

GETTING STARTED

For Windows Users only!

(Mac and Ubuntu users can skip this section and go directly to Unix for Bioinformatics).

It is strongly advised that you start using Linux or Mac computers for all bioinformatics related activity. While there are some windows software and applications available for bioinformatics purposes, most of the programming languages, tools and scripts are natively supported only on linux and mac operating systems. Here we will use some third-party applications to practice linux commands on windows machines.

To emulate linux environment on your windows laptop, you could follow one of the follow two methods.

1. **Install CygWin (recommended)** – Download CygWin from here https://www.cygwin.com/setup-x86_64.exe

What is Cygwin?

Cygwin is a collection of open source tools that allows Unix or Linux applications to be compiled and run on a Microsoft Windows operating system (OS) from within a Linux-like interface. Cygwin offers users a Linux-like experience in a Windows environment. This capability helps developers migrate applications from Unix or Linux to Windows-based systems and makes it easier to support their applications running on the Windows platform.

OR

2. **Run Linux as a subsystem within Windows** – follow instructions from here <https://www.geeksforgeeks.org/how-to-run-linux-commands-on-windows-10/>

What is Windows Subsystem Linux or WSL?

Windows Subsystem For Linux (WSL) is a tool provided by Microsoft to run Linux natively on Windows. It's designed to be a seamless experience, essentially providing a full Linux shell that can interact with your Windows filesystem.

WSL is an optional Windows feature, so you'll need to turn it on.

What next?

Now that you have installed CygWin or WSL, you are all set to practice linux commands on Windows. Just open CygWin program or the WSL Powershell and start your tutorial.

If you are a Linux or Mac user, your computer should already support all the linux commands below, just open the Terminal app and type away!!!

UNIX FOR BIOINFORMATICS

OUTLINE:

1. What is the command line?
2. Directory Structure
3. Syntax of a Command
4. Options of a Command
5. Command Line Basics (ls, pwd, Ctrl-C, man, alias, ls -lthra)
6. Getting Around (cd)
7. Absolute and Relative Paths
8. Tab Completion
9. History Repeats Itself (history, head, tail,)
10. Editing Yourself (Ctrl-A, Ctrl-E, Ctrl-K, Ctrl-W)
11. Create and Destroy (echo, cat, rm, rmdir)
12. Transferring Files (scp)
13. Piping and Redirection (l, >, », cut, sort, grep)
14. Compressions and Archives (tar, gzip, gunzip)
15. Forced Removal (rm -r)
16. BASH Wildcard Characters (?, *, find, environment variables(\$), quotes/ticks)
17. Manipulation of a FASTA file (cp, mv, wc -l/-c)
18. Symbolic Links (ln -s)
19. STDOUT and STDERR (>1, >2)
20. Paste Command (paste, for loops)
21. Shell Scripts and File Permissions (chmod, nano, ./)

WHAT IS UNIX?

UNIX is an operating system which was first developed in the 1960s, and has been under constant development ever since. By operating system, we mean the suite of programs which make the computer work. It is a stable, multi-user, multi-tasking system for servers, desktops and laptops.

UNIX systems also have a graphical user interface (GUI) similar to Microsoft Windows which provides an easy to use environment. However, knowledge of UNIX is required for operations which aren't covered by a graphical program, or for when there is no windows interface available, for example, in a telnet session.

TYPES OF UNIX

There are many different versions of UNIX, although they share common similarities. The most popular varieties of UNIX are Sun Solaris, GNU/Linux, and MacOS X.

THE UNIX OPERATING SYSTEM

The UNIX operating system is made up of three parts; the kernel, the shell and the programs.

The kernel

The kernel of UNIX is the hub of the operating system: it allocates time and memory to programs and handles the filestore and communications in response to system calls.

As an illustration of the way that the shell and the kernel work together, suppose a user types `rm myfile` (which has the effect of removing the file `myfile`). The shell searches the filestore for the file containing the program `rm`, and then requests the kernel, through system calls, to execute the program `rm` on `myfile`. When the process `rm myfile` has finished running, the shell then returns the UNIX prompt `%` to the user, indicating that it is waiting for further commands.

The shell

The shell acts as an interface between the user and the kernel. When a user logs in, the login program checks the username and password, and then starts another program called the shell. The shell is a command line interpreter (CLI). It interprets the commands the user types in and arranges for them to be carried out. The commands are themselves programs: when they terminate, the shell gives the user another prompt (`%` on our systems).

The adept user can customise his/her own shell, and users can use different shells on the same machine. This shell is an all-text display (most of the time your mouse doesn't work) and is accessed using an application called the "terminal" which usually looks like a black window with white letters or a white window with black letters by default.

The `tcsh` shell has certain features to help the user inputting commands.

Filename Completion - By typing part of the name of a command, filename or directory and pressing the [Tab] key, the `tcsh` shell will complete the rest of the name automatically. If the shell finds more than one name beginning with those letters you have typed, it will beep, prompting you to type a few more letters before pressing the tab key again.

History - The shell keeps a list of the commands you have typed in. If you need to repeat a command, use the cursor keys to scroll up and down the list or type `history` for a list of previous commands.

Files and processes

Everything in UNIX is either a file or a process.

A process is an executing program identified by a unique PID (process identifier).

A file is a collection of data. They are created by users using text editors, running compilers etc.

Examples of files:

- a document (report, essay etc.)
- the text of a program written in some high-level programming language

- instructions comprehensible directly to the machine and incomprehensible to a casual user, for example, a collection of binary digits (an executable or binary file);
- a directory, containing information about its contents, which may be a mixture of other directories (subdirectories) and ordinary files.

THE DIRECTORY STRUCTURE

All the files are grouped together in the directory structure. The file-system is arranged in a hierarchical structure, like an inverted tree. The top of the hierarchy is traditionally called root (written as a slash /)

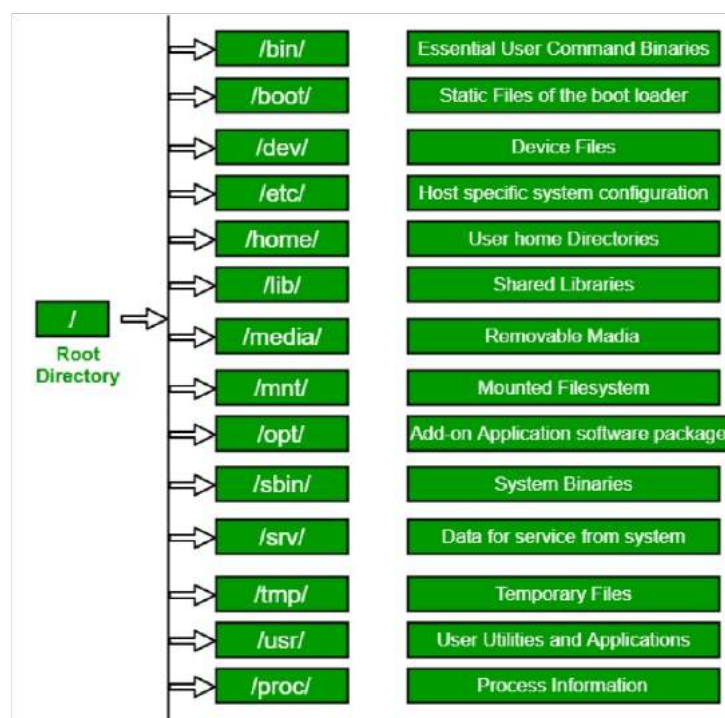
Absolute path: always starts with "/" - the root folder

[/home/ubuntu/workspace](#)

the folder (or file) "workspace" in the folder "ubuntu" or "student #1" in the folder "home" in the folder from the root.

Relative path: always relative to our current location.

a single dot (.) refers to the current directory two dots (..) refers to the directory one level up.



Usually, /home is where the user accounts reside, ie. users' 'home' directories. For example, for a user that has a username of "student #1": their home directory is /home/student1. It is the directory that a user starts in after starting a new shell or logging into a remote server.

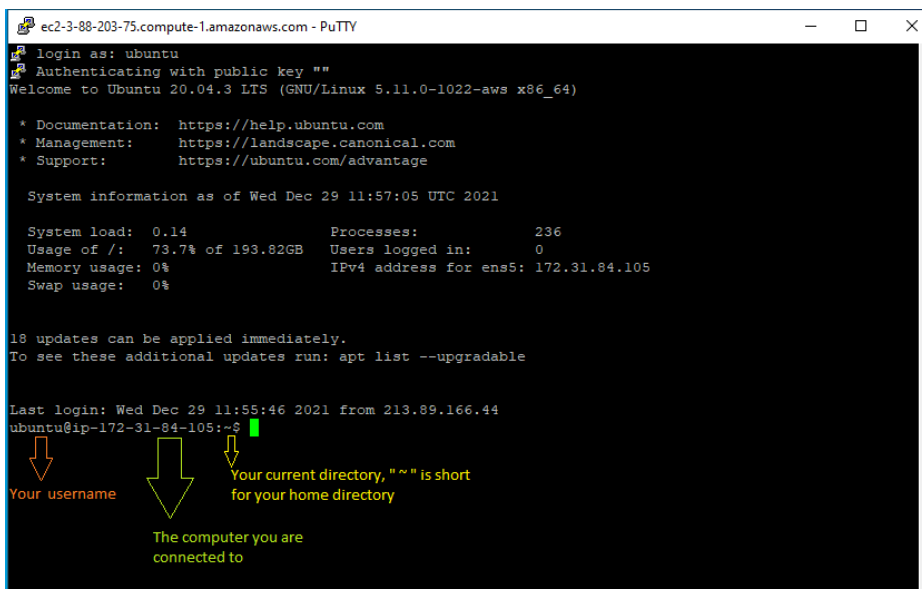
The tilde (~) is a short form of a user's home directory.

STARTING AN UNIX TERMINAL



After opening or logging into a terminal, system messages are often displayed, followed by the “prompt”. A prompt is a short text message at the start of the command line and ends with a \$ in bash shell, commands are typed after the prompt. The prompt typically follows the form `username@server:current_directory $`.

If your screen looks like the one below, i.e. you see a bunch of messages and then your ubuntu student instance number followed by “@172.31.84.105:~\$” at the beginning of the line, then you are successfully logged in.



UNIX BASICS

First some basics - how to look at your surroundings (PRESENT WORKING DIRECTORY... WHERE AM I?)

