# labibi Documentation

***Release 0.1***

**C. Titus Brown**

November 27, 2012

# CONTENTS

NOTE: For real-time help either before or during the workshop, you are welcome to try posting in our chat room: http://j.mp/SWCchat

`instructors`

CHAPTER

ONE

# WORKSHOP MATERIALS

## 1.1 Installation and setup instructions

Please try to accomplish all of the downloading before Thursday morning – Thanks!

### 1.1.1 Accounts and software install

1. Sign up for a (free) account on Github: http://github.com/.
2. Download and install Anaconda CE.
3. Download and install git.

### 1.1.2 Virtual Machine stuff

These instructions are especially important if you're running Windows.

1. Download and install VirtualBox on your laptop

   (click the 'x86/amd64' blue link next to your platform).

2. Download this very large (2 GB) file to your laptop. (NOTE: from the Scripps network, this link will go much faster.)
3. Run VirtualBox and import the virtual machines in the very large file. To do this, go to File -> Import Appliance, browse to the 2 GB file you just downloaded, and click 'OK' after all the prompts.

## 1.2 Day 1 – the shell, and basic programming

### 1.2.1 Introductions and Getting Started

Intro:

- Instructors: Tracy Teal, Titus Brown
- TAs: Qingpeng Zhang, Cait Pickens
- Where are the materials? Here! http://swc-scripps.idyll.org/
- Online chat? Here! http://j.mp/SWCchat
- **Comments? Questions? Fixes?**

  - You can add comments at the bottom of each page.
  - You can also *edit* each page on github, and suggest them as changes.

- The schedule is adaptable! Day 2 will be adjusted to how Day 1 goes, plus some.
- Everything we talk about will be available through this Web site, one way or another...
- Ask questions!
- Watch for the cliffs: everything is going along fine and whups, that's a big step...

Getting started:

1. Please partner with someone that you know and has similar challenges.
2. Please send to Tracy and Titus (tnt@idyll.org) a ~one paragraph description of a specific research problem that you are having. For example, "I am working on a marine organism and we got 5000 genes from sequencing and I need to figure out what their matches are in another organism and I want to automate the process."

### 1.2.2 The Shell

**Updated and presented by: Tracy Teal**

**Based on materials originally developed by: Milad Fatenejad, Katy Huff, Jonathon Dursi, and Sasha Wood**

#### What is 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.

Use a browser to open the tutorial on github and type in the URL:

    github.com/swcarpentry/2012-11-scripps

Click on the directory named *1-Shell*.

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.

There are many reasons to learn about the shell. In my opinion, the most important reasons are that:

1. It is very common to encounter the shell and command-line-interfaces in scientific computing, so you will probably have to learn it eventually
2. The shell is a really powerful way of interacting with your computer. GUIs and the shell are complementary - by knowing both you will greatly expand the range of tasks you can accomplish with your computer. You will also be able to perform many tasks more efficiently.

The shell is just a program and there are many different shell programs that have been developed. The most common shell (and the one we will use) is called the Bourne-Again SHell (bash). Even if bash is not the default shell, it usually installed on most systems and can be started by typing *bash* in the terminal. Many commands, especially a lot of the basic ones, work across the various shells but many things are different. I recommend sticking with bash and learning it well.

To open a terminal, just single click on the "Terminal" icon on the Desktop.

### The Example: Manipulating Experimental Data Files

We will spend most of our time learning about the basics of the shell by manipulating some experimental data from a set of hearing tests. To get the data for this test, you will need internet access. Just enter the command:

```
git clone https://github.com/swcarpentry/2012-11-scripps.git
```

This will grab all of the data needed for this workshop from the internet.

### Let's get started

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

#### 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 really 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 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 we are in the */home/username* directory. This is the home directory for the *username* user. That is our user name. You can always find out your user name by entering the command *whoami*.

**File Types**

When you enter the *ls* command lists the contents of the current directory. There are several items in the home directory, notice that they are all colored blue. This tells us that all of these items are directories as opposed to files.

Lets 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. It is colored white meaning that it is a file, as opposed to a directory. The *touch* command just creates an empty file.

Some terminals will not color the directory entries in this very convenient way. In those terminals, use *ls -F* instead of *ls*. The *-F* argument modifies the results so that a slash is placed at the end of directories. If the file is *executable* meaning that it can be run like a program, then a star fill be placed of the file name.

You can also use the command *ls -l* to see whether items in a directory are files or directories. *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.

To get the size of the files in human readable format add a -h. This is very nice when you're working with big files and get tired of counting all the zeros.

