VLAAMS SUPERCOMPUTER CENTRUM | *Innovative Computing for A Smarter Flanders*

Linux Tutorial

Last updated: May 4 2022

For Linux Users

Authors:

Jasper Devreker (UGent), Ewan Higgs (UGent), Kenneth Hoste (UGent)

Acknowledgement: VSCentrum.be

Audience:

This document is a hands-on guide for using the Linux command line in the context of the **UGent** HPC infrastructure. The command line (sometimes called 'shell') can seems daunting at first, but with a little understanding can be very easy to use. Everything you do starts at the prompt. Here you have the liberty to type in any commands you want. Soon, you will be able to move past the limited point and click interface and express interesting ideas to the computer using the shell.

Gaining an understanding of the fundamentals of Linux will help accelerate your research using the HPC infrastructure. You will learn about commands, managing files, and some scripting basics.

Notification:

```
$ commands
```

These should be entered by the reader at a command line in a terminal on the UGent-HPC. They appear in all exercises preceded by a $ and printed in **bold**. You'll find those actions in a grey frame.

Button are menus, buttons or drop down boxes to be pressed or selected.

"Directory" is the notation for directories (called "folders" in Windows terminology) or specific files. (e.g., "/user/home/gent/vsc400/vsc40000")

"Text" Is the notation for text to be entered.

Tip: A "Tip" paragraph is used for remarks or tips.

They can also be downloaded from the VSC website at https://www.vscentrum.be. Apart from this HPC Tutorial, the documentation on the VSC website will serve as a reference for all the operations.

Tip: The users are advised to get self-organised. There are only limited resources available at the HPC, which are best effort based. The HPC cannot give support for code fixing, the user applications and own developed software remain solely the responsibility of the end-user.

More documentation can be found at:

1. VSC documentation: https://www.vscentrum.be/user-portal
2. External documentation (TORQUE, Moab): http://docs.adaptivecomputing.com

This tutorial is intended for users working on **Linux** who want to connect to the HPC of the **UGent**.

This tutorial is available in a Windows, Mac or Linux version.

This tutorial is available for UAntwerpen, UGent, KU Leuven, UHasselt and VUB users.

Request your appropriate version at hpc@ugent.be.

Contact Information:

We welcome your feedback, comments and suggestions for improving the Linux Tutorial (contact: hpc@ugent.be).

For all technical questions, please contact the HPC staff:

1. Website: http://www.ugent.be/hpc
2. By e-mail: hpc@ugent.be
3. In real: Directie Informatie- en Communicatietechnologie, Krijgslaan 291 Building S9, 9000 Gent
4. Follow us on Twitter: http://twitter.com/HPCUGent

Mailing-lists:

1. Announcements: hpc-announce@lists.ugent.be (for official announcements and communications)
2. Users: hpc-users@lists.ugent.be (for discussions between users)

Glossary

cluster A group of compute nodes.

compute node The computational units on which batch or interactive jobs are processed. A compute node is pretty much comparable to a single personal computer. It contains one or more sockets, each holding a single CPU. Some nodes also contain one or more GPGPUs. The compute node is equipped with memory (RAM) that is accessible by all its CPUs.

core An individual compute unit inside a CPU. A CPU typically contains one or more cores.

Linux An operating system, similar to UNIX.

login node On HPC clusters, login nodes serve multiple functions. From a login node you can submit and monitor batch jobs, analyse computational results, run editors, plots, debuggers, compilers, do housekeeping chores as adjust shell settings, copy files and in general manage your account. You connect to these servers when want to start working on the UGent-HPC.

memory A quantity of physical memory (RAM). Memory is provided by compute nodes. It is required as a constraint or consumed as a consumable resource by jobs. Within Moab, memory is tracked and reported in megabytes (MB).

Moab Moab is a job scheduler, which allocates resources for jobs that are requesting resources.

modules HPC uses an open source software package called "Environment Modules" (Modules for short) which allows you to add various path definitions to your shell environment.

MPI MPI stands for Message-Passing Interface. It supports a parallel programming method designed for distributed memory systems, but can also be used well on shared memory systems.

node See compute node.

queue PBS/TORQUE queues, or "classes" as Moab refers to them, represent groups of computing resources with specific parameters. A queue with a 12 hour runtime or "walltime" would allow jobs requesting 12 hours or less to use this queue.

scratch Supercomputers generally have what is called scratch space: storage available for temporary use. Use the scratch filesystem when, for example you are downloading and uncompressing applications, reading and writing input/output data during a batch job, or when you work with large datasets. Scratch is generally a lot faster then the Data or Home filesystem.

Contents

Glossary 4

I Beginner's Guide 8

1 Getting Started 9

1.1 Logging in . . . 9

1.2 Getting help . . . 9

1.2.1 Errors . . . 9

1.3 Basic terminal usage . . . 10

1.3.1 Command history . . . 10

1.3.2 Stopping commands . . . 10

1.4 Variables . . . 10

1.4.1 Defining variables . . . 10

1.4.2 Using non-defined variables . . . 11

1.4.3 Restoring your default environment . . . 12

1.5 Basic system information . . . 12

1.6 Exercises . . . 12

2 Navigating 14

2.1 Current directory: "pwd" and "$PWD" . . . 14

2.2 Listing files and directories: "ls" . . . 14

2.3 Changing directory: "cd" . . . 15

2.4 Inspecting file type: "file" . . . 15

2.5 Absolute vs relative file paths . . . 16

2.6 Permissions . . . 16

2.7 Finding files/directories: “find” . . . 17
2.8 Exercises . . . 17

3 Manipulating files and directories **18**
3.1 File contents: “cat”, “head”, “tail”, “less”, “more” . . . 18
3.2 Copying files: “cp” . . . 18
3.3 Creating directories: “mkdir” . . . 19
3.4 Renaming/moving files: “mv” . . . 19
3.5 Removing files: “rm” . . . 19
3.6 Changing permissions: “chmod” . . . 19
3.6.1 Access control lists (ACLs) . . . 21
3.7 Zipping: “gzip”/“gunzip”, “zip”/“unzip” . . . 21
3.7.1 “zip” and “unzip” . . . 21
3.8 Working with tarballs: “tar” . . . 22
3.8.1 Order of arguments . . . 22
3.9 Exercises . . . 22

4 Uploading/downloading/editing files **24**
4.1 Uploading/downloading files . . . 24
4.2 Symlinks for data/scratch . . . 24
4.3 Editing with `nano` . . . 24
4.4 Copying faster with `rsync` . . . 25
4.5 Exercises . . . 25

