JGI Modules Tutorial

Tony Wildish
Daniel Udwary
NERSC Data Science Engagement Group

November 30, 2017

https://bitbucket.org/TWildish/jgi-modules-tutorial.git
Introduction

• What’s a module?
• Why should you use them
• Why shouldn’t you use them
• Module structure
• How to build a simple module
• How to build a more complex module
• Best practices
• Exercises

https://bitbucket.org/TWildish/jgi-modules-tutorial.git
• I assume you know basic module use:
  – module load
  – module unload
  – module avail
  – module swap

https://bitbucket.org/TWildish/jgi-modules-tutorial.git
What’s a module?

• **A module is a way of using software**
  – It sets up the runtime environment for installed software
    • Essentially it just sets/unsets environment variables:
      • $PATH, $LD_LIBRARY_PATH, $MANPATH, $PKG_CONFIG_PATH,
        $PERL5LIB, $PYTHONPATH, any others you may want

  – It allows specifying and enforcing dependencies
    • Modules X can’t load without module Y etc

  – It allows specifying multiple versions of a piece of software
    • Module XYZ/1.0, XYZ/1.2, XYZ/2.0…

  – It doesn’t describe how the software was built
  – It doesn’t describe how the software was installed
  – It’s not a version control system

https://bitbucket.org/TWildish/jgi-modules-tutorial.git
Why should you use modules?

• **Reproducibility**
  – Guarantee that your runtime environment is controlled properly

• **Flexibility**
  – Use several applications, that are built/maintained separately, in a coherent manner

• **Simplicity**
  – Module files are easy to create

https://bitbucket.org/TWildish/jgi-modules-tutorial.git
Why shouldn’t you use modules?

• **Reproducibility**
  – Software may be tied to the system it’s built on, may not be easily portable

• **Flexibility**
  – Software that has conflicting requirements is harder to manage with modules than by other means

• **Simplicity**
  – You still have to build and install the software yourself

https://bitbucket.org/TWildish/jgi-modules-tutorial.git
Portability, Flexibility

- Install in base OS
- Install in user-space ($HOME etc)
- Modules
- Conda environments
- Docker/Shifter images
- Virtual machines

Harder

Easier

https://bitbucket.org/TWildish/jgi-modules-tutorial.git
Module structure

• To use modules you need:
  – A directory, added to your $MODULEPATH (see later)
  – Subdirectories for each package
  – Version-specific modulefiles in each package subdirectory

• E.g. on Denovo:
  – $MODULEPATH contains /usr/common/jgi/Modules/modulefiles, which contains, among others: falcon/1.8.8

  denovo1> module avail falcon
  -------------------------- /usr/common/jgi/Modules/modulefiles --------------------------
  falcon/1.8.8

https://bitbucket.org/TWildish/jgi-modules-tutorial.git
denovo1> cat /usr/common/jgi/Modules/modulefiles/falcon/1.8.8

#%Module1.0
##
## Required internal variables
set name falcon
set version 1.8.8
set root /usr/common/jgi/assemblers/$name/$version

## List conflicting modules here
set mod_conflict {$name}

## List prerequisite modules here
set mod_prereq_autoload {python/2.7-anaconda}
set mod_prereq {python/2.7-anaconda}

## Source the common modules code-base
source /usr/common/usg/Modules/include/usgModInclude.tcl

## Software-specific settings exported to user environment
prepend-path PATH $root/fc_env/bin
prepend-path PYTHONPATH $root/fc_env/lib/python2.7/site-packages:$root/FALCON
setenv FALCON_DIR $root
setenv PYTHONUSERBASE $root/fc_env
setenv FALCON_PREFIX $root/fc_env

Module files are written in Tcl
N.B. This is a simplified version of the real file

https://bitbucket.org/TWildish/jgi-modules-tutorial.git
Required header

## Required internal variables

set name falcon
set version 1.8.8
set root {/usr/common/jgi/assemblers/$name/$version}

## List conflicting modules here

set mod_conflict { $name }

## List prerequisite modules here

set mod_prereq_autoload { python/2.7-anaconda }
set mod_prereq { python/2.7-anaconda }

