Unix Basics

Most of bioinformatics happens on Unix-like operating systems. Almost all our tools run from the Unix command line, so bioinformaticians must know how to move data around, run programs, and chain the output of one program into another to create analysis pipelines.

Getting access to systems

Given that our class will have students on a variety of operating systems (Windows, OSX, Linux), we will use the HPC (high performance computing) cluster at the University of Arizona for our work. Students using the Windows operating system might find the Gitbash or Cygwin tools useful for getting access to tools like "ssh" and "scp." Students on OSX and Linux machines may choose to install software locally using package management tools like "homebrew" (OSX) or "apt-get" and "yum" (Linux).

University of Arizona HPC

If you're on Linux or OSX, open a terminal and type:

```
ssh <NetID>@hpc.arizona.edu
```

If all goes well, you should see something like this:

```
Last login: Sat Jan  5 07:53:30 2019 from ip72-200-123-88.tc.ph.cox.net
This is a bastion host used to access the rest of the environment.

Shortcut commands to access each resource
-----------------------------------------
Ocelote:                El Gato:
$ ocelote               $ elgato
```

Or you may see this (e.g., if you have enabled the menu with the `menuon` command):

```
Last login: Sat Jan  5 07:53:30 2019 from ip72-200-123-88.tc.ph.cox.net
This is a bastion host used to access the rest of the environment.

Shortcut commands to access each resource
-----------------------------------------
Ocelote:                El Gato:
$ ocelote               $ elgato
```

If you would like to avoid the 2-factor authentication, then copy your SSH private key to the target system like so:

1) On your local machine and `cd ~/.ssh`. If you do not have one, then run `ssh-keygen`.
2) Copy the contents of the `id_rsa.pub` file (the "public" part of your key). If you do not have one, then run `ssh-keygen`.
3) Login to the target system and `cd ~/.ssh`. If you do not have one, then run `ssh-keygen`.
4) Edit the `authorized_keys` file (e.g., with `nano`) and paste in the public key.
5) Set the permissions with `chmod 600 authorized_keys`
6) Test your login from your local machine.

Stampede (TACC)

The "stampede" cluster is another HPC resource that is freely available to our students. It is located at TACC, the Texas Advanced Computing Center at the University of Texas in Austin. Go to `https://portal.tacc.utexas.edu/` to create an account. We recommend our students use the same username at TACC and Cyverse.

Cyverse (UA)

Cyverse is an NSF-funding cyberinfrastructure project headquartered at the University of Arizona. It began life as "iPlant" in 2008. Our students may choose to make use of Cyverse infrastructure, as well, such as "apps" for assemblies, gene calling, protein clustering, and such. Students should go to https://user.cyverse.org/ to create an account. Again, we recommend you use the same username as your TACC account to make communication between the TACC and UA systems easier.

Docker, VirtalBox

If cannot gain access to the UA HPC or TACC, you can follow along at home using a virtual machine such as a Docker (https://www.docker.com/) or VirtualBox (https://www.virtualbox.org/) image installed on your machine.

Common Unix Commands

I assume you are on a command line by now, so let's look at some commands.

- **whoami**: reports your username
- **w**: shows who is currently on a system
- **man**: show the manual page for a command

- **echo**: say something
- **cowsay**: have a cow say something
- **env**: print your environment
- **printenv**: print some/all of your environment
- **which/type**: tells you the location of a program
- **touch**: create an empty regular file
- **file**: briefly describe a file or directory
- **pwd**: print working directory, where you are right now
- **ls**: list files in current directory
- **find**: find files or directories matching some conditions
- **locate**: find files using a database, requires daemon to run
- **cd**: change directory (with no arguments, `cd $HOME`)
- **cp**: copy a file (or "cp -r" to copy a directory)
- **mv**: move a file or diretory
- **mkdir**: create a directory
- **rmdir**: remove a directory
- **rm**: remove a file (or "rm -r" to remove a directory)
- **cat**: concatenate files (cf. http://porkmail.org/era/unix/award.html)
- **column**: arrange text into columns
- **paste**: merge lines of files
- **sort**: sort text or numbers
- **uniq**: remove duplicates *from a sorted list*
- **sed**: stream editor for altering text
- **awk/gawk**: pattern scanning and processing language
- **grep**: global regular expression program (maybe?), cf. https://en.wikipedia.org/wiki/Regular_expression
- **history**: look at past commands, cf. CTRL-R for searching your history directly from the command line
- **head**: view the first few (10) lines of a file
- **tail**: view the last (10) lines of a file
- **comm**: find lines in common/unique in two sorted files
- **top**: view the programs taking the most system resources (memory, I/O, time, CPUs, etc.), cf. "htop"
- **ps**: view the currently running processes
- **cut**: select columns from the output of a program
- **wc**: (character) word (and line) count
- **more/less**: pager programs that show you a "page" of text at a time; cf. https://unix.stackexchange.com/questions/81129/what-are-the-differences-between-most-more-and-less/81131
- **bc**: calculator
- **df**: report file system disk space usage; useful to find a place to land your data
- **du**: report disk usage; recommend "du -shc"; useful to identify large directories that need to be removed
- **ssh**: secure shell, like telnet only with encryption
- **scp**: secure copy a file from/to remote systems using ssh