Also, some files are hidden. They usually start with a . If you want to see those files, you can add a -a.

So, a standard 'ls' command might be 'ls -alFh' or 'ls -lFh'

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.

**Changing Directories**

Now, let's move to a different directory. The command *cd* (change directory) is used to move around. Let's move into the *2012-11-scripps* directory. Enter the following command:

```
cd 2012-11-scripps
```

Now use the *ls* command to see what is inside this directory. You will see that there is an entry which is green. This means that this is an executable. If you use *ls -F* you will see that this file ends with a star.

This directory contains all of the material for this boot camp. Now move to the directory containing the data for the shell tutorial:

```
cd 1-Shell
```

If you enter the *cd* command by itself, you will return to the home directory. Try this, and then navigate back to the *1-Shell* directory.

### Arguments

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.

**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). You can always find the directory you are in using the *pwd* command. However, you can also give *ls* the names of other directories to view. Navigate to the home directory if you are not already there. Then enter the command:

```
ls 2012-11-scripps
```

This will list the contents of the *2012-11-scripps* directory without you having to navigate there. Now enter:

```
ls 2012-11-scripps/1-Shell
```

This prints the contents of *1-Shell*. The *cd* command works in a similar way. Try entering:

```
cd 2012-11-scripps/1-Shell
```

and you will jump directly to *1-Shell* without having to go through the intermediate directory.

### Full vs. Relative Paths

The *cd* command takes an argument which is the directory name. Directories can be specified using either a *relative* path 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. Navigate to the home directory. Now, enter the *pwd* command and you should see:

```
/home/username
```

which is the full name of your home directory. This tells you that you are in a directory called *username*, which sits inside a directory called *home* which sits inside the very top directory in the hierarchy. The very top of the hierarchy is a directory called */* which is usually referred to as the *root directory*. So, to summarize: *username* is a directory in *home* which is a directory in */*.

Now enter the following command:

```
cd /home/username/2012-11-scripps/1-Shell
```

This jumps to *1-Shell*. Now go back to the home directory. We saw earlier that the command:

```
cd 2012-11-scripps/1-Shell
```

had the same effect - it took us to the *1-Shell* directory. But, instead of specifying the full path (*/home/username/2012-11-scripps/1-Shell*), 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?

### 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 *1-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 */home/username/2012-11-scripps*. You can chain these together, so:

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

prints the contents of */home/username* which is 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 /home/username* all do exactly the same thing. These shortcuts are not necessary, they are provided for your convenience.

#### Our data set: Cochlear Implants

A cochlear implant is a small electronic device that is surgically implanted in the inner ear to give deaf people a sense of hearing. More than a quarter of a million people have them, but there is still no widely-accepted benchmark to

measure their effectiveness. In order to establish a baseline for such a benchmark, a researcher got teenagers with CIs to listen to audio files on their computer and report:

1. the quietest sound they could hear
2. the lowest and highest tones they could hear
3. the narrowest range of frequencies they could discriminate

To participate, subjects came to a laboratory and one of the lab techs played an audio sample, and recorded their data - when they first heard the sound, or first heard a difference in the sound. Each set of test results were written out to a text file, one set per file. Each participant has a unique subject ID, and a made-up subject name. Each experiment has a unique experiment ID. The experiment has collected 351 files so far.

The data is a bit of a mess! There are inconsistent file names, there are extraneous "NOTES" files that we'd like to get rid of, and the data is spread across many directories. We are going to use shell commands to get this data into shape. By the end we would like to:

1. Put all of the data into one directory called "alldata"
2. Have all of the data files in there, and ensure that every file has a ".txt" extension
3. Get rid of the extraneous "NOTES" files

If we can get through this example in the available time, we will move onto more advanced shell topics...

**Wild cards**

Navigate to the *~/2012-11-scripps/1-Shell/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 see all of the contents of a given directory. 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 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.

---

**Short Exercise**

Do each of the following using a single *ls* command without navigating to a different directory.

