Using Modules

Modules are the way Research Computing provides a lot of software without it all conflicting with each other. Modules are also the way we provide multiple versions of the same software.

NOTE: Sometimes module is not the best option for loading software. Instead, you may want to use spack load; we have a guide for when to use each.

Getting to Know Modules

After logging on to a computer managed by Research Computing you will be able to use the module command. module works by manipulating your environment to give you access to the software. The following is a list of subcommands for `module` that will allow to take full advantage of `module`. Note: `[module]` is the name of the module you want to load.

`module avail [module]`	Lists all the available modules with their version in a formatted list. Specifying the module will list the multiple versions of the module available. Notice (default) next to some modules, this indicates that if you load the module without specifying which version, this version will be loaded.
`module whatis [module]`	Outputs a brief description of the module specified. Note: if no module is specified, all the modules will be listed
`module load [module]`	Loads the specified module. A version can also be specified, see the demonstration below.
`module unload [module]`	Unloads the specified module.
`module list`	Lists all the currently loaded modules.
`module help`	Explains how to use the module command, includes a list of all available command.
`module help [module]`	Get help information for the specified module. Many modules do not have very much information however.
`module show [module]`	Shows information about the module and the changes the module will make to your environment.

A Demonstration of `module`

The following is a demonstration of the module commands to give you an idea of how the commands listed above behave. For this demonstration we will be trying to load the newest version of Python available on our system (3.7.2).

1. We will check to see if a version of module is already available

   ```
   [abc1234@computer ~]$ python -V

   Python 2.7.5
   ```

2. So now that we know that version currently loaded for Python is not version we want, we look to see if that version is available:

   ```
   [abc1234@computer ~]$ module avail python
   -------- /tools/spack/Modulefiles.tcl/linux-rhel7-x86_64 --------

   python/2.7.16-gcc-7.4.0-2geicowv python/3.7.2--gcc-7.4.0-6on2qnpy

   python/2.7.16-gcc-7.4.0-6yshniaz

   ------------------- /tools/Modules/modulefiles ------------------

   python/2.7.10 python/2.7.12 python/2.7.9 python/3.5.2
   ```

   Note: `module avail` would have worked just as well to find which versions of Python are available. I recommend taking a look at all the modules to see what is available to you when you are first getting started.
3. Now that we know it exists, we are ready to load the module:

```
[abc1234@computer ~]$ module load python/3.7.2--gcc-7.4.0-6on2qnpy
```

Note: If we saw (default) next to version of python we wanted to load we could have just typed `module load python` and that would have loaded the version Research Computing recommends. We encourage you to load the default version unless you know you need another version.

Protip: For long module names you can hit tab on your keyboard to autocomplete after you have typed enough for the computer to understand what you are trying to type. In this case `module load python/3.7` would have been enough to autocomplete `python/3.7.2--gcc-7.4.0-6on2qnpy`. Autocomplete works for many commands, try it.

4. Now we can check that the module was installed:

   ```
   [abc1234@computer ~]$ module list

   Currently Loaded Modulefiles

     1) tools-dir

     2) module_defaults

     3) system_defaults

     4) module_rhel7

     5) module_class

     6) ncurses/6.1-gxx-7.4.0-6f66ubcf

     . . .

    17) python/3.7.2-gcc-7.4.0-6on2qnpy
   ```

   If we ran this command prior to loading Python you would only see the first five modules that are loaded as default. We can also see many other modules that were loaded alongside Python. Not all modules load other with it; it depends on what the module needs and how it changes the environment.
5. To make sure the version was updated we can check the version again:

   ```
   [abc1234@computer ~]$ python -V

   Python 3.7.2
   ```

6. Now let's say we are done using that version of Python or we loaded the wrong one and we want to change it. We need to unload it:

   ```
   [abc1234@computer ~]$ module unload python/3.7.2--gcc-7.4.0-6on2qnpy
   ```

7. Now if we run `python -V` we will see python is once again 2.7.5. If we run `module list` we will see `python/3.7.2`… is gone, but not the modules loaded with it. Those can be removed easily by listing the modules after the unload command:

   ```
   [abc1234@computer ~]$ module unload[module1][module2] [module3]…
   ```

   Note: The autocomplete protip would speed up the process significantly and prevent typos that would break the command.

Tricky Modules

Sometimes modules are not explicitly listed in module avail. For example, idl, which is used by many imaging scientists does not look to be available. This is because it is contained in another module, envi. The following demonstrates this:

```
[abc1234@computer ~]$ which idl

/usr/bin/which: no idl in (/tools/spack/spackages/...)

[abc1234@computer ~]$ module load envi

[abc1234@computer ~]$ which idl

/tools/envi/5.2/idl84/bin/idl
```

The `which` command here is useful to see if you can run a program without trying to open it.

