

Installing Software and Writing Modules

Introduction

Model, CoC and URLs

- Carpentries model: hands-on portion, aka *live coding*
 - Carpentries Code of Conduct
- <https://github.com/SmithsonianWorkshops/>
 - view *slides* or the *markdown* version

In the intro portion of the workshop you will learn:

- About downloading code
- About compiling code
- How to build a package from source code
 - configure
 - build
 - install
- What are yum, rpm, get-apt, & sudo
- How to write modules

Downloading Code

Source vs Executable

- In most cases you are better off downloading the source and building the code (aka the executable) yourself.
- Downloading an executable is easier but likely will not to work.

Downloading Executables

Some developers provide pre-built executables of their software. There are instances when available executables will run flawlessly on Hydra, but make sure that:

- 1 you can trust the origin,
- 2 you get a version compatible with Hydra,
 - *i.e.*, CentOS 7.x for Intel/AMD CPUs (x86_64)

Remember

- Hydra configuration is specific:
 - pre-built code may need *stuff* (dependencies) not on Hydra.

Notes on Downloading Executables

Risks

- Since users on Hydra do not have elevated privileges (root access) you are very unlikely to damage the cluster, but malicious software can still damage your files.
- In rare cases it may install a *Trojan horse* that could exploit a known vulnerability.
 - Be vigilant and responsible.
 - In case of doubt, never hesitate to contact us.

Compiling code

Steps

Creating executable from source code is typically done as follows:

- 1 compile the source file(s) to produce object file(s),
- 2 link the object file(s) and libraries into an executable.

In Practice

- Often aided by a makefile,
- *Configuring* is creating such makefile or equivalent.

This will be illustrated in the hands-on section.

Building from Source

1. Configure

- Most packages come with a configuration script, a list of prerequisites (dependencies/libraries) and instructions,
- Some packages allow you to build the code without some features in case you cannot satisfy some of the prerequisites,
- You most likely need to load the right module(s) to use the appropriate tools (compilers).
- The configuration step will test if the code can be built:
 - check dependencies, versions, etc.
 - if this fails, the code cannot be built as is.

1.b Makefile only

- Other (simpler) packages come with a `makefile` that needs to be edited,
 - check the instructions.

Building from Source (cont'd)

2. Build

- make sure you have loaded the right modules,
- run `make` to compile and link (aka build) the code.

2.b Test

- some packages come with the optional step of testing the built before installing it, using something like `make test`.

3. Install

- copy the executable(s) to the right place(s),
 - usually defined by the configuration,
- best practice is to separate build from install locations.

Basics about make and makefile

The command make

- make is a utility to maintain groups of programs.
- Uses instructions in a makefile to build targets from sources by following rules.
- written to help build & maintain code, can be used for a lot more (full Carpentries module).

Examples:

- build the first target listed in the makefile:

```
make
```

- build the target “this” listed in the “makefile” file:

```
make this
```

- build “that” using “makefile.special” and set “VAR” to “val”:

```
make -f makefile.special VAR=val that
```


Basics about make and makefile (cont'd)

The Makefile or makefile files

- a file that defines targets and codifies rules and dependencies to build targets;
 - dependency: has a source needed to build something changed?
- it can be very simple, but can also be quite complex.

Also

- make has implicit rules:
 - can build targets w/out a makefile or w/out rules.

This will be illustrated in the hands-on part

Setting up Your Environment to Run Your Code

Likely Needed

You likely will need to adjust your *environment* to run some code:

- 1 the location of the code: `path` or `PATH`,
- 2 the location of the libraries: `LD_LIBRARY_PATH`,
- 3 you may also need to set some environment variables, etc.

Easier Way: modules

This is where using a module makes things easy:

- compact, and
- works with any shell.

The yum, rpm, get-apt and sudo Soup

Definitions

- yum: is a package-management utility for CentOS
- rpm: pre-built software package
 - *both* are for sys-admin,
 - help handle dependencies,
 - *yet* . . .
- get-apt: Debian's version of yum, *does not work* on CentOS.

Also

- sudo: allows to run a command as 'root': **you can't!**

BTW

- Instructions that mention yum, rpm, apt-get or sudo
 - **will not work** on Hydra,
 - **yet** in most cases there is another way.

