What are modules on the Linux systems and how do I use them?

Scope

This document provides information about using *modules* on the Unified Linux Systems managed by the School of Informatics and Computing. Modules provide a way to make new software available as well as providing a mechanism whereby multiple versions of a single package are easily accessible. There are other standalone Linux systems in the school that are not part of the unified configuration and, therefore, this document does not apply to those systems.

The approach to the maintenance of software on the unified Linux systems is to keep the standard versions of software as supplied by the linux distribution as the default. We then use a package called Environment Modules to make it easy for users to get access to new packages or versions of software other than the OS default. There is more information about software installs on the unified linux systems at How do I get new or updated software installed on the unified Linux systems?

Basic Module Usage

When you log into the unified Linux systems, you will have access to the *module* command. Using *module* you can manipulate your environment (like your PATH and LD_LIBRARY_PATH) to gain access to new software or different versions of the same package. For example, to make Python version 3.4 the default, you would run:

```
module load python/3.4
```

We also set a module default, which is generally the most current stable version of a package. You can load this module default by omitting the version number. For example, if you just use:

```
module load python
```

then you will get the most recent stable version of python that we have available. *We recommend that you use the defaults unless you specifically need a particular version.*

Here are some module commands you may find useful:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>module avail</td>
<td>List the available modules. Note that if there are multiple versions of a single package that one will be denoted as (default). If you load the module without a version number you will get this default version.</td>
</tr>
<tr>
<td>module whatis</td>
<td>List all the available modules along with a short description.</td>
</tr>
<tr>
<td>module load MODULE</td>
<td>Load the named module.</td>
</tr>
<tr>
<td>module unload MODULE</td>
<td>Unload the named module, reverting back to the OS defaults.</td>
</tr>
<tr>
<td>module list</td>
<td>List all the currently loaded modules.</td>
</tr>
<tr>
<td>module help</td>
<td>Get general help information about modules.</td>
</tr>
<tr>
<td>module help MODULE</td>
<td>Get help information about the named module.</td>
</tr>
<tr>
<td>module show MODULE</td>
<td>Show details about the module, including the changes that loading the module will make to your environment.</td>
</tr>
</tbody>
</table>

Making Permanent Changes

Running the module command only changes the default for the current shell. You can make this change permanent by added this module line to your .bash_profile (if using the bash shell) or .login file (if using csh or tcsh). Note that the default shell for new accounts is bash but you can confirm the shell you are using by running "echo $SHELL". After you make changes to your .bash_profile or .login file these changes will be in effect the next time you login.

Modules and Scripts

When you are using modules, you may also need to take care if you are writing things like python scripts. A common convention in a python script is to use a first line like the following:

```
#!/usr/bin/python
```
The problem with this is that /usr/bin/python will always be the default version and not the version you get via modules. You can work around this problem using the following syntax instead:

```bash
#!/usr/bin/env python
```

What this does is run the version of python that is the default for your environment rather than the system default.

However, please note that using `/usr/bin/env` doesn't work with CGI scripts on the CGI server because in that environment your bash or tcsh config files (e.g., `.bash_profile`) are not invoked. In that case, you must explicitly use the full path to the proper script binary. For example, you can run "which python" after loading the module and then use the path displayed with something like the following in your script:

```bash
#!/l/python1.2.3/bin /python
```

## Modules and Crontabs

There are situations that you want to invoke the programs set up by "module load ..." in your cron job. Here is an example showing one way to do it

For BASH users:

```
0 4 * * * . /etc/profile.d/modules.sh; module load python/2.7; which python
```

For tcsh users:

```
0 4 * * * . /etc/profile.d/modules.csh; module load python/2.7; which python
```

## Creating Modules

If you want to create your own module files, all you have to do is create a directory named `privatemodules` in your home directory and put your modules there. Once that directory exists, the module system will search it for module files the next time you login (and you should see them if you run "module avail"). If you aren't sure how to create module files, one easy way to get started is to look at the files in `/etc/modulefiles` for examples.