5 Beyond the basics **26**
5.1 Input/output . . . 26
5.1.1 Redirecting `stdout` . . . 26
5.1.2 Reading from `stdin` . . . 27
5.1.3 Redirecting `stderr` . . . 27
5.1.4 Combining `stdout` and `stderr` . . . 27
5.2 Command piping . . . 27
5.3 Shell expansion . . . 28
5.4 Process information . . . 28
5.4.1 `ps` and `pstree` . . . 28

5.4.2 kill . . . 29
5.4.3 top . . . 29
5.4.4 ulimit . . . 29
5.5 Counting: wc . . . 29
5.6 Searching file contents: grep . . . 30
5.7 cut . . . 30
5.8 sed . . . 30
5.9 awk . . . 30
5.10 Basic Shell Scripting . . . 31
5.10.1 Shebang . . . 31
5.10.2 Conditionals . . . 31
5.10.3 Loops . . . 32
5.10.4 Subcommands . . . 32
5.10.5 Errors . . . 32
5.11 .bashrc login script . . . 33
5.12 Scripting for the cluster . . . 34
5.12.1 Example job script . . . 34
5.12.2 PBS pragmas . . . 34
5.13 Exercises . . . 35

6 Common Pitfalls **36**
6.0.1 Files . . . 36
6.0.2 Help . . . 37

7 More information **39**

8 Q & A **40**

9 More on the HPC infrastructure **41**
9.1 Filesystems . . . 41
9.1.1 Quota . . . 41
9.2 Modules . . . 41
9.3 Using the clusters . . . 42
9.4 Exercises . . . 42

Part I

Beginner's Guide

Chapter 1

Getting Started

1.1 Logging in

To get started with the HPC-UGent infrastructure, you need to obtain a VSC account, see Chapter 2 of the HPC manual. **Keep in mind that you must keep your private key to yourself!**

You can look at your public/private key pair as a lock and a key: you give us the lock (your public key), we put it on the door, and then you can use your key to open the door and get access to the HPC infrastructure. **Anyone who has your key can use your VSC account!**

Details on connecting to the HPC infrastructure are available in Chapter 3 of the HPC manual.

1.2 Getting help

To get help:

1. use the documentation available on the system, through the `help`, `info` and `man` commands (use `q` to exit).

```
help cd
info ls
man cp
```

2. use Google
3. contact hpc@ugent.be in case of problems or questions (even for basic things!)

1.2.1 Errors

Sometimes when executing a command, an error occurs. Most likely there will be error output or a message explaining you this. Read this carefully and try to act on it. Try googling the error first to find any possible solution, but if you can't come up with something in 15 minutes, don't hesitate to mail hpc@ugent.be.

1.3 Basic terminal usage

The basic interface is the so-called shell prompt, typically ending with `$` (for `bash` shells).

You use the shell by executing commands, and hitting `<enter>`. For example:

```
$ echo hello
hello
```

You can go to the start or end of the command line using `Ctrl-A` or `Ctrl-E`.

To go through previous commands, use `<up>` and `<down>`, rather than retyping them.

1.3.1 Command history

A powerful feature is that you can "search" through your command history, either using the `history` command, or using `Ctrl-R`:

```
$ history
    1  echo hello

# hit Ctrl-R, type 'echo'
(reverse-i-search)`echo': echo hello
```

1.3.2 Stopping commands

If for any reason you want to stop a command from executing, press `Ctrl-C`. For example, if a command is taking too long, or you want to rerun it with different arguments.

1.4 Variables

At the prompt we also have access to shell variables, which have both a *name* and a *value*.

They can be thought of as placeholders for things we need to remember.

For example, to print the path to your home directory, we can use the shell variable named `HOME`:

```
$ echo $HOME
/user/home/gent/vsc400/vsc40000
```

This prints the value of this variable.

1.4.1 Defining variables

There are several variables already defined for you when you start your session, such as `$HOME` which contains the path to your home directory.

For a full overview of defined environment variables in your current session, you can use the `env` command. You can sort this output with `sort` to make it easier to search in:

```
$ env | sort
...
HOME=/user/home/gent/vsc400/vsc40000
...
```

You can also use the `grep` command to search for a piece of text. The following command will output all VSC-specific variable names and their values:

```
$ env | sort | grep VSC
```

But we can also define our own. this is done with the `export` command (note: variables are always all-caps as a convention):

```
$ export MYVARIABLE="value"
```

It is important you don't include spaces around the = sign. Also note the lack of `$` sign in front of the variable name.

If we then do

```
$ echo $MYVARIABLE
```

this will output `value`. Note that the quotes are not included, they were only used when defining the variable to escape potential spaces in the value.

Changing your prompt using `$PS1`

You can change what your prompt looks like by redefining the special-purpose variable `$PS1`.

For example: to include the current location in your prompt:

```
$ export PS1='\w $'
~ $ cd test
~/test $
```

Note that ~ is short representation of your home directory.

To make this persistent across session, you can define this custom value for `$PS1` in your `.profile` startup script:

```
$ echo 'export PS1="\w $ " ' >> ~/.profile
```

1.4.2 Using non-defined variables

One common pitfall is the (accidental) use of non-defined variables. Contrary to what you may expect, this does *not* result in error messages, but the variable is considered to be *empty* instead.

This may lead to surprising results, for example:

```
$ export WORKDIR=/tmp/test
$ cd $WROKDIR
$ pwd
/user/home/gent/vsc400/vsc40000
$ echo $HOME
/user/home/gent/vsc400/vsc40000
```

To understand what's going on here, see the section on `cd` below.

The moral here is: **be very careful to not use empty variables unintentionally**.

Tip for job scripts: use `set -e -u` to avoid using empty variables accidentally.

The `-e` option will result in the script getting stopped if any command fails.

The `-u` option will result in the script getting stopped if empty variables are used. (see https://ss64.com/bash/set.html for a more detailed explanation and more options)

More information can be found at http://www.tldp.org/LDP/abs/html/variables.html .

1.4.3 Restoring your default environment

If you've made a mess of your environment, you shouldn't waste too much time trying to fix it. Just log out and log in again and you will be given a pristine environment.

1.5 Basic system information

Basic information about the system you are logged into can be obtained in a variety of ways.

