### Quick Links

<table>
<thead>
<tr>
<th>HPC at NYU</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>HPC Newsletter</td>
<td></td>
</tr>
<tr>
<td>Getting started on Prince</td>
<td></td>
</tr>
<tr>
<td>Getting an account</td>
<td></td>
</tr>
<tr>
<td>Prince How-to Articles</td>
<td></td>
</tr>
<tr>
<td>Logging in</td>
<td></td>
</tr>
<tr>
<td>Windows</td>
<td></td>
</tr>
<tr>
<td>Mac / Linux</td>
<td></td>
</tr>
<tr>
<td>Clusters and Storage July 2017</td>
<td></td>
</tr>
<tr>
<td>Mercer (HPC, retired)</td>
<td></td>
</tr>
<tr>
<td>Prince (HPC)</td>
<td></td>
</tr>
<tr>
<td>Dumbo (Hadoop)</td>
<td></td>
</tr>
<tr>
<td>Dalmata (NYU Abu Dhabi)</td>
<td></td>
</tr>
<tr>
<td>-------------------------</td>
<td></td>
</tr>
<tr>
<td><strong>Transfering data to/from the clusters</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Transfering data to/from Prince cluster using Globus</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Submittin jobs with sbatch</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Available software</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Building Software Tutorials</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Programming for Biologists</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Research Gallery</strong></td>
<td></td>
</tr>
<tr>
<td><strong>FAQs</strong></td>
<td></td>
</tr>
<tr>
<td><strong>NYU HPC Cheat Sheet</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Wiki Map</strong></td>
<td></td>
</tr>
<tr>
<td><strong>HPC Policies</strong></td>
<td></td>
</tr>
</tbody>
</table>
What are Environment Modules?

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 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.

To understand how Environment Modules work, it is helpful to think about what the shell does when you enter a command:

How does the shell know what 'ls' means? (click to expand)

A significant component of the shell is its *environment* - a set of shell variables and environment variables (such as $USER) that scripts and programs can set and use.

A variable has a name, which can contain letters, numbers and underscores, and a value which is simply a text string. To access a variable place a `-' in front of it. Try:
echo $USER

and compare the result with:

echo USER

Sometimes you will see a variable reference like $(USER) or ${USER:-abc}. The first is a more explicit usage which is needed in certain cases, and the second is a more advanced usage, in this case "$USER if it is set, or abc otherwise". To learn more about advanced usage of variables (also called parameters), type "man bash".

There is a subtle difference between shell variables and environment variables: shell variables are only visible in the current shell, while environment variables are visible to programs started from the shell, including subshells. By convention, shell variables are usually given a lowercase name while environment variables are given an uppercase name.

You can set a shell variable by making it equal something:

my_var="hello there"

An environment variable is a shell variable, exported to the environment:

export MY_VAR="hello there"

Spaces are important! There must be no spaces on either side of the equal sign.

You can see what environment variables are set with "env".

Environment variables are especially useful when writing job scripts: you can set locations or other options once at the top of the script and reference them later. Changing a run directory or where an input file is kept then becomes much more manageable.

One particularly important environment variable is $PATH. This is a colon-separated list of locations in which the shell looks for commands:

```
$ echo $PATH
/usr/kerberos/bin:/usr/java/latest/bin:/usr/local/bin:/bin:/usr/bin:/opt/ganglia/bin:
/opt/ganglia/sbin:/opt/rocks/bin:/opt/rocks/sbin:/opt/dell/srvadmin/bin:/opt/torque/bin:/opt/torque/sbin:
/home/ab123/bin
```

So when I enter "ls" at the command prompt, the shell looks for an executable file in /usr/kerberos/bin, then in /usr/java/latest/bin, and so on until it finds one.

This behavior becomes more significant in session 3, when we start to use Environment Modules to make software packages accessible. One of the things that loading an Environment Module does is to add the appropriate directories to your $PATH variable.

To see which executable will be run when you type a command, there is a command called `which`. For example: "which ls" will (probably) show you that entering "ls" at the command prompt will run /bin/ls.

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.
Finding a software package on the NYU HPC clusters

The command for seeing what software packages are available is:

```bash
$ module avail
```

The module command selects its subcommand based on the first unique match it finds for the letters typed so far, hence "avail" matches "available". You can in fact shorten it further, to "ava".