- **rsync**: remote sync; uses scp but only copies data that is different
- **ftp**: use "file transfer protocol" to retrieve large data sets (better than HTTP/browsers, esp for getting data onto remote systems)
- **ncftp**: more modern FTP client that automatically handles anonymous logins
- **wget**: web get a file from an HTTP location, cf. "wget is not a crime" and Aaron Schwartz
- |: pipe the output of a command into another command
- **>, >>**: redirect the output of a command into a file; the file will be created if it does not exist; the single arrow indicates that you wish to overwrite any existing file, while the double-arrows say to append to an existing file
- **<**: redirect contents of a file into a command
- **nano**: a very simple text editor; until you're ready to commit to vim or emacs, start here
- **md5sum**: calculate the MD5 checksum of a file
- **diff**: find the differences between two files

Pronunciation

- /: "slash"; the thing leaning the other way is a "backslash"
- **sh**: "shuh" or "ess-ach"
- **etc**: "et-see"
- **usr**: "user"
- **src**: "source"
- #: "hash" (NOT "hashtag") or "pound"
- **$**: "dollar"
- **!**: "bang"
- **#!**: "shebang"
- ^: "caret"
- **PID**: "pid" (not pee-eye-dee)
- ~: "twiddle" or "tilde"; shortcut to your home directory when alone, shortcut to another user's home directory when used like "~bhurwitz"

Variables

You will see things like `$USER` and `$HOME` that start with the `$` sign. These are variables because they can change from person to person, system to system. On most systems, my username is "kyclark" but I might be "kclark" or "kyclark1" on others, but on all systems `$USER` refers to whatever value is defined for my username. Similarly, my `$HOME` directory might be "/Users/kyclark," "/home1/03137/kyclark," or "/home/u20/kyclark," but I can always refer to the

idea of my home directory with the variable `$HOME`.

Make it stop!

If you launch a program that won't stop, you can use CTRL-C to send an "interrupt" signal to the program. If it is well-behaved, it should stop, but it may not. You can open another terminal on the machine and run `ps -fu $USER` to find all the programs you are running.

```
$ ps -fu $USER
UID        PID  PPID  C STIME TTY          TIME CMD
kyclark  31718 31692  0 12:16 ?        00:00:00 sshd: kyclark@pts/75
kyclark  31723 31718  0 12:16 pts/75   00:00:00 -bash
kyclark  33265 33247  0 12:16 ?        00:00:00 sshd: kyclark@pts/86
kyclark  33277 33265  1 12:16 pts/86   00:00:00 -bash
kyclark  33792 33277  9 12:17 pts/86   00:00:00 vim run.p6
kyclark  33806 31723  0 12:17 pts/75   00:00:00 ps -fu kyclark
```

The PID is the "process ID" and the PPID is the "parent process ID." In the above table, let's assume my "vim" session was locked up. I could `kill 33792`. If in a reasonable amount of time (a minute or so) that doesn't work, common wisdom says you `kill -9` to really, really tell it to shut down, but:

> No no no. Don't use kill -9.
>
> It doesn't give the process a chance to cleanly:
>
> - shut down socket connections
> - clean up temp files
> - inform its children that it is going away
> - reset its terminal characteristics
>
> and so on and so on and so on.
>
> Generally, send 15, and wait a second or two, and if that doesn't work, send 2, and if that doesn't work, send 1. If that doesn't, REMOVE THE BINARY because the program is badly behaved!
>
> Don't use kill -9. Don't bring out the combine harvester just to tidy up the flower pot. cf. http://porkmail.org/era/unix/award.html#uuk9letter

Along with CTRL-C, you should learn about CTRL-Z to put a process into the background. This could be handy if, say, you were in an editor, you make a change to your code, you CTRL-Z to background the editor, you run the script to see if it worked, then you `fg` to bring it back to the foreground or `bg` it to have it resume running in the background. I would consider this a sub-optimal

work environment, but it's fine if you were for some reason limited to a single terminal window.

Generally if you want a program to run in the background, you would launch it from the command line with an ampersand ("&") at the end:

```
$ my-background-prog.sh &
```

Lastly, know that most Unix programs interpret CTRL-D as the end-of-input signal. You can use this to send the "exit" command to most any interactive program, even your shell. Here's a way to enter some text into a file directly from the command line without using a text editor. After typing the last line (i.e., type "chickens.<Enter>"), type CTRL-D:

```
$ cat > wheelbarrow
so much depends
upon

a red wheel
barrow

glazed with rain
water

beside the white
chickens.
<CTRL-D>
$ cat wheelbarrow
so much depends
upon

a red wheel
barrow

glazed with rain
water

beside the white
chickens.
```

Handy command line shortcuts

- Tab: hit the Tab key for command completion; hit it early and often!
- **!!**: (bang-bang) execute the last command again
- **!$**: (bang-dollar) the last argument from your previous command line (think of the $ as the right anchor in a regex)

- `!^`: (band-caret) the first argument from your previous command line (think of the ^ as the left anchor in a regex)
- CTRL-R: reverse search of your history
- Up/down cursor keys: go backwards/forwards in your history
- CTRL-A, CTRL-E: jump to the start, end of the command line when in emacs mode (default)

  N.B.: If you are on a Mac, it's easy to remap your (useless) CAPSLOCK key to CTRL. Much less strain on your hand as you will find you need CTRL quite a bit, even more so if you choose emacs for your $EDITOR.

