labibi Documentation

Release 0.1

C. Titus Brown

September 28, 2015

Contents

3. Titus Brown <ctb@msu.edu> and Alexandra Pawlik <a.pawlik@software.ac.uk>

We will use this Etherpad site to distribute commands.

Explain: minute cards; stick notes on monitors.

CHAPTER 1

Session 1: command line & BLAST

1.1 Welcome!

Welcome lecture

Materials from the Next-Gen Sequence Analysis Workshop including the instructions on how to install and configure all software that you will need can be found here: http://ged.msu.edu/angus/tutorials-2013/

TODO:

1. start up your virtual machine
2. start up a browser
3. start up a terminal!
4. copying and pasting commands
5. copying common files around.

1.1.1 Copying & pasting commands

Tips on copying and pasting commands:

- left mouse button to drag-select (copy) from this Web page;
- middle mouse button (push scroll wheel) to paste.

Try copy/pasting this into the terminal:

```
echo 'hello, world'
```

Try copying multiple lines at once:

```
ls
echo 'this is my directory'
```

Done! You're now an expert!

1.2 BLAST

BLAST lecture

Download and install some useful scripts:

```
git clone https://github.com/ngs-docs/ngs-scripts ~/software/ngs-scripts
```

Create a working directory under the main user directory, and change to that working directory:

```
cd
mkdir blast
cd blast
```

('pwd' should show that you are in /home/student/blast).

Download the E. coli MG1655 protein data set:

```
curl -O http://ftp.ncbi.nlm.nih.gov/genomes/Bacteria/Escherichia_coli_K_12_substr__MG1655_uid57779/NC
```

This grabs that URL and saves the contents of 'NC_000913.faa' to the local disk.

Grab a Prokka-generated set of proteins (see e.g. http://2013-caltech-workshop.readthedocs.org/en/latest/prokka-annotation.html):

```
curl -O http://athyra.idyll.org/~t/ecoli0104.faa
```

Let's take a quick look at these files:

```
head ecoli0104.faa
head NC_000913.faa
```

These are FASTA protein files.

Format it for BLAST and run BLAST of the O104 protein set against the MG1655 protein set:

```
formatdb -i NC_000913.faa -o T -p T
blastall -i ecoli0104.faa -d NC_000913.faa -p blastp -e 1e-12 -o 0104.x.NC -a 8
```

Look at the output file:

```
head -20 0104.x.NC
```

Let's convert 'em to a CSV file:

```
python ~/software/ngs-scripts/blast/blast-to-csv-with-names.py ecoli0104.faa NC_000913.faa 0104.x.NC
```

This creates a file '0104.x.NC.csv', which you can open in a spreadsheet program like Excel or Google Docs/Spreadsheet.

```
grep ^'>' fasta_file | wc -l
```

1.2.1 Reciprocal BLAST calculation

Do the reciprocal BLAST, too:

```
formatdb -i ecoli0104.faa -o T -p T
blastall -i NC_000913.faa -d ecoli0104.faa -p blastp -e 1e-12 -o NC.x.0104 -a 8
```

Extract reciprocal best hit:

```
python ~/software/ngs-scripts/blast/blast-to-ortho-csv.py ecoli0104.faa NC_000913.faa 0104.x.NC NC.x
```

This generates a file 'ortho.csv', containing the ortholog assignments and their annotations.

CHAPTER 2

Session 2: mRNAseq assembly and annotation

2.1 Copying common files around

Go to Activities (upper left) & click; select the file drawer.

Go to Bookmarks, Training materials. Enter the Super Secret Password (on the board at the front).

Double click on 'NGS bootcamp biologists'.

Copy the desired files (right click, copy); then go over the Bookmarks again, select 'Home'. Do 'paste'.