## Source the common modules code-base

source /usr/common/usg/Modules/include/usgModInclude.tcl

## Software-specific settings exported to user environment

prepend-path PATH $root/fc_env/bin
prepend-path PYTHONPATH $root/fc_env/lib/python2.7/site-packages:$root/FALCON
setenv FALCON_DIR $root
setenv PYTHONUSERBASE $root/fc_env
setenv FALCON_PREFIX $root/fc_env

https://bitbucket.org/TWildish/jgi-modules-tutorial.git
## Required internal variables

set name falcon
set version 1.8.8
set root /usr/common/jgi/assemblers/$name/$version

## List conflicting modules here

set mod_conflict {$name}

## List prerequisite modules here

set mod_prereq autoload {python/2.7-anaconda}
set mod_prereq {python/2.7-anaconda}

## Source the common modules code-base

source /usr/common/usg/Modules/include/usgModInclude.tcl

## Software-specific settings exported to user environment

prepend-path PATH $root/fc_env
prepend-path PYTHONPATH $root/fc_env/lib/python2.7/site-packages:
$root/FALCON

setenv FALCON_DIR $root
setenv PYTHONUSERBASE $root/fc_env
setenv FALCON_PREFIX $root/fc_env

List any module conflicts:

Most modules conflict with themselves – you won’t want to use module X/1.0 and X/2.0 at the same time.

List any module dependencies:

Use mod_prereq to prevent the module loading unless the dependencies are already loaded.

Use mod_prereq_autoload to automatically load those dependencies.

https://bitbucket.org/TWildish/jgi-modules-tutorial.git
## Required internal variables

set name falcon
set version 1.8.8
set root /usr/common/jgi/assemblers/$name/$version

## List conflicting modules here

set mod_conflict {$name}

## List prerequisite modules here

set mod_prereq_autoload { python/2.7-anaconda }
set mod_prereq { python/2.7-anaconda }

## Source the common modules code-base

source /usr/common/usg/Modules/include/usgModInclude.tcl

## Software-specific settings exported to user environment

prepend-path PATH $root/fc_env/bin
prepend-path PYTHONPATH $root/fc_env/lib/python2.7/site-packages:$root/FALCON
setenv FALCON_DIR $root
setenv PYTHONUSERBASE $root/fc_env
setenv FALCON_PREFIX $root/fc_env

N.B. $PACKAGE_DIR is set to point to the base of the installation, by local convention

Set $PATH, $PYTHONPATH etc, using variables defined earlier

prepend-path prefixes to $PATH-like variables

setenv simply sets a value

https://bitbucket.org/TWildish/jgi-modules-tutorial.git
How to build a simple module

• First, create a directory to hold your module files

• Clone the repository

  > git clone https://bitbucket.org/TWildish/jgi-modules-tutorial.git
  > cd jgi-modules-tutorial
  > export HERE=`pwd`

• Create a subdirectory ‘Modules’, add it to your $MODULEPATH

  > mkdir Modules
  > module use $HERE/Modules

  – ‘module use’ adds the module to your $MODULEPATH, don’t set it by hand
  – Now, modules added under that directory can be used automatically

https://bitbucket.org/TWildish/jgi-modules-tutorial.git
How to build a simple module: FastTree

• **Run the build script**
  > cd examples/fasttree/build
  > ./build-fasttree-2.1.10.sh

  – That installs the binary in $HERE/examples/fasttree/2.1.10

• **Now create the module file to point to that installation**
  – There’s a template file, ‘2.1.10’, in the same directory as the build script
  – Edit it, change the definition for `root` to point to the installation root
    • ‘set root { /…/jgi-modules-tutorial/examples/$name/$version }’
  – N.B. give full pathname, don’t use environment variables in the module file

  – Create a ‘fasttree’ subdirectory in your modules directory, copy this file there
    • mkdir $HERE/Modules/fasttree
    • cp 2.1.10 $HERE/Modules/fasttree/

https://bitbucket.org/TWildish/jgi-modules-tutorial.git
Recap: What did we just do?