`pwd`

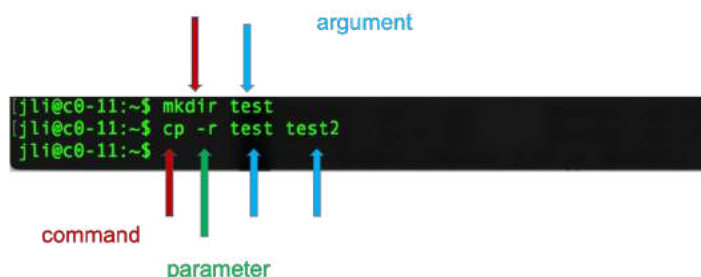
see the words separated by /. this is called the path. In unix, the location of each folder or file is shown like this.. this is more like the address or the way to find a folder or file.

for example, my Desktop, Documents and Downloads folder usually are located in my home directory so in that case, it path to the folders will be

```
/home/ubuntu/Desktop  
/home/ubuntu/Documents  
/home/ubuntu/Downloads
```

SYNTAX OF A COMMAND

A command plus the required parameters/arguments The separator used in issuing a command is space, number of spaces does not matter



Now let's try some basic and simple commands

list files here... you should see just the folders that we have created for you here and nothing else.

```
ls  
list files somewhere else, like /tmp/  
ls /tmp
```

TIP!

In unix one of the first things that's good to know is how to escape once you've started something you don't want. Use Ctrl-c (shows as '^C' in the terminal) to exit (kill) a command. In some cases, a different key sequence is required (Ctrl-d). Note that anything including and after a "#" symbol is ignored, i.e. a comment. So in all the commands below, you do not have to type anything including and past a "#".

Options

Each command can act as a basic tool, or you can add 'options' or 'flags' that modify the default behavior of the tool. These flags come in the form of '-v' ... or, when it's a more descriptive word, two dashes: '--verbose' ... that's a common (but not universal) one that tells a tool that you want it to give you output with more detail. Sometimes, options require specifying amounts or strings, like '-o results.txt' or '--output results.txt' ... or '-n 4' or '--numCPUs 4'. Let's try some, and see what the man page for the 'list files' command 'ls' is like.

```
ls -R
```

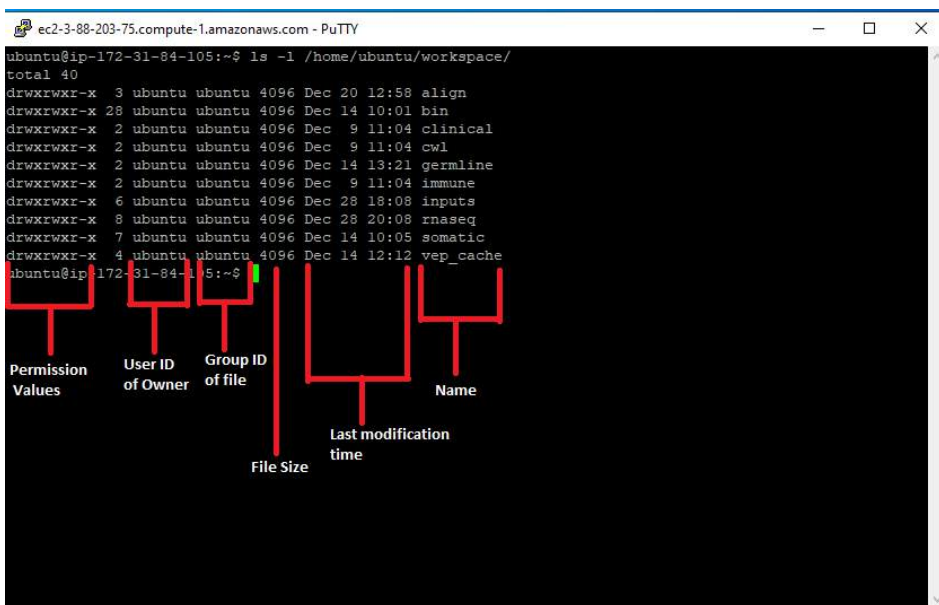
Lists directories and files recursively. This will be a very long output, so use Ctrl-C to break out of it. Sometimes you have to press Ctrl-C many times to get the terminal to recognize it. In order to know which options do what, you can use the manual pages. To look up a command in the manual pages type “man” and then the command name. So to look up the options for “ls”, type:

```
man ls
```

Navigate this page using the up and down arrow keys, PageUp and PageDown, and then use q to quit out of the manual. In this manual page, find the following options, quit the page, and then try those commands. You could even open another terminal, log in again, and run manual commands in that terminal.

```
ls -l /usr/bin/
```

#long format, gives permission values, owner, group, size, modification time, and name



```
ubuntu@ip-172-31-84-105:~$ ls -l /home/ubuntu/workspace/
total 40
drwxrwxr-x 3 ubuntu ubuntu 4096 Dec 20 12:58 align
drwxrwxr-x 28 ubuntu ubuntu 4096 Dec 14 10:01 bin
drwxrwxr-x 2 ubuntu ubuntu 4096 Dec 9 11:04 clinical
drwxrwxr-x 2 ubuntu ubuntu 4096 Dec 9 11:04 cwl
drwxrwxr-x 2 ubuntu ubuntu 4096 Dec 14 13:21 germline
drwxrwxr-x 2 ubuntu ubuntu 4096 Dec 9 11:04 immune
drwxrwxr-x 6 ubuntu ubuntu 4096 Dec 28 18:08 inputs
drwxrwxr-x 8 ubuntu ubuntu 4096 Dec 28 20:08 rnaseq
drwxrwxr-x 7 ubuntu ubuntu 4096 Dec 14 10:05 somatic
drwxrwxr-x 4 ubuntu ubuntu 4096 Dec 14 12:12 vep_cache
ubuntu@ip-172-31-84-105:~$
```

Permission Values

User ID of Owner

Group ID of file

File Size

Last modification time

Name

Exercise:

Feel free to see manual pages for these basic commands

Commands and their Meanings

```
man ls
ls -a
man mkdir
```

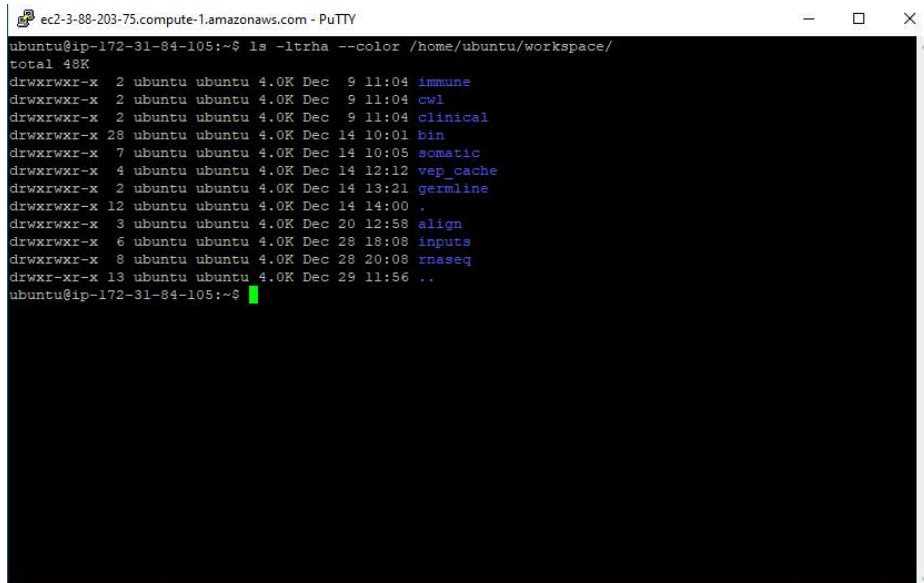
Now see the difference between these three commands

```
cd
cd ~
cd ..  change to parent directory
```

Also these commands

```
ls -l
ls -a
ls -l -a
```

```
ls -la
ls -ltrha
ls -ltrha --color
```



```
ec2-3-88-203-75.compute-1.amazonaws.com - PuTTY
ubuntu@ip-172-31-84-105:~$ ls -ltrha --color /home/ubuntu/workspace/
total 48K
drwxrwxr-x 2 ubuntu ubuntu 4.0K Dec  9 11:04 immune
drwxrwxr-x 2 ubuntu ubuntu 4.0K Dec  9 11:04 cwl
drwxrwxr-x 2 ubuntu ubuntu 4.0K Dec  9 11:04 clinical
drwxrwxr-x 28 ubuntu ubuntu 4.0K Dec 14 10:01 bin
drwxrwxr-x 7 ubuntu ubuntu 4.0K Dec 14 10:05 somatic
drwxrwxr-x 4 ubuntu ubuntu 4.0K Dec 14 12:12 vep_cache
drwxrwxr-x 2 ubuntu ubuntu 4.0K Dec 14 13:21 germline
drwxrwxr-x 12 ubuntu ubuntu 4.0K Dec 14 14:00 .
drwxrwxr-x 3 ubuntu ubuntu 4.0K Dec 20 12:58 align
drwxrwxr-x 6 ubuntu ubuntu 4.0K Dec 28 18:08 inputs
drwxrwxr-x 8 ubuntu ubuntu 4.0K Dec 28 20:08 rnaseq
drwxr-xr-x 13 ubuntu ubuntu 4.0K Dec 29 11:56 ..
ubuntu@ip-172-31-84-105:~$
```

Quick aside: what if I want to use same options repeatedly? and be lazy? You can create a shortcut to another command using 'alias'.

```
alias ll='ls -lah'
ll
```

GETTING AROUND

The filesystem you're working on is like the branching root system of a tree. The top level, right at the root of the tree, is called the 'root' directory, specified by '/' ... which is the divider for directory addresses, or 'paths'.

Now let's see a little about the commands you checked in the exercise above.

We move around using the 'change directory' command, 'cd'. The command pwd return the present working directory.

```
cd # no effect? that's because by itself it sends you home (to ~)
cd / # go to root of tree's root system
cd home # go to where everyone's homes are
pwd
cd username # use your actual home, not "username"
pwd
cd /
pwd
cd ~ # a shortcut to home, from anywhere
pwd
cd . # '.' always means *this* directory
pwd
cd .. # '..' always means *one directory up*
pwd
```


ABSOLUTE AND RELATIVE PATHS

You can think of paths like addresses. You can tell your friend how to go to a particular store from where they are currently (a 'relative' path), or from the main Interstate Highway that everyone uses (in this case, the root of the filesystem, '/' ... this is an 'absolute' path). Both are valid. But absolute paths can't be confused, because they always start off from the same place, and are unique. Relative paths, on the other hand, could be totally wrong for your friend if you assume they're somewhere they're not. With this in mind, let's try a few more:

```
cd /usr/bin # let's start in /usr/bin
#relative (start here, take one step up, then down through lib and gcc)
```

```
cd ../lib/init/
pwd
#absolute (start at root, take steps)
```

```
cd /usr/lib/init/
pwd
```

Now, because it can be a real pain to type out, or remember these long paths, we need to discuss ...

TAB COMPLETION

Using tab-completion is a must on the command line. A single <tab> auto-completes file or directory names when there's only one name that could be completed correctly. If multiple files could satisfy the tab-completion, then nothing will happen after the first <tab>. In this case, press <tab> a second time to list all the possible completing names. Note that if you've already made a mistake that means that no files will ever be completed correctly from its current state, then <tab>'s will do nothing.

touch updates the timestamp on a file, here we use it to create three empty files.

```
cd # go to your home directory
mkdir ~/tmp
cd ~/tmp
touch one seven september
ls o
tab with no enter should complete to 'one', then enter
ls s
```

tab with no enter completes up to 'se' since that's in common between seven and september. tab again and no enter, this second tab should cause listing of seven and september. type 'v' then tab and no enter now it's unique to seven, and should complete to seven. enter runs 'ls seven' command.