2.2 Working with mRNAseq data

(Based https://khmer-protocols.readthedocs.org/en/latest/

`reads_and_qc`

`mrnaseq-assembly`

CHAPTER 3

Session 3: Shell and shell scripting

An extended version of this tutorial is available here: https://github.com/swcarpentry/boot-camps/tree/2013-06-wise-beginners/shell

Want to learn more? Come to one of the Software Carpentry bootcamps! http://www.software-carpentry.org/

Material by Milad Fatenejad, Sasha Wood, Radhika Khetani and Tracy Teal

3.1 What is the shell and how do I access the shell?

The *shell* is a program that presents a command line interface which allows you to control your computer using commands entered with a keyboard instead of controlling graphical user interfaces (GUIs) with a mouse/keyboard combination.

A *terminal* is a program you run that gives you access to the shell. There are many different terminal programs that vary across operating systems.

3.2 Hello, World!

One very basic command is *echo*. This command just prints text to the terminal. Try the command:

```
echo Hello, World
```

Then press enter. You should see the text "Hello, World" printed back to you. The echo command is useful for printing from a shell script, for displaying variables, and for generating known values to pass to other programs.

3.3 Moving around the file system

Let's learn how to move around the file system using command line programs. This is really easy to do using a GUI (just click on things). Once you learn the basic commands, you'll see that it is easy to do in the shell too.

First we have to know where we are. The program *pwd* (print working directory) tells you where you are sitting in the directory tree. The command *ls* will list the files in the current directory. Directories are often called "folders" because of how they are represented in GUIs. Directories are just listings of files. They can contain other files or directories.

Whenever you start up a terminal, you will start in a special directory called the *home* directory. Every user has their own home directory where they have full access to do whatever they want. In this case, the *pwd* command tells us that the name of our home directory is. The last word in that listing should also be the name of your user. You can also find out your user name by entering the command *whoami*.

3.4 File Types

When you enter the *ls* command, it lists the contents of the current directory. There are several items in your home directory.

Let's create an empty file using the *touch* command. Enter the command:

```
touch testfile
```

Then list the contents of the directory again. You should see that a new entry, called *testfile*, exists. The *touch* command just creates an empty file. *ls -l* gives a lot more information too, such as the size of the file and information about the owner. If the entry is a directory, then the first letter will be a "d". The fifth column shows you the size of the entries in bytes. Notice that *testfile* has a size of zero.

Now, let's get rid of *testfile*. To remove a file, just enter the command:

```
rm testfile
```

The *rm* command can be used to remove files. If you enter *ls* again, you will see that *testfile* is gone.

3.5 Changing Directories

For some parts of the tutorial we will be using sample data. In order to download them, type into the terminal the following command:

```
git clone https://github.com/ngs-docs/2013-norwich-biology.git TGAC-workshop
```

Now, let's move to a different directory. The command *cd* (change directory) is used to move around. Let's move into the *TGAC-workshop* directory. Enter the following command:

```
cd TGAC-workshop
```

Now use the *ls* command to see what is inside this directory. This directory contains all of the material for this workshop. Now move to the directory containing the data for the shell tutorial:

```
cd data
```

If you enter the *cd* command by itself, you will return to the home directory.

3.6 Arguments (flags)

Most programs take additional arguments that control their exact behavior. For example, *-F* and *-l* are arguments to *ls*. The *ls* program, like many programs, take a lot of arguments. But how do we know what the options are to particular commands?

Most commonly used shell programs have a manual. You can access the manual using the *man* program. Try entering:

```
man ls
```

This will open the manual page for *ls*. Use the space key to go forward and b to go backwards. When you are done reading, just hit *q* to exit.

Programs that are run from the shell can get extremely complicated. To see an example, open up the manual page for the *find* program, which we will use later this session. No one can possibly learn all of these arguments, of course. So you will probably find yourself referring back to the manual page frequently. Note: sometimes it can be pretty difficult to understand what it says in a man file. However, each time you read a man file you will understand more of it.