Altering your $PATH

As you see above, "env" will list all the key-value pairs defining your environment. For instance, everyone has a `$HOME` directory that you can see with `echo $HOME`. The exact location of `$HOME` can vary among systems, e.g.:

- Mac: /Users/kyclark
- Ocelot: /home/u20/kyclark
- Stampede: /home1/03137/kyclark

Your `$PATH` setting is extremely important as it defines the directory locations that will be searched (in order) to find programs. Here's my `$PATH` on the UA HPC:

```
[hpc:login3@~]$ echo $PATH | sed "s/:/:\n/g"
/rsgrps/bhurwitz/hurwitzlab/bin:
/home/u20/kyclark/.cargo/bin:
/home/u20/kyclark/.local/bin:
/cm/local/apps/gcc/6.1.0/bin:
/cm/shared/uaapps/pbspro/18.2.1/sbin:
/cm/shared/uaapps/pbspro/18.2.1/bin:
/opt/TurboVNC/bin:
/cm/shared/uabin:
/usr/lib64/qt-3.3/bin:
/cm/local/apps/environment-modules/4.0.0//bin:
/usr/local/bin:
/bin:
/usr/bin:
/usr/local/sbin:
/usr/sbin:
/sbin:
/sbin:
/usr/sbin:
/cm/local/apps/environment-modules/4.0.0/bin
```

I've used "sed" to add a newline after each colon so you can more easily see that the directories are separated by colons. Notice that I have the shared "hurwitzlab" directory first in my path. Much of our work will require access to tools that are not installed by default on the HPC. You could build them into your own `$HOME` directory, but it will be easier if you just add this shared directory to your $PATH. From the command line, you can do this:

```
export PATH=/rsgrps/bhurwitz/hurwitzlab/bin:$PATH
```

You just told your shell (bash) to set the `$PATH` variable to our "hurwitzlab" directory and then whatever it was set to before. Obviously we want this to happen each time we log in, so we can add this command to `$HOME/.bashrc`:

```
echo "export PATH=/rsgrps/bhurwitz/hurwitzlab/bin:$PATH" >> ~/.bashrc
```

> N.B., "dotfiles" are files with names that begin with a dot. They are normally hidden from view unless you use `ls -a` to list "all" files. A single dot `.` means the current directory, and two dots `..` mean the parent directory. Your ".bashrc" (or maybe ".profile" or maybe ".bash_profile" depending on your system) file is read every time you login to your system, so you can remember your customizations. "Rc" may mean "resource configuration," but who really knows?

As you find or create useful programs that you would like to have available globally on your system (i.e., not just in the current working directory), you can create a location like `$HOME/bin` (or my preferred `$HOME/.local/bin`) and add this to your `$PATH` as well. You can add as many directories as you like (within reason).

Aliases

Sometimes you'll find you're using a particular command quite often and want to create a shortcut. You can assign any command to a single "alias" like so:

```
alias cx='chmod +x'
alias up2='cd ../../'
alias up3='cd ../../../'
```

If you execute this on the command line, the alias will be saved until you log out. Adding these lines to your `.bashrc` will make it available every time you log in. When you make a change and want the shell to bring those into the current environment, you need to `source` the file. The command `.` is an alias for `source`:

```
$ source ~/.bashrc
$ . ~/.bashrc
```

Permissions

When you execute `ls -l`, you'll see the "long" listing of the contents of a directory similar to this:

```
-rwxr-xr-x   1 kyclark  staff     174 Aug  9 20:21 abs.py*
drwxr-xr-x  14 kyclark  staff     476 Aug  3 12:14 anaconda3/
```

The first column of data contains a wealth of information represent in 10 bits. The first bit is:

- "-" for a regular file
- "d" for a directory
- "l" for a symlink (like a shortcut)

The other nine bits are broken into sets of three bits that represent the permissions for the "user," "group," and "other." The "abs.py" is a regular file we can tell from the first dash. The next three bits show "rwx" which means that the user ("kyclark") has read, write, and execute permissions for this file. The next three bits show "r-x" meaning that the group ("staff") can read and execute the file only. The same is true for all others.

When you create a file, the normal default is that it is not executable. You must specifically tell Unix to change the mode of the file using the "chmod" command. Often it's enough to say:

```
$ chmod +x myprog.sh
```

To turn on the execute bits for everyone, but it's possible to have much finer control of the permissions. If you only want user and group to have execute, then do:

```
$ chmod ug+x myprog.sh
```

Removing is done with a "-," so any combination of `[ugo]` `[+-]` `[rwx]` will usually get you what you want.

Sometimes you may see instructions to `chmod 775` a file. This is using octal notation where the three bits "rwx" correspond to the digits "421," so the first "7" is "4+2+1" which equals "rwx" whereas the "5" = "4+1" so only "rw":

```
 user   group   other
r w x   r w x   r w x
4 2 1 | 4 2 1 | 4 2 1
+ + +   + + +   + - +
 = 7     = 7     = 5
```

Therefore "chmod 775" is the same as:

```
$ chmod -rwx myfile
$ chmod ug+rwx myfile
$ chmod o+rw myfile
```

When you create ssh keys or config files, you are instructed to `chmod 600`:

```
 user   group   other
r w x   r w x   r w x
4 2 1 | 4 2 1 | 4 2 1
+ + -   - - -   - - -
 = 6     = 0     = 0
```

Which means that only you can read or write the file, and no one else can do anything with it. So you can see that it can be much faster to use the octal notation.

When you are trying to share data with your colleagues who are on the same system, you may put something into a shared location but they complain that they cannot read it or nothing is there. The problem is most likely permissions. The "uask" setting on a system determines the default permissions, and it may be that the directory and/or files are readable only by you. It may also be that you are not in a common group that you can use to grant permission, in which case you can either:

- politely ask your sysadmin to create a new group OR
- `chmod 777` the directory, which is probably the worst option as it makes the directory completely accessible to anyone to do anything. In short, don't do this unless you really don't care if someone accidentally or maliciously wipes out your data.

