## Tutorial 2: HPC at NYU

**Back to the NYU HPC Wiki**
**Back to Tutorials index**
**Back to the Introduction**

**Prequel:** preparing your workstation for the HPC access (and this tutorial)

### Accessing software with Environment Modules

**Job scripts and how to reserve resources**
- Introduction to job scheduling
- Submitting a job with qsub
- Requesting resources
- Requesting GPUs
- Using compute nodes interactively
Finding and using software with 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:

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 *module file* 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 module file 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:

```
$ 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:

```
$ 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
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/3.3.4
```

**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.

What modules do I currently have loaded?

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

$ module list