We limit ourselves to determining the hostname:

```
$ hostname
gligar01.gligar.os

$ echo $HOSTNAME
gligar01.gligar.os
```

And querying some basic information about the Linux kernel:

```
$ uname -a
Linux gligar01.gligar.os 2.6.32-573.8.1.el6.ug.x86_64 #1 SMP Mon Nov 16 15:12:09
  CET 2015 x86_64 x86_64 x86_64 GNU/Linux
```

1.6 Exercises

- Print the full path to your home directory
- Determine the name of the environment variable to your personal scratch directory
- What's the name of the system you're logged into? Is it the same for everyone?

- Figure out how to print the value of a variable without including a newline
- How do you get help on using the `man` command?

Chapter 2

Navigating

2.1 Current directory: "pwd" and "$PWD"

To print the current directory, use `pwd` or `$PWD`:

```
$ cd $HOME
$ pwd
/user/home/gent/vsc400/vsc40000
$ echo "The current directory is:  $PWD"
The current directory is: /user/home/gent/vsc400/vsc40000
```

2.2 Listing files and directories: "ls"

A very basic and commonly used command is `ls`, which can be used to list files and directories.

In it's basic usage, it just prints the names of files and directories in the current directory. For example:

```
$ ls
afile.txt   some_directory
```

When provided an argument, it can be used to list the contents of a directory:

```
$ ls some_directory
one.txt  two.txt
```

A couple of commonly used options include:

- detailed listing using `ls -l`:

  ```
  $ ls -l
  total 4224
  -rw-rw-r-- 1 vsc40000 vsc40000 2157404 Apr 12 13:17 afile.txt
  drwxrwxr-x 2 vsc40000 vsc40000     512 Apr 12 12:51 some_directory
  ```

- To print the size information in human-readable form, use the `-h` flag:

```
$ ls -lh
total 4.1M
-rw-rw-r-- 1 vsc40000 vsc40000 2.1M Apr 12 13:16 afile.txt
drwxrwxr-x 2 vsc40000 vsc40000  512 Apr 12 12:51 some_directory
```

- also listing hidden files using the `-a` flag:

```
$ ls -lah
total 3.9M
drwxrwxr-x   3 vsc40000 vsc40000  512 Apr 12 13:11 .
drwx------ 188 vsc40000 vsc40000 128K Apr 12 12:41 ..
-rw-rw-r--   1 vsc40000 vsc40000 1.8M Apr 12 13:12 afile.txt
-rw-rw-r--   1 vsc40000 vsc40000    0 Apr 12 13:11 .hidden_file.txt
drwxrwxr-x   2 vsc40000 vsc40000  512 Apr 12 12:51 some_directory
```

- ordering files by the most recent change using `-rt`:

```
$ ls -lrth
total 4.0M
drwxrwxr-x 2 vsc40000 vsc40000  512 Apr 12 12:51 some_directory
-rw-rw-r-- 1 vsc40000 vsc40000 2.0M Apr 12 13:15 afile.txt
```

If you try to use `ls` on a file that doesn't exist, you will get a clear error message:

```
$ ls nosuchfile
ls: cannot access nosuchfile: No such file or directory
```

2.3 Changing directory: "cd"

To change to a different directory, you can use the `cd` command:

```
$ cd some_directory
```

To change back to the previous directory you were in, there's a shortcut: `cd -`

Using `cd` without an argument results in returning back to your home directory:

```
$ cd
$ pwd
/user/home/gent/vsc400/vsc40000
```

2.4 Inspecting file type: "file"

The `file` command can be used to inspect what type of file you're dealing with:

```
$ file afile.txt
afile.txt: ASCII text

$ file some_directory
some_directory: directory
```

2.5 Absolute vs relative file paths

An *absolute* filepath starts with `/` (or a variable which value starts with `/`), which is also called the *root* of the filesystem.

Example: absolute path to your home directory: `/user/home/gent/vsc400/vsc40000`.

A *relative* path starts from the current directory, and points to another location up or down the filesystem hierarchy.

Example: `some_directory/one.txt` points to the file `one.txt` that is located in the subdirectory named `some_directory` of the current directory.

There are two special relative paths worth mentioning:

- `.` is a shorthand for the current directory
- `..` is a shorthand for the parent of the current directory

You can also use `..` when constructing relative paths, for example:

```
$ cd $HOME/some_directory
$ ls ../afile.txt
../afile.txt
```

2.6 Permissions

Each file and directory has particular *permissions* set on it, which can be queried using `ls -l`.

For example:

```
$ ls -l afile.txt
-rw-rw-r-- 1 vsc40000 agroup 2929176 Apr 12 13:29 afile.txt
```

The `-rwxrw-r--` specifies both the type of file (`-` for files, `d` for directories (see first character)), and the permissions for user/group/others:

1. each triple of characters indicates whether the read (`r`), write (`w`), execute (`x`) permission bits are set or not
2. the 1st part `rwx` indicates that the *owner* "vsc40000" of the file has all the rights
3. the 2nd part `rw-` indicates the members of the *group* "agroup" only have read/write permissions (not execute)
4. the 3rd part `r--` indicates that *other* users only have read permissions

The default permission settings for new files/directories are determined by the so-called *umask* setting, and are by default:

1. read-write permission on files for user/group (no execute), read-only for others (no write/execute)

2. read-write-execute permission for directories on user/group, read/execute-only for others (no write)

See also the chmod command later in this manual.

2.7 Finding files/directories: "find"

`find` will crawl a series of directories and lists files matching given criteria.

For example, to look for the file named `one.txt`:

```
$ cd $HOME
$ find .  -name one.txt
./some_directory/one.txt
```

To look for files using incomplete names, you can use a wildcard `*`; note that you need to escape the `*` to avoid that Bash *expands* it into `afile.txt` by adding double quotes:

```
$ find .  -name "*.txt"
./.hidden_file.txt
./afile.txt
./some_directory/one.txt
./some_directory/two.txt
```

A more advanced use of the `find` command is to use the `-exec` flag to perform actions on the found file(s), rather than just printing their paths (see `man find`).

2.8 Exercises

- Go to `/tmp`, then back to your home directory. How many different ways to do this can you come up with?
- When was your home directory created or last changed?
- Determine the name of the last changed file in `/tmp`.
- See how home directories are organised. Can you access the home directory of other users?

Chapter 3

Manipulating files and directories

Being able to manage your data is an important part of using the HPC infrastructure. The bread and butter commands for doing this are mentioned here. It might seem annoyingly terse at first, but with practice you will realise that it's very practical to have such common commands short to type.

