RNA-seq Data Analysis Bootcamp

Stephen D. Turner, Ph.D.
Bioinformatics Core Director

November 10, 2014

Contents

2 RNA-seq: reads to counts 30

3 Introduction to R 37

Preface

This book contains material from a workshop directed toward life scientists with little to no experience with statistical computing or bioinformatics. This will introduce both the Linux/UNIX operating system and the R statistical computing environment, with a focus on a biological application - analyzing RNA-seq data for differentially expressed genes. The first half will introduce basic operation in a UNIX environment, and will cover the first steps in an RNA-seq analysis including QC, alignment, and quantitation. The second half will introduce the R statistical computing environment, and will cover differential gene expression analysis using Bioconductor. At the end of the course or after reading through this book, you will:

1. Know how to provision your own computing resources using Amazon Web Services Elastic Compute Cloud.
2. Be familiar with the UNIX shell, including nagivating the filesystem, creating/examining/removing files, getting help, and batch operations.
3. Know how to align and quantitate gene expression with RNA-seq data.
4. Become familiar with the R statistical computing environment, including data types, variables, array manipulation, functions, data frames, data import/export, visualization, and using packages.
5. Know what packages to use and what steps to take to analyze RNA-seq data for differentially expressed genes.

This book is a PDF version of the online materials available at bioconnector.github.io. Please see the online materials for instructions regarding how to create an Amazon Web Services account and setting up software that will be used in this course. Be sure to stop or terminate any running AWS instances when no longer in use.

This course is sponsored by the Claude Moore Health Sciences Library, and borrows some materials from the Software Carpentry and Data Carpentry projects.

Chapter 1

Introduction to the UNIX shell

Modified from lessons for Data Carpentry with original contributions from Tracy Teal, Paul Wilson, Milad Fatenejad, Sasha Wood and Radhika Khetani.

Learn more:

- http://datacarpentry.org/
- http://software-carpentry.org/

Objectives

- What is the shell?
- How do you access it?
- Connecting to cloud computing resources
- How do you use it?
- Getting around the Unix file system
- looking at files
- manipulating files
- automating tasks
- What is it good for?
- Where are resources where I can learn more? (because the shell is awesome)

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.

Why should you care?

- For 99% of bioinformatics tools, you have to use the shell (the command line). There is no graphical interface.
- The shell gives you *power*. The command line gives you the power to do your work more efficiently and more quickly. When you need to do things tens to hundreds of times, knowing how to use the shell is transformative.
- You have to use the shell to connect to remote computers.

Automation

Have 10,000,000 files to rename, read in, analyze, and visualize? It's easy to automate things with the shell.

(image removed from PDF version)

How to access the shell

You access the shell through a program called a Terminal. We're going to use a terminal to connect to the shell of a remote computer. The terminal program is built-in on Mac and Linux. For Windows, you'll have to download a separate program called a terminal emulator that will allow you to connect to remote computers.

Mac

On Mac the shell is available through Terminal:

Applications -> Utilities -> Terminal

Go ahead and drag the Terminal application to your Dock for easy access.

Windows

On Windows machines we'll be using a terminal emulator called PuTTY.

More resources on the shell

Cheat sheets:

- http://fosswire.com/post/2007/08/unixlinux-command-cheat-sheet/

Web sites where you can see what the different components of a shell command are doing:

- explainshell.com
- commandlinefu.com

Connecting to cloud computing resources

Cloud computing means different things to different people. There's Software-as-a-Service (Saas) like Gmail, Facebook, etc., that most people think of when they think of cloud computing. Here, we're strictly talking about Infrastructure-as-a-Service (IaaS). That is, we're using (or *renting*) computing infrastructure that we don't *own*.

We'll cover in class how to start and connect to an Amazon Web Services EC2 instance. ***It's essential that you stop or terminate any running AWS instances after you're done with them***.

Shell basics

Get some data files to work with

We will spend most of our time learning about the basics of the shell by manipulating some experimental data. If you aren't using the virtual machine image you'll need to download the data for the tutorial. For this you'll need internet access, because you're going to get it off the web.

Open the shell and type the commands:

```
cd
git clone https://github.com/bioconnector/workshops.git
```

This command will grab all of the data needed for this workshop. It's using something called git that's used for version control, but we won't talk about that here.

If you're already using the machine image you might need to update things.

```
cd workshops
git pull
cd
```

Moving around and listing files

We'll start out by moving around the file system and listing files. Today we're going to go through using the command line. These commands are in the README.md file and on your handout (fixme).

Let's go in to that directory we just downloaded:

```
cd workshops
```

`cd` stands for 'change directory'. A *directory* is the same thing as a *folder*. We just call folders *directories* in the Linux/UNIX world. Here we just entered the workshops directory. If we were doing this on a graphical shell we would have double-clicked on a little folder icon. It's the same idea.

In this directory, there should be some things we just downloaded. Let's check. Type:

```
ls
```

`ls` stands for 'list' and it lists the contents of a directory. This woudld be what you would see in the folder if you were doing this graphically on a desktop.

Blue things are directories, white things are files.

Now, let's go look in the 'data' directory in the 'shell' lesson. It's nested a few directories deep. To get to it, let's enter and list each directory like so:

```
cd lessons
ls
cd shell
ls
cd data
ls
```

In there, all mixed up together are regular files, directories, and an executable program. If we want to know which is which, we can type:

```
ls -F
```

Things with a `/` after it is a directory.
Things with a `*` after them are programs.
It there's nothing there it's a regular file.

You can also use the command:

```
ls -l
```

to see whether items in a directory are files or directories. It gives a lot more information too, such as the size of the file, who owns the file, etc.

So, we can see that we have several files, directories and a program. Great!

Options

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

Most commonly used shell programs have a manual. Let's open the manual page for `ls`.

You can access the manual using the `man` program.

```
man ls
```

Space key goes forward
Or use the arrow keys to scroll up and down.
When you are done reading, just hit `q` to quit.

Programs that are run from the shell can get extremely complicated. To see an example, open up the manual page for the `find` program. No one can possibly learn all of these arguments, of course. So you will probably find yourself referring back to the manual page frequently.

The Unix directory file structure (a.k.a. where am I?)

As you've already just seen, you can move around in different directories or folders at the command line.

When you're working with bioinformatics programs, you're working with your data and it's key to be able to have that data in the right place and make sure the program has access to the data. Many of the problems people run in to with command line bioinformatics programs is not having the data in the place the program expects it to be.

Let's draw out what we just did, and some of the other files and folders we could have clicked on.

This is called a hierarchical file system structure, like an upside down tree with root (/) at the base that looks like this.

(image removed from PDF version)

That (/) at the base is often also called the 'top' level.

When you are working at your computer or log in to a remote computer, you are on one of the branches of that tree, your home directory: **/home/username**.

If we type `cd` by itself:

```
cd
```

This puts you in your home directory. That's **/home/username**

EXERCISE

- Using `cd` and `ls`, go in to the ‘workshops/shell/data’ directory and list its contents.
- How many files, how many directories and how many programs are there?

Where am I?

Let’s also check to see where we are. Sometimes when we’re wandering around in the file system, it’s easy to lose track of where we are and get lost.

If you want to know what directory you’re currently in, type:

```
pwd
```

This stands for ‘print working directory’. That’s the directory you’re currently working in, and it’s “printed” to the screen.

What if we want to move back up and out of the `data` directory? To go ‘back up a level’ we need to use `..`

```
cd ..
```

Now do `ls` and `pwd`. See now that we went back up in to the ‘shell’ directory. `..` means “the directory above,” or “the parent directory.”

EXERCISE

Let’s go on a file hunt. Move around in the “shell/data/hidden” directory and try to find the file “youfoundit.txt.”

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 using `cd` by itself, then `ls` the contents of the "workshops/lessons/shell" directory:

```
cd
ls workshops/lessons/shell
```

This listed the contents of workshops/lessons/shell without navigating there.

The `cd` command works the same way. Try entering:

```
cd
ls workshops/lessons/shell/data/hidden
```

and you will jump directly to `hidden` without having to go through the intermediate directories.

EXERCISE

Try finding the 'anotherfile.txt' file without changing directories.

HUGE Shortcut: 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 file or directory, then hit the tab key, the UNIX shell will try to fill in the rest of the directory name. For example, enter:

```
cd
cd w<tab>
```

The shell will fill in the rest of the directory name for "workshops". Now go to `workshops/lessons/rnaseq-1day/data`

```
cd
cd wo<tab>le<tab>rna<tab>da<tab>
```

Now type `ls uvb` and hit tab twice.

```
ls uvb<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 `uvb`. 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.

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. 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/bioinfo/workshops/lessons/shell/data/hidden
```

This jumps to `hidden`. Now go back to the home directory (`cd`). We saw earlier that the command:

```
cd workshops/lessons/shell/data/hidden
```

had the same effect - it took us to the `hidden` directory. But, instead of specifying the full path, 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.

A relative path is like getting directions from someone on the street. They tell you to "go right at the Stop sign, and then turn left on Main Street". That works great if you're standing there together, but not so well if you're trying to tell someone how to get there from another country. A full path is like GPS coordinates. It tells you exactly where something is no matter where you are right now.