File system layout

The top level of a Unix file system is "/" which is called "root." Confusingly, there is also an account named "root" which is basically the super-user/sysadmin (systems administrator). Unix has always been a multi-tenant system ...

Installing software

Much of the time, "bioinformatics" seems like little more than installing software and chaining them together with scripts. Sometimes you may be lucky enough to have a "sysadmin" (systems administrator) who can assist you, but most of the time you'll find yourself needing to take care of business yourself.

My suggestions for installing software are (in order):

Sysadmin

Go introduce yourself to your sysadmins. Take them to lunch or order them some pizza or drop off some good beer or whiskey. Whatever it takes to be on good terms because a good sysadmin who is responsive to your needs is an enormous help. If they are willing to install software for you, that is the way to go. Often, though, this is a task far beneath them, and they would expect you to be able to fend for yourself. They may provide `sudo` (https://xkcd.com/149/) privileges to allow you to install software into shared locations (e.g., `/usr/local`), but it's more likely they would expect you to install into your `$HOME`.

Package managers

There are several package management systems for Linux and OSX including apt-get, yum, homebrew, macports, and more. These usually relieve the problems of software compatibility and shared libraries. Unless you have `sudo` to install globally, you can configure to install into your `$HOME`.

Binary installations

Quite often you'll be happy to find that the maintainers of the software you need have gone to the trouble to build binary distributions for your system, which is likely to be a generic 64-bit Linux platform. Often you can just download the binaries and put them into your `$PATH`. There is usually a "README" or "INSTALL" file that will explain exactly what to do. To use the binaries, you can:

1) Always refer to the full path to the binary
2) Place them into a directory in your `$PATH` like `$HOME/.local/bin`
3) Add the new directory to your `$PATH`

Source installations

Installing from source usually means downloading a "tarball" ("tar" = "tape archive," a container of files, that is then compressed with a program like "gzip" to create a ".tar.gz" or ".tgz" file extension), running "configure" to figure out how it can build on your system, and then "make" to build the binaries. Usually you will run "make install" to put the binaries into their proper directory, but sometimes you just "make" and copy the files yourself.

The basic steps for installing into your `$HOME` are usually:

```
$ tar xvf package.tgz
$ ./configure --prefix=$HOME/.local
$ make && make install
```

When I'm in an environment with a directory I can share with my team (like the UA HPC), I'll configure the package to install into that shared space so that others can use the program. When I'm on a system like "stampede" where I cannot share with others, I'll usually install into my `$HOME/.local` or some sort of "work" directory.

Unix exercises

> N.B.: When you see a `$` given in the example prompts, it is a metacharacter indicating that this is the prompt for a normal (not super-user) account. Your default prompt may be different, and it is highly configurable (search for "PS1 unix prompt" to learn more). Anyway, point is that you should type (copy/paste) all the stuff *after* the `$`. If you ever see a prompt with "#" in a tutorial, it's indicating a command that should be run as the super-user/root account, e.g., installing some software into a system-wide directory so it can be shared by all users.

Find the number of unique users on a shared system

We know that `w` will tell us the users logged in. Try it now on a system that has many users (i.e., not your laptop) and see the output. Likely there are dozens of users, so we'll connect the output of `w` to `head` using a pipe `|` so that we only see the first five lines:

```
[hpc:login2@~]$ w | head -5
 09:39:27 up 65 days, 20:05, 10 users,  load average: 0.72, 0.75, 0.78
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
kyclark  pts/2    gatekeeper.hpc.a 09:38    0.00s  0.05s  0.02s w
emsenhub pts/0    gatekeeper.hpc.a 04:05   14.00s  0.87s  0.87s -bash
joneska  pts/3    gatekeeper.hpc.a 08:25    1:12m  0.16s  0.12s vim results_x2r
```

Really we want to see the first five *users*, not the first five *lines* of output. To skip the first two lines of headers from `w`, we can pipe `w` into `awk` and tell it we only want to see output when the Number of Records (`NR`) is greater than 2:

```
[hpc:login2@~]$ w | awk 'NR>2' | head -5
kyclark  pts/2    gatekeeper.hpc.a 09:38    0.00s  0.07s  0.03s w
emsenhub pts/0    gatekeeper.hpc.a 04:05   26.00s  0.87s  0.87s -bash
joneska  pts/3    gatekeeper.hpc.a 08:25    1:13m  0.16s  0.12s vim results_x2r
shawtaro pts/4    gatekeeper.hpc.a 08:06   58:34   0.17s  0.17s -bash
darrenc  pts/5    gatekeeper.hpc.a 07:58   51:07   0.14s  0.07s qsub -I -N pipe
```

`awk` takes a PREDICATE and a CODE BLOCK (contained within curly brackets `{}`). Without a PREDICATE, `awk` prints the whole line. I only want to see the

first column, so I can tell `awk` to `print` just column `$1`:

```
[hpc:login2@~]$ w | awk 'NR>2 {print $1}' | head -5
kyclark
emsenhub
joneska
shawtaro
darrenc
```

We can see that the some users like "joneska" are logged in multiple times:

```
[hpc:login2@~]$ w | awk 'NR>2 {print $1}'
kyclark
emsenhub
joneska
shawtaro
darrenc
guven
guven
guven
joneska
dmarrone
```

Let's `uniq` that output:

```
[hpc:login2@~]$ w | awk 'NR>2 {print $1}' | uniq
kyclark
emsenhub
joneska
shawtaro
darrenc
guven
joneska
dmarrone
```

