## Software and Environment Modules

### Quick Links

<table>
<thead>
<tr>
<th>Category</th>
<th>Links</th>
</tr>
</thead>
<tbody>
<tr>
<td>HPC Home</td>
<td>Getting an account, Gentle Introduction to using HPC, Getting started on Prince</td>
</tr>
<tr>
<td>Clusters and Storage</td>
<td>Prince (HPC), Dumbo (Hadoop), Brooklyn (OpenStack)</td>
</tr>
<tr>
<td>Logging in</td>
<td>Windows, Mac/Linux</td>
</tr>
</tbody>
</table>

- Wind...
Dalmia (NYU Abu Dhabi)

Transfering data to/from the clusters
Transfering data to/from Prince cluster using Globus

Submittin g jobs with sbatch
Available software
Licensed Software Available on the HPC Cluster

Building Software
Slurm Tutorial
Tutorials
FAQs
Scratch Area Cleanup

Programming for Biologists
Running jobs on the Prince Cluster

<table>
<thead>
<tr>
<th>Accessing the Prince Cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td>From Windows workstation</td>
</tr>
<tr>
<td>From Mac workstation</td>
</tr>
</tbody>
</table>

Software and Environment Module

Job script and resource request

Introduction to job scheduling

Submitting jobs with `sbatch`

Requesting resources
A typical workstation is used by one or a few users who need a small selection of software packages configured in a specific way. All software is installed under Program Files (Windows), or Applications (Mac), or /usr/bin and /usr/lib (Linux). Keeping software up to date whilst managing dependencies between specific versions of different software packages is already challenging.

A typical HPC cluster has a large number of users, each needing a different selection of software packages, often with different versions and configurations. Installing all software in /usr/bin and /usr/lib whilst meeting the disparate needs of each user under these circumstances is simply not possible.

Environment Modules is a tool for managing multiple versions and configurations of software packages, and is used by many HPC centers around the world. With Environment Modules, software packages are installed away from the base system directories, and for each package an associated modulefile describes what must be altered in a user’s shell environment – such as the $PATH environment variable – in order to use the software package. The modulefile also describes dependencies and conflicts between this software package and other packages and versions.

To use a given software package, you load the corresponding module. Unloading the module afterwards cleanly undoes the changes that loading the module made to your environment, thus freeing you to use other software packages that might have conflicted with the first one.

Exercise
Log in to Prince and, at the prompt, type each of the commands described below

Finding a software package on the NYU HPC clusters

The command for seeing what software packages are available is:

```
$ module avail
```
This will produce a long of software package. At NYU, the naming convention for modules is `package/build_configuration/version` or, for packages provided in binary form, `package/version`.

For example, on Prince we have several installations of the open-source software "fftw", including:

- `fftw/intel/3.3.5` - fftw version 3.3.5, built with the Intel compiler suite
- `fftw/openmpi/intel/2.1.5` - fftw version 2.1.5, built for MPI with OpenMPI and the Intel compiler suite
- `fftw/openmpi/intel/3.3.5` - fftw version 3.3.5, built for MPI with OpenMPI and the Intel compiler suite

Matlab on the other hand is a commercial package and comes as a binary, not source code, so the only version changes between modules:

- `matlab/2016b`

If you know what the package you need is called, or even what its name starts with, you can see a smaller list of packages by appending all or part of the package name to `module avail`, for example:

```
$ module avail fftw
```

will list only the available configurations and versions of `fftw`, while

```
$ module avail f
```

will list all packages whose name begins with "f".

### Why keep old versions of software?

There are two good reasons to keep old versions even though newer releases are installed:

- **Compatibility**: other software packages may require a specific version of this package, or may not work in conjunction with the newer package
- **Reproducibility**: the specific version and build configuration of a software package can lead to minor differences in the results of simulations using it. In order to exactly replicate an experiment, the same version of software should be used.

### Exercise

Scan the available modules for one or two software packages you expect to need. Take note of which versions are available. (we'll look more closely at them later)

**Tip:** you can append the list of module versions to a NOTES file by redirecting the output of `module avail` as shown below (recall redirection in session 2). The `module` command writes its output to stderr, not stdout, so you need to also redirect stderr to stdout with "2>&1" (assuming you are using bash). And remember to use "">>" rather than "">" so that you append, rather than overwrite, your NOTE file.

```
$ module avail package >> NOTES.txt 2>&1
```

### Finding out more about a software package

You can use "module show", "module whatis" and "module help" to find out about the package and what actions will be performed by loading the module. We won't cover that here, but it is in the Wiki.
Loading and unloading modules

To load a module:

```
$ module load module-name
```

For example:

```
$ module load fftw/intel/3.3.5
```

**Important**
Always specify the full module name, including build configuration and version. If you do not, you will get an arbitrarily chosen version of the software package.

To unload a module:

```
$ module unload module-name
```

For example:

```
$ module unload fftw/intel/3.3.5
```

**Important**
Specifying the module name is important too when unloading the module, with Lmod Environment Modules software which is used at Prince cluster, e.g. try this "module unload fftw"; it will not unload anything.

Unloading all modules

You may think it remove all loaded modules from your environment with:

```
$ module unload
```

But it does not with Lmod. So be careful, always check outcome after your run a command.

It's a good idea to use "module purge" before loading modules to ensure you have a consistent environment each time you run.

What modules do I currently have loaded?

You can check which modules are currently loaded in your environment with:

```
$ module list
```

I used "module load" and got a "module: command not found" error. What should I do?

Normally the location of the module command is set up when the shell is started, but under some circumstances that startup procedure can be bypassed. If you get this error you can explicitly prepare your environment for modules with one of the following commands:

- If your script (or interactive environment) uses `bash` (the default) or `sh` or `ksh`:
  
  ```
  $ source /etc/profile.d/lmod.sh
  ```

- If your script (or interactive environment) uses `csh` or `tcsh`:
  
  ```
  $ source /etc/profile.d/lmod.csh
  ```

In the case of a Slurm job script, add one of the above lines before the first "module" command in your script.

If you are seeing the error in an interactive shell, run one of the above commands at the prompt, then attempt the "module load" command.
Exercise
Load the modules you identified in the previous exercise. Now use "module list" to see what is in your environment.

You may have other modules there which you did not load: this is because some software packages depend on other software packages, and the convention at NYU HPC is for modules to automatically load dependencies.

Experiment with "module unload" and "module purge" too.

Tip: It may be helpful to have your NOTES file with the module names visible on the screen while you do this. You can print the contents of NOTES.txt on the terminal with "cat NOTES.txt".

In summary:

- Different users need different combinations of different versions of software packages
- Initial login is a bare Unix environment
- Explore available software with "module avail"
- Load software into your environment with "module load"
- Return to a clean environment with "module purge"