Introduction to the Unix shell for Biologists

R. Burke Squires

NIAID BCBB

This work adapted from Konrad Förstner and Chaochih Liu

Outline:

1. Setup
2. Motivation and background
3. Why GNU/Linux
4. Basic anatomy of a command line call
5. How to get help and documentation
6. Bash keyboard shortcuts
7. Files, folders, locations
8. Manipulating files and folder
9. File content - part 1
10. File content - part 2
11. Finding things in files
12. Connecting tools
13. Repeating command using the for loop
14. Shell scripting
15. Examples analysis

16. Additional resources

Setup - Create and download some test files

Use the `Makefile` in this repo and run

```
$ make example_files
```

This should create folder `unix_course_files` thank contains several examples files.

Motivation and background

In this course you will learn the basics of how to use the Unix shell. Unix is a class of operating systems with many different flavors including well-known ones like GNU/Linux (Ubuntu, RedHat) and the BSDs. The development of Unix and its shell (also known as **command line** interface) dates back to the late 1960s. Still, their concepts lead to very powerful tools. In the command line you can easily combine different tools into pipelines, avoid repetitive work and make your workflow reproducible. Knowing how to use the shell will also enable you to run programs that are only developed for this environment which is the case for many bioinformatics tools.

Why GNU/Linux?

- Software costs $0
- Advanced Multitasking
- Remote tasking (“real networking”)
- Multiuser
- Easy access to programming languages, databases, open-source projects

Why GNU/Linux?

- Software freedoms
  - Free to use for any purpose
  - Free to study and modify the source code
  - Free to share
  - Free to share modified versions
- No dependence on vendors
- Better performance
- More up-to-date
- Many more reasons...

Basics

Things to keep in mind throughout this tutorial

Remember the UNIX/LINUX command line is case sensitive!

All commands in this manual are printed in gray code boxes.

Commands given in red are considered more important for beginners than commands given in black.

The hash (pound) sign `#` indicates the start of comments for commands.

The notation `<&hellip;>` refers to variables and file names that need to be specified by the user. The symbols `<` and `>` need to be excluded.

Logging-In

Mac or LINUX

To log-in to the remote Linux shell, open the terminal and type:

```
ssh <your_username>@<host_name>
```

Where `ssh` stands for secure shell and `host_name` is the remote server's domain

name (i.e. login.msi.umn.edu). You will then be asked to enter your password, simply type it and press enter.

The format you will use to log-in to BioCompACE is:

```
ssh biocompace.icermali.org
```

To reset your password please go here.

The basic anatomy of a command line call

Running a tool in the command line interface follows a simple pattern. At first you have to write the name of the command (if it is not globally installed it's precise location needs to be given - we will get to this later). Some programs additionally require parameters. While the parameters are the requirement of the program the actual values we give are called arguments. There are two different ways how to pass those arguments to a program - via keywords parameter (also called named keywords, flags or options) or via positional parameters. The common pattern looks like this (`<>` indicates obligatory items, `[]` indicates optional items):

```
<program name> [keyword parameters] [positional parameters]
```

An example is calling the program `ls` which **ls**ts the content of a directory. You can simply call it without any argument

```
$ ls
```

or with one or more keyword argument

```
$ ls -l
$ ls -lh
```

or with one or more positional arguments

```
$ ls test_folder
```

or with one or more keyword and positional arguments

```
$ ls -l test_folder
```

The result of a command is written usually to the so called *standard output* of the shell which is the screen shown to you. We will later learn how to redirect this e.g. to the *standard input* of another program.

How to get help and documentation

Especially in the beginning you will have a lot of questions what a command does and which arguments and parameters need to be given. One rule before using a command or before asking somebody about it is called RTFM (please check the meaning yourself). Maybe the most important command is `man` which stands for *manual*. Most commands offer a manual and with `man` you can read those. To get the documentation of `ls` type

```
$ man ls
```

To close the manual use `q`. Additionally or alternatively many tools offer some help via the parameter `-h`, `-help` or `--help`. For example `ls`:

```
$ ls --help
```

Other tools present this help if they are called without any parameters or arguments.