Hmm, that's not right – "joneska" is listed twice, and that is not unique. Remember that `uniq` only works *on sorted input*? So let's sort those names first:

```
[hpc:login2@~]$ w | awk 'NR>2 {print $1}' | sort | uniq
darrenc
dmarrone
emsenhub
guven
joneska
kyclark
shawtaro
```

To count how many unique users are logged in, we can use the `wc` (word count) program with the `-l` (lines) flag to count just the lines from the previous command

```
[hpc:login2@~]$ w | awk 'NR>2 {print $1}' | sort | uniq | wc -l
7
```

So what you see is that we're connecting small, well-defined programs together using pipes to connect the "standard input" (STDIN) and "standard output (STDOUT) streams. There's a third basic file handle in Unix called"standard error" (STDERR) that we'll come across later. It's a way for programs to report problems without simply dying. You can redirect errors into a file like so:

```
$ program 2>err
$ program 1>out 2>err
```

The first example puts STDERR into a file called "err" and lets STDOUT print to the terminal. The second example captures STDOUT into a file called "out" while STDERR goes to "err."

N.B.: Sometimes a program will complain about things that you cannot fix, e.g., `find` may complain about file permissions that you don't care about. In those cases, you can redirect STDERR to a special filehandle called `/dev/null` where they are forgotten forever – kind of like the "memory hole" in 1984.

```
find / -name my-file.txt 2>/dev/null
```

Count "oo" words

On almost every Unix system, you can find `/usr/share/dict/words`. Let's use `grep` to find how many have the "oo" vowel combination. It's a long list, so I'll pipe it into "head" to see just the first five:

```
$ grep 'oo' /usr/share/dict/words | head -5
abloom
aboon
aboveproof
abrood
abrook
```

Yes, that works, so redirect those words into a file and count them. Notice the use of `!$` (bang-dollar) to reference the last argument of the previous line so that I don't have to type it again (really useful if it's a long path):

```
$ grep 'oo' /usr/share/dict/words > oo-words
$ wc -l !$
10460 oo-words
```

Let's count them directly out of `grep`:

```
$ grep 'oo' /usr/share/dict/words | wc -l
10460
```

Do any of those words additionally contain the "ow" sequence?

```
$ grep 'oo' /usr/share/dict/words | grep 'ow' | head -5
arrowroot
arrowwood
balloonflower
bloodflower
blowproof
```

How many are there?

```
$ grep 'oo' /usr/share/dict/words | grep 'ow' | wc -l
158
```

How many *do not* contain the "ow" sequence?

```
$ grep 'oo' /usr/share/dict/words | grep -v 'ow' | wc -l
10302
```

Do those numbers add up?

```
$ bc <<< 158+10302
10460
```

Excellent. Smithers, massage my brain.

Something with sequences

Now we will get some sequence data from the iMicrobe FTP site. Both `wget` and `ncftpget` will do the trick:

```
$ mkdir -p ~/contigs
$ cd !$
$ wget ftp://ftp.imicrobe.us/biosys-analytics/contigs/contigs.zip
```

> N.B.: How do we know we got the correct data? Go back and look at that FTP site, and you will see that there is a "contigs.zip.md5" file that we can `less` on the server to view the contents:

```
$ ncftp ftp://ftp.imicrobe.us/biosys-analytics/contigs
NcFTP 3.2.6 (Dec 04, 2016) by Mike Gleason (http://www.NcFTP.com/contact/).
Connecting to 150.135.44.10...
Welcome to the imicrobe.us repository
Logging in...
Login successful.
Logged in to ftp.imicrobe.us.
Current remote directory is /biosys-analytics/contigs.
ncftp /biosys-analytics/contigs > ls
contigs.zip        contigs.zip.md5
ncftp /biosys-analytics/contigs > cat contigs.zip.md5
1b7e58177edea28e6441843ddc3a68ab  contigs.zip
ncftp /biosys-analytics/contigs > exit
```

You can read up on MD5 (https://en.wikipedia.org/wiki/Md5sum) to understand that this is a signature of the file. If we calculate the MD5 of the file we dowloaded and it matches what we see on the server, then we can be sure that we have the exact file that is on the FTP site:

```
$ md5 contigs.zip
MD5 (contigs.zip) = 1b7e58177edea28e6441843ddc3a68ab
```

> Yes, those two sums match. Note that sometimes the program is also named `md5sum`.

So, back to the exercise. Let's unpack the contigs:

```
$ unzip contigs.zip
Archive:  contigs.zip
  inflating: group12_contigs.fasta
  inflating: group20_contigs.fasta
  inflating: group24_contigs.fasta
$ rm contigs.zip
```

These files are in FASTA format (https://en.wikipedia.org/wiki/FASTA_format), which basically looks like this:

```
>MCHU - Calmodulin - Human, rabbit, bovine, rat, and chicken
ADQLTEEQIAEFKEAFSLFDKDGDGTITTKELGTVMRSLGQNPTEAELQDMINEVDADGNGTID
FPEFLTMMARKMKDTDSEEEIREAFRVFDKDGNGYISAAELRHVMTNLGEKLTDEEVDEMIREA
DIDGDGQVNYEEFVQMMTAK*
>gi|5524211|gb|AAD44166.1| cytochrome b [Elephas maximus maximus]
LCLYTHIGRNIYYGSYLYSETWNTGIMLLLITMATAFMGYVLPWGQMSFWGATVITNLFSAIPYIGTNLV
EWIWGGFSVDKATLNRFFAFHFILPFTMVALAGVHLTFLHETGSNNPLGLTSDSDKIPFHPYYTIKDFLG
LLILILLLLLLALLSPDMLGDPDNHMPADPLNTPLHIKPEWYFLFAYAILRSVPNKLGGVLALFLSIVIL
GLMPFLHTSKHRSMMLRPLSQALFWTLTMDLLTLTWIGSQPVEYPYTIIGQMASILYFSIILAFLPIAGX
IENY
```

Header lines start with ">", then the sequence follows. Sequences may be broken up over several lines of 50 or 80 characters, but it's just as common to see the sequences take only one (sometimes very long) line. Sequences may be nucleotides, proteins, very short DNA/RNA, longer contigs (shorter strands assembled into contiguous regions), or entire chromosomes or even genomes.

