

What even is the `module` command?

Ed Bennett

@QuantumofEd



Swansea University
Prifysgol Abertawe

@SwanseaUni



SUPERCOMPUTING WALES
UWCHGYFRIFIADURA CYMRU

@SuperCompWales



@sa2c_swansea

SA2C Tech Chat, 2018-07-27

Slides: git.io/fNEcv

module **basics**

- HPC clusters provide a lot of software

module **basics**

- HPC clusters provide a lot of software
- Not all of it can be in the environment at once

module **basics**

- HPC clusters provide a lot of software
- Not all of it can be in the environment at once
- Install each package in its own folder

module **basics**

- HPC clusters provide a lot of software
- Not all of it can be in the environment at once
- Install each package in its own folder
- Add to the `$PATH` on request

Using module

- `module avail`

Using module

- `module avail`
- `module load`

Using module

- `module avail`
- `module load`
- `module list`

Using module

- `module avail`
- `module load`
- `module list`
- `module unload`

Using module

- `module avail`
- `module load`
- `module list`
- `module unload`
- `module purge`

Promises

“Modules can be loaded and unloaded dynamically and atomically, in an clean fashion.”

Promises

“Modules can be loaded and unloaded dynamically and atomically, in an clean fashion.”

```
[s.e.j.bennett@c12 ~]$ echo $F90
```

Promises

“Modules can be loaded and unloaded dynamically and atomically, in an clean fashion.”

```
[s.e.j.bennett@c12 ~]$ echo $F90
```

```
[s.e.j.bennett@c12 ~]$ export F90=gfortran
```

Promises

“Modules can be loaded and unloaded dynamically and atomically, in an clean fashion.”

```
[s.e.j.bennett@c12 ~]$ echo $F90
```

```
[s.e.j.bennett@c12 ~]$ export F90=gfortran
```

```
[s.e.j.bennett@c12 ~]$ module load compiler/intel
```

Promises

“Modules can be loaded and unloaded dynamically and atomically, in an clean fashion.”

```
[s.e.j.bennett@c12 ~]$ echo $F90
```

```
[s.e.j.bennett@c12 ~]$ export F90=gfortran
```

```
[s.e.j.bennett@c12 ~]$ module load compiler/intel
```

```
[s.e.j.bennett@c12 ~]$ echo $F90
```

Promises

“Modules can be loaded and unloaded dynamically and atomically, in an clean fashion.”

```
[s.e.j.bennett@c12 ~]$ echo $F90
```

```
[s.e.j.bennett@c12 ~]$ export F90=gfortran
```

```
[s.e.j.bennett@c12 ~]$ module load compiler/intel
```

```
[s.e.j.bennett@c12 ~]$ echo $F90
```

```
ifort
```

```
[s.e.j.bennett@c12 ~]$ module unload compiler/intel
```


Promises

“Modules can be loaded and unloaded dynamically and atomically, in an clean fashion.”

```
[s.e.j.bennett@c12 ~]$ echo $F90
```

```
[s.e.j.bennett@c12 ~]$ export F90=gfortran
```

```
[s.e.j.bennett@c12 ~]$ module load compiler/intel
```

```
[s.e.j.bennett@c12 ~]$ echo $F90
```

```
ifort
```

```
[s.e.j.bennett@c12 ~]$ module unload compiler/intel
```

```
[s.e.j.bennett@c12 ~]$ echo $F90
```

Promises

“Modules can be loaded and unloaded dynamically and atomically, in an clean fashion.”

```
[s.e.j.bennett@c12 ~]$ echo $F90
```

```
[s.e.j.bennett@c12 ~]$ export F90=gfortran
```

```
[s.e.j.bennett@c12 ~]$ module load compiler/intel
```

```
[s.e.j.bennett@c12 ~]$ echo $F90
```

```
ifort
```

```
[s.e.j.bennett@c12 ~]$ module unload compiler/intel
```

```
[s.e.j.bennett@c12 ~]$ echo $F90
```

```
[s.e.j.bennett@c12 ~]$
```

Wait a minute!

- Executable scripts run in a subshell

Wait a minute!

- Executable scripts run in a subshell
- They can't change the environment

Wait a minute!

- Executable scripts run in a subshell
- They can't change the environment
- To change the environment needs `source` or `.`, to run in the current shell

Wait a minute!

- Executable scripts run in a subshell
- They can't change the environment
- To change the environment needs `source` or `.,` to run in the current shell
- What even *is* `module`?

Is it an alias?

Slides: git.io/fNEcv

Three thin, dark blue wavy lines that sweep across the bottom of the slide, starting from the left and ending on the right, creating a decorative footer area.