3.1 File contents: "cat", "head", "tail", "less", "more"

To print the contents of an entire file, you can use `cat`; to only see the first or last N lines, you can use `head` or `tail`:

```
$ cat one.txt
1
2
3
4
5

$ head -2 one.txt
1
2

$ tail -2 one.txt
4
5
```

To check the contents of long text files, you can use the `less` or `more` commands which support scrolling with "<up>", "<down>", "<space>", etc.

3.2 Copying files: "cp"

```
$ cp source target
```

This is the `cp` command, which copies a file from source to target. To copy a directory, we use the `-r` option:

```
$ cp -r sourceDirectory target
```

A last more complicated example:

```
$ cp -a sourceDirectory target
```

Here we used the same `cp` command, but instead we gave it the `-a` option which tells cp to copy all the files and keep timestamps and permissions.

3.3 Creating directories: "mkdir"

```
$ mkdir directory
```

which will create a directory with the given name inside the current directory.

3.4 Renaming/moving files: "mv"

```
$ mv source target
```

`mv` will move the source path to the destination path. Works for both directories as files.

3.5 Removing files: "rm"

Note: there are NO backups, there is no 'trash bin'. If you remove files/directories, they are gone.

```
$  rm filename
```

`rm` will remove a file or directory. (`rm -rf directory` will remove every file inside a given directory). WARNING: files removed will be lost forever, there are no backups, so beware when using this command!

Removing a directory: "rmdir"

You can remove directories using `rm -r directory`, however, this is error prone and can ruin your day if you make a mistake in typing. To prevent this type of error, you can remove the contents of a directory using `rm` and then finally removing the directory with:

```
$ rmdir directory
```

3.6 Changing permissions: "chmod"

Every file, directory, and link has a set of permissions. These permissions consist of permission groups and permission types. The permission groups are:

1. User - a particular user (account)
2. Group - a particular group of users (may be user-specific group with only one member)
3. Other - other users in the system

The permission types are:

1. Read - For files, this gives permission to read the contents of a file
2. Write - For files, this gives permission to write data to the file. For directories it allows users to add or remove files to a directory.
3. Execute - For files this gives permission to execute a file as through it were a script. For directories, it allows users to open the directory and look at the contents.

Any time you run `ls -l` you'll see a familiar line of `-rwx------` or similar combination of the letters `r`, `w`, `x` and `-` (dashes). These are the permissions for the file or directory. (See also the previous section on permissions)

```
$ ls -l
total 1
-rw-r--r--. 1 vsc40000 mygroup 4283648 Apr 12 15:13 articleTable.csv
drwxr-x---. 2 vsc40000 mygroup      40 Apr 12 15:00 Project_GoldenDragon
```

Here, we see that `articleTable.csv` is a file (beginning the line with `-`) has read and write permission for the user `vsc40000` (`rw-`), and read permission for the group `mygroup` as well as all other users (`r--` and `r--`).

The next entry is `Project_GoldenDragon`. We see it is a directory because the line begins with a `d`. It also has read, write, and execute permission for the `vsc40000` user (`rwx`). So that user can look into the directory and add or remove files. Users in the `mygroup` can also look into the directory and read the files. But they can't add or remove files (`r-x`). Finally, other users can read files in the directory, but other users have no permissions to look in the directory at all (`---`).

Maybe we have a colleague who wants to be able to add files to the directory. We use `chmod` to change the modifiers to the directory to let people in the group write to the directory:

```
$ chmod g+w Project_GoldenDragon
$ ls -l
total 1
-rw-r--r--. 1 vsc40000 mygroup 4283648 Apr 12 15:13 articleTable.csv
drwxrwx---. 2 vsc40000 mygroup      40 Apr 12 15:00 Project_GoldenDragon
```

The syntax used here is `g+x` which means **g**roup was given **w**rite permission. To revoke it again, we use `g-w`. The other roles are `u` for user and `o` for other.

You can put multiple changes on the same line: `chmod o-rwx,g-rxw,u+rx,u-w somefile` will take everyone's permission away except the user's ability to read or execute the file.

You can also use the `-R` flag to affect all the files within a directory, but this is dangerous. It's best to refine your search using `find` and then pass the resulting list to `chmod` since it's not usual for all files in a directory structure to have the same permissions.

3.6.1 Access control lists (ACLs)

However, this means that all users in `mygroup` can add or remove files. This could be problematic if you only wanted one person to be allowed to help you administer the files in the project. We need a new group. To do this in the HPC environment, we need to use access control lists (ACLs):

```
$ setfacl -m u:otheruser:w Project_GoldenDragon
$ ls -l Project_GoldenDragon
drwxr-x---+ 2 vsc40000 mygroup      40 Apr 12 15:00 Project_GoldenDragon
```

This will give the **u**ser `otheruser` permissions to **w**rite to `Project_GoldenDragon`

Now there is a + at the end of the line. This means there is an ACL attached to the directory. `getfacl Project_GoldenDragon` will print the ACLs for the directory.

Note: most people don't use ACLs, but it's sometimes the right thing and you should be aware it exists.

See https://linux.die.net/man/1/setfacl for more information.

3.7 Zipping: "gzip"/"gunzip", "zip"/"unzip"

Files should usually be stored in a compressed file if they're not being used frequently. This means they will use less space and thus you get more out of your quota. Some types of files (e.g., CSV files with a lot of numbers) compress as much as 9:1. The most commonly used compression format on Linux is gzip. To compress a file using gzip, we use:

```
$ ls -lh myfile
-rw-r--r--. 1 vsc40000 vsc40000 4.1M Dec  2 11:14 myfile
$ gzip myfile
$ ls -lh myfile.gz
-rw-r--r--. 1 vsc40000 vsc40000 1.1M Dec  2 11:14 myfile.gz
```

Note: if you zip a file, the original file will be removed. If you unzip a file, the compressed file will be removed. To keep both, we send the data to `stdout` and redirect it to the target file:

```
$ gzip -c myfile > myfile.gz
$ gunzip -c myfile.gz > myfile
```

3.7.1 "zip" and "unzip"

Windows and macOS seem to favour the zip file format, so it's also important to know how to unpack those. We do this using unzip:

```
$ unzip myfile.zip
```

If we would like to make our own zip archive, we use zip:

```
$ zip myfiles.zip myfile1 myfile2 myfile3
```

3.8 Working with tarballs: "tar"

Tar stands for "tape archive" and is a way to bundle files together in a bigger file.