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.

Over time, it will become easier for you to keep a mental note of the structure of the directories that you are using and how to quickly navigate them.

EXERCISE

- List the contents of the /bin directory. Do you see anything familiar in there?

Saving time with shortcuts and wild cards

Shortcuts

There are some shortcuts which you should know about. Dealing with the home directory is very common. In the shell the tilde character, `~`, is a shortcut for your home directory.

```
ls ~
```

Try it even when you're not in your home directory:

```
cd
cd workshops
```

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.

```
ls ..
```

prints the contents of the `/home/username/`. You can chain these together, so:

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

prints the contents of `/home/`. 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, while you are in the `workshops` directory, the commands `ls ~`, `ls ../`, and `ls /home/bioinfo` all do exactly the same thing.

Our data set: FASTQ files

In this example we're going to use some RNA-seq data that we'll actually analyze later on. The data come from RNA-seq done on skin cells treated with ultraviolet light versus controls (paper; data).

We did sequencing, and what we have here are just reads that came from a 100 MB region on chromosome 4. We want to be able to look at these files and do some things with them.

Wild cards

Navigate to the `workshops/lessons/rnaseq-1day/data` directory from your home. This directory contains our FASTQ files and some other ones we'll need for analyses. If we type `ls`, we will see that there are several files in there. Some of the end with .fastq.gz. These are compressed fastq files.

The `*` character is a shortcut for "everything". Thus, if you enter `ls *`, you will see all of the contents of a given directory.

```
ls *fastq.gz
```

This lists every file that ends with a `fastq.gz`. This command:

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

Lists every file in `/usr/bin` that ends in the characters `.sh`.

If we wanted to list just the controls or just the uvb-treated samples, we could do this:

```
ls ctl*
ls uvb*
```

So how does this actually work? Well…when the shell sees a word that contains the `*` character, it automatically looks for filenames that match the given pattern. In this case, it identified 3 such files. Then, it replaced the `ctl*` with the list of files, separated by spaces.

Because we'll use these files later, let's go ahead and extract or uncompress these files. To compres a text file we would use `gzip`. To extract, we'll use `gunzip`. Let's extract all the files that end in `.gz`. This will extract both the fastq files and the counts.txt file as well:

```
gunzip *.gz
```

Now, time for some more practice with wildcards.

EXERCISE

Do each of the following using a single `ls` command without navigating to a different directory. - List all of the files in `/bin` that start with the letter 'c - List all of the files in `/bin` that contain the letter 'a' - List all of the files in `/bin` that end with the letter 'o'

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.

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

You can also review your recent commands with the `history` command to see a numbered list of commands you've run.

```
history
```

You can reuse one of these commands directly by referring to the number of that command. If your history looked like this:

```
259  ls *
260  ls /usr/bin/*.sh
261  ls ctl*
```

then you could repeat command `#260` by simply entering:

```
!260
```

EXERCISE

Find the line number in your history for the last exercise (listing files in /bin) and reissue that command.

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 print out all of the contents using the program `cat`. Enter the following command:

```
cat ctl1.fastq
```

This prints out the contents of the `ctl1.fastq` file.

EXERCISE

- Print out the contents of the `workshops/lessons/rnaseq-1day/data/coldata.csv` file. What does this file contain?
- Without changing directories, use one short command to print the contents of all of the files in the `~/workshops/posts/` directory.

Make sure we're in the right place for the next set of the lessons. We want to be in the rnaseq data directory (`workshops/lessons/rnaseq-1day/data`). Check if you're there with `pwd` and if not navigate there. One way to do that would be

```
cd ~/workshops/lessons/rnaseq-1day/data
```

`cat` is a terrific program, but when the file is really big, it can be annoying to use.

The program, `less`, is useful when files are big and you want to be able to scroll through them.

```
less ctl1.fastq
```

`less` opens the file, and lets you navigate through it. The commands are identical to the `man` program.

Some commands in `less`

Key	Action
space	go forward
b	go backwards
g	go to the beginning
G	go to the end
q	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 `ctl1.fasta` for the the sequence "GATTACA". 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 "GATTACA", `less` will not find it. You need to go to the beginning of the file with the `g` key in less, and start the search from there.

Remember, the `man` program actually uses `less` internally and therefore uses the same commands, so you can search documentation using "/" as well!

There's another way that we can look at files, and in this case, just look at part of them. This can be particularly useful if we just want to see the beginning or end of the file, or see how it's formatted.

The commands `head` and `tail` let you look at the beginning and end of a file respectively.

```
head ctl1.fastq
tail ctl1.fastq
```

The `-n` option to either of these commands can be used to print the first or last `n` lines of a file. If we want to see the first read in the file, try this:

```
head -n 4 ctl1.fastq
tail -n 4 ctl1.fastq
```

Searching files

We showed a little how to search within a file using `less`.

We can search within files without even opening them, using `grep`. Grep is a command-line utility for searching plain-text data sets for lines matching a string or regular expression.

Let's find all the lines in ctl1.fastq that contain the sequence motif "GATTACA."

```
grep GATTACA ctl1.fastq
```

We get back just the sequence line, but what if we wanted all four lines, the whole part of that FASTQ sequence, back instead.

```
grep -B 1 -A 2 GATTACA ctl1.fastq
```

The `-A` flag stands for "after match" so it's returning the line that matches plus the two after it. The `-B` flag returns that number of lines before the match, so that's returning the fastq header.

EXERCISE

Search for the sequence 'GATTTTTACA' (GATTACA with 5 T's instead of 2) in ctl1.fastq file and in the output have the sequence name. The output should look like this:

```
@SRR1145047.5880759
AAGCTAAAAAAAAAATGGATGTTTCAGTTAAATGTTTTAAAGAGGTACAGATTTTTACAAGGACATAATATAAG
```

Next, search for that sequence in all the FASTQ files.

Redirection & Pipes

We're excited we have all these sequences that we care about that we just got from the FASTQ files. Perhaps that was some really important motif that is going to help us answer some important question. But all those sequences just went whizzing by with grep. How can we capture them?

We can do that with something called "redirection". The idea is that we're redirecting the output to the terminal (all the stuff that went whizzing by) to something else. In this case, we want to write it to a file, so that we can look at it later.

We do this with: `>`.

Let's try it out and put all the sequences that contain 'GATTACA' from all the files in to another file called "gattaca-reads.txt".

```
grep GATTACA *.fastq > gattaca-reads.txt
```

The prompt should sit there a little bit, and then it should look like nothing happened. But type `ls`. You should have a new file called "gattaca-reads.txt". Take a look at it and see if it has what you think it should.

```
ls
less gattaca-reads.txt
```

There's another useful redirection command that we're going to show, and that's called the pipe: `|`. It's probably not a key on your keyboard you use very much. What `|` does is take the output that scrolling by on the terminal and then can run it through another command. When it was all whizzing by before, we wished we could just slow it down and look at it, like we can with `less`. Well it turns out that we can! We pipe the `grep` command through `less`

The pipe '|' takes the output of the first thing and then puts it in to the second part

```
grep GATTACA *.fastq | less
```

Now we can use the arrows to scroll up and down and use `q` to get out.

There's another command called `wc`. `wc` stands for`word count`. It counts the number of lines or characters. So, we can use it to count the number of lines we're getting back from our `grep` command. And that will tell us how many sequences we're finding with that motif across all the files.

```
grep GATTACA *.fastq | wc
```

That tells us the number of lines, words and characters in the file. If we just want the number of lines, we can use the `-l` flag for `lines`.

```
grep GATTACA *.fastq | wc -l
```

You could have piped the output to a file like you did the first time and ran `wc` on that file:

```
wc gattaca-reads.txt
```

But using the pipe you didn't need to create the intermediate file. Pipes are really powerful for stringing together these different commands, so you can do whatever you need to do.

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.