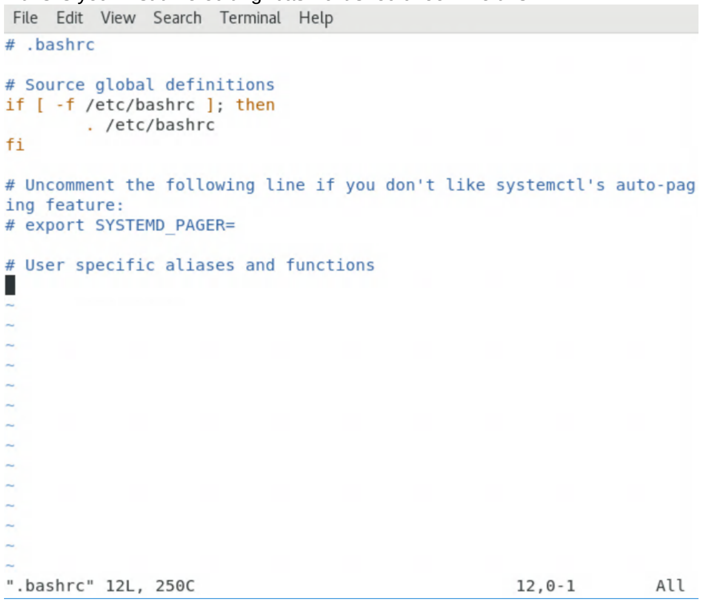
Making Your Changes Permanent

Loading modules is not permanent; when you close the terminal it will go back to its original environment. If you know that every time you open the terminal you will need to load certain modules you can edit your .bashrc to load the modules automatically when the terminal is open. To do this:

1. Open .bashrc (located as a hidden file in your home directory) with your choice of text editor:

   ```
   [abc1234@computer ~]$ vim ~/.bashrc
   ```

If this is your first time editing .bashrc it should look like this:

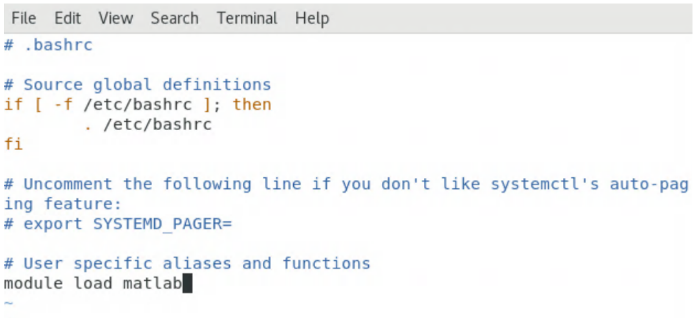


```
# .bashrc

# Source global definitions
if [ -f /etc/bashrc ]; then
        . /etc/bashrc
fi

# Uncomment the following line if you don't like systemctl's auto-pag
ing feature:
# export SYSTEMD_PAGER=

# User specific aliases and functions

~
~
~
~
~
~
~
~
~
~
~
~
~
~
".bashrc" 12L, 250C                                12,0-1        All
```

2. On the line beneath `# User specific aliases…` type:

   `module load [module]`

   In this example I will be loading the default Matlab module:



```
# .bashrc

# Source global definitions
if [ -f /etc/bashrc ]; then
        . /etc/bashrc
fi

# Uncomment the following line if you don't like systemctl's auto-pag
ing feature:
# export SYSTEMD_PAGER=

# User specific aliases and functions
module load matlab
~
```

3. Save your changes and exit the text editor
4. Now if you run `module list` you will notice the module was not loaded. .bashrc is only read when the terminal is opened. To load those modules without closing and reopening the terminal run:

   ```
   [abc1234@computer ~]$ source ~/.bashrc
   ```

5. Now when you run `module list` you will see your module there as loaded
6. The next time you open your terminal (for SSH users this is logging in and FastX users this is opening from activities) the module will be loaded.

Modules and Scripts

When you are using modules you must take caution when writing scripts that define the interpreter in the file. For example in a Python script the first line is commonly:

```
#!/usr/bin/python
```

If you are using modules and decide to use a version of Python that is not the default, such as 3.7.2, this line will not point to that interpreter, but rather the default. Instead use:

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

This will point to the version of Python you loaded into your environment.

If there are any further questions, or there is an issue with the documentation, please contact rc-help@rit.edu *for additional assistance.*