Bash keyboard shortcuts

There are different implementations of the Unix shell. You are currently working with Bash (**B**ourne-**a**gain **sh**ell). Bash has several keyboard shortcuts that improve the interaction. Here is a small selection:

- Ctrl-c - Stop the command
- Ctlr-↑ - Go backward in command history
- Ctlr-↓ - Go forward in command history
- Ctrl-a - Jump to the beginning of a line
- Ctrl-e - Jump to the end of a line
- Ctrl-u - Remove everything before the cursor position
- Ctrl-k - Remove everything after the cursor position
- Ctrl-l - Clean the screen

- Ctrl-r - Search in command history
- Tab - extend commands and file/folder names

Files, folders, locations

Topics:

- `ls`
- `pwd`
- `cd`
- `mkdir`
- Relative vs. absolute path
- `~/`

In this part you will learn how to navigate through the file system, explore the content of folders and create folders.

At first we need to know where we are. If you open a new terminal you should be in your home directory (we will explain this below). To test this, call the program `pwd` which stands for **p**rint **w**orking **d**irectory.

```
$ pwd
/home/ubuntu
```

The default user of the Ubuntu live system is called `ubuntu`. In general each user has a folder with its user name located inside the folder `home`. The next command we need and which has been already mentioned above is `ls`. It simply lists the content of a folder. If you call it without any arguments it will output the content of the current folder. Using `ls` we want to get a rough overview of what a common Unix file system tree looks like and learn how to address files and folders. The root folder of a systems starts with `/`. Call

```
$ ls /
```

to see the content of the root folder. You should see something like

```
bin   data  etc  lib    lost+found  mnt  proc  run   srv  tmp  var
boot  dev   home lib64  media       opt  root  sbin  sys  usr
```

There are several subfolders in the so-called root folder (and yes, to make it a little bit confusing there is even a folder called `root` in the root folder). Those are more important if you are the administrator of the system. Normal users do not have the permission to make changes here. Currently your home directory is your little universe in which you can do whatever you want. In here we will learn how work with paths. A file or folder can be addressed either with its *absolute* or *relative path*. As you have downloaded and decompressed the test data you should have a folder `unix_course_files` located in your home folder. Assuming you are in this folder (`/home/ubuntu/`) the relative path to the folder is simply `unix_course_files` . You can get the content of the folder listed by calling `ls` like this:

```
$ ls unix_course_files
```

This is the so called *relative path* as it is relative to the current work directory `/home/ubuntu/` . The *absolute path* would start with a `/` and is `/home/ubuntu/unix_course_files` . Call `ls` like this:

```
$ ls /home/ubuntu/unix_course_files
```

There are some conventions regarding *relative* and *absolute paths*. One is that a dot (`.`) represents the current folder. The command

```
$ ls ./
```

should return the same as simply calling

```
$ ls
```

Two dots (`..`) represent the parent folder. If you call

```
$ ls ../
```

you should see the content of `/home` . If you call

```
$ ls ../../
```

you should see the content of the parent folder of the parent folder which is the root folder (`/`) assuming you are in `/home/ubuntu/` . Another convention is that `~/` represents the home directory of the user. The command

```
$ ls ~/
```

should list the content of your home directory independent of your current

location in the file system.

Now as we know where we are and what is there we can start to change our location. For this we use the command `cd` (change directory). If you are in your home directory `/home/ubuntu/` you can go into the folder `unix_course_files` by typing

```
$ cd unix_course_files
```

After that call `pwd` to make sure that you are in the correct folder.

```
$ pwd
/home/ubuntu/unix_course_files
```

To go back into your home directory you have different options. Use the *absolute path*

```
$ cd /home/ubuntu/
```

or the above mentioned convention for the home directory `~/`:

```
$ cd ~/
```

or the *relative path*, in this case the parent directory of `/home/ubuntu/unix_course_files`:

```
$ cd ../
```

As the home directory is such an important place `cd` uses this as default argument. This means if you call `cd` without argument you will go to the home directory. Test this behavior by calling

```
$ cd
```

Try now to go to different locations in the file system and list the files and folders located there.