For example, draw a cow saying the largest word in the english language that contains the word "purple":

```
# Read in the system's dictionary.
cat /usr/share/dict/words
# grep for "purple"
cat /usr/share/dict/words | grep purple
# count letters in each word
cat /usr/share/dict/words | grep purple | awk '{print length($1), $1}'
# numeric sort
cat /usr/share/dict/words | grep purple | awk '{print length($1), $1}' | sort -n
# take the last line
cat /usr/share/dict/words | grep purple | awk '{print length($1), $1}' | sort -n | tail
# cut out the second field (-f 2) if the file were delimited by a space (-d " ")
cat /usr/share/dict/words | grep purple | awk '{print length($1), $1}' | sort -n | tail
# make the cow say it
cat /usr/share/dict/words | grep purple | awk '{print length($1), $1}' | sort -n | tail
```

This program piped together 7 UNIX programs (`cat`, `grep`, `awk`, `sort`, `tail`, `cut`, `cowsay`), each of which does a very small job on its own, and strung them together to do something pretty powerful (as silly as it may be).

```
 _____________
< unimpurpled >
 -------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Creating, moving, copying, and removing

Now we can move around in the file structure, look at files, search files, redirect. But what if we want to do normal things like copy files or move them around or get rid of them. Sure, if we had a desktop, we could do most of these things without the command line, but what fun would that be?! Besides it's usually faster to do it at the command line, or you'll be on a remote server like Amazon where you won't have another option.

The `coldata.csv` file tells us which sample names are which treatment (in this example the filenames are pretty informative, but when they come off the sequencer usually they won't be). This is a really important file, so we want to make a copy so we don't lose it.

Lets copy the file using the `cp` command. The `cp` command backs up the file. Navigate to the `rnaseq-1day/data` directory and enter:

```
cp coldata.csv coldata.csv-backup
```

Now `coldata.csv-backup` has been created as a copy of `coldata.csv`.

Let's make a `backup` directory where we can put this file.

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

```
mkdir backup
```

We can now move our backed up file in to this directory. We can move files around using the command `mv`. Enter this command:

```
mv coldata.csv-backup backup
```

This moves `coldata.csv-backup` into the directory `backup/` or the full path would be `~/workshops/lessons/rnaseq-1day/data/backup/coldata.csv-backup`

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

```
ls
mv coldata.csv coldata-IMPORTANT.csv
ls
```

Now let's delete the backup copy:

```
ls
ls backup
rm backup/coldata.csv-backup
ls
ls backup
```

The `rm` file removes the file. Be careful with this command. It doesn't just nicely put the files in the Trash. They're really gone.

EXERCISE

Do the following:

1. Rename the `coldata-IMPORTANT.csv` file back to `coldata.csv`.
2. Create a new directory in the rnaseq-1day directory called `new`.
3. Then, copy the `coldata.csv` file into `new`

By default, `rm`, will NOT delete directories. You can tell `rm` to delete a directory and everything in it *recursively* using the `-r` option. Let's delete that `new` directory we just made. Enter the following command:

```
rm -r new
```

Be careful with this.

Writing files

We've been able to do a lot of work with files that already exist, but what if we want to write our own files. Obviously, we're not going to type in a FASTA file, but there are a lot of reasons we'll want to edit a file.

To write in files, we're going to use the program `nano`. We're going to create a file that contains the favorite grep command so you can remember it for later. We'll name this file 'awesome.sh'.

```
nano awesome.sh
```

Enter the following into the file:

```
grep -B 1 -A 2 GATTACA *.fastq
```

At the bottom of nano, you see the "^X Exit". That means that we use Ctrl-X to exit. Type `Ctrl-X`. It will ask if you want to save it. Type `y` for yes. Then it asks if you want that file name. Hit 'Enter'.

Now you've written a file. You can take a look at it with less or cat, or open it up again and edit it.

EXERCISE

Open 'awesome.sh' and add "echo AWESOME!" (no quotes) after the grep command and save the file.

We're going to come back and use this file in just a bit.

Running programs

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 pwd
```

Will return “/bin/pwd”. Thus, we can see that `pwd` 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.

Remember that file where we wrote our favorite grep command in there? Since we like it so much, we might want to run it again, or even all the time. Instead of writing it out every time, we can just run it as a script. Let’s try to run that script:

```
awesome.sh
```

You should get an error saying that awesome.sh cannot be found. That is because the directory `~/workshops/lessons/rnaseq-1day/data` is not in the`PATH`. You can try again to run the `awesome.sh` program by entering:

```
./awesome.sh
```

Alas, we get `-bash: ./awesome.sh: Permission denied`. This is because we haven’t told the computer that it’s a program that can be executed. To do that we have to make it ‘executable’. We do this by changing its mode. The command for that is `chmod` - change mode. We’re going to change the mode of this file, so that it’s executable and the computer knows it’s OK to run it as a program.

To run a program, you have to set the right permissions, make it executable rather than just a text file.

```
chmod +x awesome.sh
ls -l
```

Now we can run the program

```
./awesome.sh
```

Now you should have seen some output, and of course, it's AWESOME! Congratulations, you just created your first shell script!

EXERCISE

1. In the `data` directory, use `nano` to write a script called `quickpeek.sh` that:
    - Runs `head` on all the fastq files in the current directory
    - Runs `wc` on all the fastq files
    - `echo`s "Done!" when finished.
2. Make the program executable.
3. Run the program.

Simple parallel computing with `find` and `parallel`

We've learned how to do a few things already using wildcards. For instance, we extracted all the fastq files with

```
gunzip *.fastq
```

which the shell interpreted the same as:

```
gunzip ctl1.fastq  ctl2.fastq   ctl3.fastq  uvb1.fastq  uvb2.fastq  uvb3.fastq
```

But what if we wanted to do something more complicated on lots of files, and do it in parallel across multiple processors? For example, what if we wanted to pull out all the reads containing the "GATTACA" motif from each file independently and write those all to separate files for each read?

We can't just do something like:

```
grep GATTACA *.fastq > gattacareads.txt
```

Because that would write one big file with all the GATTACA reads from all the files smashed together. What we want to do is something like this:

```
grep GATTACA ctl1.fastq > ctl1.fastq.gattaca.txt
grep GATTACA ctl2.fastq > ctl2.fastq.gattaca.txt
grep GATTACA ctl3.fastq > ctl3.fastq.gattaca.txt
grep GATTACA uvb1.fastq > uvb1.fastq.gattaca.txt
grep GATTACA uvb2.fastq > uvb2.fastq.gattaca.txt
grep GATTACA uvb3.fastq > uvb3.fastq.gattaca.txt
```

Now, for one, that's a lot of typing. What if you had 100 fastq files you wanted to do this with? And secondly, while this is example data and things run pretty quickly, what if each of these files were 10s of gigabytes, having millions of reads in each? Each `grep` command would take a few minutes to run. But most modern computers have multiple processors or cores in them, and we should be able to take advantage of that and send out each of those processes to a separate core to be done in parallel.

`find`

The UNIX `find` command is a simple program can be used to find files based on arbitrary criteria. Go back up to the parent `rnaseq-1day` directory, and type this command:

```
find .
```

That prints out all the files and directories, and everything in those directories, recursively. Let's print out only files with the `-type f` option:

```
find . -type f
```

Now, let's limit the search to find only fastq files with the `-name` option. Here, we just pass in quotes the pattern to match. Here it's anything that ends with `.fastq`.

```
find . -name "*.fastq"
```

That will find anything ending in `.fastq` living in any directory down from where we are currently standing on the filesystem.

Now, what if we wanted to actually do something with those files? There are a few ways to do this, but one we're going to use today involves using a program called `parallel`. If you run the `find` command and pipe the output of `find` into `parallel`, you can run arbitrary commands on the input files you found. Let's do a `--dry-run` so `parallel` will *only show us what would have been run.* Let's run a fake example:

```
#first find the files
find . -name "*.fastq"
# then pipe to parallel with dry run
find . -name "*.fastq" | parallel --dry-run "dowhatever {}"
```

```
dowhatever ./data/ctl1.fastq
dowhatever ./data/ctl2.fastq
dowhatever ./data/ctl3.fastq
dowhatever ./data/uvb1.fastq
dowhatever ./data/uvb2.fastq
dowhatever ./data/uvb3.fastq
```

- First, we're running the same `find` command as before. Remember, this prints out the path for all the fastq files it found, with the path relative to where we ran the find command.
- Next, we're calling the `parallel` program with the `--dry-run` option. This tells `parallel` to not actually run anything, but to just tell us what it *would* do.
- Next, we have the stuff we want to run in parallel inside the quotes.
- The open/closed curly braces `{}` is a special placeholder for `parallel`, which assumes the values of whatever was passed in on the pipe. In this example, each of the fastq files found by `find` will take the place of `{}` wherever it's found.

Let's try one for real. Let's get the word count of all the fastq files in parallel.

```
# First find the files
find . -name "*.fastq"

# Next do a dry-run to see what would be run
find . -name "*.fastq" | parallel --dry-run "wc {}"

# Finally, run the commands in parallel
find . -name "*.fastq" | parallel "wc {}"
```

What if we wanted to do something like search for a particular nucleotide sequence like "GATTACA" inside of each file, pull out those sequences, and write those to a separate results file for each input?

```
# First find the files
find . -name "*.fastq"

# Next do a dry-run to see what would be run
find . -name "*.fastq" | parallel --dry-run "grep GATTACA {} > {}.gattaca.txt"

# Finally, run the commands in parallel
find . -name "*.fastq" | parallel "grep GATTACA {} > {}.gattaca.txt"
```

When you do the dry run, you'll get something like that looks like this:

```
grep GATTACA ./data/ctl1.fastq > ./data/ctl1.fastq.gattaca.txt
grep GATTACA ./data/ctl2.fastq > ./data/ctl2.fastq.gattaca.txt
grep GATTACA ./data/ctl3.fastq > ./data/ctl3.fastq.gattaca.txt
grep GATTACA ./data/uvb1.fastq > ./data/uvb1.fastq.gattaca.txt
grep GATTACA ./data/uvb2.fastq > ./data/uvb2.fastq.gattaca.txt
grep GATTACA ./data/uvb3.fastq > ./data/uvb3.fastq.gattaca.txt
```

These are the commands that *would be run* in `parallel` if you didn't use the `--dry-run` flag. Now, if we go back and re-run that command without the `--dry-run` flag.