It cannot be overstated how useful tab completion is. You should get used to using it constantly. Watch experienced programmers type and they maniacally hit tab once or twice in between almost every character. You don't have to go that far, of course, but get used to constantly getting feedback from hitting tab and you will save yourself a huge amount of typing and trying to remember weird directory and filenames.

TIME TO SHIFT GEARS AND PICK UP SOME SPEED NOW!

HISTORY REPEATS ITSELF

Linux remembers everything you've done (at least in the current shell session), which allows you to pull steps from your history, potentially modify them, and redo them. This can obviously save a lot of time and typing.

The 'head' command views the first 10 (by default) lines of a file. The 'tail' command views the last 10 (by default) lines of a file. Type 'man head' or 'man tail' to consult their manuals.

```
<up arrow> # last command
<up> # next-to-last command
<down> # last command, again
<down> # current command, empty or otherwise
history # usually too much for one screen, so ...
history | head # we discuss pipes (the vertical bar) below
history | tail
history | less # use 'q' to exit less
ls -l
pwd
history | tail
!560 # re-executes 560th command (yours will have different numbers; choose the
one that recreates your really important result!)
```

EDITING YOURSELF

Here are some more ways to make editing previous commands, or novel commands that you're building up, easier:

```
<up><up> # go to some previous command, just to have something to work on
<ctrl-a> # go to the beginning of the line
<ctrl-e> # go to the end of the line
#now use left and right to move to a single word (surrounded by whitespace:
spaces or tabs)
<ctrl-k> # delete from here to end of line
<ctrl-w> # delete from here to beginning of preceeding word
blah blah blah<ctrl-w><ctrl-w> # leaves you with only one 'blah'
```

You can also search your history from the command line:

```
<ctrl-r>fir # should find most recent command containing 'fir' string: echo
'first' > test.txt
<enter> # to run command
<ctrl-c> # get out of recursive search
<ctr-r> # repeat <ctrl-r> to find successively older string matches
```

CREATE AND DESTROY

We already learned one command that will create a file, touch. Now let's look at create and removing files and directories.

```
cd # home again
mkdir ~/tmp2
cd ~/tmp2
echo 'Hello, world!' > first.txt
```

echo text then redirect ('>') to a file.

```
cat first.txt # 'cat' means 'concatenate', or just spit the contents of the file to the screen
```

WHY 'CONCATENATE'? TRY THIS:

```
cat first.txt first.txt first.txt > second.txt
cat second.txt
```

OK, let's destroy what we just created:

```
cd ../
rmdir tmp2 # 'rmdir' means 'remove directory', but this shouldn't work!
rm tmp2/first.txt
rm tmp2/second.txt # clear directory first
rmdir tmp2 # should succeed now
```

So, 'mkdir' and 'rmdir' are used to create and destroy (empty) directories. 'rm' to remove files. To create a file can be as simple as using 'echo' and the '>' (redirection) character to put text into a file. Even simpler is the 'touch' command.

```
mkdir ~/cli
cd ~/cli
touch newFile
ls -ltra # look at the time listed for the file you just created
cat newFile # it's empty!
sleep 60 # go grab some coffee
touch newFile
ls -ltra # same time?
```

So 'touch' creates empty files, or updates the 'last modified' time. Note that the options on the 'ls' command you used here give you a Long listing, of All files, in Reverse Time order (l, a, r, t).

FORCED REMOVAL (CAUTION!!!)

When you're on the command line, there's no 'Recycle Bin'. Since we've expanded a whole directory tree, we need to be able to quickly remove a directory without clearing each subdirectory and using 'rmdir'.

```
cd
mkdir -p rmtest/dir1/dir2 # the -p option creates all the directories at once
rmdir rmtest # gives an error since rmdir can only remove directories that are empty
rm -rf rmtest # will remove the directory and EVERYTHING in it
```

Here -r = recursively remove sub-directories, -f means force. **Obviously, be careful with 'rm -rf', there is no going back**, if you delete something with rm, rmdir its gone! **There is no Recycle Bin on the Command-Line!**

PIPING AND REDIRECTION

Pipes ('|') allow commands to hand output to other commands, and redirection characters ('>' and '»') allow you to put output into files.

```
echo 'first' > test.txt
cat test.txt # outputs the contents of the file to the terminal
echo 'second' > test.txt
```

```
cat test.txt
echo 'third' >> test.txt
cat test.txt
```

The ‘>’ character redirects output of a command that would normally go to the screen instead into a specified file. ‘>’ overwrites the file, ‘>>’ appends to the file.

The ‘cut’ command pieces of lines from a file line by line. This command cuts characters 1 to 3, from every line, from file ‘test.txt’

```
cut -c 1-3 test.txt
```

same thing, piping output of one command into input of another

```
cat test.txt | cut -c 1-3
```

This pipes (i.e., sends the output of) cat to cut to sort (-r means reverse order sort), and then grep searches for pattern (‘s’) matches (i.e. for any line where an ‘s’ appears anywhere on the line)

```
cat test.txt | cut -c 1-3 | sort -r
cat test.txt | cut -c 1-3 | sort -r | grep s
```

This is a great way to build up a set of operations while inspecting the output of each step in turn. We’ll do more of this in a bit.

COMPRESSION AND ARCHIVES

As file sizes get large, you’ll often see compressed files, or whole compressed folders. Note that any good bioinformatics software should be able to work with compressed file formats.

```
gzip test.txt
cat test.txt.gz
To uncompress a file
gunzip -c test.txt.gz
```

The ‘-c’ leaves the original file alone, but dumps expanded output to screen

```
gunzip test.txt.gz # now the file should change back to uncompressed test.txt
```

Tape archives, or .tar files, are one way to compress entire folders and all contained folders into one file. When they’re further compressed they’re called ‘tarballs’. We can use wget (web get) to download the file.

```
wget -L -O PhiX_Illumina_RTA.tar.gz http://igenomes.illumina.com.s3-website-us-east-1.amazonaws.com/PhiX/Illumina/RTA/PhiX_Illumina_RTA.tar.gz
```

The .tar.gz and .tgz are commonly used extensions for compressed tar files, when gzip compression is used. The application tar is used to uncompress .tar files

```
tar -xzvf PhiX_Illumina_RTA.tar.gz
```

Here -x = extract, -z = use gzip/gunzip, -v = verbose (show each file in archive), -f filename

Note that, unlike Windows, linux does not depend on file extensions to determine file behavior. So you could name a tarball ‘fish.puppy’ and the extract command above should work just fine. The only thing that should

be different is that tab-completion doesn't work within the 'tar' command if it doesn't see the 'correct' file extension.

BASH WILDCARD CHARACTERS

We can use 'wildcard characters' when we want to specify or operate on sets of files all at once.

```
ls ?hiX/Illumina
```

list files in Illumina sub-directory of any directory ending in 'hiX'

```
ls PhiX/Illumina/RTA/Sequence/*/*.fa
```

list all files ending in '.fa' a few directories down. So, '?' fills in for zero or one character, '*' fills in for zero or more characters. The 'find' command can be used to locate files using a similar form.

```
find . -name "*.f*"
find . -name "*.f?"
```

how is this different from the previous ls commands?

Quick Note About the Quote(s)

The quote characters " and ' are different. In general, single quotes preserve the literal meaning of all characters between them. On the other hand, double quotes allow the shell to see what's between them and make substitutions when appropriate. For example:

```
VRBL=someText
echo '$VRBL'
echo "$VRBL"
```

However, some commands try to be 'smarter' about this behavior, so it's a little hard to predict what will happen in all cases. It's safest to experiment first when planning a command that depends on quoting ... list filenames first, instead of changing them, etc. Finally, the 'backtick' characters ` (same key - unSHIFTED - as the tilde ~) causes the shell to interpret what's between them as a command, and return the result.

Counts the number of lines in file and stores result in the lines variable

```
LINES=`cat PhiX/Illumina/RTA/Sequence/Bowtie2Index/genome.1.bt2 | wc -l`
echo $LINES
```

SYMBOLIC LINKS

Since copying or even moving large files (like sequence data) around your filesystem may be impractical, we can use links to reference 'distant' files without duplicating the data in the files. Symbolic links are disposable pointers that refer to other files, but behave like the referenced files in commands. I.e., they are essentially 'Shortcuts' (to use a Windows term) to a file or directory.

The 'ln' command creates a link. You should, by default, always create a symbolic link using the -s option.

```
ln -s PhiX/Illumina/RTA/Sequence/WholeGenomeFasta/genome.fa .
ls -ltrhaF # notice the symbolic link pointing at its target
grep -c ">" genome.fa
```

STDOUT & STDERR

Programs can write to two separate output streams, ‘standard out’ (STDOUT), and ‘standard error’ (STDERR). The former is generally for direct output of a program, while the latter is supposed to be used for reporting problems. I’ve seen some bioinformatics tools use STDERR to report summary statistics about the output, but this is probably bad practice. Default behavior in a lot of cases is to dump both STDOUT and STDERR to the screen, unless you specify otherwise. In order to nail down what goes where, and record it for posterity:

```
wc -c genome.fa 1> chars.txt 2> any.err
```

the 1st output, STDOUT, goes to ‘chars.txt’ the 2nd output, STDERR, goes to ‘any.err’

```
cat chars.txt
```

Contains the character count of the file genome.fa

```
cat any.err
```

Empty since no errors occurred.

Saving STDOUT is pretty routine (you want your results, yes?), but remember that explicitly saving STDERR is important on a remote server, since you may not directly see the ‘screen’ when you’re running jobs.

THE SED COMMAND

Let’s take a look at the ‘sed’ command. NOTE: On Macs use ‘gsed’. sed (short for stream editor) is a command that allows you to manipulate character data in various ways. One useful thing it can do is substitution. Let’s download a simple file to work on:

```
wget https://course-cg-5534.s3.amazonaws.com/unix_exercise/region.bed -O region.bed
```

Take a look at the file:

```
cat region.bed
```

Now, let’s make all the uppercase “CHR”s into lowercase:

```
cat region.bed | sed 's/CHR/chr/'
```

What happened? Only the first CHR changed. That is because we need to add the “g” option:

```
cat region.bed | sed 's/CHR/chr/g'
```

We can also do the the substitution without regards to case:

```
cat region.bed | sed 's/chr/chr/gi'
```

Let’s break down the argument to sed (within the single quotes)... The “s” means “substitute”, the word between the 1st and 2nd forward slashes (i.e. /) is the word the substitute for, the word between the 2nd and 3rd slashes is the word to substitute with, and finally the “gi” at the end are flags for global substitution (i.e.

substituting along an entire line instead of just the first occurrence on a line), and for case insensitivity (i.e. it will ignore the case of the letters when doing the substitution).