How about Hydra

Using yum

- While you **cannot** install packages with yum,
- you can check if we've installed a prerequisite package

In practice

- if the instructions say

```
sudo yum -y install <package>
```

- you can run

```
yum info <package>
```

Using yum info

Example

```
yum info libXt-devel
... stuff and may be slow the first time ...
Installed Packages
Name           : libXt-devel
Arch           : x86_64
Version        : 1.1.5
...
Description    : X.Org X11 libXt development package
...
```

You want the Arch: x86_64 to be listed as “Installed” not *just* “Available”.

How to avoid sudo

```
sudo make install
```

- if the instructions says

```
sudo make install
```

- instead, set the installation directory to be under your control,
- in most cases at the configuration step

```
./configure --prefix=/home/<username>/big-package/3.5
```

- and use

```
make install
```

Replace <username> by your username.

Final Notes

Remember

- there is a way to use `yum` as a non privileged user
 - not recommended, unless you're an **expert**!
- you can always ask about a missing prerequisite,
- most of those can be built from source since Linux is an open source OS.

Module and Module Files

The Command `module`

- convenient mechanism to configure your *environment*,
- reads a file, the *module file*, that holds instructions,
- a shell independent way to configure your environment:
 - *same* module file whether `sh/bash` or `csh/tcsh`.

Examples

- We provide module files, users can write their own.
 - look at all the module files we wrote,
 - they can be found in `/share/apps/modulefiles/`

Module File Syntax and Concepts

Special Instructions

- Instructions to configure your environment:

```
prepend-path PATH /location/of/the/code
```

```
setenv      BASE /scratch/demo
```

```
set-alias   crunch "crunch --with-that-option \*"
```

Syntax

- Module files can be complex, using tc1 language
 - you **do not** need to know tc1 to write module files.

Simple or Complex

- A simple module file can just list the modules that must be loaded to run some analysis.
- Can write complex module files and leverage tc1.

Example of module Commands

Basic

	Info	Config	Details
module	avail	load	list
module	whatis	unload	help <name>
module	whatis <name>	swap	show <name>

More help

man module

A Simple Module File

Example

```
#!/Module1.0
#
# load two modules and set the HEASOFT env variable
module load gcc/10.1.10
module load python/3.8
setenv HEASOFT /home/<username>/heasoft/6.3.1
```

Replace <username> by your username.

Example of More Elaborate and Complex Module Files

Will be illustrated in the hands-on section.

Module Files Organization

Recommended Approach

- use a central location under you home directory
~/modulefiles,
- use a tree structure
- use version numbers if/when applicable,
- let module know where to find the module files.

Customization/Examples

Tree structure

```
~/modulefiles/crunch/  
~/modulefiles/crunch/1.0  
~/modulefiles/crunch/1.2  
~/modulefiles/crunch/2.1  
~/modulefiles/crunch/.version  
~/modulefiles/viewit
```

Define a Default Version

An optional file `.version` can be used to set the default version:

```
#%Module1.0  
set ModulesVersion "1.2"
```

Hence

```
module load crunch  
module swap crunch/2.1
```

Customization (cont'd)

Let module Know Where to Find the Module Files

```
module use --append ~/modulefiles
```

Either

- 1 in your initialization file `~/ .bashrc` or `~/ .cshrc`
- 2 or better yet in a `~/ .modulerc` file

```
##Module1.0  
# adding my own module files  
module use --append /home/username/modulefiles
```


Hands-on Section

Hands-On

In the hands-on portion of the workshop you will

- Build and install software using best-practices,
 - trivial case,
 - simple/didactic example,
 - somewhat complex examples.
- Write simple and more elaborate module files.
- Run the software you installed in jobs.

But first, log in to Hydra

- If you need a reminder about how to log into Hydra and how to change your password, check the *Intro to Hydra* tutorial.
 - If the link does not work:

<https://github.com/SmithsonianWorkshops>

- > Hydra-introduction
- > hydra_intro.md



Let's pause here for 5-10 minutes



Switch to github for the Hands-on

Go to

<https://github.com/SmithsonianWorkshops/advanced-hydra-workshops/>

Convention

- I use % as prompt
 - your prompt might be different, like \$
 - you type what is **after** the prompt
 - no prompt: result from previous command.
- I where you see <genomics|sao>, you need to use either genomics or sao,
- I where you see <username>, you need to substitute your username.