EXERCISE

1. `cd` into the `data` directory and take a look at what files were created.
2. Open up one of the new files with `less` and use the `/` key to search for the "GATTACA" motif. Does it actually occur on every line?
3. Use `rm` to delete all the files ending with `.gattaca.txt`.

Installing software

There are a few different ways to install software. We're using an Ubuntu Linux distribution, and Ubuntu has a very nice software package management system called apt. You can read more about it at the online documentation. For what we'll do later on we'll need java, which isn't installed by default. If we try running `which java`, we'll see nothing is returned. If we try running `java`, Ubuntu will suggest a packages that we might try downloading to get java enabled. We'll want the default Java Runtime Environment. To install software we need to temporarily elevate our permissions to the level of the "super user". We do this temporarily by prefacing any command we want to run as super user with the command `sudo`. Let's install Java.

```
which java
java
sudo apt-get install default-jre
```

Where can I learn more about the shell?

- Software Carpentry tutorial - The Unix shell
- The shell handout - Command Reference
- explainshell.com
- http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html
- http://stackoverflow.com/
- Google - if you don't know how to do something, try Googling it. Other people have probably had the same question.
- Learn by doing. There's no real other way to learn this than by trying it out. Write your next paper in nano, open pdfs from the command line, automate something you don't really need to automate or actually do.

Chapter 2

RNA-seq: reads to counts

back to course contents

In this section of the workshop we will start with raw RNA-seq reads and do everything that we need to do to get a *count matrix* - a table or "spreadsheet" in a single file containing the the number of reads mapping to each gene for each sample. This count matrix is the input into R for differential gene expression analysis. A count matrix is a table in a single file with a row for every gene and a column for every sample. Celli,j will contain the number of genes mapping to the ith gene for the jth sample. For example, in the table below, 865 reads mapped to GeneB for sample ctl2.

	ctl1	ctl2	ctl3	uvb1	uvb2	uvb3
GeneA	33	54	10	12	50	49
GeneB	255	865	864	5446	2664	4665
GeneC	0	0	1	2	3	15

Objectives

- Quality assessment for raw sequencing read
- Quality control to trim low-quality sequence data from the ends of reads
- Align reads to the genome for all samples
- Count the number of reads mapping to each gene for each sample

Data management

First, let's extract the fastq files if we haven't done so already:

```
cd workshops/lessons/rnaseq-1day/data
ls -l
gunzip *.fastq.gz
ls -l
```

Quality control

Quality assessment

Next, let's run some quality control analysis. Google search "FastQC" to find the FastQC package from Simon Andrews. It's a program that pretty much everyone uses for QC analysis that can be run at the command line on a batch of FASTQ files.

You'll also need java to run it (`sudo apt-get install default-jre`).

First let's get some help on FastQC. Most bioinformatics programs have a `-h` or `--help` option that gives you some very basic documentation on how to run it. These aren't the same as built-in UNIX man pages. For more in-depth documentation you need to read the paper or visit the tool's website to download a manual. For instance, take a look at the online FastQC documentation and look specifically at the differences between good Illumina data and bad Illumina data.

```
fastqc -h
fastqc *.fastq --threads 4 --outdir .
```

Open up your SFTP client and enter your IP address and credentials. Transfer this data over to your computer to view it. You'll see how the last 5 bases of each read is pretty terrible quality, so let's trim those reads.

Trimming: one sample

Google "FASTX Toolkit" to find Greg Hannon's FASTX-Toolkit software. This lets you do lots of things for manipulating FASTQ and FASTA files. We're just going to use a feature to trim the end of the reads. You can see which tool you need by looking at the online documentation.

First, let's run `fastx_trimmer` to get some help.

```
fastx_trimmer -h
```

It tells us we can use the `-t` option followed by a number to trim that number of nucleotides off the end of the read. It tells us that the default way to use the program is to feed data off the `STDIN`, which is UNIX-speak for piping in data. It's default output is the `STDOUT`, which is UNIX-speak for "printing" the data to the screen. Let's see what that means:

```
head -n 8 ctl1.fastq
head -n 8 ctl1.fastq | fastx_trimmer -t 5
```

We get some error about an invalid quality score value. Let's learn a little bit about quality scores. Google FASTQ format and click the Wikipedia article, and scroll down to the section on quality. Next scroll down to encoding. Quality scores are encoded in a FASTQ file different ways. You just have to know that this is quality score encoding is Sanger, or Phred+33. There are other tools out there that can tell you what kind of quality score you have, and convert between different quality score encodings. But Sanger/Phred+33 is the most common. Unfortunately `fastx_trimmer` doesn't seem to recognize this encoding. Let's Google that error message specifically. Google "fastx_trimmer invalid quality score value" and you'll arrive at a SEQAnswers thread that tells you the answer. (Look who provided that answer, by the way, Simon Andrews, the author of FASTQC!). You can also go to the FASTX-Toolkit homepage, and if you scroll down a bit, you'll see that back in 2009 this was added as an option, but it's still not documented in the command line help.

I show you this because this is what doing bioinformatics is like in the real world. You think you have the right tool for the job, you try to run it like the documentation says to run it, and it doesn't work. You Google around to try to find the solution to the problem or better documentation. SEQAnswers is a great resource, and so is Biostars.

Let's try that again:

```
head -n 8 ctl1.fastq
head -n 8 ctl1.fastq | fastx_trimmer -t 5 -Q33
```

Look at that! `fastx_trimmer` trimmed 5 bases off the end of two reads coming in from a pipe and spit it back out to the screen. We could do this to the whole file if we did something like `cat ctl1.fastq | fastx_trimmer ... > trimmed_ctl1.fastq`. Or we could use the command line options to take input files and write output files.

```
fastx_trimmer -h
fastx_trimmer -Q33 -t 5 -i ctl1.fastq -o trimmed_ctl1.fastq
```

Let's take a look at those files to make sure it worked right. Also, let's do a word count as a sanity check to make sure we didn't lose any reads or anything.

```
head -n 4 ctl1.fastq trimmed_ctl1.fastq
wc ctl1.fastq trimmed_ctl1.fastq
```

EXERCISE

Run `ls` on the current directory. We need to clean things up before they get out of hand.

- Remove the trimmed file you just created
- Make a directory called “QC”
- Move all the fastqc output directories and zip files into the QC directory (hint, you can do this all with a wildcard and a single `mv` command)
- Use `find ... | parallel ...` to run `wc` on all the fastq files in parallel

Trimming: all samples in parallel

Let’s use GNU parallel to run these trimming jobs in parallel:

```
find *.fastq | parallel --dry-run "fastx_trimmer -t 5 -Q33 -i {} -o trimmed_{}"
```

Tip: `*` matches everything or nothing. If we wanted to look at the first few lines of one of these files, we could do this:

```
head *ctl1.fastq
```

Which would print the first few lines of the trimmed and untrimmed version.

Let’s make a directory for all the untrimmed files:

```
mkdir untrimmed
mv ctl*.fastq uvb*.fastq untrimmed
```

Alignment

Building an index

For the sake of time and compute requirements I’ve extracted reads that originate from a 100MB region of chromosome 4. In reality, you would want to do this with the entire genome. The first step here is to create an index.

Google “Ensembl FTP” to get to the FTP download page. Scroll down to the Human FASTA sequences. Click that link and copy the link to download the FASTA sequence for chromsome 4. While we’re at it let’s go ahead and download the gene set annotation (GTF file). We’re looking at chromosome 4.

This is what you *would* do if you were doing this on your own. But this can take a while so I’ve already downloaded and extracted these two files for you. See `~/genomedata`.

```
wget ftp://ftp.ensembl.org/pub/release-77/fasta/homo_sapiens/dna/Homo_sapiens.GRCh38.dna
wget ftp://ftp.ensembl.org/pub/release-77/gtf/homo_sapiens/Homo_sapiens.GRCh38.77.gtf.gz
gunzip Homo_sapiens*.gz
```

Or just move them from genomedata in the home directory:

```
ls -l ~/genomedata
mv ~/genomedata/* .
```

Now, let's build an index. Tophat is a spliced aligner. It uses the Bowtie aligner under the hood. To run Bowtie we have to first build an index using a utility that comes with Bowtie. You only have to do this once each time you build a reference. First, let's get a little help, then get it started running while we talk about alignment and aligner indexes.

```
bowtie2-build -h
bowtie2-build chr4.fa chr4
```

Alignment with TopHat

Let's try a small sample first. First, run `tophat -h` to get a little help. Let's also look at the TopHat documentation online.

```
tophat -h
head -n 40000 trimmed_ctl1.fastq > test10k.fastq
tophat --no-coverage-search -o test10k_tophat chr4 test10k.fastq
```

EXERCISE

First, look around at the results, then delete those files when you're done:

- Go into the output directory that was just created.
- Look at the `align_summary.txt` file that was created (**don't try to cat the .bam files in that directory!**)
- When you're done, remove the entire directory containing that test output, and remove the test10k.fastq file you created.