You will normally want to unpack these files more often than you make them. To unpack a `.tar` file you use:

```
$ tar -xf tarfile.tar
```

Often, you will find `gzip` compressed `.tar` files on the web. These are called tarballs. You can recognize them by the filename ending in `.tar.gz`. You can uncompress these using `gunzip` and then unpacking them using `tar`. But `tar` knows how to open them using the `-z` option:

```
$ tar -zxf tarfile.tar.gz
$ tar -zxf tarfile.tgz
```

3.8.1 Order of arguments

Note: Archive programs like `zip`, `tar`, and `jar` use arguments in the "opposite direction" of copy commands.

```
# cp, ln: <source(s)> <target>
$ cp source1 source2 source3 target
$ ln -s source target

# zip, tar: <target> <source(s)>
$ zip zipfile.zip source1 source2 source3
$ tar -cf tarfile.tar source1 source2 source3
```

If you use `tar` with the source files first then the first file will be overwritten. You can control the order of arguments of `tar` if it helps you remember:

```
$ tar -c source1 source2 source3 -f tarfile.tar
```

3.9 Exercises

1. Create a subdirectory in your home directory named `test` containing a single, empty file named `one.txt`.
2. Copy `/etc/hostname` into the `test` directory and then check what's in it. Rename the file to `hostname.txt`.
3. Make a new directory named `another` and copy the entire `test` directory to it. `another/test/one.txt` should then be an empty file.
4. Remove the `another/test` directory with a single command.
5. Rename `test` to `test2`. Move `test2/hostname.txt` to your home directory.
6. Change the permission of `test2` so only you can access it.
7. Create an empty job script named `job.sh`, and make it executable.

8. gzip `hostname.txt`, see how much smaller it becomes, then unzip it again.

Chapter 4

Uploading/downloading/editing files

4.1 Uploading/downloading files

To transfer files from and to the HPC, see the section about transferring files in chapter 3 of the HPC manual.

4.2 Symlinks for data/scratch

As we end up in the home directory when connecting, it would be convenient if we could access our data and VO storage. To facilitate this we will create symlinks to them in our home directory. This will create 4 symbolic links (they're like "shortcuts" on your desktop) pointing to the respective storages:

```
$ cd $HOME
$ ln -s $VSC_SCRATCH scratch
$ ln -s $VSC_DATA data
$ ls -l scratch data
lrwxrwxrwx 1 vsc40000 vsc40000 31 Mar 27  2009 data ->
    /user/data/gent/vsc400/vsc40000
lrwxrwxrwx 1 vsc40000 vsc40000 34 Jun  5  2012 scratch ->
    /user/scratch/gent/vsc400/vsc40000
```

4.3 Editing with **nano**

Nano is the simplest editor available on Linux. To open Nano, just type `nano`. To edit a file, you use `nano the_file_to_edit.txt`. You will be presented with the contents of the file and a menu at the bottom with commands like `^O Write Out` The `^` is the Control key. So `^O` means `Ctrl-O`. The main commands are:

1. Open ("Read"): `^R`
2. Save ("Write Out"): `^O`
3. Exit: `^X`

More advanced editors (beyond the scope of this page) are `vim` and `emacs`. A simple tutorial on how to get started with `vim` can be found at https://www.openvim.com/.

4.4 Copying faster with `rsync`

`rsync` is a fast and versatile copying tool. It can be much faster than `scp` when copying large datasets. It's famous for its "delta-transfer algorithm", which reduces the amount of data sent over the network by only sending the differences between files.

You will need to run `rsync` from a computer where it is installed. Installing `rsync` is the easiest on Linux: it comes pre-installed with a lot of distributions.

For example, to copy a folder with lots of CSV files:

```
$ rsync -rzv testfolder vsc40000@login.hpc.ugent.be:data/
```

will copy the folder `testfolder` and its contents to `$VSC_DATA` on the HPC, assuming the `data` symlink is present in your home directory, see section 4.2.

The `-r` flag means "recursively", the `-z` flag means that compression is enabled (this is especially handy when dealing with CSV files because they compress well) and the `-v` enables more verbosity (more details about what's going on).

To copy large files using `rsync`, you can use the `-P` flag: it enables both showing of progress and resuming partially downloaded files.

To copy files from the HPC to your local computer, you can also use `rsync`:

```
$ rsync -rzv vsc40000@login.hpc.ugent.be:data/bioset local_folder
```

This will copy the folder `bioset` and its contents that on `$VSC_DATA` of the HPC to a local folder named `local_folder`.

See `man rsync` or https://linux.die.net/man/1/rsync for more information about rsync.

4.5 Exercises

1. Download the file `/etc/hostname` to your local computer.
2. Upload a file to a subdirectory of your personal `$VSC_DATA` space.
3. Create a file named `hello.txt` and edit it using `nano`.

Chapter 5

Beyond the basics

Now that you've seen some of the more basic commands, let's take a look at some of the deeper concepts and commands.

5.1 Input/output

To redirect output to files, you can use the redirection operators: `>`, `>>`, `&>`, and `<`.

First, it's important to make a distinction between two different output channels:

1. `stdout`: standard output channel, for regular output
2. `stderr`: standard error channel, for errors and warnings

5.1.1 Redirecting `stdout`

`>` writes the (`stdout`) output of a command to a file and *overwrites* whatever was in the file before.

```
$  echo hello > somefile
$ cat somefile
hello
$ echo hello2 > somefile
$ cat somefile
hello2
```

`>>` appends the (`stdout`) output of a command to a file; it does not clobber whatever was in the file before:

```
$ echo hello > somefile
$ cat somefile
hello
$ echo hello2 >> somefile
$ cat somefile
hello
hello2
```

5.1.2 Reading from `stdin`

< reads a file from standard input (piped or typed input). So you would use this to simulate typing into a terminal. `< somefile.txt` is largely equivalent to `cat somefile.txt | .`

One common use might be to take the the results of a long running command and store the results in a file so you don't have to repeat it while you refine your command line. For example, if you have a large directory structure you might save a list of all the files you're interested in and then reading in the file list when you are done:

```
$ find .  -name .txt > files
$ xargs grep banana < files
```

5.1.3 Redirecting `stderr`

To redirect the `stderr` output (warnings, messages), you can use `2>`, just like >

```
$ ls one.txt nosuchfile.txt 2> errors.txt
one.txt
$ cat errors.txt
ls: nosuchfile.txt: No such file or directory
```