Now we will create our first folder using the command `mkdir` (*make directory*). Go into the home directory and type:

```
$ mkdir my_first_folder
```

Here we can discuss the implementation of another Unix philosophy: "No news is good news." The command successfully created the folder `my_first_folder`. You can check this by calling `ls`, but `mkdir` did not tell you this. If you do not get a

message this usually means everything went fine. If you call the above `mkdir` command again you should get an error message like this:

```
$ mkdir my_first_folder
mkdir: cannot create directory &lsquo;my_first_folder&rsquo;: File
exists
```

So if a command does not complain you can usually assume there was no error.

Manipulating files and folder

Topics:

- `touch`
- `cp`
- `mv`
- `rm`

touch

Next we want to manipulate files and folders. We create some dummy files using `touch` which is usually used to change the time stamps of files. But you can also create empty files with it easily. Let's create a file called `test_file_1.txt`:

```
$ touch test_file_1.txt
```

Use `ls` to check that it was created.

cp

The command `cp` (*copy*) can be used to copy files. For this it requires at least two arguments: the source and the target file. In the following example we generate a copy of the file `test_file_1.txt` called `a_copy_of_test_file.txt`.

```
$ cp test_file_1.txt a_copy_of_test_file.txt
```

Use `ls` to confirm that this worked. We can also copy the file in the folder `my_first_folder` which we have created above:

```
$ cp test_file_1.txt my_first_folder
```

Now there should be also a file `test_file_1.txt` in the folder `my_first_folder`. If you want to copy a folder and its content you have to use the parameter `-r`.

```
$ cp -r my_first_folder a_copy_of_my_first_folder
```

You can use the command `mv` (*move*) to rename or relocate files or folders. To rename the file `a_copy_of_test_file.txt` to `test_file_with_new_name.txt` call

mv

```
$ mv a_copy_of_test_file.txt test_file_with_new_name.txt
```

With `mv` you can also move a file into a folder. For this the second argument has to be a folder. For example, to move the file now named `test_file_with_new_name.txt` into the folder `my_first_folder` use

```
$ mv test_file_with_new_name.txt my_first_folder
```

You are not limited to one file if you want to move them into a folder. Let's create and move two files `file1` and `file2` into the folder `my_first_folder`.

```
$ touch file1 file2
$ mv file1 file2 my_first_folder
```

At this point we can introduce another handy feature most shells offer which is called *globbing*. Let us assume you want to apply the same command to several files. Instead of explicitly writing all the file names you can use a *globbing pattern* to address them. There are different wildcards that can be used for these patterns. The most important one is the asterisk (`*`). It can replace none, one or more characters. Let us explore this with a small example:

```
$ touch file1.txt file2.txt file3
$ ls *txt
$ mv *txt my_first_folder
```

The `ls` shows the two files matching the given pattern (i.e. `file1.txt` and `file2.txt`) while dismissing the one not matching (i.e. `file3`). Same for `mv` - it will only move the two files ending with `txt`.

rm

We accumulated several test files that we do not need anymore. Time to clean up a little bit. With the command `rm` (*remove*) you can delete files and folders. Please

be aware that there is no such thing as a trash bin if you remove items this way. They will be gone for good and without further notice.

To delete a file in `my_first_folder` call:

```
$ rm my_first_folder/file1.txt
```

To remove a folder use the parameter `-r` (*recursive*):

```
$ rm -r my_first_folder
```

Alternatively you can use the command `rmdir`:

```
$ rmdir my_first_folder
```

Finding Files in Directories and Applications

Here are some important commands to know:

```
find -name "*pattern*"            # searches for *pattern* in and below
current directory
find /usr/local -name "*blast*"   # finds file names *blast* in
specified directory
find /usr/local -iname "*blast*"  # Same as above, but case insensitive
```

Here are some additional useful arguments that would be good to know exist:

- `-user <user name>`
- `-group <group name>`
- `-ctime <number of days ago changed>`

```
find ~ -type f -mtime -2        # finds all files you have modified in
the last two days
locate <pattern>                # finds files and directories that are
written into update file
```

```
which <application_name>        # location of application
whereis <application_name>      # searches for executeables in set of
directories
dpkg -l | grep mypattern        # find Debian packages and refine
search with grep pattern
```

Example of a `find` command we will be using frequently later on today

```
find `pwd` -name "filename" | sort
```

File content - part 1

Topics:

- `less` / `more`
- `cat`
- `echo`
- `head`
- `tail`
- `cut`

Until now we did not care about the content of the files. This will change now. Please go into the folder `unix_course_files`:

```
$ cd unix_course_files
```

less/more

There should be some files waiting for you. To read the content with the possibility to scroll around we need a so called pager program. Most Unix systems offer the programs `more` and `less` which have very similar functionalities ("more or less are more or less the same"). We will use the later one here. Let's open the file `origin_of_species.txt`

```
$ less origin_of_species.txt
```

cat

The file contains Charles Darwin's *Origin of species* in plain text. You can scroll up and down line-wise using the arrow keys or page-wise using the page-up/page-down keys. To quit use the key `q`. With pager programs you can read file content interactively, but sometimes you just want to have the content of a file given to you (i.e. on the *standard output*). The command `cat` (*concatenate*) does that for one or more files. Let us use it to see what is in the example file `two_lines.txt`. Assuming you are in the folder `unix_course_files` you can call

```
$ cat two_lines.txt
```

The content of the file is shown to you. You can apply the command to two files and the content is concatenated and returned:

```
$ cat two_lines.txt three_lines.txt
```

This is a good time to introduce the *standard input* and *standard output* and what you can do with it. Above I wrote the output is given to you. This means it is written to the so called *standard output*. You can redirect the *standard output* into a file by using `>`. Let us use the call above to generate a new file that contains the combined content of both files:

```
$ cat two_lines.txt three_lines.txt > five_lines.txt
```

Please have a look at the content of this file:

```
$ cat five_lines.txt
```

echo

The *standard output* can also be redirected to other tools as *standard input*. More about this below. With `cat` we can reuse the existing file content. To create something new we use the command `echo` which writes a given string to the standard output.

```
$ echo "Something very creative"
```

To redirect the output into a target file use `>`.

```
$ echo "Something very creative." > creative.txt
```

Be aware that this can be dangerous. You will overwrite the content of an existing

file. For example if you call now

```
$ echo "Something very uncreative." > creative.txt
```

there will be only the latest string written to the file and the previous one will be overwritten. To append the output of a command to a file without overwriting the content use `>>`.

```
$ echo "Something very creative." > creative.txt
$ echo "Something very uncreative." >> creative.txt
```

Now `creative.txt` should contain two lines.

head \ tail

Sometimes you just want to get an excerpt of a file e.g. just the first or last lines of it. For this the commands `head` and `tail` can be used. Per default 10 lines are shown. You can use the parameter `-n <NUMBER>` (e.g. `-n 20` or just `-<NUMBER>` (e.g. `-20`) to specify the number of lines to be displayed. Test the tools with the file `origin_of_species.txt`:

```
$ head origin_of_species.txt
$ tail origin_of_species.txt
```

cut

You cannot only select vertically but also horizontally using the command `cut`. Let us extract only the first 10 characters of each line in the file `origin_of_species.txt`:

```
$ cut -c 1-10 origin_of_species.txt
```

The tool `cut` can be very useful to extract certain columns from CSV files (*comma/character separated values*). Have a look at the content of the file `genes.csv`. You see that it contains different columns that are tabular-separated. You can extract selected columns with `cut`:

```
$ cut -f 1,4 genes.csv
```

File content - part 2

Topics:

- `wc`
- `sort`
- `uniq`

wc

There are several tools that let you manipulate the content of a plain text file or return information about it. If you want for example some statistics about the number of character, words and lines use the command `wc`. Let us count the number of lines in the file `origin_of_species.txt`:

```
$ wc -l origin_of_species.txt
```

sort

You can use the command `sort` to sort a file alpha-numerically. Test the following calls

```
$ sort unsorted_numbers.txt
$ sort -n unsorted_numbers.txt
$ sort -rn unsorted_numbers.txt
```

and try to understand the output.

