command`
`done`
	looping

`*`	wild card file matching
`$(command)`	command substitution

Extracting results

```
$ ls *.out
ethane.out    methane.out    ...
$ grep 'energy' *.out
ethane.out:The energy is -8.00
methane.out:The energy is -5.00
propane.out:The energy is -11.00
```

`grep` searches a file or files for the pattern given in the 1st argument

Another way to edit

```
propane.out:The energy is -11.00
$ grep 'energy' *.out >raw-energies
$ sed 's/.out:The energy is//' raw-ene..
ethane -8.00
methane -5.00
propane -11.00
```

sed stands for 'stream editor'. Does not change the file – sends results to stdout.

History

```
$ history 5
  952  grep energy *.out
  953  grep energy *.out >raw-energies
  954  sed '/.out:The energy is//' raw-energies
  955  sed 's/.out:The energy is//' raw-energies
  956  history 20
```

Similar function to the up-arrow.

Joining things up

```
$ grep 'energy' *.out | sed \
>   's/.out:The energy is//' >results
$ cat results
ethane -8.00
methane -5.00
propane -11.00
```

The pipe `'|'` redirects the stdout from one cmd to the stdin of the next.

Now seen:

`grep`	search files for strings
`sed`	edit a text stream
`s/regular expression/something else/`	substitution in `vim` or `sed`
`history`	prior commands
`cmd1` **`|`** ***`cmd2`***	"pipe", output becomes input

More tools:

`ls -l`	show file permissions
`chmod +x`	change permissions
`# comments`	
`#!/bin/bash`	"hash-bang"

Still more:

`cut`	extract columns
**`cmd	cmd	cmd	...`**	"pipeline"
`^`*`line`*`$`	beginning- and end-of-line anchors in regexes