5.1.4 Combining `stdout` and `stderr`

To combine both output channels (`stdout` and `stderr`) and redirect them to a single file, you can use `&>`

```
$ ls one.txt nosuchfile.txt &> ls.out
$ cat ls.out
ls: nosuchfile.txt: No such file or directory
one.txt
```

5.2 Command piping

Part of the power of the command line is to string multiple commands together to create useful results. The core of these is the pipe: `|`. For example to see the number of files in a directory, we can pipe the (`stdout`) output of `ls` to `wc` (**w**ord **c**ount, but can also be used to count the number of lines with the `-l` flag).

```
$ ls | wc -l
     42
```

A common pattern is to to pipe the output of a command to `less` so you can examine or search the output:

```
$ find .  | less
```

Or to look through your command history:

```
$ history | less
```

You can put multiple pipes in the same line. For example, which `cp` commands have we run?

```
$ history | grep cp | less
```

5.3 Shell expansion

The shell will expand certain things, including:

1. `*` wildcard: for example `ls t*txt` will list all files starting with 't' and ending in 'txt'
2. tab completion: hit the `<tab>` key to make the shell complete your command line; works for completing file names, command names, etc.
3. `$...` or `${...}`: environment variables will be replaced with their value; example: `echo "I am $USER"` or `echo "I am ${USER}"`
4. square brackets can be used to list a number of options for a particular characters; example: `ls *.[oe][0-9]`. This will list all files starting with whatever characters (`*`), then a dot (`.`), then either an 'o' or an 'e' (`[oe]`), then a character from '0' to '9' (so any digit) (`[0-9]`). So this filename will match: `anything.o5`, but this one won't: `anything.o52`.

5.4 Process information

5.4.1 `ps` and `pstree`

`ps` lists processes running. By default, it will only show you the processes running in the local shell. To see all of your processes running on the system, use:

```
$  ps -fu $USER
```

To see all the processes

```
$ ps -elf
```

To see all the processes in a forest view, use:

```
$ ps auxf
```

The last two will spit out a lot of data, so get in the habit of piping it to `less`.

`pstree` is another way to dump a tree/forest view. It looks better than `ps auxf` but it has much less information so its value is limited.

`pgrep` will find all the processes where the name matches the pattern and print the process IDs (PID). This is used in piping the processes together as we will see in the next section.

5.4.2 `kill`

`ps` isn't very useful unless you can manipulate the processes. We do this using the `kill` command. Kill will send a message (SIGINT) to the process to ask it to stop.

```
$ kill 1234
$ kill $(pgrep misbehaving_process)
```

Usually this ends the process, giving it the opportunity to flush data to files, etc. However, if the process ignored your signal, you can send it a different message (SIGKILL) which the OS will use to unceremoniously terminate the process:

```
$ kill -9 1234
```

5.4.3 `top`

`top` is a tool to see the current status of the system. You've probably used something similar in Task Manager on Windows or Activity Monitor in macOS. `top` will update every second and has a few interesting commands.

To see only your processes, type `u` and your username after starting `top`, (you can also do this with `top -u $USER`). The default is to sort the display by `%CPU`. To change the sort order, use < and > like arrow keys.

There are a lot of configuration options in `top`, but if you're interested in seeing a nicer view, you can run `htop` instead. Be aware that it's not installed everywhere, while `top` is.

To exit `top`, use `q` (for 'quit').

For more information, see Brendan Gregg's excellent site dedicated to performance analysis.

5.4.4 ulimit

`ulimit` is a utility to get or set the user limits on the machine. For example, you may be limited to a certain number of processes. To see all the limits that have been set, use:

```
$ ulimit -a
```

5.5 Counting: `wc`

To count the number of lines, words and characters (or bytes) in a file, use `wc` (**w**ord **c**ount):

```
$ wc example.txt
      90     468    3189  example.txt
```

The output indicates that the file named `example.txt` contains 90 lines, 468 words and 3189 characters/bytes.

To only count the number of lines, use `wc -l`:

```
$ wc -l example.txt
      90    example.txt
```

5.6 Searching file contents: `grep`

`grep` is an important command. It was originally an abbreviation for "globally search a regular expression and print" but it's entered the common computing lexicon and people use 'grep' to mean searching for anything. To use grep, you give a pattern and a list of files.

```
$ grep banana fruit.txt
$ grep banana fruit_bowl1.txt fruit_bowl2.txt
$ grep banana fruit*txt
```

`grep` also lets you search for Regular Expressions, but these are not in scope for this introductory text.

5.7 `cut`

`cut` is used to pull fields out of files or pipes streams. It's a useful glue when you mix it with `grep` because `grep` can find the lines where a string occurs and `cut` can pull out a particular field. For example, to pull the first column (`-f 1`, the first **f**ield) from (an unquoted) CSV (comma-separated values, so `-d ','`: **d**elimited by `,`) file, you can use the following:

```
$ cut -f 1 -d ',' mydata.csv
```

5.8 `sed`

`sed` is the stream editor. It is used to replace text in a file or piped stream. In this way it works like grep, but instead of just searching, it can also edit files. This is like "Search and Replace" in a text editor. `sed` has a lot of features, but most everyone uses the extremely basic version of string replacement:

```
$ sed 's/oldtext/newtext/g' myfile.txt
```

By default, sed will just print the results. If you want to edit the file inplace, use `-i`, but be very careful that the results will be what you want before you go around **destroying your data**!

5.9 `awk`

`awk` is a basic language that builds on `sed` to do much more advanced stream editing. Going in depth is far out of scope of this tutorial, but there are two examples that are worth knowing.

First, `cut` is very limited in pulling fields apart based on whitespace. For example, if you have padded fields then `cut -f 4 -d ' '` will almost certainly give you a headache as there might

be an uncertain number of spaces between each field. `awk` does better whitespace splitting. So, pulling out the fourth field in a whitespace delimited file is as follows:

```
$ awk '{print $4}' mydata.dat
```

You can use `-F ':'` to change the delimiter (F for field separator).

The next example is used to sum numbers from a field:

```
$ awk -F ',' '{sum += $1} END {print sum}' mydata.csv
```

5.10 Basic Shell Scripting

The basic premise of a script is to execute automate the execution of multiple commands. If you find yourself repeating the same commands over and over again, you should consider writing one script to do the same. A script is nothing special, it is just a text file like any other. Any commands you put in there will be executed from the top to bottom.