Is it an alias?

```
[s.e.j.bennett@cl2 ~]$ which module
```


Is it an alias?

```
[s.e.j.bennett@cl2 ~]$ which module
```

```
/usr/bin/which: no module in (/usr/lib64/qt-3.3/bin:  
/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin:  
/home/s.e.j.bennett/bin)
```

Is it an alias?

```
[s.e.j.bennett@cl2 ~]$ which module
```

```
/usr/bin/which: no module in (/usr/lib64/qt-3.3/bin:  
/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin:  
/home/s.e.j.bennett/bin)
```

- Nope

Is it an alias?

```
[s.e.j.bennett@cl2 ~]$ which module
```

```
/usr/bin/which: no module in (/usr/lib64/qt-3.3/bin:  
/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin:  
/home/s.e.j.bennett/bin)
```

- Nope
- Where *is* it!?

Aside

```
[s.e.j.bennett@cl2 ~]$ module avail | grep intel
```

Aside

```
[s.e.j.bennett@cl2 ~]$ module avail | grep intel
```

```
----- /apps/modules/legacy -----
```

```
hpcw  raven
```

```
...
```

- module outputs *everything* to stderr

Aside

```
[s.e.j.bennett@cl2 ~]$ module avail | grep intel
```

```
----- /apps/modules/legacy -----
```

```
hpcw  raven
```

```
...
```

- module outputs *everything* to stderr
- Why?

Progress

```
[s.e.j.bennett@cl2 ~]$ cat /etc/profile.d/modules.sh
```

Progress

```
[s.e.j.bennett@cl2 ~]$ cat /etc/profile.d/modules.sh
```

```
shell=`/bin/basename \`${bin}/ps -p $$ -ocomm=\`  
if [ -f /usr/share/Modules/init/$shell ]  
then  
  . /usr/share/Modules/init/$shell  
else  
  . /usr/share/Modules/init/sh  
fi
```


Eureka

```
[s.e.j.bennett@cl2 ~]$ cat /usr/share/Modules/init/sh
```

Eureka

```
[s.e.j.bennett@cl2 ~]$ cat /usr/share/Modules/init/sh  
  
module() { eval ` /usr/bin/modulecmd sh $*`; }  
  
MODULESHOME=/usr/share/Modules  
export MODULESHOME  
  
...
```

- So module is a *function*

Eureka

```
[s.e.j.bennett@cl2 ~]$ cat /usr/share/Modules/init/sh  
  
module() { eval ` /usr/bin/modulecmd sh $*`; }  
  
MODULESHOME=/usr/share/Modules  
export MODULESHOME  
  
...
```

- So module is a *function*
- Now, how does it work?

modulecmd

- 7392 lines of Tcl

modulecmd

- 7392 lines of Tcl
- ...

modulecmd

- 7392 lines of Tcl
- ...
- Let's look at a modulefile instead

modulecmd

- 7392 lines of Tcl (<https://git.io/fNEgB>)
- ...
- Let's look at a modulefile instead

Finding some module files

```
[s.e.j.bennett@c12 ~]$ cat /etc/profile.d/02-default-module
```


Finding some module files

```
[s.e.j.bennett@c12 ~]$ cat /etc/profile.d/02-default-modules
```

```
#!/bin/sh
```

```
# Ansible managed
```

```
module use /apps/modules/physics
```

```
module use /apps/modules/medical
```

```
module use /apps/modules/materials
```

```
module use /apps/modules/genomics
```

```
module use /apps/modules/financial
```

```
...
```

Module file directory structure

```
[s.e.j.bennett@c12 ~]$ find /apps/modules/
```

Module file directory structure

```
[s.e.j.bennett@c12 ~]$ find /apps/modules/
```

```
/apps/modules/  
/apps/modules/legacy  
/apps/modules/legacy/raven  
/apps/modules/legacy/hpcw  
/apps/modules/medical  
/apps/modules/materials  
/apps/modules/materials/fluidity  
/apps/modules/materials/OpenFOAM  
/apps/modules/materials/OpenFOAM/5.x-20180613  
/apps/modules/materials/QuantumEspresso  
/apps/modules/materials/QuantumEspresso/6.1  
...
```

OK OK, show me a modulefile already!

```
[s.e.j.bennett@c12 ~]$ more /app/modules/materials/cp2k/2.4
```

OK OK, show me a modulefile already!

```
[s.e.j.bennett@c12 ~]$ more /app/modules/materials/cp2k/2.4
#%Module1.0#####
source /app/modules/system/logger.tcl
##
## CP2K modulefile
##
set appname "cp2k"
set version "2.4"
set para_ver "mpi"
set type "popt"
--More--
```

- Modulefiles are Tcl programs

OK OK, show me a modulefile already!

```
[s.e.j.bennett@c12 ~]$ more /app/modules/materials/cp2k/2.4
#%Module1.0#####
source /app/modules/system/logger.tcl
##
## CP2K modulefile
##
set appname "cp2k"
set version "2.4"
set para_ver "mpi"
set type "popt"
--More--
```

- Modulefiles are Tcl programs
- modulecmd executes these, and does stuff with the result