Note that this doesn't change the file itself, it is simply piping the output of the cat command to sed and outputting to the screen. If you wanted to change the file itself, you could use the "-i" option to sed:

```
cat region.bed  
sed -i 's/chr/chr/gi' region.bed
```

Now if you look at the file, the lines have changed.

```
cat region.bed
```

Another useful use of sed is for capturing certain lines from a file. You can select certain lines from a file:

```
sed '4q;d' region.bed
```

This will just select the 4th line from the file.

You can also extract a range of lines from a file:

```
sed -n '10,20p' region.bed
```

This gets the 10th through 20th lines from the file.

CHALLENGE: See if you can find a way to use sed to remove all the "CHR"s from the file.

MORE PIPES

Now, let's delve into pipes a little more. Pipes are a very powerful way to look at and manipulate complex data using a series of simple programs. Let's look at some fastq files. Get a few small fastq files:

```
wget https://course-cg-5534.s3.amazonaws.com/unix_exercise/C61.subset.fq.gz -O  
C61.subset.fq.gz  
wget https://course-cg-5534.s3.amazonaws.com/unix_exercise/I561.subset.fq.gz -O  
I561.subset.fq.gz  
wget https://course-cg-5534.s3.amazonaws.com/unix_exercise/I894.subset.fq.gz -O  
I894.subset.fq.gz
```

Since the files are gzipped files we need to use "zcat" to look at them. zcat is just like cat except for gzipped files:

```
zcat C61.subset.fq.gz | head
```

Fastq records are 4 lines per sequence, a header line, the sequence, a plus sign (which is historical), and then the quality encoding for the sequence. Notice that each header line has the barcode for that read at the end of the line. Let's count the number of each barcode. In order to do that we need to just capture the header lines from this file. We can use "sed" to do that:

```
zcat C61.subset.fq.gz | sed -n '1~4p' | head
```

By default sed prints every line. In this case we are giving the "-n" option to sed which will not print every line. Instead, we are giving it the argument "1~4p", which means to print the first line, then skip 4 lines and print again, and then continue to do that.

Now that we have a way to get just the headers, we need to isolate the part of the header that is the barcode. There are multiple ways to do this... we will use the cut command:

```
zcat C61.subset.fq.gz | sed -n '1~4p' | cut -d: -f10 | head
```

So we are using the “-d” option to cut with “:” as the argument to that option, meaning that we will be using the delimiter “:” to split the input. Then we use the “-f” option with argument “10”, meaning that we want the 10th field after the split. In this case, that is the barcode.

Finally, as before, we need to sort the data and then use “uniq -c” to count. Then put it all together and run it on the entire dataset (This will take about a minute to run):

```
zcat C61.subset.fq.gz | sed -n '1~4p' | cut -d: -f10 | sort | uniq -c
```

Now you have a list of how many reads were categorized into each barcode. Here is a sed tutorial for more exercises.

CHALLENGE: Find the distribution of the first 5 bases of all the reads in C61_S67_L006_R1_001.fq.gz. i.e., count the number of times the first 5 bases of every read occurs across all reads.

LOOPS

Loops are useful for quickly telling the shell to perform one operation after another, in series. For example:

```
for i in {1..21}; do echo $i >> a; done # put multiple lines of code on one
line, each line terminated by ';'
cat a
<1 THROUGH 21 ON SEPARATE LINES>
```

The general form is:

```
for name in {list}; do
    commands
done
```

The list can be a sequence of numbers or letters, or a group of files specified with wildcard characters:

```
for i in {3,2,1,liftoff}; do echo $i; done # needs more excitement!
for i in {3,2,1,"liftoff!"}; do echo $i; done # exclamation point will confuse
the shell unless quoted
```

A “while” loop is more convenient than a “for” loop ... if you don’t readily know how many iterations of the loop you want:

```
while {condition}; do
    commands
done
```

Now, let’s do some bioinformatics-y things with loops and pipes. First, let’s write a command to get the nucleotide count of the first 10,000 reads in a file. Use zcat and sed to get only the read lines of a file, and then only take the first 10,000:

```
zcat C61.subset.fq.gz | sed -n '2~4p' | head -10000 | less
```


Use grep's "-o" option to get each nucleotide on a separate line (take a look at the man page for grep to understand how this works):

```
zcat C61.subset.fq.gz | sed -n '2~4p' | head -10000 | grep -o . | less
```

Finally, use sort and uniq to get the counts:

```
zcat C61.subset.fq.gz | sed -n '2~4p' | head -10000 | grep -o . | sort | uniq -c

264012 A
243434 C
215045 G
278 N
277231 T
```

And, voila, we have the per nucleotide count for these reads!

We just did this for one file, but what if we wanted to do it for all of our files? We certainly don't want to type the command by hand dozens of times. So we'll use a while loop. You can pipe a command into a while loop and it will iterate through each line of the input. First, get a listing of all your files:

```
ls -l *.fq.gz
```

Pipe that into a while loop and read in the lines into a variable called "x". We use "\$x" to get the value of the variable in that iteration of the loop:

```
ls -l *.fq.gz | while read x; do echo $x is being processed...; done
```

Add the command we created above into the loop, placing \$x where the filename would be and semi-colons inbetween commands:

```
ls -l *.fq.gz | while read x; do echo $x is being processed...; zcat $x | sed -n '2~4p' | head -10000 | grep -o . | sort | uniq -c; done
```

When this runs it will print the name of every single file being processed and the nucleotide count for the reads from those files.

Now, let's say you wanted to write the output of each command to a separate file. We would redirect the output to a filename, but we need to create a different file name for each command and we want the file name to reflect its contents, i.e. the output file name should be based on the input file name. So we use "parameter expansion", which is fancy way of saying substitution:

```
ls -l *.fq.gz | while read x; do echo $x is being processed...; zcat $x | sed -n '2~4p' | head -10000 | grep -o . | sort | uniq -c > ${x%.fq.gz}.nucl_count.txt; done
```

This will put the output of the counting command into a file whose name is the prefix of the input file plus ".nucl_count.txt". It will do this for every input file.

MANIPULATION OF A FASTA FILE

Let's copy the phiX-174 genome (using the 'cp' command) to our current directory so we can play with it:

```
cp ./PhiX/Illumina/RTA/Sequence/WholeGenomeFasta/genome.fa phix.fa
```

Similarly we can also use the move command here, but then
./PhiX/Illumina/RTA/Sequence/WholeGenomeFasta/genome.fa will no longer be there:

```
cp ./PhiX/Illumina/RTA/Sequence/WholeGenomeFasta/genome.fa  
./PhiX/Illumina/RTA/Sequence/WholeGenomeFasta/genome2.fa  
ls ./PhiX/Illumina/RTA/Sequence/WholeGenomeFasta/  
mv ./PhiX/Illumina/RTA/Sequence/WholeGenomeFasta/genome2.fa phix.fa  
ls ./PhiX/Illumina/RTA/Sequence/WholeGenomeFasta/
```

This functionality of mv is why it is used to rename files.

Note how we copied the 'genome.fa' file to a different name: 'phix.fa'

```
wc -l phix.fa
```

count the number of lines in the file using 'wc' (word count) and parameter '-l' (lines).

We can use the 'grep' command to search for matches to patterns. 'grep' comes from 'globally search for a regular expression and print'.

```
grep -c '>' phix.fa
```

#Only one FASTA sequence entry, since only one header line ('>gilsomethingsomething...')

```
cat phix.fa
```

This may not be useful for anything larger than a virus! Let's look at the start codon and the two following codons:

```
grep --color "ATG....." phix.fa
```

#'.' characters are the single-character wildcards for grep. So "ATG....." matches any set of 9 characters that starts with ATG.

#Use the -color '-o' option to only print the pattern matches, one per line

```
grep -o "ATG....." phix.fa
```

#Use the 'cut' command with '-c' to select characters 4-6, the second codon

```
grep --color -o "ATG....." phix.fa | cut -c4-6
```

'sort' the second codon sequences (default order is same as ASCII table; see 'man ascii')

```
grep --color -o "ATG....." phix.fa | cut -c4-6 | sort
```

#Combine successive identical sequences, but count them using the 'uniq' command with the '-c' option

```
grep --color -o "ATG....." phix.fa | cut -c4-6 | sort | uniq -c
```

#Finally sort using reverse numeric order ('-rn')

```
grep --color -o "ATG....." phix.fa | cut -c4-6 | sort | uniq -c | sort -rn
```

... which gives us the most common codons first

This may not be a particularly useful thing to do with a genomic FASTA file, but it illustrates the process by which one can build up a string of operations, using pipes, in order to ask quantitative questions about sequence content. More generally than that, this process allows one to ask questions about files and file contents and the operating system, and verify at each step that the process so far is working as expected. The command line is, in this sense, really a modular workflow management system.

SHELL SCRIPTS, FILE PERMISSIONS

Often it's useful to define a whole string of commands to run on some input, so that (1) you can be sure you're running the same commands on all data, and (2) so you don't have to type the same commands in over and over! Let's use the 'nano' text editor program that's pretty reliably installed on most linux systems.

```
nano test.sh
```

nano now occupies the whole screen; see commands at the bottom. Let's type in a few commands. First we need to put the following line at the top of the file:

```
#!/bin/bash
```

The “#!” at the beginning of a script tells the shell what language to use to interpret the rest of the script. In our case, we will be writing “bash” commands, so we specify the full path of the bash executable after the “#!”. Then, add some commands:

```
#!/bin/bash  
  
echo "Start script..."  
pwd  
ls -l  
sleep 10  
echo "End script."
```

Hit Ctrl-X and then enter Y to save the file and exit nano.

Though there are ways to run the commands in test.sh right now, it's generally useful to give yourself (and others) 'execute' permissions for test.sh, really making it a shell script. Note the characters in the first (left-most) field of the file listing:

```
ls -lh test.sh  
  
-rw-rw-r-- 1 ubuntu workspace 79 Dec 19 15:05 test.sh
```

The first '-' becomes a 'd' if the 'file' is actually a directory. The next three characters represent read, write, and execute permissions for the file owner (you), followed by three characters for users in the owner's group, followed by three characters for all other users. Run the 'chmod' command to change permissions for the 'test.sh' file, adding execute permissions ('+x') for the user (you) and your group ('ug'):

```
chmod ug+x test.sh  
ls -lh test.sh  
  
-rwxr-xr-- 1 ubuntu workspace 79 Dec 19 15:05 test.sh
```

The first 10 characters of the output represent the file and permissions. The first character is the file type, the next three sets of three represent the file permissions for the user, group, and everyone respectively.

r = read w = write x = execute So let's run this script. We have to provide a relative reference to the script './' because its not our our "PATH".:

#you can do either

```
./test.sh
```

#or you can run it like this

```
bash test.sh
```

And you should see all the commands in the file run in sequential order in the terminal.