Next, Using `find` and `parallel --dry-run`, try to construct the command that would run all the samples through tophat in parallel. Using the `-o` option, make the output for each run be a directory with `_tophat` appended to it. The command generated should look like this:

```
tophat --no-coverage-search -o trimmed_ctl1.fastq_tophat chr4 trimmed_ctl1.fastq
tophat --no-coverage-search -o trimmed_ctl2.fastq_tophat chr4 trimmed_ctl2.fastq
tophat --no-coverage-search -o trimmed_ctl3.fastq_tophat chr4 trimmed_ctl3.fastq
tophat --no-coverage-search -o trimmed_uvb1.fastq_tophat chr4 trimmed_uvb1.fastq
tophat --no-coverage-search -o trimmed_uvb2.fastq_tophat chr4 trimmed_uvb2.fastq
tophat --no-coverage-search -o trimmed_uvb3.fastq_tophat chr4 trimmed_uvb3.fastq
```

But, don't launch the jobs just yet

Now, before we launch those jobs, let's take a look at the help for GNU parallel. We can specify an upper limit to the number of jobs run in parallel using the -j option. Let's only run three jobs in parallel so we don't crash our system. Tophat consumes quite a bit of memory, and we only have 15GB available to us.

```
parallel -h
find trimmed_*.fastq | parallel -j 3 --dry-run "tophat --no-coverage-search -o {}_tophat
```

EXERCISE

- When that's done, from the main data directory, look at all the `align_summary.txt` files with one command.
- Using `grep`, pull out the line that shows you the number of mapped reads.

View the alignment

If time allows let's look around with samtools Tview. Much better to do this with software such as the Broad's Integrative Genomics Viewer (IGV).

```
cd trimmed_ctl1.fastq_tophat/
samtools view accepted_hits.bam | less
samtools index accepted_hits.bam
samtools tview
samtools tview accepted_hits.bam ../chr4.fa
# ?
# g 4:82426400
# . to turn on off dot view
# n to turn on nt colors
# b to turn on base call qualities
# r to toggle read names
# space/backspace to move around
```

Counting

Remember what we want to end up with is a table that gives us the number of reads mapping to each gene for each sample.

All we have now is an alignment which tells us for each sample the genomic positions of where each read aligned on the reference sequeunce. It doesn't tell us anything about genes. For that we'll need a gene annotation, which tells us the genomic position of genes (more specifically, exons). Then we can kind of intersect the two.

```
head chr4.gtf
```

There's a tool called featureCounts that's part of a larger package called subread that very quickly goes through all your alignments for all your samples and counts which reads map to exons in a supplied annotation file.

http://subread.sourceforge.net/

If you run `featureCounts` without any arguments, it gives you some help.

You're required to give it an annotation file, an output file, and one ore more input files.

You might want to pay special attention to the `-t` and `-g` options. Here we want to count all the reads mapping to any exon of a gene (`-t exon`), and summarize the counts for that gene across all the exons that gene has (`-g gene_name`). Use `less genes.gtf` to look at the annotation to see what the names of the meta-features are.

We can also use the `-T` option to specify that we want to run multiple threads to speed things up.

Take a look at other options for dealing with paired-end data, strand-specific data, multi-mapping reads, etc.

```
featureCounts -a chr4.gtf -o counts.txt -t exon -g gene_name -T 4 */accepted_hits.bam
```

After mapping is complete, take a look at the summary file produced. Open up cyberduck and using SFTP, download the counts.txt file you just created. ***Finally, terminate the AWS EC2 instance after you've downloaded all the data you need.***

Resources

- Illumina iGenomes: Gene annotations and pre-computed indexes for a variety of organisms.
- Bowtie2 Manual: http://bowtie-bio.sourceforge.net/bowtie2/manual.shtml
- Tophat2
- Tutorial: http://ccb.jhu.edu/software/tophat/tutorial.shtml
- Manual: http://ccb.jhu.edu/software/tophat/manual.shtml
- featureCounts tutorial: http://bioinf.wehi.edu.au/featureCounts/

Chapter 3

Introduction to R

back to course contents

The first part of this workshop will demonstrate very basic functionality in R, including functions, functions, vectors, creating variables, getting help, subsetting, data frames, plotting, and reading/writing files.

Link to slides.

Link to R Cheat Sheet.

Before coming

{% include setup-r.md %}

RStudio

Let's start by learning about RStudio. **R** is the underlying statistical computing environment, but using R alone is no fun. **RStudio** is a graphical integrated development environment that makes using R much easier.

- Panes in RStudio: I set up my window to have the editor in the top left, console top right, environment/history on the bottom left, and plots/help on the bottom right.
- Code that you type into the console is code that R executes. From here forward we will use the editor window to write a script that we can save to a file and run it again whenever we want to. We usually give it a `.R` extension, but it's just a plain text file. If you want to send commands from your editor to the console, use `CMD+Enter` (`Ctrl+Enter` on Windows).
- Anything after a `#` sign is a comment. Use them liberally to *comment your code.*

Basic operations

R can be used as a glorified calculator. Try typing this in directly into the console. Make sure you're typing into into the editor, not the console, and save your script. Use the run button, or press `CMD+Enter` (`Ctrl+Enter` on Windows).

```
2 + 2
5 * 4
2^3
```

R Knows order of operations and scientific notation.

```
2 + 3 * 4/(5 + 3) * 15/2^2 + 3 * 4^2
50000
```

However, to do useful and interesting things, we need to assign *values* to *objects*. To create objects, we need to give it a name followed by the assignment operator `<-` and the value we want to give it:

```
weight_kg <- 55
```

`<-` is the assignment operator. Assigns values on the right to objects on the left, it is like an arrow that points from the value to the object. Mostly similar to `=` but not always. Learn to use `<-` as it is good programming practice. Using `=` in place of `<-` can lead to issues down the line.

Objects can be given any name such as `x`, `current_temperature`, or `subject_id`. You want your object names to be explicit and not too long. They cannot start with a number (`2x` is not valid but `x2` is). R is case sensitive (e.g., `weight_kg` is different from `Weight_kg`). There are some names that cannot be used because they represent the names of fundamental functions in R (e.g., `if`, `else`, `for`, see here for a complete list). In general, even if it's allowed, it's best to not use other function names, which we'll get into shortly (e.g., `c`, `T`, `mean`, `data`, `df`, `weights`). In doubt check the help to see if the name is already in use. It's also best to avoid dots (`.`) within a variable name as in `my.dataset`. It is also recommended to use nouns for variable names, and verbs for function names.

When assigning a value to an object, R does not print anything. You can force to print the value by typing the name:

```
weight_kg
```

Now that R has `weight_kg` in memory, we can do arithmetic with it. For instance, we may want to convert this weight in pounds (weight in pounds is 2.2 times the weight in kg):

```
2.2 * weight_kg
```

We can also change a variable's value by assigning it a new one:

```
weight_kg <- 57.5
2.2 * weight_kg
```

This means that assigning a value to one variable does not change the values of other variables. For example, let's store the animal's weight in pounds in a variable.

```
weight_lb <- 2.2 * weight_kg
```

and then change `weight_kg` to 100.

```
weight_kg <- 100
```

What do you think is the current content of the object `weight_lb`? 126.5 or 220?

You can see what objects (variables) are stored by viewing the Environment tab in Rstudio. You can also use the `ls()` function. You can remove objects (variables) with the `rm()` function. You can do this one at a time or remove several objects at once.

```
ls()
rm(weight_lb, weight_kg)
ls()
weight_lb  # oops! you should get an error because weight_lb no longer exists!
```

EXERCISE

What are the values after each statement in the following?

```
mass <- 50  # mass?
age <- 30  # age?
mass <- mass * 2  # mass?
age <- age - 10  # age?
mass_index <- mass/age  # massIndex?
```

Functions

R has built-in functions.

```
# Notice that this is a comment.  Anything behind a # is 'commented out' and
# is not run.
sqrt(144)
log(1000)
```

Get help by typing a question mark in front of the function's name, or `help(functionname)`:

```
help(log)
?log
```

Note syntax highlighting when typing this into the editor. Also note how we pass *arguments* to functions. The `base=` part inside the parentheses is called an argument, and most functions use arguments. Arguments modify the behavior of the function. Functions some input (e.g., some data, an object) and other options to change what the function will return, or how to treat the data provided. Finally, see how you can *next* one function inside of another (here taking the square root of the log-base-10 of 1000).

```
log(1000)
log(1000, base = 10)
log(1000, 10)
sqrt(log(1000, base = 10))
```

EXERCISE

See `?abs` and calculate the square root of the log-base-10 of the absolute value of `-4*(2550-50)`. Answer should be `2`.