uniq

The tool `uniq` takes a sorted list of lines and removes line-wise the redundancy. Please have a look at the content of the file `redundant.txt`. Then use `uniq` to generate a non-redundant list:

```
$ uniq redundant.txt
```

If you call `uniq` with `-c` you get the number of occurrence for each remaining entry:

```
$ uniq -c redundant.txt
```

Finding Things in Files

Grep

* `grep` searches **within files** whereas `find` searches **directories**

```
grep pattern file           # provides lines in 'file' where pattern
'appears'
                            # if pattern is shell function use single
quotes: '>'

grep -H pattern             # -H prints out file name in front of
pattern

grep 'pattern' file | wc    # pipes lines with pattern into word count
'wc'
                            # wc arguments:
                            #   -c: show only bytes
                            #   -w: show only words
                            #   -l: show only lines
                            # help on regular expressions:
                            #   $ man 7 regex
                            #   man perlre
```

With the tool `grep` you can extract lines that match a given pattern. For instance, if you want to find all lines in `origin_of_species.txt` that contain the word `species` call

```
$ grep species origin_of_species.txt
```

As you can see we only get the lines that contain `species` but not the ones that contain `Species`. To make the search case-insensitive use the parameter `-i`.

```
$ grep -i species origin_of_species.txt
```

If you are only interested in the number of lines that match the pattern use `-c`:

```
$ grep -ic species origin_of_species.txt
```

Connecting tools

Another piece of the Unix philosophy is to build small tools that do one thing optimally and use the standard input and standard output. The real power of Unix builds on the capability to easily connect tools. For this so-called *pipes* are used. To use the *standard output* of one tool as *standard input* of another tool the vertical bar `|` is used. For example, in order to extract the first 1000 lines from `origin_of_species.txt`, search for lines that contain `species`, then search in those lines the ones which contain `wild` and finally replace the `w`s by `m`s call (Please write this in one line in the shell and remove the `\`):

Redirections

Before we jump into redirections, I would like to review wildcards.

Wildcards are denoted by * and it is used to specify many files (I'll discuss more details about this later and show an example).

A few examples of formats are:

- <beginning-of-filename>*
- *<end-of-filename>
- *<middle-of-filename>*

Now, we'll go back to redirections.

The following commands are redirections:

```
ls > file               # stores ls output into specified file
command < my_file       # uses file after '<' as STDIN
command >> my_file      # appends output of one command to file (will
not overwrite file with same filename)
command | tee my_file   # writes STDOUT to file and print to screen
command > my_file; cat my_file  # writes STDOUT to file,
                                # then prints it to screen
                                # semi-colon designates end of command
grep my_pattern my_file | wc    # Pipes (|) output of 'grep' to 'wc'
```

Piping

Piping is another form of redirects.

It is a way to chain commands together

- Can take the STDOUT of one command and send it to STDIN of another
- Denoted by `|` symbol

Example:

```
find `pwd` -name "name_of_file" | sort

head -n 1000 origin_of_species.txt | grep species | grep wild | tr w m
```

Repeating command using the `for` loop

Assuming you want to generate a copy of each of your files ending with ´.txt´. A

```
cp *txt copy_of_*txt
```

would not work.

With `for` loops you can solve this problem. Let's start with a simple one.

```
for FILE in three_lines.txt two_lines.txt
> do
> head -n 1 $FILE
> done
```

The variable `FILE` (you can give it also any other name) can be used inside of the loop.