COMMAND LINE ARGUMENTS FOR SHELL SCRIPTS

Now let's modify our script to use command line arguments, which are arguments that can come after the script name (when executing) to be part of the input inside the script. This allows us to use the same script with different inputs. In order to do so, we add variables \$1, \$2, \$3, etc.... in the script where we want our input to be. So, for example, use nano to modify your test.sh script to look like this:

```
#!/bin/bash

echo "Start script..."
PWD=`pwd`
echo "The present working directory is $PWD"
ls -l $1
sleep $2
wc -l $3
echo "End script."
```

Now, rerun the script using command line arguments like this:

```
./test.sh genome.fa 15 PhiX/Illumina/RTA/Annotation/Archives/archive-2013-03-06-19-09-31/Genes/ChromInfo.txt
```

Note that each argument is separated by a space, so \$1 becomes "genome.fa", \$2 becomes "15", and \$3 becomes "PhiX/Illumina/RTA/Annotation/Archives/archive-2013-03-06-19-09-31/Genes/ChromInfo.txt". Then the commands are run using those values. Now rerun the script with some other values:

```
./test.sh .. 5 genome.fa
```

Now, \$1 becomes "..", \$2 is "5", and \$3 is "genome.fa".

PIPES AND LOOPS INSIDE SCRIPTS

Open a new text file using the text editor "nano":

```
nano get_nucl_counts.sh
```

Copy and Paste the following into the file:

```
#!/bin/bash

zcat $1 | sed -n '2~4p' | head -$2 | grep -o . | sort | uniq -c
```

Save the file and exit. Change the permissions on the file to make it executable:

```
chmod a+x get_nucl_counts.sh
```

Now, we can run this script giving it different arguments every time. The first argument (i.e. the first text after the script name when it is run) will get put into the variable “\1”. The second argument (delimited by spaces) will get put into “\2”. In this case, “\1” is the file name, and “\2” is the number of reads we want to count. So, then we can run the script over and over again using different values and the command will run based on those values:

```
./get_nucl_counts.sh I561.subset.fq.gz 1000
./get_nucl_counts.sh I561.subset.fq.gz 10000
./get_nucl_counts.sh C61.subset.fq.gz 555
```

We can also put loops into a script. We’ll take the loop we created earlier and put it into a file, breaking it up for readability and using backslashes for line continuation:

```
nano get_nucl_counts_loop.sh
```

Put this in the file and save it:

```
#!/bin/bash

ls -l *.fq.gz | \
while read x; do \
    echo $x is being processed...; \
    zcat $x | sed -n '2~4p' | head -$1 | \
    grep -o . | sort | uniq -c > ${x%.fq.gz}.nucl_count.txt; \
done
```

Make it executable:

```
chmod a+x get_nucl_counts_loop.sh
```

And now we can execute the entire loop using the script. Note that there is only one argument now, the number of reads to use:

```
./get_nucl_counts_loop.sh 100
```

FINALLY - A GOOD SUMMARY FOR ROUNDING UP THE TUTORIAL

[TEN SIMPLE RULES FOR GETTING STARTED WITH COMMAND-LINE BIOINFORMATICS](#)

A SIMPLE UNIX CHEAT SHEET

[Download Unix Cheat Sheet](#)

AN INTRODUCTION ON HOW TO ANALYZE TABLE DATA WITHOUT MS EXCEL

AWK CRASH COURSE FOR BIOINFORMATICS

WHAT IS AWK?

AWK is an interpreted programming language designed for text processing and typically used as a data extraction and reporting tool.

The AWK language is a data-driven scripting language consisting of a set of actions to be taken against streams of textual data - either run directly on files or used as part of a pipeline - for purposes of extracting or transforming text, such as producing formatted reports. The language extensively uses the string datatype, associative arrays (that is, arrays indexed by key strings), and regular expressions.

AWK has a limited intended application domain, and was especially designed to support one-liner programs.

It is a standard feature of most Unix-like operating systems.

source: [Wikipedia](#)

WHY AWK?

You can replace a pipeline of 'stuff | grep | sed | cut...' with a single call to awk. For a simple script, most of the timelag is in loading these apps into memory, and it's much faster to do it all with one. This is ideal for something like an openbox pipe menu where you want to generate something on the fly. You can use awk to make a neat one-liner for some quick job in the terminal, or build an awk section into a shell script.

Simple AWK commands

An AWK program consists of a sequence of pattern-action statements and optional function definitions. It processes text files. AWK is a line oriented language. It divides a file into lines called records. Each line is broken up into a sequence of fields. The fields are accessed by special variables: \$1 reads the first field, \$2 the second and so on. The \$0 variable refers to the whole record.

The structure of an AWK program has the following form:

```
pattern { action }
```

The pattern is a test that is performed on each of the records. If the condition is met then the action is performed. Either pattern or action can be omitted, but not both. The default pattern matches each line and the default action is to print the record.

```
awk -f program-file [file-list]  
awk program [file-list]
```

An AWK program can be run in two basic ways: a) the program is read from a separate file; the name of the program follows the -f option, b) the program is specified on the command line enclosed by quote characters.

Print all the lines from a file

By default, awk prints all lines of a file, so to print every line of above created file use below command

```
awk '{print}' file
```

Note: In awk command '{print}' is used to print all fields along with their values.

Print only specific field like 2nd & 3rd

In awk command, we use \$ (dollar) symbol followed by field number to print field values.

```
awk -F "," '{print $2, $3}' file
```

In the above command we have used the option -F “,” which specifies that comma (,) is the field separator in the file. This is usually good practice when dealing with tables where the separators between each column could be single or multiple white-space (" ") or tab ("\t") or a colon (":") or a semicolon (";").

AWK one-liners

AWK one-liners are simple one-shot programs run from the command line. Let us have the following text file: words.txt

We want to print all words included in the words.txt file that are longer than five characters.

```
wget https://course-cg-5534.s3.amazonaws.com/awk_exercise/words.txt -O  
words.txt  
awk 'length($1) > 5 {print $0}' words.txt
```

```
storeroom  
existence  
ministerial  
falcon  
bookworm  
bookcase
```

The AWK program is placed between two single quote characters. The first is the pattern; we specify that the length of the record is greater than five. The length function returns the length of the string. The \$1 variable refers to the first field of the record; in our case there is only one field per record. The action is placed between curly brackets.

```
awk 'length($1) > 5' words.txt
```

```
storeroom  
existence  
ministerial  
falcon  
bookworm  
bookcase
```

As we have specified earlier, the action can be omitted. In such a case a default action is performed — printing of the whole record.


```
awk 'length($1) == 3' words.txt
```

```
cup  
sky  
top  
war
```

We print all words that have three characters.

```
awk '!(length($1) == 3)' words.txt
```

```
storeroom  
tree  
store  
book  
cloud  
existence  
ministerial  
falcon  
town  
bookworm  
bookcase
```

With the ! operator, we can negate the condition; we print all lines that do not have three characters.

```
awk '(length($1) == 3) || (length($1) == 4)' words.txt
```

```
tree  
cup  
book  
town  
sky  
top  
war
```

Next we apply conditions on numbers.

We have a file with scores of students - scores.txt.

```
wget https://course-cg-5534.s3.amazonaws.com/awk_exercise/scores.txt -O  
scores.txt  
awk '$2 >= 90 { print $0 }' scores.txt
```

```
Lucia 95  
Joe 92  
Sophia 90
```

We print all students with scores 90+.

```
awk '$2 >= 90 { print }' scores.txt
```

```
Lucia 95  
Joe 92  
Sophia 90
```

If we omit an argument for the print function, the \$0 is assumed.

```
awk '$2 >= 90' scores.txt
```

```
Lucia 95
Joe 92
Sophia 90
```

A missing { action } means print the matching line.

```
awk '{ if ($2 >= 90) print }' scores.txt
```

```
Lucia 95
Joe 92
Sophia 90
```

Instead of a pattern, we can also use an if condition in the action.

```
awk '{sum += $2} END { printf("The average score is %.2f\n", sum/NR) }'
scores.txt
```

```
The average score is 77.56
```

This command calculates the average score. In the action block, we calculate the sum of scores. In the END block, we print the average score. We format the output with the built-in printf function. The %.2f is a format specifier; each specifier begins with the % character. The .2 is the precision -- the number of digits after the decimal point. The f expects a floating point value. The \n is not a part of the specifier; it is a newline character. It prints a newline after the string is shown on the terminal.

AWK WORKING WITH PIPES

AWK can receive input and send output to other commands via the pipe.

```
echo -e "1 2 3 5\n2 2 3 8" | awk '{print $(NF)}'
```

```
5
8
```

In this case, AWK receives output from the echo command. It prints the values of last column.

```
awk -F: '$7 ~ /bash/ {print $1}' /etc/passwd | wc -l
```

```
3
```

Here, the AWK program sends data to the wc command via the pipe. In the AWK program, we find out those users who use bash. Their names are passed to the wc command which counts them. In our case, there are three users using bash.

AWK for Bioinformatics - looking at transcriptome data

You can find a lot of online tutorials, but here we will try out a few steps which show how a bioinformatician analyses a **GTF file** using awk.

GTF is a special file format that contains information about different regions of the genome and their associated annotations. More on that here - [Ensembl File Formats-GTF](#).

```
wget https://course-cg-5534.s3.amazonaws.com/awk_exercise/transcriptome.gtf
-O transcriptome.gtf
```

```
head transcriptome.gtf | less -S
```

```
##description: evidence-based annotation of the human genome (GRCh37),
version 18 (Ensembl 73)
##provider: GENCODE
##contact: gencode@sanger.ac.uk
##format: gtf
##date: 2013-09-02
chr1  HAVANA  exon    173753  173862  .   -   .   gene_id
"ENSG00000241860.2"; transcript_id "ENST00000466557.2"; gene_type
"processed_transcript"; gene_status "NOVEL"; gene_name "RP11-34P13.13";
transcript_type "lincRNA"; transcript_status "KNOWN"; transcript_name
"RP11-34P13.13-001"; exon_number 1; exon_id "ENSE00001947154.2"; level 2;
tag "not_best_in_genome_evidence"; havana_gene "OTTHUMG00000002480.3";
havana_transcript "OTTHUMT00000007037.2";
chr1  HAVANA  transcript 1246986 1250550 .   -   .   gene_id
"ENSG00000127054.14"; transcript_id "ENST00000478641.1"; gene_type
"protein_coding"; gene_status "KNOWN"; gene_name "CPSF3L"; transcript_type
"retained_intron"; transcript_status "KNOWN"; transcript_name "CPSF3L-006";
level 2; havana_gene "OTTHUMG00000003330.11"; havana_transcript
"OTTHUMT00000009365.1";
chr1  HAVANA  CDS     1461841 1461911 .   +   0   gene_id
"ENSG00000197785.9"; transcript_id "ENST00000378755.5"; gene_type
"protein_coding"; gene_status "KNOWN"; gene_name "ATAD3A"; transcript_type
"protein_coding"; transcript_status "KNOWN"; transcript_name "ATAD3A-003";
exon_number 13; exon_id "ENSE00001664426.1"; level 2; tag "basic"; tag
"CCDS"; ccidsid "CCDS31.1"; havana_gene "OTTHUMG00000000575.6";
havana_transcript "OTTHUMT00000001365.1";
chr1  HAVANA  exon    1693391 1693474 .   -   .   gene_id
"ENSG00000008130.11"; transcript_id "ENST00000341991.3"; gene_type
"protein_coding"; gene_status "KNOWN"; gene_name "NADK"; transcript_type
"protein_coding"; transcript_status "KNOWN"; transcript_name "NADK-002";
exon_number 3; exon_id "ENSE00003487616.1"; level 2; tag "basic"; tag
"CCDS"; ccidsid "CCDS30565.1"; havana_gene "OTTHUMG00000000942.5";
havana_transcript "OTTHUMT00000002768.1";
chr1  HAVANA  CDS     1688280 1688321 .   -   0   gene_id
"ENSG00000008130.11"; transcript_id "ENST00000497186.1"; gene_type
"protein_coding"; gene_status "KNOWN"; gene_name "NADK"; transcript_type
"nonsense_mediated_decay"; transcript_status "KNOWN"; transcript_name
"NADK-008"; exon_number 2; exon_id "ENSE00001856899.1"; level 2; tag
"mRNA_start_NF"; tag "cds_start_NF"; havana_gene "OTTHUMG00000000942.5";
havana_transcript "OTTHUMT00000002774.3";
```