3.7 Examining the contents of other directories

By default, the *ls* commands lists the contents of the working directory (i.e. the directory you are in). However, you can also give *ls* the names of other directories to view. Let's explore the directory where we downloaded the sample data:

```
ls TGAC-workshop
```

This will list the contents of the *TGAC-workshop* directory without you having to navigate there. Now enter:

```
ls TGAC-workshop/data
```

This prints the contents of *data*. The *cd* command works in a similar way. Try entering:

```
cd TGAC-workshop/data
```

and you will jump directly to *data* without having to go through the intermediate directory.

3.8 Full vs. Relative Paths

The *cd* command takes an argument which is the directory name. Directories can be specified using either a *relative* path or a full *path*. The directories on the computer are arranged into a hierarchy. The full path tells you where a directory is in that hierarchy, all the way from the root and downwards.

Navigate to the home directory. Now, enter the *pwd* command and you should see the full name of your home directory. This tells you that you are in a directory that is named the same as your user, which sits inside one or more other directories. The very top of the hierarchy is a directory called / which is usually referred to as the *root directory*.

First, figure out again what the full path to your home directory was. Now enter the following command (replace the stuff in <> with the results from *pwd*).

```
cd <pwd-results>/TGAC-workshop/data
```

This jumps to *data*. Now go back to the home directory. We saw earlier that the command

```
cd TGAC-workshop/data
```

had the same effect - it took us to the *data* directory. But, instead of specifying the full path which started with a /, we specified a *relative path*. In other words, we specified the path relative to our current directory. A full path always starts with a /. A relative path does not. You can usually use either a full path or a relative path depending on what is most convenient. If we are in the home directory, it is more convenient to just enter the relative path since it involves less typing.

Now, list the contents of the /bin directory. Do you see anything familiar in there?

3.9 Saving time with shortcuts, wild cards, and tab completion

Shortcuts

There are some shortcuts which you should know about. Dealing with the home directory is very common. So, in the shell the tilde character, ~, is a shortcut for your home directory. Navigate to the *shell* directory, then enter the command:

```
ls ~
```

This prints the contents of your home directory, without you having to type the full path. The shortcut .. always refers to the directory above your current directory. Thus:

```
ls ..
```

prints the contents of the ~/TGAC-workshop directory. You can chain these together, so:

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

prints the contents of what should be your home directory. Finally, the special directory . always refers to your current directory. So, *ls*, *ls .*, and *ls ./././.* all do the same thing, they print the contents of the current directory. This may seem like a useless shortcut right now, but we'll see when it is needed in a little while.

To summarize, the commands *ls ~*, *ls ~/.*, *ls ../../*, and *ls <absolute path to home directory>* all do exactly the same thing. These shortcuts are not necessary, they are provided for your convenience.

Wild cards

Navigate to the *~/TGAC-workshop/data/THOMAS* directory. This directory contains our hearing test data for THOMAS. If we type *ls*, we will see that there are a bunch of files which are just four digit numbers. By default, *ls* lists all of the files in a given directory. The * character is a shortcut for "everything". Thus, if you enter *ls **, you will again see all of the contents of a given directory. This * can be combined with other characters. Now try this command:

```
ls *1
```

This lists every file that ends with a *1*. This command:

```
ls /usr/bin/*.sh
```

Lists every file in */usr/bin* that ends in the characters *.sh*. And this command:

```
ls *4*1
```

lists every file in the current directory which contains the number *4*, and ends with the number *1*. There are four such files: *0241*, *0341*, *0431*, and *0481*.

So how does this actually work? Well...when the shell (bash) sees a word that contains the * character, it automatically looks for files that match the given pattern. In this case, it identified four such files. Then, it replaced the **4*1* with the list of files, separated by spaces. In other words, the two commands:

```
ls *4*1
ls 0241 0341 0431 0481
```

are exactly identical. The *ls* command cannot tell the difference between these two things.

Tab Completion

Navigate to the home directory. Typing out directory names can waste a lot of time. When you start typing out the name of a directory, then hit the tab key, the shell will try to fill in the rest of the directory name. For example, enter:

```
cd T<tab>
```

The shell will fill in the rest of the directory name for *TGAC-workshop*. Press enter to enter the workshop directory. Next, go into the data directory and do:

```
ls 3<tab><tab>
```

When you hit the first tab, nothing happens. The reason is that there are multiple file in this directory which start with 3. Thus, the shell does not know which one to fill in. When you hit tab again, the shell will list the possible choices.