So, how many sequences are in "group12_contigs.fasta"? To answer, we just need to count how many times we see ">". We can do that with "grep":

```
$ grep > group12_contigs.fasta
Usage: grep [OPTION]... PATTERN [FILE]...
Try 'grep --help' for more information.
```

What is going on? Remember when we captured the "oo" words that we used the ">" symbol to tell Unix to *redirect* the output of `grep` into a file. We need to tell Unix that we mean a literal greater-than sign by placing it in single or double quotes or putting a backslash in front of it:

```
$ grep '>' group12_contigs.fasta
$ grep \> group12_contigs.fasta
```

You should actually see nothing because something quite insidious happened with that first "grep" statement – it overwrote our original "group12_contigs.fasta" with the result of "grep"ing for nothing, which is nothing:

```
$ ls -l group12_contigs.fasta
-rw-rw---- 1 kyclark staff 0 Aug 10 15:08 group12_contigs.fasta
```

Ugh, OK, I have to go back and `wget` the "contigs.zip" file to restore it. That's OK. Things like this happen all the time.

```
$ ls -lh group12_contigs.fasta
-rw-rw---- 1 kyclark staff 2.9M Aug 10 14:38 group12_contigs.fasta
```

Now that I have restored my data, I want to count how many greater-than signs in the file:

```
$ grep '>' group12_contigs.fasta | wc -l
132
```

Hey, I could see doing that often. Maybe we should make this into an "alias" (see above). The problem is that the "argument" to the function (the filename) is stuck in the middle of the chain of commands, so it would make it tricky to use an alias for this. We can create a bash function that we add to our `$HOME/.bashrc`:

```
function countseqs() {
  grep '>' $1 | wc -l
}
```

After you add this, remember to source this file to make it available:

```
$ source ~/.bashrc
$ countseqs group12_contigs.fasta
132
```

Same answer. Good. However, someone beat us to the punch. There is a powerful tool called "seqmagick" (https://github.com/fhcrc/seqmagick) that will do this (and much, much more). It's installed into the "hurwitzlab/bin" directory, or you can install it locally:

```
$ seqmagick info group12_contigs.fasta
name                  alignment    min_len   max_len   avg_len  num_seqs
group12_contigs.fasta FALSE           5136    116409  22974.30       132
```

Run "seqmagick -h" to see everything it can do.

Moving on, let's find how many contig IDs in "group12_contigs.fasta" contain the number "47":

```
$ grep 47 group12_contigs.fasta > group12_ids_with_47
```

```
[login3@~/work/sequences]$ cat !$
cat group12_ids_with_47
>Contig_247
>Contig_447
>Contig_476
>Contig_1947
>Contig_4764
>Contig_4767
>Contig_13471
```

Here we did a little "useless use of cat," but it's OK. We also could have used "less" to view the file. Here's another useless use of cat to copy a file:

```
$ cat group12_ids_with_47 > temp1_ids
```

Additionally, we want to copy the file again to make duplicates:

```
$ cp group12_ids_with_47 temp2_ids
```

How can we be sure these files are the same? Let's use "diff":

```
$ diff temp1_ids temp2_ids
```

You should see nothing, which is a case of "no news is good news." They don't differ in any way. We can verify this with "md5sum":

```
$ md5sum temp*
957390ab4c31db9500d148854f542eee  temp1_ids
957390ab4c31db9500d148854f542eee  temp2_ids
```

They are the same file. If there were even one character difference, they would generate different hashes.

Now we will create a file with duplicate IDs:

```
$ cat temp1_ids temp2_ids > duplicate_ids
```

Check contents of "duplicate_ids" using "less" or "cat." Now grab all of the contigs IDs from "group20_contigs.fasta" that contain the number "51." Concatenate the new IDs to the duplicate_ids file in a file called "multiple_ids":

```
$ cp duplicate_ids multiple_ids
$ grep 51 group20_contigs.fasta >> !$
grep 51 group20_contigs.fasta >> multiple_ids
```

Notice the ">>" arrows to indicate that we are *appending* to the existing "multiple_ids" file.