Vectors and classes

Let's create some numeric vectors. Vectors (aka "arrays" in Perl, "lists" in Python) are single *objects* containing an ordered collection of *elements*. A simple vector is a numeric vector, a single *object* containing several numbers. Here let's display a few vectors. We can also do vector arithmetic. When printing vectors to the screen that have lots of elements, notice that the bracketed number in the gutter of the output is just a counter indexing the number of elements in the vector.

```
# Some simple numeric vectors:
1:5
6:10
1:5 + 6:10
1:100
```

We can also create arbitrary vectors with the `c()` function (short for "combine").

```
c(1, 2, 5)
c(1:5, 11:15)
```

What if we wanted to create a vector from 2 to 10 by 2's? What about 2 to 200 by 4's? This might be useful for setting up an experiment where every other sample is an experimental group and every other is a control.

```
c(2, 4, 6, 8, 10)

# Get some help with the seq() function, then create a vector from 2 to 200
# by 2s.  Notice how the seq() function works -- the `to` argument will
# never be exceeded.
help(seq)
seq(from = 2, to = 200, by = 4)
```

You can assign this vector of values to an object, just like you would for one item. For example we can create a vector of animal weights:

```
weights <- c(50, 60, 65)
weights
```

A vector can also contain characters:

```
animals <- c("mouse", "rat", "dog")
animals
```

There are many functions that allow you to inspect the content of a vector. `length()` tells you how many elements are in a particular vector:

```
length(weights)
length(animals)
```

`class()` indicates the class (the type of element) of an object:

```
class(weights)
class(animals)
```

EXERCISE

- Use the `c()` function to create/assign a new object that combines the `weights` and `animals` vectors into a single vector called `combined`.
- What happened to the numeric values? *Hint*: What's the `class()` of `combined`?
- Why do you think this happens?

The function `str()` provides an overview of the structure of an object and the elements it contains. It is a really useful function when working with large and complex objects:

```
str(weights)
str(animals)
```

You can add elements to your vector simply by using the `c()` function:

```
weights
weights <- c(weights, 90)  # adding at the end
weights <- c(30, weights)  # adding at the beginning
weights
```

What happens here is that we take the original vector `weights`, and we are adding another item first to the end of the other ones, and then another item at the beginning. We can do this over and over again to build a vector or a dataset. When you're programming this may be useful to autoupdate results that we are collecting or calculating.

Certain *functions* operate only on certain *classes* of object. Here, `weights` is a `numeric` vector. The built-in `sum()` function will operate on numeric objects, but not characters.

```
sum(weights)
sum(animals)
```

EXERCISE

Sum the integers 1 through 100 and 501 through 600 (e.g. 1+2+...+99+100+501+502+...+599+600)

Slicing/indexing vectors

Let's create a vector of 50 integers going from 101 to 150. We can access certain elements of that vector by putting the element's *index(es)* in square brackets. E.g., `x[1]` will return the first element in vector `x`. Calling `x[c(3,5)]` will access the third and fifth elements. Calling `x[1:10]` will return the first ten elements of `x`.

Special note: R indexes vectors starting at 1. This is different from many other languages, including Perl and Python, which index starting from zero.

```
# Create the vector.
x <- 101:150

# Get the first element.
x[1]

# Get the 42nd element.
x[42]

# Get the 20th through the 25th elements.
x[20:25]

# If you try to access elements that don't exist, you'll return missing
# values.  Missing values are represented as NA
x[45:55]  #NA is missing value!
```

Data Frames

We use **data frames** to store heterogeneous tabular data in R: tabular, meaning that individuals or observations are typically represented in rows, while variables or features are represented as columns; heterogeneous, meaning that columns/features/variables can be different classes (on variable, e.g. age, can be numeric, while another, e.g., cause of death, can be text).

Later on we'll read in data from a text file into a data frame object using one of the functions `read.table()` for text files or `read.csv()` for comma-separated tables. But for now, let's use a built-in data frame called `mtcars`. This data was extracted from the 1974 Motor Trend US magazine, and comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973–74 models). We can load this built-in data with `data(mtcars)`. By the way, running `data()` without any arguments will list all the available built-in datasets included with R.

Let's load the data first. Type the name of the object itself (`mtcars`) to view the entire data frame. *Note: doing this with large data frames can cause you trouble.*

```
data(mtcars)
class(mtcars)
mtcars
```

Inspecting data.frame objects

There are several built-in functions that are useful for working with data frames.

- Content:
  - `head()`: shows the first 6 rows
  - `tail()`: shows the last 6 rows
- Size:
  - `dim()`: returns a 2-element vector with the number of rows in the first element, and the number of columns as the second element (the dimensions of the object)
  - `nrow()`: returns the number of rows
  - `ncol()`: returns the number of columns
- Summary:
  - `colnames()` (or just `names()`): returns the column names
  - `rownames()`: returns the row names
  - `str()`: structure of the object and information about the class, length and content of each column
  - `summary()`: works differently depending on what kind of object you pass to it. Passing a data frame to the `summary()` function prints out some summary statistics about each column (min, max, median, mean, etc.)

```
head(mtcars)
tail(mtcars)
dim(mtcars)
nrow(mtcars)
ncol(mtcars)
colnames(mtcars)
rownames(mtcars)
str(mtcars)
summary(mtcars)
```

Accessing variables & subsetting data frames

We can access individual variables within a data frame using the `$` operator, e.g., `mydataframe$specificVariable`. Let's print out the number of cylinders for every car, and calculate the average miles per gallon for ever car in the dataset (using the built-in `mean()` function).

```
# display the number of cylinders for each car.
mtcars$cyl
# first display MPG for all vehicles, then calculate the average.
mtcars$mpg
mean(mtcars$mpg)
```

We can also use the `subset()` function to return a subset of the data frame that meets a specific condition.

1. The first argument is the data frame you want to subset, e.g. `subset(mtcars....`
2. The second argument is a condition you must satisfy, e.g. `subset(mtcars, cyl==6)`. If you want to satisfy *all* of multiple conditions, you can use the "and" operator, `&`. The "or" operator `|` (the pipe character, usually shift-backslash) will return a subset that meet *any* of the conditions.
    - `==`: Equal to
    - `!=`: Not equal to
    - `>`, `>=`: Greater than, greater than or equal to
    - `<`, `<=`: Less than, less than or equal to

Try it out:

```
# Return only cars with 6 cylinder engines.
subset(mtcars, cyl == 6)
# Return only cars having more than 6 cylinders **and** an engine
# displacement volume less than 300cc.
subset(mtcars, cyl > 6 & disp < 300)
# Return only the cars that get at least 20 miles per gallon or have a
# displacement volume of less than 100cc.
subset(mtcars, mpg >= 20 | disp < 100)
```

Finally, take a look at the class of what's returned by a `subset()` function. The `subset()` function takes a data.frame and returns a data.frame. You can operate on this new data.frame just as you would any other data.frame using the `$` operator. E.g., print the MPG for all the 6 cylinder vehicles:

```
subset(mtcars, cyl == 6)$mpg
```

EXERCISE

1. Print out the part of the mtcars dataset only for cars that have greater than or equal to 6 cylinders *and* get at least 15 miles per gallon.
2. What's the mean engine displacement of these vehicles?

---

with()

The `with()` function is particularly helpful. Let's say you wanted to compute some (senseless) value by computing the MPG times the number of cylinders divided by the car's displacement. You could access the dataset's variables using the $ notation, or you could use `with()` to temporarily *attach* the data frame, and call the variables directly, as if they were just vectors hanging out in your workspace. The first argument to `with()` is the name of the data frame, and the second argument is all the stuff you'd like to do with the particular features in that data frame.

Try typing the following commands:

```r
# Display the number of cylinders.
mtcars$cyl
with(mtcars, cyl)

# Compute the senseless value described above. Both return the same results.
mtcars$mpg * mtcars$cyl/mtcars$disp
with(mtcars, mpg * cyl/disp)
```

EXERCISE

Using the `with()` and `subset()` functions, compute the average value of the ratio of the engine displacement (`disp`) divided by the engine's horsepower (`hp`) for all the 8-cylinder vehicles.

```
## [1] "Answer: 1.754801425962"
```

Plotting basics

Plotting a single numeric variable goes down the rows and plots a value on the y-axis for each observation (index) in the data frame.

```r
plot(mtcars$mpg)
```

This isn't a very useful figure. More appropriate might be a histogram. We can try to let R decide how many breaks to insert in the histogram, or we can set that manually. We can also set the color of the bars.

```r
hist(mtcars$mpg)
hist(mtcars$mpg, breaks = 10)
hist(mtcars$mpg, breaks = 10, col = "black")
```

We can create a scatterplot between two variables with `plot(varX, varY)`.

```r
# This would also work, but let's use with().  plot(mtcars$disp, mtcars$mpg)
with(mtcars, plot(disp, mpg))
```

There are hundreds of plotting parameters you can use to make your plot look exactly like you want. Let's use a solid-filled point instead of an open circle with the `pch=` argument (point character), color the points red with the `col=` argument, give it a title by passing a character object to the `main=` argument, and change the x and y axis titles with the `xlab=` and `ylab=` arguments, respectively. Let's go through this one step at a time.

```r
with(mtcars, plot(disp, mpg, pch = 16))
with(mtcars, plot(disp, mpg, pch = 16, col = "red"))
with(mtcars, plot(disp, mpg, pch = 16, col = "red", main = "MPG vs Displacement"))
with(mtcars, plot(disp, mpg, pch = 16, col = "red", main = "MPG vs Displacement",
    ylab = "Fuel Economy (MPG)", xlab = "Displacement (cu. in.)"))
```

Notice how on that last line I broke the command up into two lines for better readability. I broke the command at the comma separating arguments, and indented the following line for readability.