1. List all of the files in */bin* that contain the letter *a*
2. List all of the files in */bin* that contain the letter *a* or the letter *b*
3. List all of the files in */bin* that contain the letter *a* AND the letter *b*

**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 2<tab>
```

The shell will fill in the rest of the directory name for *2012-11-scripps*. Now enter:

```
ls i<tab><tab>
```

When you hit the first tab, nothing happens. The reason is that there are multiple directories in the home directory which start with i. 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.

^-R will do a reverse-search through your command history. This is very useful.

#### 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. More on environment variables later. 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 *1-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 */home/username/2012-11-scripps/1-Shell* 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:

```
/home/username/2012-11-scripps/1-Shell/hello
```

Or by entering:

```
../1-Shell/hello
```

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

### 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. 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).

If there's a bunch of things on your screen and you want to clean it up a bit, you can type 'clear' and that will clear your screen so you have a shiny prompt at the top of your screen.

---

**Short Exercises**

1. Print out the contents of the *~/2012-11-scripps/1-Shell/dictionary.txt* file. What does this file contain?
2. Without changing directories, (you should still be in *1-Shell*), use one short command to print the contents of all of the files in the /home/username/2012-11-scripps/1-Shell/data/THOMAS directory.

---

*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 ~/2012-11-scripps/1-Shell/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. Also, the arrow keys work for navigating up and down. Finally, hit "q" to quit.

*less* also gives you a way of searching through files. Just hit the "/" key to begin a search. Enter the name of 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. You can also type "n" for it to go to the next one it finds. *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!

---

**Short Exercise**

Use the commands we've learned so far to figure out how to search in reverse while using *less*.

---

### Redirection

Let's turn to the experimental data from the hearing tests that we began with. This data is located in the *~/2012-11-scripps/1-Shell/data* directory. Each subdirectory corresponds to a particular participant in the study. Navigate to the *Bert* subdirectory in *data*. There are a bunch of text files which contain experimental data results. Lets print them all:

```
cat au*
```

Now enter the following command:

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

This tells the shell to take the output from the *cat au** 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 what ever 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.

---

**Short Exercise**

Use >>, to append the contents of all of the files which contain the number 4 in the directory:

```
/home/username/2012-11-scripps/1-Shell/data/gerdal
```

to the existing *all_data* file. Thus, when you are done *all_data* should contain all of the experiment data from Bert and any experimental data file from gerdal that contains the number 4.

---

### Creating, moving, copying, and removing

We've created a file called *all_data* using the redirection operator >. This is critical file so we have 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
```

Now the 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.

---

**Short Exercise**

Do the following:

1. Rename the *all_data_IMPORTANT* file to *all_data*.
2. Create a directory in the *data* directory called *foo*
3. Then, copy the *all_data* file into *foo*

---

By default, *rm*, will NOT delete directories. You can tell *rm* to delete a directory using the *-r* option. Enter the following command:

```
rm -r foo
```

### 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 Bert/* gerdal/*4*
```

For each of the files indicated, *wc* has printed a line with three numbers. The first is the number of lines in that file. The second is the number of words. Finally, the total number of characters is indicated. The final line contains this information summed over all of the files. Thus, there were 10445 characters in total.

Remember that the *Bert/** and *gerdal/*4** 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. Thus, the size of the file in bytes should also be 10445. 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.

---

**Short Exercise**

Figure out how to get *wc* to print just the number of lines in *all_data*.

---

#### The awesome power of the Pipe

Suppose I wanted to only see the total number of character, words, and lines across the files *Bert/** and *gerdal/*4**. 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. Thus, I have wasted a precious 7062 bytes of hard disk space. 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 Bert/* gerdal/*4** and send it into the *tail -n 1* command. The | character (called pipe) is used for this purpose. Enter the following command:

```
wc Bert/* gerdal/Data0559 | 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 is the lines:

```
are
good
```

printed back at you. 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 is 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:: |, >, >>.

**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. There are many such programs, the easiest one which is installed on almost all systems is called *nano*. Navigate to */tmp* and enter the following command:

```
nano toBeSorted
```

Now enter the four names as shown above. When you are done, press CONTROL+O to write out the file. Press enter to use the file name *toBeSorted*. Then press CONTROL+x to exit *nano*.

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

```
sort toBeSorted
```

Notice that the names are now printed in alphabetical order.

---

**Short Exercise**

Use the *echo* command and the append operator, >>, to append your name to the file, then sort it.

---

Let's navigate back to *~/2012-11-scripps/1-Shell/data*. You should still have the *all_data* file hanging around here. Enter the following command:

```
wc Bert/* | 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 *Bert* 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.

**Short Exercise**

Combine the *wc*, *sort*, *head* and *tail* commands so that only the *wc* information for the largest file is listed

Hint: To print the smallest file, use:

```
wc Bert/* | sort -k 3 -n | head -n 1
```

---

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*. We'll use *nano* to create this file. Navigate to the *data* directory, then:

```
nano smallest
```

Then enter the following text:

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

Now, *cd* into the *Bert* 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. 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 ../smallest
```

You will see that the file name is green. Congratulations, you just created your first shell script!