• We created a directory to hold module files
  — $HERE/Modules

• Added that directory to our MODULEPATH
  — With the ‘module use’ command

• Compiled & installed fasttree

• Created a module file to add the fasttree directory to the $PATH environment variable

• Added that module file to the module directory, in a subdirectory named for the software we installed
  — $HERE/Modules/fasttree/2.1.10

• Now, ‘module avail fasttree’ will show our module!
A more complex module

- The ‘examples’ directory contains build scripts for other tools
  - last, mash, mummer, prodigal, vsearch, zlib
    - There’s a template module file for last, you can do the others as an exercise
    - The actual module files aren’t more complex, it’s only the software build procedure that is more involved

- The build scripts all run out-of-the-box on Denovo
  - Several best-practices illustrated, please take a look at them
  - The ‘last’ build script, in particular, is well-documented

https://bitbucket.org/TWildish/jgi-modules-tutorial.git
Best practices: module files

• Keep your build scripts and module templates together
  – Can automate generating module files for most packages, ask me later if you want to do that

• Make sure you define all the environment variables your package needs
  – PATH, PYTHONPATH, MANPATH, package-specific variables etc

• Keep them under version control
  – Knowing how a module was built is essential for reproducible science
  – Makes your environment more (likely to be) portable to new platforms

https://bitbucket.org/TWildish/jgi-modules-tutorial.git
Best practices: building software

• **Build scripts require a *lot* of care to do well**
  – It’s not difficult to do properly, but it is necessary for reproducibility
    • Build-scripts should be self-documenting
  – Prefer building from source over installing binaries
    • Controls dependencies better, not relying on the system so much
    • Can make a *huge* difference in performance, optimizing for modern CPU architecture (anyone using 32-bit or i386 binaries?)
  – Record the original location of the source code (the URL), so you can refer back there for more information etc
    • Document where you got the package, and how
    • Keep the source after you download it, it may disappear from the web
  – Make your scripts abort on error
    • Don’t assume they will work every time
  – READ THE BUILD INSTRUCTIONS for each package
    • Don’t just accept defaults without understanding what they do

https://bitbucket.org/TWildish/jgi-modules-tutorial.git
Best practices: building software

– Clean the build environment, then build it up from scratch
  • Minimize dependencies on the OS (GCC, Perl, Python, graphics libraries…)

– Don’t hardwire the location of the build or the installation
  • That reduces portability and flexibility, makes development harder

– Remove the installation & working directories before building
  • Don’t risk incorporating stale artifacts from previous builds

– Clean up after building too, so you don’t leave cruft

– If the package has built-in tests, run them before installing
  • Look for ‘make check’ or ‘make test’ targets
  • If there are no built-in tests, can you provide something minimal?

– Bonus points: add $GCC_RECORD_SWITCHES to your compilation flags
  • Set for compilers that support it, records compile options in the build products
  • Read the options back with ‘readelf’:
    – > readelf -p .GCC.command.line $SPADES_DIR/bin/spades

https://bitbucket.org/TWildish/jgi-modules-tutorial.git
Best practices: building software

• Building a library? Be kind to your users!
  - Build it for all available compilers
    • Don’t force the user to use a specific compiler unnecessarily
    • Use ‘mod_variations’ in your module file to load the right flavor
    • See examples/zlib/build/build-zlib-1.2.11.sh
  - Add extra environment variables in your module file to make it easier to use the library
    • See examples/zlib/1.2.11, which sets ZLIB_INC, ZLIB_LIB, & others
    • See the examples for mash and vsearch for how to use them

https://bitbucket.org/TWildish/jgi-modules-tutorial.git
Final comments

• **That’s a lot to remember, is it worth it?**
  – Binary-only installs are OK if you can trust the person who built the software
    • Fine with Anaconda, there’s a strong community
    • Docker images vary in quality, some good, some not so good
    • If you really care about the results from your applications, and the performance you get running them, you should build them yourself

• **So you’ve built your module, what next?**
  – You’re one step away from building a container, why not try it!
    • Much less work for the container than for the original software
    • Can make the base container look like the OS you build the s/w on, then your build script should work out of the box
    • Gives you a migration path to other platforms (cloud etc) for only a little extra investment
Exercises

• **Build and install a module for each of the examples**
  – Follow the recipe from slide 14 (after preliminaries on slide 13)
  – Run the build scripts in the examples directory for each package
    • Just cd into the directory and execute the script
  – Create a module file for each package, install it
    • …/Modules/${package}/${version}
  – Verify that you can ‘module load’ the package and run the software

• **Bonus exercise:**
  – One of my recommended practices isn’t followed by any of the example scripts. Can you fix them?
  – The example for metabat only provides the download link, create the build script and module file yourself.

https://bitbucket.org/TWildish/jgi-modules-tutorial.git
National Energy Research Scientific Computing Center

https://bitbucket.org/TWildish/jgi-modules-tutorial.git