With plotting parameters, **Google is your friend.**

- Forget what each point character represents? Google *R pch*.
- Forget the names of R's colors? Google *R colors*. Want to learn more about color schemes in R? Google *RColorBrewer*.
- Try googling *R graphical parameters*.

EXERCISE

Plot horsepower (y-axis) vs displacement (x-axis) for vehicles with more than 4 cylinders. Give the graph a title and label the axes. Make the points solid (hint, `pch=16`) blue (hint, `col="blue"`) circles.

Reading in / writing out data

First, lets create a small dataset consisting of only 8 cylinder cars.

```
mtcars_8cyl <- subset(mtcars, cyl == 8)
mtcars_8cyl
```

Next, check what your working directory is with `getwd()` with no arguments, and look up some help for `write.table()` and `write.csv()`.

```
getwd()
help(write.table)
help(write.csv)
```

Using RStudio, go to the Session menu, and select the directory (folder) you want to work from under the "Set Working Directory" menu. You can also do this manually with the `setwd()` command.

```
getwd()
setwd("~/Desktop/R")
```

Once you've set your working directory either using RStudio or on the command line, save the new reduced data frame to a comma-separated file called `cars8.csv` using the `write.csv()` function.

```
write.csv(mtcars_8cyl, file = "cars8.csv")
```

Data can be loaded using the Tools – Import Dataset – From text file menu in R studio. Or you can also load a dataset manually using `read.table()` or `read.csv()`. First, read the help on these functions:

```
help(read.table)
help(read.csv)
```

Here let's remove the dataset, and re-import it into an object called cars8 from the file we just saved.

```
rm(mtcars_8cyl)
mtcars_8cyl
cars8 <- read.table(file = "cars8.csv", header = TRUE, sep = ",", row.names = 1)
cars8
rm(cars8)
cars8 <- read.csv(file = "cars8.csv", header = TRUE, row.names = 1)
cars8
```

Chapter 4

RNA-seq: differential gene expression analysis

back to course contents

This is an introduction to RNAseq analysis involving reading in count data from an RNAseq experiment, exploring the data using base R functions and then analysis with the DESeq2 package.

Install and load packages

First, we'll need to install some add-on packages. Most generic R packages are hosted on the Comprehensive R Archive Network (CRAN, http://cran.us.r-project.org/). To install one of these packages, you would use `install.packages("packagename")`. You only need to install a package once, then load it each time using `library(packagename)`. Let's install the **gplots** and **calibrate** packages.

```
install.packages("gplots")
install.packages("calibrate")
```

Bioconductor packages work a bit differently, and are not hosted on CRAN. Go to http://bioconductor.org/ to learn more about the Bioconductor project. To use any Bioconductor package, you'll need a few "core" Bioconductor packages. Run the following commands to (1) download the installer script, and (2) install some core Bioconductor packages. You'll need internet connectivity to do this, and it'll take a few minutes, but it only needs to be done once.

```
# Download the installer script
source("http://bioconductor.org/biocLite.R")
```

```
# biocLite() is the bioconductor installer function. Run it without any
# arguments to install the core packages or update any installed packages.
# This requires internet connectivity and will take some time!
biocLite()
```

To install specific packages, first download the installer script if you haven't done so, and use `biocLite("packagename")`. This only needs to be done once then you can load the package like any other package. Let's download the DESeq2 package:

```
# Do only once
source("http://bioconductor.org/biocLite.R")
biocLite("DESeq2")
```

Now let's load the packages we'll use:

```
library(DESeq2)
library(gplots)
library(calibrate)
```

Bioconductor packages usually have great documentation in the form of *vignettes*. For a great example, take a look at the DESeq2 vignette for analyzing count data.

Introduction and data import

Analyzing an RNAseq experiment begins with sequencing reads. These are aligned to a reference genome, then the number of reads mapped to each gene can be counted.

The data for this tutorial comes from a PLOS ONE paper, Genome-Wide Transcriptional Profiling of Skin and Dorsal Root Ganglia after Ultraviolet-B-Induced Inflammation, and the raw data can be downloaded from Gene Expression Omnibus database (GEO).

This data has already been downloaded and aligned to the human genome. The command line tool featureCounts was used to count reads mapped to human genes from the Ensembl annotation.

The output from this tool is provided in the `counts.txt` file in the `data` directory. Have a look at this file in the shell, using `head`.

First, set your working directory to the top level of the RNA-seq course. Import the data into R as a `data.frame` and examine it again. You can set the arguments of `read.table` to import the first row as a header giving the column names, and the first column as row names.

```
# Filename with output from featureCounts
countfile <- "data/counts.txt"
```

```
# Read in the data
countdata <- read.table(countfile, header = TRUE, row.names = 1)
head(countdata)
colnames(countdata)
class(countdata)
```

The data.frame contains information about genes (one gene per row) with the gene positions in the first five columns and then information about the number of reads aligning to the gene in each experimental sample. There are three replicates for control (column names starting with "ctl") and three for samples treated with ultraviolet-B light (starting "uvb"). We don't need the information on gene position for this analysis, just the counts for each gene and sample, so we can remove it from the data frame.

```
# Remove first five columns (chr, start, end, strand, length)
countdata <- countdata[, -(1:5)]
head(countdata)
colnames(countdata)
```

We can rename the columns to something shorter and a bit more readable.

```
# Manually
c("ctl1", "ctl2", "ctl3", "uvb1", "uvb2", "uvb3")
```

We can do it manually, but what if we have 600 samples instead of 6? This would become cumbersome. Also, it's always a bad idea to hard-code sample phenotype information at the top of the file like this. A better way to do this is to use the `gsub` command to strip out the extra information. This more robust to introduced errors, for example if the column order changes at some point in the future or you add additional replicates.

```
# Using gsub -- robust. Get help with ?gsub
gsub(pattern = "trimmed_|.fastq_tophat.accepted_hits.bam", replacement = "",
    x = colnames(countdata))
colnames(countdata) <- gsub(pattern = "trimmed_|.fastq_tophat.accepted_hits.bam",
    replacement = "", x = colnames(countdata))
head(countdata)
```

EXERCISE

There's an R function called `rowSums()` that calculates the sum of each row in a numeric matrix, like the count matrix we have here, and it returns a vector. There's also a function called `which.max()` that determines the index of the maximum value in a vector.

0. Find the gene with the highest expression across all samples – remember, each row is a gene.
1. Extract the expression data for this gene for all samples.
2. In which sample does it have the highest expression?
3. What is the function of the gene? Can you suggest why this is the top expressed gene?

Data investigation using base R

We can investigate this data a bit more using some of the basic R functions before going on to use more sophisticated analysis tools.

First make a copy of the data, because we'll need it later. We will work on the copy. We will calculate the mean for each gene for each condition and plot them.

```
cd2 <- countdata  #make a copy

# get Control columns
colnames(cd2)

# grep searches for matches to a pattern. Get help with ?grep

# get the indexes for the controls
grep("ctl", colnames(cd2))
ctlCols <- grep("ctl", colnames(cd2))
head(cd2[, ctlCols])

# use the rowMeans function
cd2$ctlmean <- rowMeans(cd2[, ctlCols])
head(cd2)

# same for uvb
uvbCols <- grep("uvb", colnames(cd2))
cd2$uvbmean <- rowMeans(cd2[, uvbCols])

head(cd2)
```

EXERCISE

Using the `subset()` function, print out all the columns where the control mean does not equal 0 **and** where the UVB mean does not equal zero.

Bonus: code golf – use the fewest characters to get the same solution.

EXERCISE

1. Plot the mean expression of each gene in control against the UVB sample mean. Are there any outliers?
2. How could you make this plot more informative and look more professional? Hint: try plotting on the log scale and using a different point character.

```
plot(cd2$ctlmean, cd2$uvbmean)
with(cd2, plot(log10(ctlmean), log10(uvbmean), pch = 16))
with(cd2, plot(ctlmean, uvbmean, log = "xy", pch = 16))
```

Poor man's differential gene expression

We can find candidate differentially expressed genes by looking for genes with a large change between control and UVB samples. A common threshold used is log2 fold change more than 2 or less than -2. We will calculate log2 fold change for all the genes and colour the genes with log2 fold change of more than 2 or less than -2 on the plot.

First, check for genes with a mean expression of 0. Putting zeroes into the log2 fold change calculation will produce NAs, so we might want to remove these genes. Note: this is for mathematical reasons, although different software may produce different results when you try to do `log2(0)`.

```
head(cd2$ctlmean)
head(cd2$ctlmean > 0)
```

In R, `TRUE` and `FALSE` can be represented as `1` and `0`, respectively. When we call `sum(cd2$ctlmean > 0)`, we're really asking, "how many genes have a mean above 0 in the control group?"

```
sum(cd2$ctlmean > 0)
sum(cd2$uvbmean > 0)
```

Now, let's subset the data and keep things where either the control or UVB group means are greater than zero.

```
nrow(cd2)
cd2 <- subset(cd2, cd2$ctlmean > 0 | cd2$uvbmean > 0)
nrow(cd2)
head(cd2)
```

Mathematically things work out better for us when we test things on the log scale. On the absolute scale, upregulation goes from 1 to infinity, while downregulation is bounded by 0 and 1. On the log scale, upregulation goes from 0 to infinity, and downregulation goes from 0 to negative infinity. Let's compute a log-base-2 of the fold change.