### 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 *Bert* directory. Every data file in this directory has a line which says "Range". The range represents the smallest frequency range that can be discriminated. Lets list all of the ranges from the tests that Bert conducted:

```
grep Range *
```

---

**Short Exercise**

Create an executable script called *smallestrange* in the *data* directory, that is similar to the *smallest* script, but prints the file containing the file with the smallest Range. Use the commands *grep*, *sort*, and *tail* to do this.

---

### 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 these commands:

```
find . -type f -name "*1*"
find . -type f -name "*1*" -or -name "*2*" -print
find . -type f -name "*1*" -and -name "*2*" -print
```

The *find* command can acquire a list of files and perform some operation on each file. Try this command out:

```
find . -type f -exec grep Volume {} \;
```

This command finds every file starting from .. Then it searches each file for a line which contains the word "Volume". The *{}* refers to the name of each file. The trailing *;* is used to terminate the command. This command is slow, because it is calling a new instance of *grep* for each item the *find* returns.

A faster way to do this is to use the *xargs* command:

```
find . -type f -print | xargs grep Volume
```

*find* generates a list of all the files we are interested in, then we pipe them to *xargs*. *xargs* takes the items given to it and passes them as arguments to *grep*. *xargs* generally only creates a single instance of *grep* (or whatever program it is running).

---

**Short Exercise**

Navigate to the *data* directory. Use one find command to perform each of the operations listed below (except number 2, which does not require a find command):

1. Find any file whose name is "NOTES" within *data* and delete it
2. Create a new directory called *cleaneddata*
3. Move all of the files within *data* to the *cleaneddata* directory
4. Rename all of the files to ensure that they end in *.txt* (note:: it is ok for the file name to end in *.txt.txt*

Hint: If you make a mistake and need to start over just do the following:

1. Navigate to the *1-Shell* directory
2. Delete the *data* directory
3. Enter the command: *git checkout – data* You should see that the data directory has reappeared in its original state

**BONUS**

Redo exercise 4, except rename only the files which do not already end in *.txt*. You will have to use the *man* command to figure out how to search for files which do not match a certain name.

---

## Bonus

**backtick, xargs**: Example find all files with certain text

**alias** -> rm -i

**variables** -> use a path example

**.bashrc**

**du**

**ln**

**ssh and scp**

**Regular Expressions**

**Permissions**

**Chaining commands together**

### 1.2.3 Introductory Python

We'll be looking at the following IPython Notebooks, all of which are under the python/ directory of the git repository:

Scientific Python basics: python-full.ipynb

Plotting with matplotlib: matplotlib-full.ipynb

Reading and writing files: readwrite-full.ipynb

—

Loading and plotting files: loading-and-plotting-data.ipynb

make_figure.py script

End of day data Analysis mix: end-of-day-data-analysis.ipynb

#### Running IPython Notebook

Honestly, the hardest part is just getting things running :(. Pick whichever one of the solutions below works...

##### Running the notebook on Mac OS X using Anaconda CE

Once you get the Anaconda CE file (see *Installation and setup instructions* for download links!), go run it *at the command line* – double clicking it doesn't seem to work :(.

This should be as simple as opening up a Terminal window and typing:

```
bash ~/Downloads/AnacondaCE-1.2.0-macosx.sh
```

and answering all the questions with the defaults ('yes' where appropriate).

Then, once it's all done installing, cd to the git directory that you downloaded earlier, cd into the python/ subdirectory, and type

```
~/anaconda/bin/ipython notebook --pylab=inline
```

Your Web browser should pop up. Tada!

##### Running the notebooks in the Virtual Box virtual machine

Start up your virtual machine (see *Installation and setup instructions* for instructions on installing VirtualBox), and then click 'Terminal'. Inside of terminal, run the following commands:

```
git clone https://github.com/swcarpentry/2012-11-scripps
ls
cd 2012-11-scripps/
ls
cd python
ls
./run-in-vm.sh
```

(You can copy on your Web browser, and then paste into the Terminal in your VM with 'ctrl-shift-V'.)

This will start up a Firefox browser pointing at IPython Notebook

#### Extra – upgrading ipython notebook

You can upgrade ipython notebook to a newer version like so. Type:

```
sudo apt-get install python-pip
sudo pip install --upgrade ipython
```

This will take a few minutes to do, because it has to download some files...

#### IPython Notebook, a brief intro

The IPython Notebook (ipynb for short) is a simple notebook interface to Python that lets you interactively run Python code and view figures and graphics. You can load, save, and download notebooks as a record of your research as well as for interaction with colleagues.