Remove the existing "temp" files using a "*" wildcard:

```
$ rm temp*
```

Now let's explore more of what "sort" and "uniq" can do for us. We want to find which IDs are unique and which are duplicated. If we read the manpage

("man uniq"), we see that there are "-d" and "-u" flags for doing just that. However, we've already seen that input to "uniq" needs to be sorted, so we need to remember to do that:

```
$ sort multiple_ids | uniq -d > temp1_ids
$ sort multiple_ids | uniq -u > temp2_ids
$ diff temp*
1,7c1,11
< >Contig_13471
< >Contig_1947
< >Contig_247
< >Contig_447
< >Contig_476
< >Contig_4764
< >Contig_4767
---
> >Contig_10051
> >Contig_1651
> >Contig_4851
> >Contig_5141
> >Contig_5143
> >Contig_5164
> >Contig_5170
> >Contig_5188
> >Contig_6351
> >Contig_9651
> >Contig_9851
```

Let's remove our temp files again and make a "clean_ids" file:

```
$ rm temp*
$ sort multiple_ids | uniq > clean_ids
$ wc -l multiple_ids clean_ids
 25 multiple_ids
 18 clean_ids
 43 total
```

We can use "sed" to alter the IDs. The "s//" command say to "substitute" the first thing with the second thing, e.g., to replace all occurences of "foo" with "bar", use "s/foo/bar" (http://stackoverflow.com/questions/4868904/what-is-the-origin-of-foo-and-bar).

```
$ sed 's/C/c/' clean_ids
$ sed 's/_/./' clean_ids
$ sed 's/>//' clean_ids > newclean_ids
```

That last one removes the FASTA file artifact that identifies the beginning of an ID but is not part of the ID. We can use this with "seqmagick" now to extract those sequences and find out how many were found:

```
$ seqmagick convert --include-from-file newclean_ids group12_contigs.fasta newgroup12_contig
$ seqmagick info !$
seqmagick info newgroup12_contigs.fasta
name                      alignment    min_len   max_len   avg_len  num_seqs
newgroup12_contigs.fasta FALSE           5587     30751  16768.14         7
```

We can get stats on all our files:

```
$ seqmagick info *fasta > fasta-info
$ cat !$
name                      alignment    min_len   max_len   avg_len  num_seqs
group12_contigs.fasta     FALSE           5136    116409  22974.30       132
group20_contigs.fasta     FALSE           5029     22601   7624.38       203
group24_contigs.fasta     FALSE           5024     81329  12115.70       139
newgroup12_contigs.fasta FALSE           5587     30751  16768.14         7
```

We can use "cut" to view various columns:

```
$ cut -f 2 fasta-info
$ cut -f 2,4 fasta-info
$ cut -f 2-4 fasta-info
```

But it does not line up very nicely. We can use "column" to fix this:

```
$ cut -f 2-4 fasta-info | column -t
alignment  min_len  max_len
FALSE      5136     116409
FALSE      5029     22601
FALSE      5024     81329
FALSE      5587     30751
```

Gapminder

Do the following:

```
$ git clone https://github.com/kyclark/metagenomics-book
$ cd metagenomics-book/problems/gapminder/data
```

How many files are in the "data" directory?

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

How many lines are in each/all of the files?

```
$ wc -l *
```

You can use `cat` to spew at the entire contents of a file into your shell, but if you'd just like to see the top of a file, you can use:

```
$ head Trinidad_and_Tobago.cc.txt
```

If you only want to see 5 lines, use `-n 5` or `-5` .

For our exercise, we'd like to combine all the files into one file we can analyze. That's easy enough with:

```
$ cat * > all.txt
```

Let's use `head` to look at the top of file:

```
$ head -5 all.txt
Afghanistan    1997    22227415    Asia    41.763    635.341351
Afghanistan    2002    25268405    Asia    42.129    726.7340548
Afghanistan    2007    31889923    Asia    43.828    974.5803384
Afghanistan    1952    8425333    Asia    28.801    779.4453145
Afghanistan    1957    9240934    Asia    30.332    820.8530296
```

Hmm, there are no column headers. Let's fix that. There's one file that's pretty different in content (it has only one line) and name ("country.cc.txt"):

```
$ cat country.cc.txt
country    year    pop    continent    lifeExp    gdpPercap
```

Those are the headers that you can combine to all the other files to get named columns, something very important if you want to look at the data in Excel and R/Python data frames.

```
$ rm all.txt
$ mv country.cc.txt headers
$ cat headers *.txt > all.txt
$ head -5 all.txt | column -t
country      year  pop       continent  lifeExp  gdpPercap
Afghanistan  1997  22227415  Asia       41.763   635.341351
Afghanistan  2002  25268405  Asia       42.129   726.7340548
Afghanistan  2007  31889923  Asia       43.828   974.5803384
Afghanistan  1952  8425333   Asia       28.801   779.4453145
```

Yes, that looks much better. Double-check that the number of lines in the `all.txt` match the number of lines of input:

```
$ wc -l *.cc.txt headers
$ wc -l all.txt
```

How many observations do we have for 1952?

```
$ grep 1952 all.txt | wc -l
$ cut -f 2 *.cc.txt | grep 1952 | wc -l
```

Those numbers aren't the same! Why is that?

```
$ grep 1952 all.txt | cut -f 2 | sort | uniq -c
 142 1952
   1 1982
   1 1987
```

```
$ grep 1952 all.txt | grep 198[27]
Lebanon    1982    3086876    Asia    66.983    7640.519521
Mozambique    1987    12891952    Africa    42.861    389.8761846
```

How many observations for every year?

```
$ cut -f 2 *.cc.txt | sort | uniq -c
```

How many observations are present for Africa?

```
$ grep Africa all.txt | wc -l
```

How many for each continent?

```
$ cut -f 4 *.cc.txt | sort | uniq -c
```

What was the world population in 1952? As we've seen, just using `grep 1952` is not sufficient. We want to take the 3rd column if the 2nd column is equal to "1952." `awk` will let us do just that. Normally `awk` will split on whitespace, so we need to use `-F"\t"` to tell it to split on the tab (`\t`) character. Use `man awk` to learn more.

```
$ awk -F"\t" '$2 == "1952" { print $3 }' *.cc.txt
```