When we do this we'll see some `Inf` and `-Inf` values. This is what happens when we take `log2(Inf)` or `log2(0)`.

```
# calculate the log2 fold change
cd2$log2FC <- log2(cd2$uvbmean/cd2$ctlmean)
head(cd2)

# see how many are up, down, or both (using the absolute value function)
sum(cd2$log2FC > 2)
sum(cd2$log2FC < -2)
sum(abs(cd2$log2FC) > 2)
```

We can few just the "outliers" with `subset(cd2, abs(log2FC)>2)`, which gets us just the things that have a large fold change in either direction. Let's plot these.

```
with(cd2, plot(ctlmean, uvbmean, log = "xy", pch = 16))
subset(cd2, abs(log2FC) > 2)
# use the points function to add more points to the same axes
with(subset(cd2, abs(log2FC) > 2), points(ctlmean, uvbmean, pch = 16, col = "red"))
```

What do you notice about the positions of the outliers on these plots? How would you interpret this? What are some of the problems with this simple approach?

DESeq2 analysis

DESeq2 is an R package for analyzing count-based NGS data like RNA-seq. It is available from Bioconductor. Bioconductor is a project to provide tools for analysing high-throughput genomic data including RNA-seq, ChIP-seq and arrays. You can explore Bioconductor packages here.

Just like R packages from CRAN, you only need to install Bioconductor packages once, then load them every time you start a new R session.

```
source("http://bioconductor.org/biocLite.R")
biocLite("DESeq2")
```

```
library("DESeq2")
citation("DESeq2")
```

It requires the count data to be in matrix form, and an additional dataframe describing sample metadata. Notice that the **colnames of the countdata** match the **rownames of the metadata*.

```
mycoldata <- read.csv("data/coldata.csv", row.names = 1)
mycoldata
```

DESeq works on a particular type of object called a DESeqDataSet. The DESeqDataSet is a single object that contains input values, intermediate calculations like how things are normalized, and all results of a differential expression analysis. You can construct a DESeqDataSet from a count matrix, a metadata file, and a formula indicating the design of the experiment.

```
dds <- DESeqDataSetFromMatrix(countData = countdata, colData = mycoldata, design = ~conc
dds
```

Next, let's run the DESeq pipeline on the dataset, and reassign the resulting object back to the same variable. Before we start, `dds` is a bare-bones DESeqDataSet. The `DESeq()` function takes a DESeqDataSet and returns a DESeqDataSet, but with lots of other information filled in (normalization, results, etc). Here, we're running the DESeq pipeline on the `dds` object, and reassigning the whole thing back to `dds`, which will now be a DESeqDataSet populated with results.

```
dds <- DESeq(dds)
```

Now, let's use the `results()` function to pull out the results from the `dds` object. Let's re-order by the adjusted p-value.

```
# Get differential expression results
res <- results(dds)
head(res)

# Order by adjusted p-value
res <- res[order(res$padj), ]
head(res)
```

Combine DEseq results with the original counts data. Write significant results to a file.

```
sig <- subset(res, padj < 0.05)
dir.create("results")
write.csv(sig, file = "results/sig.csv")  # tab delim data
```

You can open this file in Excel or any text editor (try it now).

Data Visualization

We can also do some exploratory plotting of the data.

The differential expression analysis above operates on the raw (normalized) count data. But for visualizing or clustering data as you would with a microarray experiment, you ned to work with transformed versions of the data. First, use a *regularlized log* transofmration while re-estimating the dispersion ignoring any information you have about the samples (`blind=TRUE`). Perform a principal components analysis and hierarchical clustering.

```
# Transform
rld <- rlogTransformation(dds)

# Principal components analysis
plotPCA(rld, intgroup = "condition")

# Hierarchical clustering analysis let's get the actual values for the first
# few genes
head(assay(rld))
## now transpose those
t(head(assay(rld)))
## now get the sample distances from the transpose of the whole thing
dist(t(assay(rld)))
sampledist <- dist(t(assay(rld)))
plot(hclust(sampledist))
```

Let's plot a heatmap.

```
# ?heatmap for help
sampledist
as.matrix(sampledist)
sampledistmat <- as.matrix(sampledist)
heatmap(sampledistmat)
```

That's a horribly ugly default. You can change the built-in heatmap function, but others are better.

```
# better heatmap with gplots
library("gplots")
heatmap.2(sampledistmat)
heatmap.2(sampledistmat, key = FALSE, trace = "none")
colorpanel(10, "black", "white")
heatmap.2(sampledistmat, col = colorpanel(64, "black", "white"), key = FALSE,
    trace = "none")
```

```
heatmap.2(sampledistmat, col = colorpanel(64, "steelblue", "white"), key = FALSE,
    trace = "none")
heatmap.2(sampledistmat, col = colorpanel(64, "red", "white", "blue"), key = FALSE,
    trace = "none")
```

What about a histogram of the p-values?

```
# Examine plot of p-values
hist(res$pvalue, breaks = 50, col = "grey")
```

Let's plot an MA-plot. This shows the fold change versus the overall expression values.

```
with(res, plot(baseMean, log2FoldChange, pch = 16, cex = 0.5, log = "x"))
with(subset(res, padj < 0.05), points(baseMean, log2FoldChange, col = "red",
    pch = 16))

# optional: label the points with the calibrate package. see ?textxy for
# help
library("calibrate")
res$Gene <- rownames(res)
head(res)
with(subset(res, padj < 0.05), textxy(baseMean, log2FoldChange, labs = Gene,
    cex = 1, col = "red"))
```

Let's create a volcano plot.

```
par(pch = 16)
with(res, plot(log2FoldChange, -log10(pvalue), main = "Volcano plot"))
with(subset(res, padj < 0.05), points(log2FoldChange, -log10(pvalue), col = "red"))
with(subset(res, abs(log2FoldChange) > 2), points(log2FoldChange, -log10(pvalue),
    col = "orange"))
with(subset(res, padj < 0.05 & abs(log2FoldChange) > 2), points(log2FoldChange,
    -log10(pvalue), col = "green"))
# Add legend
legend("topleft", legend = c("FDR<0.05", "|LFC|>2", "both"), pch = 16, col = c("red",
    "orange", "green"))
# Label points
with(subset(res, padj < 0.05 & abs(log2FoldChange) > 2), textxy(log2FoldChange,
    -log10(pvalue), labs = Gene, cex = 1))
```

Record package and version info with `sessionInfo()`

The `sessionInfo()` prints version information about R and any attached packages. It's a good practice to always run this command at the end of your R session and record it for the sake of reproducibility in the future.

```
sessionInfo()
```

```
## R version 3.1.0 (2014-04-10)
## Platform: x86_64-apple-darwin13.1.0 (64-bit)
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## attached base packages:
## [1] parallel  methods   stats     graphics  grDevices utils     datasets
## [8] base
##
## other attached packages:
##  [1] calibrate_1.7.2         MASS_7.3-33
##  [3] gplots_2.14.1           DESeq2_1.4.5
##  [5] RcppArmadillo_0.4.320.0 Rcpp_0.11.2
##  [7] GenomicRanges_1.16.3    GenomeInfoDb_1.0.2
##  [9] IRanges_1.22.9          BiocGenerics_0.10.0
## [11] knitr_1.6               BiocInstaller_1.14.2
##
## loaded via a namespace (and not attached):
##  [1] annotate_1.42.1      AnnotationDbi_1.26.0 Biobase_2.24.0
##  [4] bitops_1.0-6         caTools_1.17         DBI_0.3.1
##  [7] evaluate_0.5.5       formatR_0.10         gdata_2.13.3
## [10] genefilter_1.46.1    geneplotter_1.42.0   grid_3.1.0
## [13] gtools_3.4.1         KernSmooth_2.23-12   lattice_0.20-29
## [16] locfit_1.5-9.1       RColorBrewer_1.0-5   RSQLite_0.11.4
## [19] splines_3.1.0        stats4_3.1.0         stringr_0.6.2
## [22] survival_2.37-7      tools_3.1.0          XML_3.98-1.1
## [25] xtable_1.7-3         XVector_0.4.0
```

Going further

- After the course, download the Integrative Genome Viewer from the Broad Institute. Download all your .bam files from your AWS instance, and load them into IGV. Try navigating to regions around differentially expressed genes to view how reads map to genes differently in the controls versus the irradiated samples.

- Can you see any genes where differential expression is likely attributable to a specific isoform?
- Do you see any instances of differential exon usage? You can investigate this formally with the DEXSeq package.
- Read about pathway analysis with GOSeq or SeqGSEA - tools for gene ontology analysis and gene set enrichment analysis using next-generation sequencing data.
- Read about multifactor designs in the DESeq2 vignette for cases where you have multiple variables of interest (e.g. irradiated vs controls in multiple tissue types).

After the course, make sure you stop any running AWS instances.