More of this cp2k modulefile

```
proc ModulesHelp { } {  
    puts stderr "\tLoads PATH and LD_LIBRARY_PATH settings for  
    puts stderr "\tAn example jobscript and input file can be  
    puts stderr "\tThis, along with a README file, is in the  
    puts stderr "\tThe cp2k application directory can be found  
    puts stderr "\tlooking at the output of `which cp2k` and  
    puts stderr "\tMore information on this package can be found  
}  
--More--
```

- Module help gets output to stderr

More of this cp2k modulefile

```
proc ModulesHelp { } {  
    puts stderr "\tLoads PATH and LD_LIBRARY_PATH settings for  
    puts stderr "\tAn example jobscript and input file can be  
    puts stderr "\tThis, along with a README file, is in the  
    puts stderr "\tThe cp2k application directory can be found  
    puts stderr "\tlooking at the output of `which cp2k` and  
    puts stderr "\tMore information on this package can be found  
}  
--More--
```

- Module help gets output to stderr
- Why?

More of this cp2k modulefile

```
proc ModulesHelp { } {  
    puts stderr "\tLoads PATH and LD_LIBRARY_PATH settings for  
    puts stderr "\tAn example jobscript and input file can be  
    puts stderr "\tThis, along with a README file, is in the  
    puts stderr "\tThe cp2k application directory can be found  
    puts stderr "\tlooking at the output of `which cp2k` and  
    puts stderr "\tMore information on this package can be found  
}  
--More--
```

- Module help gets output to stderr
- Why?
- Let's keep going...

Oh no...

```
# Set ulimit unlimited
if { [module-info shelltype sh] } {
    puts "ulimit -s unlimited;"
} elseif { [module-info shelltype csh] } {
    puts "limit stacksize unlimited;"
}
--More--
```

- Why are we putsing shell commands to stdout?

Oh no...

```
# Set ulimit unlimited
if { [module-info shelltype sh] } {
    puts "ulimit -s unlimited;"
} elseif { [module-info shelltype csh] } {
    puts "limit stacksize unlimited;"
}
--More--
```

- Why are we putsing shell commands to stdout?
- Wait

Oh no...

```
# Set ulimit unlimited
if { [module-info shelltype sh] } {
    puts "ulimit -s unlimited;"
} elseif { [module-info shelltype csh] } {
    puts "limit stacksize unlimited;"
}
--More--
```

- Why are we putsing shell commands to stdout?
- Wait
- Surely not

```
module() {  
    eval `/usr/bin/modulecmd sh $*`;  
}
```

`eval` considered harmful (to sanity)

- All actual work is done inside `modulecmd`

`eval` **considered harmful (to sanity)**

- All actual work is done inside `modulecmd`
- The output of which is passed straight to `eval`

`eval` **considered harmful (to sanity)**

- All actual work is done inside `modulecmd`
- The output of which is passed straight to `eval`
- Commands are run by outputting them to `stdout`

`eval` considered harmful (to sanity)

- All actual work is done inside `modulecmd`
- The output of which is passed straight to `eval`
- Commands are run by outputting them to `stdout`
- So messages can only be displayed via `stderr`

`eval` considered harmful (to sanity)

- All actual work is done inside `modulecmd`
- The output of which is passed straight to `eval`
- Commands are run by outputting them to `stdout`
- So messages can only be displayed via `stderr`
- We can cross-check this in the `modulecmd` source...

modulecmd revisited

```
switch -- $::g_shellType {  
  ...  
  {sh} {  
    puts stdout "$var=[charEscaped $::env($var)];\  
    export $var;"  
  }  
  ...  
  {tcl} {  
    set val $::env($var)  
    puts stdout "set ::env($var) {$val};"  
  }  
  {cmd} {  
    set val $::env($var)  
    puts stdout "set $var=$val"  
  }  
  ...  
}
```

Private modules

- Recall module use `/apps/modules/physics`

Private modules

- Recall module use `/apps/modules/physics`
- Last argument is just a path

Private modules

- Recall module use /apps/modules/physics
- Last argument is just a path
- Paths are scanned for modulefile-like things

Private modules

- Recall `module use /apps/modules/physics`
- Last argument is just a path
- Paths are scanned for modulefile-like things
- So we can add a path in `$HOME`

Private modules

- Recall `module use /apps/modules/physics`
- Last argument is just a path
- Paths are scanned for modulefile-like things
- So we can add a path in `$HOME`
- This is built in via the `use.own` module, which looks in `$HOME/privatemodules`

Writing our own module

```
#!/Module -*- tcl -*-  
## This is a module to access something  
proc ModulesHelp { } {  
    puts stderr "This module sets up access to something"  
}  
module-whatis "sets up access to something"  
prereq something/else  
conflict that/other/module  
module load gcc  
setenv      SOMEVERION      0.95  
append-path PATH             /home/[user]/[somedir]/bin  
append-path MANPATH          /home/[user]/[somedir]/man  
append-path LD_LIBRARY_PATH  /home/[user]/[somedir]/lib
```

Thanks for listening!



Slides: git.io/fNEcv