This will produce a long list of software packages. 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/3.3.4` - fftw version 3.3.4, built with the Intel compiler suite
- `fftw/mvapich2/intel/2.1.5` - fftw version 2.1.5, built for MPI with MVAPICH2 and the Intel compiler suite
- `fftw/mvapich2/intel/3.3.4` - fftw version 3.3.4, built for MPI with MVAPICH2 and 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

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/2014a`

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:

```bash
$ module avail fftw
```

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

```bash
$ 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.

### Finding out more about a software package

Most modulefiles include a one-line description of what the software package does, which can be seen with the command:

```bash
$ module whatis module-name
```

For example:
A module file may include more detailed help for the software package, which can be seen with:

```bash
$ module help module-name
```

For example:

```bash
$ module help cufflinks/intel/2.1.1
```

```
--------- Module Specific Help for 'cufflinks/intel/2.1.1'
---------------------------
Adds cufflinks to your environment.
```

As the example illustrates, the NYU HPC module files currently have limited help information. We recommend visiting the website of the software package for accurate and up-to-date documentation.

You can also see exactly what effect loading the module will have with:

```bash
$ module show module-name
```

For example:

```bash
$ module show cufflinks/intel/2.1.1
```

```
-----------------------------------------------
/share/apps/modules/modulefiles/cufflinks/intel/2.1.1:
module-whatis Cufflinks assembles transcripts, estimates their abundances, and tests for differential expression and regulation in RNA-Seq samples
conflict cufflinks
module load intel/11.1.046
module load zlib/intel/1.2.5
module load python/intel/2.7.2
module load boost/intel/serial/1.47.0
module load eigen/3.0.5
prepend-path PATH /share/apps/cufflinks/2.1.1/intel/bin
setenv CUFFLINKS_ROOT /share/apps/cufflinks/2.1.1/intel
-----------------------------------------------
```

Note that the cufflinks module, like many others on the NYU HPC clusters, loads a number of other modules. This is to ensure that when you use a module, your environment is consistent with the environment in which the package was built and tested. To see what modules are currently loaded in your environment use "module list" as described below (TODO link).

What modules do I currently have loaded?

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

```bash
$ module list
```
### Loading and unloading modules

To load a module:

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

For example:

```
$ module load fftw/intel/3.3.4
```

**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 the module:

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

For example:

```
$ module unload fftw
```

Specifying the full module name is not as important when unloading the module.

### Unloading all modules

You can remove all loaded modules from your environment with:

```
$ module purge
```

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

### Installing a software package on the NYU HPC clusters

Most software is installed under /share/apps/. Users can always build software packages and install in your $HOME folder, but only the system administrators can install in /share/apps.

You can always email us at hpc@nyu.edu to request a software package you need. It helps if you can include a link to its web page or download page. If it is for Python or R, it also helps if you tell us which Python or R module you are using.

Python and R also allow you to install modules in your $HOME area.

**Python**

For Python packages from PyPI and for Python 2.7.6, you can use pip:

```
$ module load python/intel/2.7.6
$ pip install --user <pkg_name>
```

*package will be installed in $HOME/.local.*
$ module load r/intel/3.2.0
$ R CMD INSTALL "<pkg_name>"

The package will be installed in $HOME/R/x86_64-centos-linux-gnu-library.

Some R packages have dependencies on other modules, and you will need to tell R where to find libraries (PKG_LIB) and possibly headers (PKG_CFLAGS). For example, the "topicmodels" package requires the GNU Scientific library (gsl):

$ module load r/intel/3.2.0
$ module load gsl/intel/1.16
$ export PKG_CFLAGS=-I$GSL_INC
$ export PKG_LIBS="-L $GSL_LIB"
$ R CMD INSTALL topicmodels

# sometimes installing from within R is more successful with finding the package:
$ R
> install.packages('topicmodels')

R packages generally have a file src/Makevars with additional compilation options, such as PKG_LIBS. Unfortunately, many specify these options in a way that overrides your setting of PKG_LIBS. To fix this you will need to download the .tar.gz file for the package, edit the Makevars file, and install the edited package, such as illustrated below:
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/env-modules.sh
  ```

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

In the case of a PBS job script, add one of the above lines before the first "module load" 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 again.