I'll bet you didn't notice that one of those numbers was in scientific notation. That's going to cause a problem. Here it is:

```
$ awk -F"\t" '$2 == "1952" { print $3 }' *.cc.txt | grep [a-z]
3.72e+08
```

We have to throw in a `grep -v` to get rid of that (the `-v` reverses the match), then use the `paste` command is used to put a "+" in between all the numbers:

```
$ awk -F"\t" '$2 == "1952" { print $3 }' *.cc.txt | grep -v [a-z] | paste -sd+ -
```

and then we pipe that to the `bc` calculator:

```
$ awk -F"\t" '$2 == "1952" { print $3 }' *.cc.txt | grep -v [a-z] | paste -sd+ - | bc
2034957150.999989
```

It bothers me that it's not an integer, so I'm going to use `printf` in the `awk` command to trim that:

```
$ awk -F"\t" '$2 == "1952" { printf "%d\n", $3 }' *.cc.txt | grep -v [a-z] | paste -sd+ - |
2406957150
```

How did population change over the years? Let's put a list of the unique years into a file called "years" and then `cat` over that to run the above for each year:

```
$ cut -f 2 *.txt | sort | uniq > years
$ for year in `cat years`; do echo -n $year ": " && awk -F"\t" "\$2 == $year { printf \"%d\n
1952 : 2406957150
1957 : 2664404580
1962 : 2899782974
1967 : 3217478384
```

```
1972 : 3576977158
1977 : 3930045807
1982 : 4289436840
1987 : 4691477418
1992 : 5110710260
1997 : 5515204472
2002 : 5886977579
2007 : 6251013179
```

That's kind of useful! Here's how I might put that into a script:

```
$ cat pop-years.sh
#!/bin/bash

set -u

YEARS="years"

cut -f 2 ./*.cc.txt | sort | uniq > "$YEARS"
NUM=$(wc -l $YEARS | awk '{print $1}')

if [[ "$NUM" -lt 1 ]]; then
  echo "No years ($NUM)!"
  exit 1
fi

while read -r YEAR; do
    echo -n "$YEAR: "
    awk -F"\t" "\$2 == $YEAR { printf \"%d\\n\", \$3 }" ./*.cc.txt | grep -v "[a-z]" | paste
done < "$YEARS"
$ ./pop-years.sh
1952: 2406957150
1957: 2664404580
1962: 2899782974
1967: 3217478384
1972: 3576977158
1977: 3930045807
1982: 4289436840
1987: 4691477418
1992: 5110710260
1997: 5515204472
2002: 5886977579
2007: 6251013179
```

How has life expectancy changed over the years? For this we'll need to write a little Python program. I'll `cat` the program so you can see it. You can type this in with `nano` and then do `chmod +x avg.py` to make it executable (or use the

one I added):

```
$ cat avg.py
#!/usr/bin/env python3

import sys

args = list(map(float, sys.argv[1:]))
print(str(sum(args) // len(args)))
$ for year in `cat years`; do echo -n "$year: " && grep $year *.txt | cut -f 5 | xargs ./avg
1952: 49.0
1957: 51.0
1962: 53.0
1967: 55.0
1972: 57.0
1977: 59.0
1982: 61.0
1987: 63.0
1992: 64.0
1997: 65.0
2002: 65.0
2007: 66.0
```

How many observations where the life expectancy ("lifeExp," field #5) is greater than 40? For this, let's use the `awk` tool.

```
$ awk -F"\t" '$5 > 40' all.txt | wc -l
```

How many of those are from Africa? We can either use `cut` to get the 4th field or ask `awk` to print the 4th field for us:

```
$ awk -F"\t" '$5 > 40' all.txt | cut -f 4 | grep Africa | wc -l
$ awk -F"\t" '$5 > 40 {print $4}' all.txt | grep Africa | wc -l
```

How many countries had a life expectancy greater than 70, grouped by year?

```
$ awk -F"\t" '$5 > 70 { print $2 }' *.cc.txt | sort | uniq -c
   5 1952
   9 1957
  16 1962
  25 1967
  30 1972
  38 1977
  44 1982
  49 1987
  54 1992
  65 1997
  75 2002
  83 2007
```

How could we add continent to this?

```
$ awk -F"\t" '$5 > 70 { print $2 ":" $4 }' *.cc.txt | sort | uniq -c
```

As you look at the data and want to ask more complicated questions like how does `gdpPercap` affect `lifeExp`, you'll find you need more advanced tools like Python or R. Now that the data has been collated and the columns named, that will be much easier.

What if we want to add headers to each of the files?

```
$ mkdir wheaders
$ for file in *.txt; do cat headers $file > wheaders/$file; done
$ wc -l wheaders/* | head -5
      13 wheaders/Afghanistan.cc.txt
      13 wheaders/Albania.cc.txt
      13 wheaders/Algeria.cc.txt
      13 wheaders/Angola.cc.txt
      13 wheaders/Argentina.cc.txt
$ head wheaders/Vietnam.cc.txt
country    year    pop    continent    lifeExp    gdpPercap
Vietnam    1952    26246839    Asia    40.412    605.0664917
Vietnam    1957    28998543    Asia    42.887    676.2854478
Vietnam    1962    33796140    Asia    45.363    772.0491602
Vietnam    1967    39463910    Asia    47.838    637.1232887
Vietnam    1972    44655014    Asia    50.254    699.5016441
Vietnam    1977    50533506    Asia    55.764    713.5371196
Vietnam    1982    56142181    Asia    58.816    707.2357863
Vietnam    1987    62826491    Asia    62.82    820.7994449
Vietnam    1992    69940728    Asia    67.662    989.0231487
```