However there are some rules you need to abide by.

Here is a very detailed guide should you need more information.

5.10.1 Shebang

The first line of the script is the so called shebang (`#` is sometimes called hash and `!` is sometimes called bang). This line tells the shell which command should execute the script. In the most cases this will simply be the shell itself. The line itself looks a bit weird, but you can copy paste this line as you need not worry about it further. It is however very important this is the very first line of the script! These are all valid shebangs, but you should only use one of them:

```
1 #!/bin/sh
```

```
1 #!/bin/bash
```

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

5.10.2 Conditionals

Sometimes you only want certain commands to be executed when a certain condition is met. For example, only move files to a directory if that directory exists. The syntax:

```
1  if [ -d directory ] && [ -f file ]
2  then
3     mv file directory
4  fi
5
6  Or you only want to do something if  a file exists:
7
8  if [ -f filename ]
9  then
10    echo "it exists"
```

```
11 fi
```

Or only if a certain variable is bigger than one

```
1 if [ $AMOUNT -gt 1 ]
2 then
3   echo "More than one"
4   # more commands
5 fi
```

Several pitfalls exist with this syntax. You need spaces surrounding the brackets, the **then** needs to be on the beginning of a line. It is best to just copy this example and modify it.

In the initial example we used `-d` to test if a directory existed. There are several more checks.

Another useful example, to test if a variable contains a value (so it's not empty):

```
1 if [ -z $PBS_ARRAYID ]
2 then
3   echo "Not an array job, quitting."
4   exit 1
5 fi
```

the `-z` will check if the length of the variable's value is greater than zero.

5.10.3 Loops

Are you copy pasting commands? Are you doing the same thing with just different options? You most likely can simplify your script by using a loop.

Let's look at a simple example:

```
1 for i in 1 2 3
2 do
3   echo $i
4 done
```

5.10.4 Subcommands

Subcommands are used all the time in shell scripts. What they basically do is storing the output of a command in a variable. So this can later be used in a conditional or a loop for example.

```
CURRENTDIR=`pwd`  # using backticks
CURRENTDIR=$(pwd)  # recommended (easier to type)
```

In the above example you can see the 2 different methods of using a subcommand. `pwd` will output the current working directory, and its output will be stored in the CURRENTDIR variable. The recommend way to use subcommands is with the `$()` syntax.

5.10.5 Errors

Sometimes some things go wrong and a command or script you ran causes an error. How do you properly deal with these situations?

Firstly a useful thing to know for debugging and testing is that you can run any command like this:

```
command 2>&1 output.log   # one single output file, both output and errors
```

If you add `2>&1 output.log` at the end of any command, it will combine `stdout` and `stderr`, outputting it into a single file named `output.log`.

If you want regular and error output separated you can use:

```
command > output.log 2> output.err  # errors in a separate file
```

this will write regular output to `output.log` and error output to `output.err`.

You can then look for the errors with `less` or search for specific text with `grep`.

In scripts you can use

```
set -e
```

this will tell the shell to stop executing any subsequent commands when a single command in the script fails. This is most convenient as most likely this causes the rest of the script to fail as well.

Advanced error checking

Sometimes you want to control all the error checking yourself, this is also possible. Everytime you run a command, a special variable `$?` is used to denote successful completion of the command. A value other than zero signifies something went wrong. So an example use case:

```
1  command_with_possible_error
2  exit_code=$?  # capture exit code of last command
3  if [ $exit_code -ne 0 ]
4  then
5     echo "something went wrong"
6  fi
```

5.11 `.bashrc` login script

If you have certain commands executed every time you log in (which includes every time a job starts), you can add them to your `$HOME/.bashrc` file. This file is a shell script that gets executed every time you log in.

Examples include:

- modifying your `$PS1` (to tweak your shell prompt)
- printing information about the current/jobs environment (echoing environment variables, etc.)
- selecting a specific cluster to run on with `module swap cluster/...`

Some recommendations:

- Avoid using `module load` statements in your `$HOME/.bashrc` file
- Don't directly edit your `.bashrc` file: if there's an error in your `.bashrc` file, you might not be able to log in again. In order to prevent that, use another file to test your changes, then copy them over when you tested the script.

5.12 Scripting for the cluster

When writing scripts to be submitted on the cluster there are some tricks you need to keep in mind.

5.12.1 Example job script

```
1  #!/bin/bash
2  #PBS -l nodes=1:ppn=1
3  #PBS -N FreeSurfer_per_subject-time-longitudinal
4  #PBS -l walltime=48:00:00
5  #PBS -q long
6  #PBS -m abe
7  #PBS -j oe
8  export DATADIR=$VSC_DATA/example
9  # $PBS_JOBID is unique for each job, so this creates a unique directory
10 export WORKDIR=$VSC_SCRATCH_NODE/$PBS_JOBID
11 mkdir -p $WORKDIR
12 # copy files to local storage
13 cp -a $DATADIR/workfiles $WORKDIR/
14
15 # load software we need
16 module load FreeSurfer
17 cd $WORKDIR
18 # recon-all ... &> output.log  # this command takes too long, let's show a more
     practical example
19 echo $PBS_ARRAYID > $WORKDIR/$PBS_ARRAYID.txt
20 # create results directory if necessary
21 mkdir -p $DATADIR/results
22 # copy work files back
23 cp $WORKDIR/$PBS_ARRAYID.txt $DATADIR/results/
```

5.12.2 PBS pragmas

The scheduler needs to know about the requirements of the script, for example: how much memory will it use, how long will it run? These things can be specified inside a script with what we call PBS pragmas.

This pragma (a pragma is a special comment) tells PBS to use 1 node and 1 core:

```
1  #PBS -l nodes=1:ppn=1   # single-core
```

For parallel software, you can request multiple cores (OpenMP) and/or multiple nodes (MPI). *Only use this when the software you use is capable of working in parallel.*. Here is an example:

```
#PBS -l nodes=1:ppn=16  # single-node, multi-core
#PBS -l nodes=5:ppn=16  # multi-node
```

We intend to submit it on the long queue:

```
#PBS -q long
```

We request a total running time of 48 hours (2 days).

```
#PBS -l walltime=48:00:00
```

We specify a desired name of our job:

```
#PBS -N FreeSurfer_per_subject-time-longitudinal
```

This specifies mail options:

```
#PBS -m abe
```

1. `a` means mail is sent when the job is aborted.
2. `b` means mail is sent when the job begins.
3. `e` means mail is sent when the job ends.