The main thing you need to know about IPython is that to execute code in a cell, you hit Shift-ENTER.

---

**The shell - focus: automating stuff**

- files & directories;
- creating things;
- pipes and filters;
- loops;
- scripts;

**Intro Python - focus: data processing**

- Basic operations;
- For loops & if statements;
- Reading, transforming, and writing data;
- Organizing code in functions and modules;
- Scripting with Python;

## 1.3 Day 2 – many miscellaneous topics

You should all start reading XKCD.

### 1.3.1 Python data structures

See Python data structures

### 1.3.2 Installing Python packages; useful Python packages

See `python-packages`

### 1.3.3 Useful UNIX tools

ssh and screen; BLAST.

SSH instructions here

Installing and using command line BLAST

### 1.3.4 Testing

When we say "testing" we really mean *automated testing*. The central problems addressed by testing are *correctness* and *reproducibility*. (While these are linked, they are not the same!)

There are two basic kinds of tests that I'd like to briefly discuss. One kind of test is a *unit test*. The other kind of test is a *regression test*. (There are also many more.)

Unit tests address *small units of code*, like functions. They are used to isolate and nail down and prove the functionality of potentially complicated little functions.

Regression tests address *the overall function of code*, and they are used to make sure that your code is doing the same thing *today* as it was *yesterday*.

I'll show you examples of both, but quickly :).

#### Writing tests

We're going to be using the nose testing framework, which is just a framework that makes it easy to find and execute tests.

Basically, 'nose' creates a command 'nosetests' that finds and runs tests. The idea is that you won't need to register new tests.

A test function looks like this:

```
def test_something():
   # run some code
   # fail loudly or succeed silently
```

See testing-with-nose.ipynb.

#### More reading

For more reading, see:

> http://software-carpentry.org/4_0/test/

and

> http://ivory.idyll.org/articles/nose-intro.html

and also

> http://ivory.idyll.org/blog/software-quality-death-spiral.html

### 1.3.5 Version Control

(An abbreviated version of: http://ged.msu.edu/angus/git-intro.html)

The purpose of version control is to serve as a method for tacking changes to files, which enables lots of things:

1. track changes (wanted and unwanted) in your files.
2. keep track of an entire history of changes.
3. track multiple independent "branches" of work.
4. collaborate sanely.

---

A brief introduction:

1. Web interface and editing files.
2. One-person repositories.
3. One-person repositories with multiple locations.
4. One-person repositories with branches.
5. Collaborations!

### 1.3.6 Pipelines

Operating on more complex files.

Doing stuff with BLAST output.

Other materials:

> The Analyzing Next-Generation Sequencing Data summer course is a good resource for people interested in NGS specifically.
>
> For people interested in learning Python, I got really strong positive recommendations for the MIT OpenCourseware and Khan Academies lectures from a course attendee.

CHAPTER

TWO

# SCRIPPS COMPUTATIONAL INFO

- IT has set up a temporary server for use by class participants. The server name is it-test7.lj.ad.scripps.edu
- iPython Notebook is available at http://it-test7.lj.ad.scripps.edu:8888/ (NOTE: only available from the Scripps network)
- This server is also available for interactive shell use – SSH to username@it-test7.lj.ad.scripps.edu (replacing 'username' with your Scripps username)
- IT also invites users who want to do more computational work to request accounts on garabaldi
- To request an account, please send an email to hpc_ca@scripps.edu
- Account will provide access to two nodes for interactive use, and also the potential of using our high-performance computing (HPC) cluster.
- IT also maintains two general-purpose compute servers: home.scripps.edu, app.scripps.edu
- Both are older machines that currently aren't well supported.
- We suggest you *not* use these servers to use/practice the skills learned in this workshop
- To encourage IT to make these more up-to-date and publicized resources, use it-test7
- Also, sign up for the mailing list of the Computational Biology and Bioinformatics (CBB) Affinity Group
- Web sign up: http://lists.scripps.edu/majorcool.cgi
- Or, send email to majordomo@scripps.edu with 'subscribe cbb' (no quotes) in the message body

CHAPTER

THREE

# HIPCHAT

- For the moment, we will leave up the hipchat room. If you feel like it, post your questions and/or answer others' questions: http://j.mp/SWCchat
- Transcripts
- Wednesday 14th
- Thursday 15th
- Friday 16th

CHAPTER

FOUR

# INDICES AND TABLES

- *genindex*
- *search*

dumb edit