The transcriptome has 9 columns. The first 8 are separated by tabs and look reasonable (chromosome, annotation source, feature type, start, end, score, strand, and phase), the last one is kind of hairy: it is made up of key-value pairs separated by semicolons, some fields are mandatory and others are optional, and the values are surrounded in double quotes. That's no way to live a decent life. (*text copied from the source*)

let's get only the lines that have gene in the 3th column.

```
$ awk -F "\t" ' $3 == "gene" ' transcriptome.gtf | head | less -S
```

```
chr1  HAVANA  gene     11869   14412   .   +   .   gene_id
"ENSG00000223972.4"; transcript_id "ENSG00000223972.4"; gene_type
"pseudogene"; gene_status "KNOWN"; gene_name "DDX11L1"; transcript_type
"pseudogene"; transcript_status "KNOWN"; transcript_name "DDX11L1"; level
2; havana_gene "OTTHUMG00000000961.2";
chr1  HAVANA  gene     14363   29806   .   -   .   gene_id
"ENSG00000227232.4"; transcript_id "ENSG00000227232.4"; gene_type
```

```
"pseudogene"; gene_status "KNOWN"; gene_name "WASH7P"; transcript_type
"pseudogene"; transcript_status "KNOWN"; transcript_name "WASH7P"; level 2;
havana_gene "OTTHUMG00000000958.1";
chr1 HAVANA gene 29554 31109 . + . gene_id
"ENSG00000243485.2"; transcript_id "ENSG00000243485.2"; gene_type
"lincRNA"; gene_status "NOVEL"; gene_name "MIR1302-11"; transcript_type
"lincRNA"; transcript_status "NOVEL"; transcript_name "MIR1302-11"; level
2; havana_gene "OTTHUMG00000000959.2";
chr1 HAVANA gene 34554 36081 . - . gene_id
"ENSG00000237613.2"; transcript_id "ENSG00000237613.2"; gene_type
"lincRNA"; gene_status "KNOWN"; gene_name "FAM138A"; transcript_type
"lincRNA"; transcript_status "KNOWN"; transcript_name "FAM138A"; level 2;
havana_gene "OTTHUMG00000000960.1";
chr1 HAVANA gene 52473 54936 . + . gene_id
"ENSG00000268020.2"; transcript_id "ENSG00000268020.2"; gene_type
"pseudogene"; gene_status "KNOWN"; gene_name "OR4G4P"; transcript_type
"pseudogene"; transcript_status "KNOWN"; transcript_name "OR4G4P"; level 2;
havana_gene "OTTHUMG00000185779.1";
chr1 HAVANA gene 62948 63887 . + . gene_id
"ENSG00000240361.1"; transcript_id "ENSG00000240361.1"; gene_type
"pseudogene"; gene_status "KNOWN"; gene_name "OR4G11P"; transcript_type
"pseudogene"; transcript_status "KNOWN"; transcript_name "OR4G11P"; level
2; havana_gene "OTTHUMG00000001095.2";
chr1 HAVANA gene 69091 70008 . + . gene_id
"ENSG00000186092.4"; transcript_id "ENSG00000186092.4"; gene_type
"protein_coding"; gene_status "KNOWN"; gene_name "OR4F5"; transcript_type
"protein_coding"; transcript_status "KNOWN"; transcript_name "OR4F5"; level
2; havana_gene "OTTHUMG00000001094.1";
chr1 HAVANA gene 89295 133566 . - . gene_id
"ENSG00000238009.2"; transcript_id "ENSG00000238009.2"; gene_type
"lincRNA"; gene_status "NOVEL"; gene_name "RP11-34P13.7"; transcript_type
"lincRNA"; transcript_status "NOVEL"; transcript_name "RP11-34P13.7"; level
2; havana_gene "OTTHUMG00000001096.2";
chr1 HAVANA gene 89551 91105 . - . gene_id
"ENSG00000239945.1"; transcript_id "ENSG00000239945.1"; gene_type
"lincRNA"; gene_status "NOVEL"; gene_name "RP11-34P13.8"; transcript_type
"lincRNA"; transcript_status "NOVEL"; transcript_name "RP11-34P13.8"; level
2; havana_gene "OTTHUMG00000001097.2";
chr1 HAVANA gene 131025 134836 . + . gene_id
"ENSG00000233750.3"; transcript_id "ENSG00000233750.3"; gene_type
"pseudogene"; gene_status "KNOWN"; gene_name "CICP27"; transcript_type
"pseudogene"; transcript_status "KNOWN"; transcript_name "CICP27"; level 1;
tag "pseudo_consens"; havana_gene "OTTHUMG00000001257.3";
```

Perhaps filter a bit more and print the content of the 9th column in the file.

```
$ awk -F "\t" '$3 == "gene" { print $9 }' transcriptome.gtf | head | less -
S
```

```
gene_id "ENSG00000223972.4"; transcript_id "ENSG00000223972.4"; gene_type
"pseudogene"; gene_status "KNOWN"; gene_name "DDX11L1"; transcript_type
"pseudogene"; transcript_status "KNOWN"; transcript_name "DDX11L1"; level
2; havana_gene "OTTHUMG00000000961.2";
gene_id "ENSG00000227232.4"; transcript_id "ENSG00000227232.4"; gene_type
"pseudogene"; gene_status "KNOWN"; gene_name "WASH7P"; transcript_type
"pseudogene"; transcript_status "KNOWN"; transcript_name "WASH7P"; level 2;
havana_gene "OTTHUMG00000000958.1";
gene_id "ENSG00000243485.2"; transcript_id "ENSG00000243485.2"; gene_type
"lincRNA"; gene_status "NOVEL"; gene_name "MIR1302-11"; transcript_type
"lincRNA"; transcript_status "NOVEL"; transcript_name "MIR1302-11"; level
2; havana_gene "OTTHUMG00000000959.2";
```

```
gene_id "ENSG00000237613.2"; transcript_id "ENSG00000237613.2"; gene_type
"lincRNA"; gene_status "KNOWN"; gene_name "FAM138A"; transcript_type
"lincRNA"; transcript_status "KNOWN"; transcript_name "FAM138A"; level 2;
havana_gene "OTTHUMG00000000960.1";
gene_id "ENSG00000268020.2"; transcript_id "ENSG00000268020.2"; gene_type
"pseudogene"; gene_status "KNOWN"; gene_name "OR4G4P"; transcript_type
"pseudogene"; transcript_status "KNOWN"; transcript_name "OR4G4P"; level 2;
havana_gene "OTTHUMG00000185779.1";
gene_id "ENSG00000240361.1"; transcript_id "ENSG00000240361.1"; gene_type
"pseudogene"; gene_status "KNOWN"; gene_name "OR4G11P"; transcript_type
"pseudogene"; transcript_status "KNOWN"; transcript_name "OR4G11P"; level
2; havana_gene "OTTHUMG00000001095.2";
gene_id "ENSG00000186092.4"; transcript_id "ENSG00000186092.4"; gene_type
"protein_coding"; gene_status "KNOWN"; gene_name "OR4F5"; transcript_type
"protein_coding"; transcript_status "KNOWN"; transcript_name "OR4F5"; level
2; havana_gene "OTTHUMG00000001094.1";
gene_id "ENSG00000238009.2"; transcript_id "ENSG00000238009.2"; gene_type
"lincRNA"; gene_status "NOVEL"; gene_name "RP11-34P13.7"; transcript_type
"lincRNA"; transcript_status "NOVEL"; transcript_name "RP11-34P13.7"; level
2; havana_gene "OTTHUMG00000001096.2";
gene_id "ENSG00000239945.1"; transcript_id "ENSG00000239945.1"; gene_type
"lincRNA"; gene_status "NOVEL"; gene_name "RP11-34P13.8"; transcript_type
"lincRNA"; transcript_status "NOVEL"; transcript_name "RP11-34P13.8"; level
2; havana_gene "OTTHUMG00000001097.2";
gene_id "ENSG00000233750.3"; transcript_id "ENSG00000233750.3"; gene_type
"pseudogene"; gene_status "KNOWN"; gene_name "CICP27"; transcript_type
"pseudogene"; transcript_status "KNOWN"; transcript_name "CICP27"; level 1;
tag "pseudo_consens"; havana_gene "OTTHUMG00000001257.3";
```

What about if we want just a specific piece from this information? We can take the output from the first awk script in to a second one. Note that we will use different field separator ";".

```
$ awk -F "\t" '($3 == "gene" { print $9 })' transcriptome.gtf | awk -F "; " "
'{ print $3 }' | head
```

```
gene_type "pseudogene"
gene_type "pseudogene"
gene_type "lincRNA"
gene_type "lincRNA"
gene_type "pseudogene"
gene_type "pseudogene"
gene_type "protein_coding"
gene_type "lincRNA"
gene_type "lincRNA"
gene_type "pseudogene"
```

CHAINING AWK CALLS

We will start with the AWK call that we were using before, and we will append a pipe | so it can be used as input for the next AWK call, this time using a space and a semicolon as the delimiter to define what a column is:

```
awk -F "\t" '($3 == "gene" { print $9 })' transcriptome.gtf | awk -F "; " '{
print $3 }' | head | less -S
```

```
gene_type "pseudogene"
gene_type "pseudogene"
gene_type "lincRNA"
gene_type "lincRNA"
```

```
gene_type "pseudogene"
gene_type "pseudogene"
gene_type "protein_coding"
gene_type "lincRNA"
gene_type "lincRNA"
gene_type "pseudogene"
```