Joins error output with regular output:

```
#PBS -j oe
```

All of these options can also be specified on the command-line and will overwrite any pragmas present in the script.

5.13 Exercises

1. Create a file that contains this message: "Hello, I am <user>", where `<user>` is replaced by your username. Don't cheat by using an editor, use a command to create the file.
2. Use another command to add this line to the same file: "I am on system <hostname> in directory <current directory>". Words between <> should be replaced with their value (hint: use environment variables).
3. How many files and directories are in `/tmp`?
4. What's the name of the 5th file/directory in alphabetical order in `/tmp`?
5. List all files that start with `t` in `/tmp`.
6. Create a file containing "My home directory <home> is available using $HOME". `<home>` should be replaced with your home directory, but `$HOME` should remain as-is.
7. How many processes are you currently running? How many are you allowed to run? Where are they coming from?

Chapter 6

Common Pitfalls

6.0.1 Files

Location

If you receive an error message which contains something like the following:

```
No such file or directory
```

It probably means that you haven't placed your files in the correct directory or you have mistyped the file name or path.

Try and figure out the correct location using `ls`, `cd` and using the different `$VSC_*` variables.

Spaces

Filenames should **not** contain any spaces! If you have a long filename you should use underscores or dashes (e.g., `very_long_filename`).

```
$ cat some file
No such file or directory 'some'
```

Spaces are permitted, however they result in surprising behaviour. To cat the file `'some file'` as above, you can escape the space with a backslash ("`\ `") or you can put the filename in quotes:

```
$ cat some\ file
...
$ cat "some file"
...
```

This is especially error prone if you are piping results of `find`:

```
$ find .  -type f | xargs cat
No such file or directory name 'some'
No such file or directory name 'file'
```

This can be worked around using the `-print0` flag:

```
$ find .  -type f -print0 | xargs -0 cat
...
```

But, this is tedious and you can prevent errors by simply colouring within the lines and not using spaces in filenames.

Missing/mistyped environment variables

If you use a command like `rm -r` with environment variables you need to be careful to make sure that the environment variable exists. If you mistype an environment variable then it will resolve to a blank string. This means the following resolves to `rm -r ~/*` which will remove every file in your home directory!

```
$ rm -r ~/$PROJETC/*
```

Typing dangerous commands

A good habit when typing dangerous commands is to precede the line with `#`, the comment character. This will let you type out the command without fear of accidentally hitting enter and running something unintended.

```
$ #rm -r ~/$POROJETC/*
```

Then you can go back to the beginning of the line (`Ctrl-A`) and remove the first character (`Ctrl -D`) to run the command. You can also just press enter to put the command in your history so you can come back to it later (e.g., while you go check the spelling of your environment variables).

Permissions

```
$ ls -l script.sh # File with correct permissions
-rwxr-xr-x 1 vsc40000 vsc40000 2983 Jan 30 09:13 script.sh
$ ls -l script.sh # File with incorrect permissions
-rw-r--r-- 1 vsc40000 vsc40000 2983 Jan 30 09:13 script.sh
```

Before submitting the script, you'll need to add execute permissions to make sure it can be executed:

```
$ chmod +x script_name.sh
```

6.0.2 Help

If you stumble upon an error, don't panic! Read the error output, it might contain a clue as to what went wrong. You can copy the error message into Google (selecting a small part of the error without filenames). It can help if you surround your search terms in double quotes (for example `"No such file or directory"`), that way Google will consider the error as one thing, and won't show results just containing these words in random order.

If you need help about a certain command, you should consult its so called "man page":

```
$ man command
```

This will open the manual of this command. This manual contains detailed explanation of all the options the command has. Exiting the manual is done by pressing 'q'.

Don't be afraid to contact hpc@ugent.be. They are here to help and will do so for even the smallest of problems!

Chapter 7

More information

1. Unix Power Tools - A **fantastic** book about most of these tools (see also The Second Edition)
2. `http://linuxcommand.org/`: A great place to start with many examples. There is an associated book which gets a lot of good reviews
3. The Linux Documentation Project - More guides on various topics relating to the Linux command line
4. basic shell usage
5. Bash for beginners
6. MOOC

Chapter 8

Q & A

Please don't hesitate to contact hpc@ugent.be in case of questions or problems.

Chapter 9

More on the HPC infrastructure

9.1 Filesystems

Multiple different shared filesystems are available on the HPC infrastructure, each with their own purpose. See chapter 6, section titled "Where to store your data on the HPC" of the HPC manual for a list of available locations.

VO storage

If you're member of a (non-default) virtual organisation (VO), see Chapter 6, section titled "Virtual Organisations", you have access to additional directories (with more quota) on the data and scratch filesystems, which you can share with other members in the VO.

9.1.1 Quota

Space is limited on the cluster's storage. To check your quota, see Chapter 6, section titled "Pre-defined quota" of the HPC manual.

To figure out where your quota is being spent, the `du` (**d**isk **u**sage) command can come in useful:

```
$ du -sh test
59M   test
```

Do *not* (frequently) run `du` on directories where large amounts of data are stored, since that will:

1. take a long time
2. result in increased load on the shared storage since (the metadata of) every file in those directories will have to be inspected.

9.2 Modules

Software is provided through so-called environment modules.

The most commonly used commands are:

1. `module avail`: show *all* available modules
2. `module avail <software name>`: show available modules for a specific software name
3. `module list`: show list of loaded modules
4. `module load <module name>`: load a particular module

More information is available in chapter 3 of the HPC manual, section named "Modules".

9.3 Using the clusters

The use the clusters beyond the login node(s) which have limited resources, you should create job scripts and submit them to the clusters.

Detailed information is available in chapter 4 of the HPC manual, section named "Defining and submitting your job".

9.4 Exercises

Create and submit a job script that computes the sum of 1-100 using Python, and prints the numbers to a *unique* output file in `$VSC_SCRATCH`.

Hint: `python -c "print(sum(range(1, 101)))"`

- How many modules are available for Python version 3.6.4?
- How many modules get loaded when you load the `Python/3.6.4-intel-2018a` module?
- Which `cluster` modules are available?

- What's the full path to your personal home/data/scratch directories?
- Determine how large your personal directories are.
- What's the difference between the size reported by `du -sh $HOME` and by `ls -ld $HOME`?