Tab completion can also fill in the names of programs. For example, enter *e<tab><tab>*. You will see the name of every program that starts with an *e*. One of those is *echo*. If you enter *ec<tab>* you will see that tab completion works.

Command History

You can easily access previous commands. Hit the up arrow. Hit it again. You can step backwards through your command history. The down arrow takes your forwards in the command history.

^-C will cancel the command you are writing, and give you a fresh prompt.

You can display all your command history (since the last login) using command:

```
history
```

3.10 Which program?

Commands like *ls*, *rm*, *echo*, and *cd* are just ordinary programs on the computer. A program is just a file that you can *execute*. The program *which* tells you the location of a particular program. For example:

```
which ls
```

Will return "/bin/ls". Thus, we can see that *ls* is a program that sits inside of the */bin* directory. Now enter:

```
which find
```

You will see that *find* is a program that sits inside of the */usr/bin* directory.

So ... when we enter a program name, like *ls*, and hit enter, how does the shell know where to look for that program? How does it know to run */bin/ls* when we enter *ls*. The answer is that when we enter a program name and hit enter, there are a few standard places that the shell automatically looks. If it can't find the program in any of those places, it will print an error saying "command not found". Enter the command:

```
echo $PATH
```

This will print out the value of the *PATH* environment variable. Notice that a list of directories, separated by colon characters, is listed. These are the places the shell looks for programs to run. If your program is not in this list, then an error is printed. The shell ONLY checks in the places listed in the *PATH* environment variable.

Navigate to the *shell* directory and list the contents. You will notice that there is a program (executable file) called *hello* in this directory. Now, try to run the program by entering:

```
hello
```

You should get an error saying that hello cannot be found. That is because the directory <*your home directory*>*/TGAC-workshop* is not in the *PATH*. You can run the *hello* program by entering:

```
./hello
```

Remember that . is a shortcut for the current working directory. This tells the shell to run the *hello* program which is located right here. So, you can run any program by entering the path to that program. You can run *hello* equally well by specifying:

```
<path to home directory>/TGAC-workshop/hello
```

Or by entering:

```
../TGAC-workshop/hello
```

When there are no / characters, the shell assumes you want to look in one of the default places for the program.

3.11 Examining Files

We now know how to switch directories, run programs, and look at the contents of directories, but how do we look at the contents of files?

The easiest way to examine a file is to just print out all of the contents using the program *cat*. Enter the following command:

```
cat ex_data.txt
```

This prints out the contents of the *ex_data.txt* file. This file contains an example of how our data looks like. If you enter:

```
cat ex_data.txt ex_data.txt
```

It will print out the contents of *ex_data.txt* twice. *cat* just takes a list of file names and writes them out one after another (this is where the name comes from, *cat* is short for concatenate).

cat is a terrific program, but when the file is really big, it can be annoying to use. The program, *less*, is useful for this case. Enter the following command:

```
less ~/TGAC-workshop/data/dictionary.txt
```

less opens the file, and lets you navigate through it. The commands are identical to the *man* program. Use "space" to go forward and hit the "b" key to go backwards. The "g" key goes to the beginning of the file and "G" goes to the end. When you are done, hit "q" to quit.

less also gives you a way of searching through files. Just hit the "/" key to begin a search. Enter the word you would like to search for and hit enter. It will jump to the next location where that word is found. Try searching the *dictionary.txt* file for the word "cat". If you hit "/" then "enter", *less* will just repeat the previous search. *less* searches from the current location and works its way forward. If you are at the end of the file and search for the word "cat", *less* will not find it. You need to go to the beginning of the file and search.

Remember, the *man* program uses the same commands, so you can search documentation using "/" as well!