Now that we see what the third column looks like, we can filter for protein-coding genes

```
awk -F "\t" '$3 == "gene" { print $9 }' transcriptome.gtf | \
awk -F ";" " '$3 == \"gene_type \\\"protein_coding\\\"\"' | \
head | less -S
```

```
gene_id "ENSG00000186092.4"; transcript_id "ENSG00000186092.4"; gene_type
"protein_coding"; gene_status "KNOWN"; gene_name "OR4F5"; transcript_type
"protein_coding"; transcript_status "KNOWN"; transcript_name "OR4F5"; level
2; havana_gene "OTTHUMG00000001094.1";
gene_id "ENSG00000237683.5"; transcript_id "ENSG00000237683.5"; gene_type
"protein_coding"; gene_status "KNOWN"; gene_name "AL627309.1";
transcript_type "protein_coding"; transcript_status "KNOWN";
transcript_name "AL627309.1"; level 3;
gene_id "ENSG00000235249.1"; transcript_id "ENSG00000235249.1"; gene_type
"protein_coding"; gene_status "KNOWN"; gene_name "OR4F29"; transcript_type
"protein_coding"; transcript_status "KNOWN"; transcript_name "OR4F29";
level 2; havana_gene "OTTHUMG000000002860.1";
gene_id "ENSG00000185097.2"; transcript_id "ENSG00000185097.2"; gene_type
"protein_coding"; gene_status "KNOWN"; gene_name "OR4F16"; transcript_type
"protein_coding"; transcript_status "KNOWN"; transcript_name "OR4F16";
level 2; havana_gene "OTTHUMG000000002581.1";
gene_id "ENSG00000269831.1"; transcript_id "ENSG00000269831.1"; gene_type
"protein_coding"; gene_status "NOVEL"; gene_name "AL669831.1";
transcript_type "protein_coding"; transcript_status "NOVEL";
transcript_name "AL669831.1"; level 3;
gene_id "ENSG00000269308.1"; transcript_id "ENSG00000269308.1"; gene_type
"protein_coding"; gene_status "NOVEL"; gene_name "AL645608.2";
transcript_type "protein_coding"; transcript_status "NOVEL";
transcript_name "AL645608.2"; level 3;
gene_id "ENSG00000187634.6"; transcript_id "ENSG00000187634.6"; gene_type
"protein_coding"; gene_status "KNOWN"; gene_name "SAMD11"; transcript_type
"protein_coding"; transcript_status "KNOWN"; transcript_name "SAMD11";
level 2; havana_gene "OTTHUMG000000040719.8";
gene_id "ENSG00000268179.1"; transcript_id "ENSG00000268179.1"; gene_type
"protein_coding"; gene_status "NOVEL"; gene_name "AL645608.1";
transcript_type "protein_coding"; transcript_status "NOVEL";
transcript_name "AL645608.1"; level 3;
gene_id "ENSG00000188976.6"; transcript_id "ENSG00000188976.6"; gene_type
"protein_coding"; gene_status "KNOWN"; gene_name "NOC2L"; transcript_type
"protein_coding"; transcript_status "KNOWN"; transcript_name "NOC2L"; level
2; havana_gene "OTTHUMG000000040720.1";
gene_id "ENSG00000187961.9"; transcript_id "ENSG00000187961.9"; gene_type
"protein_coding"; gene_status "KNOWN"; gene_name "KLHL17"; transcript_type
"protein_coding"; transcript_status "KNOWN"; transcript_name "KLHL17";
level 2; havana_gene "OTTHUMG000000040721.6";
```

I added a space and a backslash (not to be confused with the regular slash /) after the first and second pipes to split the code into two lines; this makes it easier to read and it highlights that we are taking two separate steps.

The double quotes around protein_coding are escaped (also with a backslash \) because they are already contained inside double quotes. To avoid the backslashing drama we can use the partial matching operator ~ instead of the total equality operator ==.

```
awk -F "\t" '$3 == "gene" { print $9 }' transcriptome.gtf | \
awk -F "; " '$3 ~ "protein_coding"' | \
head | less -S
```

The output is the same as before: those lines that contain a `protein_coding` somewhere in their third column make the partial matching rule true, and they get printed (which is the default behavior when there are no curly braces).

Now we have all the protein-coding genes, but how do we get to the genes that only have one exon? Well, we have to revisit our initial AWK call: we selected lines that corresponded to genes, but we actually want lines that correspond to exons. That's an easy fix, we just change the word "gene" for the word "exon". Everything else stays the same.

```
awk -F "\t" '$3 == "exon" { print $9 }' transcriptome.gtf | \
awk -F "; " '$3 ~ "protein_coding"' | \
head | less -S
```

```
gene_id "ENSG00000186092.4"; transcript_id "ENST000000335137.3"; gene_type
"protein_coding"; gene_status "KNOWN"; gene_name "OR4F5"; transcript_type
"protein_coding"; transcript_status "KNOWN"; transcript_name "OR4F5-001";
exon_number 1; exon_id "ENSE000002319515.1"; level 2; tag "basic"; tag
"CCDS"; ccdsid "CCDS30547.1"; havana_gene "OTTHUMG000000001094.1";
havana_transcript "OTTHUMT000000003223.1";
gene_id "ENSG00000237683.5"; transcript_id "ENST000000423372.3"; gene_type
"protein_coding"; gene_status "KNOWN"; gene_name "AL627309.1";
transcript_type "protein_coding"; transcript_status "KNOWN";
transcript_name "AL627309.1-201"; exon_number 1; exon_id
"ENSE000002221580.1"; level 3; tag "basic";
gene_id "ENSG00000237683.5"; transcript_id "ENST000000423372.3"; gene_type
"protein_coding"; gene_status "KNOWN"; gene_name "AL627309.1";
transcript_type "protein_coding"; transcript_status "KNOWN";
transcript_name "AL627309.1-201"; exon_number 2; exon_id
"ENSE000002314092.1"; level 3; tag "basic";
gene_id "ENSG00000235249.1"; transcript_id "ENST000000426406.1"; gene_type
"protein_coding"; gene_status "KNOWN"; gene_name "OR4F29"; transcript_type
"protein_coding"; transcript_status "KNOWN"; transcript_name "OR4F29-001";
exon_number 1; exon_id "ENSE000002316283.1"; level 2; tag "basic"; tag
"CCDS"; ccdsid "CCDS41220.1"; havana_gene "OTTHUMG000000002860.1";
havana_transcript "OTTHUMT000000007999.1";
gene_id "ENSG00000185097.2"; transcript_id "ENST000000332831.2"; gene_type
"protein_coding"; gene_status "KNOWN"; gene_name "OR4F16"; transcript_type
"protein_coding"; transcript_status "KNOWN"; transcript_name "OR4F16-001";
exon_number 1; exon_id "ENSE000002324228.1"; level 2; tag "basic"; tag
"CCDS"; ccdsid "CCDS41221.1"; havana_gene "OTTHUMG000000002581.1";
havana_transcript "OTTHUMT000000007334.1";
gene_id "ENSG00000269831.1"; transcript_id "ENST000000599533.1"; gene_type
"protein_coding"; gene_status "NOVEL"; gene_name "AL669831.1";
transcript_type "protein_coding"; transcript_status "NOVEL";
transcript_name "AL669831.1-201"; exon_number 1; exon_id
"ENSE000003063549.1"; level 3; tag "basic";
gene_id "ENSG00000269831.1"; transcript_id "ENST000000599533.1"; gene_type
"protein_coding"; gene_status "NOVEL"; gene_name "AL669831.1";
transcript_type "protein_coding"; transcript_status "NOVEL";
transcript_name "AL669831.1-201"; exon_number 2; exon_id
"ENSE000003084653.1"; level 3; tag "basic";
gene_id "ENSG00000269831.1"; transcript_id "ENST000000599533.1"; gene_type
"protein_coding"; gene_status "NOVEL"; gene_name "AL669831.1";
transcript_type "protein_coding"; transcript_status "NOVEL";
transcript_name "AL669831.1-201"; exon_number 3; exon_id
"ENSE000003138540.1"; level 3; tag "basic";
```



```
gene_id "ENSG00000269308.1"; transcript_id "ENST00000594233.1"; gene_type
"protein_coding"; gene_status "NOVEL"; gene_name "AL645608.2";
transcript_type "protein_coding"; transcript_status "NOVEL";
transcript_name "AL645608.2-201"; exon_number 1; exon_id
"ENSE00003079649.1"; level 3; tag "basic";
gene_id "ENSG00000269308.1"; transcript_id "ENST00000594233.1"; gene_type
"protein_coding"; gene_status "NOVEL"; gene_name "AL645608.2";
transcript_type "protein_coding"; transcript_status "NOVEL";
transcript_name "AL645608.2-201"; exon_number 2; exon_id
"ENSE00003048391.1"; level 3; tag "basic";
```

CLEANING UP THE OUTPUT

Before we try to count how many exons belong to the same protein-coding gene, let's simplify the output so we only get the gene names (which are in column 5).

```
awk -F "\t" '$3 == "exon" { print $9 }' transcriptome.gtf | \
awk -F "; " '$3 ~ "protein_coding" {print $5}' | \
head
```

```
gene_name "OR4F5"
gene_name "AL627309.1"
gene_name "AL627309.1"
gene_name "OR4F29"
gene_name "OR4F16"
gene_name "AL669831.1"
gene_name "AL669831.1"
gene_name "AL669831.1"
gene_name "AL645608.2"
gene_name "AL645608.2"
```

This is sort of what we want. We could chain another AWK call using `-F " "`, and pick the second column (which would get rid of the `gene_name`). Feel free to try that approach if you are curious.

We can also take a shortcut by using the `tr -d` command, which deletes whatever characters appear in double quotes. For example, to remove every vowel from a sentence:

```
echo "This unix thing is cool" | tr -d "aeiou" # Ths nx thng s cl
```

Let's try deleting all the semicolons and quotes before the second AWK call:

```
awk -F "\t" '$3 == "exon" { print $9 }' transcriptome.gtf | \
tr -d ";\"" | \
awk -F " " '$6 == "protein_coding" {print $10}' | \
head
```

```
OR4F5
AL627309.1
AL627309.1
OR4F29
OR4F16
AL669831.1
AL669831.1
AL669831.1
AL645608.2
AL645608.2
```


Run `bash awk -F "\t" ' $3 == "exon" { print $9 } ' transcriptome.gtf | tr -d ";\" | head` to understand what the input to the second AWK call looks like. It's just words separated by spaces; the sixth word corresponds to the gene type, and the tenth word to the gene name.