If you press now Ctr-↑ you will get the line

```
for FILE in three_lines.txt two_lines.txt; do head -n 1 $FILE; done
```

which is equivalent to the call before. You can not only call one command inside of a loop but several:

```
for FILE in three_lines.txt two_lines.txt
> do
> head -n 1 $FILE
> echo "-----------------"
> done
```

```
for FILE in *txt
```

```
> do
> head -n 1 $FILE
> echo "-----------------"
> done
```

```
for FILE in *txt
> do
> cp $FILE copy_of_$FILE
> done
```

Shell scripting

Open a new file in a text editor of you choice, call it `count_lines.sh` and add the following text:

```
#!/bin/bash
# count_lines.sh
echo "Number of lines in the given file":
wc -l origin_of_species.txt
```

Save the file, make sure the file `origin_of_species.txt` is in the same folder and run it the script:

```
$ bash count_lines.sh
```

You should get someting like

```
Number of lines that contains species:
15322 origin_of_species.txt
```

This a very first shell script. Now we want to make it more flexible. Instead of hard coding the input file for `wc -l` we want to be able to give this as argument to the shell script. For this we change the shell script to:

```
echo "Number of lines in the given file":
wc -l $1
```

The `$1` is a varible that represents the first argument given to the shell scrip. Now you can call the script in the following way

```
$ bash count_lines.sh origin_of_species.txt
```

You should get the same results as before. If you also like to take the second argument use the variable `$2`. For using all arguments given to the shell script use the variable "$@". E.g change the shell script to:

```
echo "Number of lines in the given file(s)":
wc -l $@
```

and run it with several input files:

```
bash count_lines.sh origin_of_species.txt genes.csv
```

You should get something like:

```
Number of lines that contains species:
 15322 origin_of_species.txt
     5 genes.csv
 15327 total
```

If/then Statements

Different lines of code can be executed depending on whether or not something tests true or not.

```
#   The condition is whether or not SAMTools is installed
#   If SAMTools is installed, 'THEN' indexes the fasta file
#   'ELSE' tells us that SAMTools is not installed

if `command -v samtools > /dev/null 2> /dev/null`
then
    echo "SAMTools is installed"
    samtools my_sequence_data.fasta reference.fasta
else
    echo "SAMTools is not installed"
fi
```

Examples analysis

Equipped with a fine selection of useful programs and basic understanding of how to combine them, we will no apply them to analyze real biological data.

Retrieving data

You have used the tool `wget` above to download the example files. It is very useful, especially, if you want to retrieve large data sets. We download the fasta file of the *Salmonella* Thyphimuirum SL1344 chromosome by calling (in this document the URL is split into three lines. Please write it in one line in the shell and remove the `\`).

```
$ wget ftp://ftp.ncbi.nlm.nih.gov/genomes/archive/old_refseq/Bacteria/\
Salmonella_enterica_serovar_Typhimurium_SL1344_uid86645/\
NC_016810.fna
```

Additionally, we download the annotation in GFF format of the same replicon:

```
$ wget ftp://ftp.ncbi.nlm.nih.gov/genomes/archive/old_refseq/Bacteria/\
Salmonella_enterica_serovar_Typhimurium_SL1344_uid86645/\
NC_016810.gff
```

Counting the number of features

Use `less` to have a look at `NC_016810.gff`. It is a tabular-separated file. The first 5 lines start with `#` and are called header. Then several lines with 9 columns follow. The third column contains the type of the entry (gene, CDS, tRNA, rRNA, etc). If we want to know the numbers of tRNA entries we could try to apply `grep` and use `-c` to count the number of matching lines.

```
$ grep -c tRNA NC_016810.gff
```

This leads to a suspiciously large number. The issue is that the string `tRNA` also occurs in the attribute column (the 9th column). We just want to select lines with a match in the third column. This can be achieved by combining `cut` and `grep`.

```
$ cut -f 3 NC_016810.gff | grep -c tRNA
```

To get the number of entries for all other features we could just replace the `tRNA` e.g. by `rRNA`. But we can also get the number for all of them at once using this constellation:

```
$ grep -v "#" NC_016810.gff | cut -f 3 | sort | uniq -c
```

Try to understand what we did here. You can use a similar call to count the number

genes on the plus and minus strand:

```
$ cut -f 3,7 NC_016810.gff | grep gene | sort | uniq -c
```

Additional Resources

- A number of UNIX shell cheatsheets can be found here
- Learn bash in Y minutes
- Learning the Shell
- Writing Shell Scripts
- Shell Scripting Tutorial
- Unix for Biologits