3.12 Redirection

Let's turn to the experimental data from the hearing tests. This data is located in the *data* directory. Each subdirectory corresponds to a particular participant in the study. Navigate to the *Lawrence* subdirectory in *data*. First, press *ls* to look at the files. There are a bunch of text files which contain experimental data results. Lets print them all:

```
cat *
```

Now enter the following command:

```
cat * > ../all_data
```

This tells the shell to take the output from the *cat* * command and dump it into a new file called *../all_data*. To verify that this worked, examine the *all_data* file. If *all_data* had already existed, we would overwritten it. So the > character tells the shell to take the output from whatever is on the left and dump it into the file on the right. The >> characters do almost the same thing, except that they will append the output to the file if it already exists.

3.13 Creating, moving, copying, and removing

We've created a file called *all_data* using the redirection operator >. This file is critical - it's our analysis results - so we want to make copies so that the data is backed up. Lets copy the file using the *cp* command. The *cp* command

backs up the file. Navigate to the *data* directory and enter:

```
cp all_data all_data_backup
```

Now *all_data_backup* has been created as a copy of *all_data*. We can move files around using the command *mv*. Enter this command:

```
mv all_data_backup /tmp/
```

This moves *all_data_backup* into the directory */tmp*. The directory */tmp* is a special directory that all users can write to. It is a temporary place for storing files. Data stored in */tmp* is automatically deleted when the computer shuts down.

The *mv* command is also how you rename files. Since this file is so important, let's rename it:

```
mv all_data all_data_IMPORTANT
```

Type in *ls*, and you will see that file name has been changed to all_data_IMPORTANT. Let's delete the backup file now:

```
rm /tmp/all_data_backup
```

The *mkdir* command is used to create a directory. Just enter *mkdir* followed by a space, then the directory name.

3.14 Count the words

The *wc* program (word count) counts the number of lines, words, and characters in one or more files. Make sure you are in the *data* directory, then enter the following command:

```
wc Lawrence/*
```

For each of the files indicated, *wc* has printed a line with three numbers and also the relative file name. The first is the number of lines in that file. The second is the number of words. Third, the total number of characters is indicated. The bottom line contains this information summed over all of the files.

Remember that the *Lawrence/** files were merged into the *all_data* file. So, we should see that *all_data* contains the same number of characters:

```
wc all_data
```

Every character in the file takes up one byte of disk space. Let's confirm this:

```
ls -l all_data
```

Remember that *ls -l* prints out detailed information about a file and that the fifth column is the size of the file in bytes.

3.15 The awesome power of the Pipe

Suppose I wanted to only see the total number of character, words, and lines across the files *Lawrence/**. I don't want to see the individual counts, just the total. Of course, I could just do:

```
wc all_data
```

Since this file is a concatenation of the smaller files. Sure, this works, but I had to create the *all_data* file to do this. We can do this *without* creating a temporary file, but first I have to show you two more commands: *head* and *tail*. These commands print the first few, or last few, lines of a file, respectively. Try them out on *all_data*:

```
head all_data
tail all_data
```

The *-n* option to either of these commands can be used to print the first or last *n* lines of a file. To print the first/last line of the file use:

```
head -n 1 all_data
tail -n 1 all_data
```

Let's turn back to the problem of printing only the total number of lines in a set of files without creating any temporary files. To do this, we want to tell the shell to take the output of the *wc Lawrence/** and send it into the *tail -n 1* command. The | character (called pipe) is used for this purpose. Enter the following command:

```
wc Lawrence/* | tail -n 1
```

This will print only the total number of lines, characters, and words across all of these files. What is happening here? Well, *tail*, like many command line programs will read from the *standard input* when it is not given any files to operate on. In this case, it will just sit there waiting for input. That input can come from the user's keyboard *or from another program*. Try this:

```
tail -n 2
```

Notice that your cursor just sits there blinking. Tail is waiting for data to come in. Now type:

```
French
fries
are
good
```

then CONTROL+d. You should get the lines:

```
are
good
```

printed back at you due to you asking tail to return the last two by doing -n 2.The CONTROL+d keyboard shortcut inserts an *end-of-file* character. It is sort of the standard way of telling the program "I'm done entering data". The | character replaces the data from the keyboard with data from another command. You can string all sorts of commands together using the pipe.