COUNTING GENES

There is one more concept we need to introduce before we start counting. AWK uses a special rule called END, which is only true once the input is over. See an example:

```
echo -e "a\na\nb\nb\nb\nc" | \
awk '
    { print $1 }

END { print "Done with letters!" }
'
```

```
a
a
b
b
b
c
```

Done with letters! The -e option tells echo to convert each \n into a new line, which is a convenient way of printing multiple lines from a single character string.

In AWK, any amount of whitespace is allowed between the initial and the final quote '. I separated the first rule from the END rule to make them easier to read.

Now we are ready for counting.

```
echo -e "a\na\nb\nb\nb\nc" | \
awk '
    { counter[$1] += 1 }

END {
    for (letter in counter){
        print letter, counter[letter]
    }
}
'
```

```
a 2
b 3
c 1
```

Wow, what is all that madness?

Instead of printing each letter, we manipulate a variable that we called counter. This variable is special because it is followed by brackets [], which makes it an associative array, a fancy way of calling a variable that stores key-value pairs.

In this case we chose the values of the first column \1 to be the keys of the counter variable, which means there are 3 keys ("a", "b" and "c"). The values are initialized to 0. For every line in the input,

we add a 1 to the value in the array whose key is equal to `\1`. We use the addition operator `+=`, a shortcut for `counter[\1] = counter[\1] + 1`.

When all the lines are read, the `END` rule becomes true, and the code between the curly braces `{ }` is executed. The structure for `(key in associate_array) { some_code }` is called a for loop, and it executes `some_code` as many times as there are keys in the array. `letter` is the name that we chose for the variable that cycles through all the keys in `counter`, and `counter[letter]` gives the value stored in `counter` for each letter (which we calculated in the previous curly brace chunk).

Now we can apply this to the real example:

```
awk -F "\t" '$3 == "exon" { print $9 }' transcriptome.gtf | \
tr -d ";\" | \
awk -F " " '{
    $6 == "protein_coding" {
        gene_counter[$10] += 1
    }
}' > number_of_exons_by_gene.txt

head number_of_exons_by_gene.txt

CAPN6 24
ARL14EPL 3
DACH1 38
IFNA13 1
HSP90AB1 36
CAPN7 52
DACH2 84
IFNA14 1
LARS 188
CAPN8 78
```

If you are using the real transcriptome, it takes less than a minute to count up one million exons. Pretty impressive.

We saved the output to a file, so now we can use AWK to see how many genes are made up of a single exon.

```
awk '$2 == 1' number_of_exons_by_gene.txt | wc -l # 1362
```

BASICS OF PROGRAMMING IN R

Tip: In order to take most out of this tutorial you should not miss reading any lines in this tutorial and follow the flow and write codes on your computer.

- **FOR WINDOWS USERS -- You can now quit CygWin or WSL. R does not require Linux. You can use R directly with the help of R-Studio.**

What is R?

R is a language and environment for statistical computing and graphics developed in 1993. It provides a wide variety of statistical and graphical techniques (linear and nonlinear modeling, statistical tests, time series analysis, classification, clustering, ...), and is highly extensible, meaning that the user community can write new R tools. It is a GNU project (Free and Open Source).

The R language has its roots in the S language and environment which was developed at Bell Laboratories (formerly AT&T, now Lucent Technologies) by John Chambers and colleagues. R was created by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand, and now, R is developed by the R Development Core Team, of which Chambers is a member. R is named partly after the first names of the first two R authors (Robert Gentleman and Ross Ihaka), and partly as a play on the name of S. R can be considered as a different implementation of S. There are some important differences, but much code written for S runs unaltered under R.

Some of R's strengths:

1. The ease with which well-designed publication-quality plots can be produced, including mathematical symbols and formulae where needed. Great care has been taken over the defaults for the minor design choices in graphics, but the user retains full control.
2. It compiles and runs on a wide variety of UNIX platforms and similar systems (including FreeBSD and Linux), Windows and MacOS.
3. R can be extended (easily) via packages.
4. R has its own LaTeX-like documentation format, which is used to supply comprehensive documentation, both on-line in a number of formats and in hardcopy.
5. It has a vast community both in academia and in business.
6. It's FREE!

The R environment

R is an integrated suite of software facilities for data manipulation, calculation and graphical display. It includes

- an effective data handling and storage facility,
- a suite of operators for calculations on arrays, in particular matrices,
- a large, coherent, integrated collection of intermediate tools for data analysis,
- graphical facilities for data analysis and display either on-screen or on hardcopy, and
- a well-developed, and effective programming language which includes conditionals, loops, user-defined recursive functions and input and output facilities.

The term “environment” is intended to characterize it as a fully planned and coherent system, rather than an incremental accretion of very specific and inflexible tools, as is frequently the case with other data analysis software.

R, like S, is designed around a true computer language, and it allows users to add additional functionality by defining new functions. Much of the system is itself written in the R dialect of S, which makes it easy for users to follow the algorithmic choices made. For computationally-intensive tasks, C, C++ and

Fortran code can be linked and called at run time. Advanced users can write C code to manipulate R objects directly.

Many users think of R as a statistics system. The R group prefers to think of it of an environment within which statistical techniques are implemented.

The R Homepage

The R homepage has a wealth of information on it, [R-project.org](https://www.r-project.org)

On the homepage you can:

- Learn more about R
- Download R
- Get Documentation (official and user supplied)
- Get access to CRAN 'Comprehensive R archival network'
- RStudio
- RStudio started in 2010, to offer R a more full featured integrated development environment (IDE) and modeled after matlabs IDE.

We will be using RStudio: a free, open source R integrated development environment. It provides a built in editor, works on all platforms (including on servers) and provides many advantages such as integration with version control and project management.

We will follow this book as our guide - [Data Analysis with Rstudio - An Easygoing Introduction](#)

INSTALLATION ON MAC

To Install R

1. Open an internet browser and go to www.r-project.org.
2. Click the "download R" link in the middle of the page under "Getting Started."
3. Select a CRAN location (a mirror site) and click the corresponding link.
4. Click on the "Download R for (Mac) OS X" link at the top of the page.
5. Click on the file containing the latest version of R under "Files."
6. Save the .pkg file, double-click it to open, and follow the installation instructions.
7. Now that R is installed, you need to download and install RStudio.

To Install RStudio

1. Go to <https://www.rstudio.com/products/rstudio/#rstudio-desktop> and click on the "Download RStudio" button.
2. Click on "Download RStudio Desktop."
3. Click on the version recommended for your system, or the latest Mac version, save the .dmg file on your computer, double-click it to open, and then drag and drop it to your applications folder.

INSTALLATION ON WINDOWS

To Install R:

1. Open an internet browser and go to www.r-project.org.
2. Click the "download R" link in the middle of the page under "Getting Started."
3. Select a CRAN location (a mirror site) and click the corresponding link.
4. Click on the "Download R for Windows" link at the top of the page.
5. Click on the "install R for the first time" link at the top of the page.
6. Click "Download R for Windows" and save the executable file somewhere on your computer. Run the .exe file and follow the installation instructions.
7. Now that R is installed, you need to download and install RStudio.

To Install RStudio

1. Go to <https://www.rstudio.com/products/rstudio/#rstudio-desktop> and click on the "Download RStudio" button.
2. Click on "Download RStudio Desktop."
3. Click on the version recommended for your system, or the latest Windows version, and save the executable file. Run the .exe file and follow the installation instructions.

RStudio has many features:

- syntax highlighting
- code completion
- smart indentation
- "Projects"
- workspace browser and data viewer
- embedded plots
- Markdown notebooks, Sweave authoring and knitr with one click pdf or html
- runs on all platforms and over the web
- etc. etc. etc.

1. Getting started

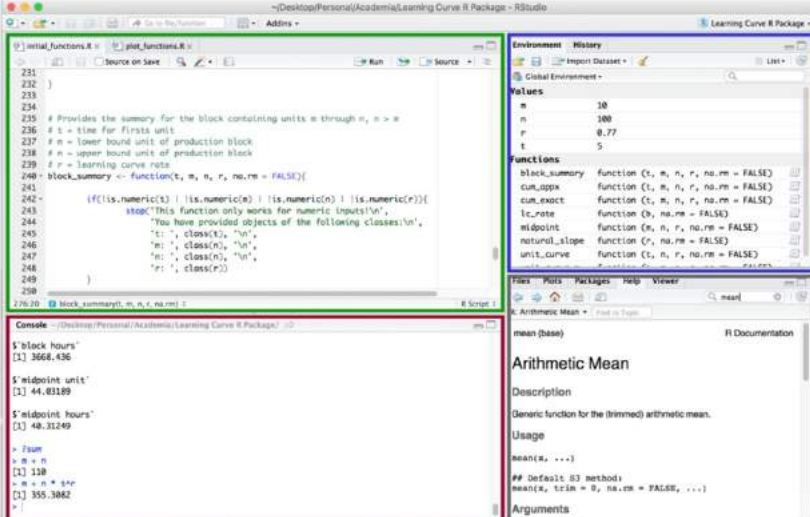
Basic layout

When you first open RStudio, you will be greeted by three panels:

The interactive R console (entire left)

Environment/History (tabbed in upper right)

Files/Plots/Packages/Help/Viewer (tabbed in lower right)



Script files

- Saves your script
- Allows code & comments
- Can have multiple files open at a time

Console/Command line

- Can use as calculator
- Does not save code
- This is where your output is displayed

Workspace environment

- Holds your objects
- Can review history

Misc - Displays:

- files in working directory
- plots when produced
- help files/search

2. Open a new RScript File

File -> New File -> R Script

RStudio_newfile

Then save the new empty file as Intro2R.R

File -> Save as -> Intro2R.R

3. Basics of your environment

The R prompt is the '>', when R is expecting more (command is not complete) you see a '+'

```
> ls()
character(0)
> ls(
+ )
character(0)
> |
```

4. Writing and running R commands

In the source editor (top left by default) type

```
getwd()
```

Then on the line Control + Enter (Linux/Windows), Command + Enter (Mac) to execute the line.

5. The assignment operator (<-) vs equals (=)

The assignment operator is used assign data to a variable

```
x <- 1:10
x
[1] 1 2 3 4 5 6 7 8 9 10
```

In this case, the equal sign works as well

```
x = 1:10
x
[1] 1 2 3 4 5 6 7 8 9 10
```

But you should NEVER EVER DO THIS

```
1:10 -> x
x
[1] 1 2 3 4 5 6 7 8 9 10
```

The two act the same in most cases. The difference in assignment operators is clearer when you use them to set an argument value in a function call. For example:

```
median(x = 1:10)
x
Error: object 'x' not found
```

In this case, x is declared within the scope of the function, so it does not exist in the user workspace.

```
median(x <- 1:10)
x
[1] 1 2 3 4 5 6 7 8 9 10
```

In this case, x is declared in the user workspace, so you can use it after the function call has been completed. There is a general preference among the R community for using <- for assignment (other than in function signatures)

6. The RStudio Cheat Sheet

[rstudio-ide.pdf](#)

spend 15 minutes getting to know RStudio a little.