The philosophy behind these command line programs is that none of them really do anything all that impressive. BUT when you start chaining them together, you can do some really powerful things really efficiently. If you want to be proficient at using the shell, you must learn to become proficient with the pipe and redirection operators: |, >, >>.

3.16 A sorting example

Let's create a file with some words to sort for the next example. We want to create a file which contains the following names:

```
Bob
Alice
Diane
Charles
```

To do this, we need a program which allows us to create text files. We will use *gedit*. Navigate to */tmp* and enter the following command:

```
gedit toBeSorted
```

Now enter the four names as shown above.

When you are back to the command line, enter the command:

```
sort toBeSorted
```

Notice that the names are now printed in alphabetical order.

Try looking at this file with *less* - note that the file itself has not changed.

Let's navigate back to *data*. Enter the following command:

```
wc Lawrence/* | sort -k 3 -n
```

We are already familiar with what the first of these two commands does: it creates a list containing the number of characters, words, and lines in each file in the *Lawrence* directory. This list is then piped into the *sort* command, so that it can be sorted. Notice there are two options given to sort:

1. *-k 3*: Sort based on the third column
2. *-n*: Sort in numerical order as opposed to alphabetical order

Notice that the files are sorted by the number of characters.

Printing the smallest file seems pretty useful. We don't want to type out that long command often. Let's create a simple script, a simple program, to run this command. The program will look at all of the files in the current directory and print the information about the smallest one. Let's call the script *smallest*. Navigate to the *data* directory, then:

```
gedit smallest
```

Then enter the following text:

```
#!/bin/bash
wc * | sort -k 3 -n | head -n 1
```

Now, *cd* into the *Lawrence* directory and enter the command *../smallest*. Notice that it says permission denied. This happens because we haven't told the shell that this is an executable file. If you do *ls -l ../smallest*, it will show you the permissions on the left of the listing.

Enter the following commands:

```
chmod a+x ../smallest
../smallest
```

The *chmod* command is used to modify the permissions of a file. This particular command modifies the file *../smallest* by giving all users (notice the *a*) permission to execute (notice the *x*) the file. If you enter:

```
ls -l ../smallest
```

You will see that the file permissions have changed. Congratulations, you just created your first shell script!

Let's see if we can create a more useful script based on what we learnt from the mRNAseq normalization and assembly module. The module is actually in the teaching materials *trim-and-assemble.sh*. Copy it over to your home directory.

3.17 Searching files

You can search the contents of a file using the command *grep*. The *grep* program is very powerful and useful especially when combined with other commands by using the pipe. Navigate to the *Lawrence* directory. Many data files in this directory have a line which says "Volume". Let's see how many times Volume occurs

```
grep Volume *
```

3.18 Finding files

The *find* program can be used to find files based on arbitrary criteria. Navigate to the *data* directory and enter the following command:

```
find . -print
```

This prints the name of every file or directory, recursively, starting from the current directory. Let's exclude all of the directories:

```
find . -type f -print
```

This tells *find* to locate only files. Now try this command:

```
find . -type f -name "*1*"
```

CHAPTER 4

Session 4: data analysis in IPython Notebook

Copy `rsem-final` out of the biology bootcamp shared directory into your home directory ('Home').

At the command line, now do

```
cd ~/rsem-final && ipython notebook --pylab=inline
```

We'll work through a new notebook first, and then go through 'rsem-ebseq'.

Additional IPython resources:

- The ipynb site: http://ipython.org/notebook.html
- A gallery of interesting notebooks: https://github.com/ipython/ipython/wiki/A-gallery-of-interesting-IPython-Notebooks
- The matplotlib gallery: http://matplotlib.org/gallery.html

Note that you can use '%loadpy' in IPython Notebook to grab code from online and import it into your notebook automagically.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search