The Searchable Compiled "Textbook" of BIOS/CSE 60132.01 Spring 2016, 2nd Ed.

Note that this document may refer to files you do not have access to. If in dire need, please email the current instructor or ssander5@alumni.nd.edu

TABLE OF CONTENTS:

Resources

Cheat Sheet (Google for more):

Commands:

Basic navigation:
ssh - secure shell, log into a machine
clear - clears the terminal of previous commands
bash - enters into the bash shell (default on ND machines is -tch)
echo - echo typed words, or a variable (which has $ before it)
declare - defines a variable, i.e. declare myvar=hello. No spaces permitted without escape or quotes.
pwd - displays the absolute path to the current directory from root (/)
ls - listing or "let's see". lists the files in the current folder.
man COMMAND - displays the manual pages for the command
cd - change current directory
mkdir - make a new directory
less FILE - opens a readable file, does not allow you to edit it.
nano - opens a file (or creates a new one) in a word pad style editor
mv FILE DESTINATION - moves a file from one point to another
mv FILE NEWNAME - renames a file
cp FILE DESTINATION - copies a file from one point to another
rm FILE - removes a file (-r flag needed to remove a directory)
scp FILE COMPUTER:DESTINATION - moves a file from home computer to another computer, in the file designated after the :
scp COMPUTER:FILE DESTINATION - moves a file from another computer to the current computer.
top - list the users on the machine and the resources in use
head - list out the top ten lines of a file to the terminal
tail - list the last ten lines of a file to the terminal
info COMMAND - more extensive info than on the man pages
date - outputs date and time
which COMMAND - tells you where a program exists if it is within you PATH variable
ln -s DESTINATION LINKNAME - creates a symbolic link from the destination (ie:dropbox in course directory) to a ./linkname/ (ie:linktodropbox). Deleting a softlink does not effect the actual directory (unless you use rm -r). **gzip FILE** - compress a file
gzip -d FILE - decompress a file, file must end in .gz
tar cf FILE - tar files together
tar xfz FILE - untar files: e(x)tract (f)rom g(z)ip

File System/System Information:
free -mg - gives you the amount of RAM on the current machine in GB (-g)
du -h - gives you disk usage information (in human readable format)
df -h - gives you info about amount of disk free (in human readable format)
top - shows all processes running and who owns them, their CPU usage, Memory usage, and other things (q to quit)
fs lq DIRECTORY - get the quota and free space for a directory.

Permissions:
chmod < ugo > < +- > < rwx > - change local file system permissions
fs listacl < dir > - AFS - list ACLs for directory < dir >
fs setacl -dir < dir > -acl < user/group > < rlidkwa > - AFS - set ACLs for < user > in < dir >
fs help - AFS - AFS permissions help
fs help setacl - AFS - setacl help

Obtaining Data and Software:

wget http://... - downloads file at http link (can also use with ftp link)
gzip -d file.gz - decompresses file.gz (can be a tar.gz file)
bunzip2 file.bz2 - decompresses file.bz2 (can be tarred)
unzip file.zip - decompresses file.zip (can be tarred)
tar -xf file.tar - unpacks file.tar
tar -zxf file.tar.gz - decompresses and unpacks a tar.gz file
tar -zcvf file.tar.gz targetdir - makes file.tar.gz out of targetdir (z = gzip, c = create)

Installing Software:
./configure --prefix=/path/to/install - configures your source to install. Leave off prefix if you have root and want to install to /usr/bin
make - creates binaries
make install - installs to the path specified in prefix
arch - check to see if we're running a 32- or 64-bit machine

Special Characters:

/ - separates directory names in a file path (ex: MyDocuments/MyMovies/movie.mov). Also refers to "root" directory (/)
\ - "escape character", causes the single character following it to be read literally (ex: echo \\ echoes a single \)
' - used in pairs to take a phrase and use it and its contents literally (ex: echo 'Yay bioinformatics')
" - used in pairs to take a phrase and use it and its contents literally except for ! and $, to allow variable subsitution (ex: echo "My path is $PATH")
$ - signals a variable. When inside a set of "" CANNOT be escaped and always signifies a variable. (i.e. echo "$HOME is my home")
* - used to match any number of characters. i.e. test*.txt, test*, *
? - used to match any single character. i.e. test1?.txt, test?.txt
; - used to signal the end of a command on a single line.
- designates a comment in bash. Anything following will not be read by computer.
#! - hash bang or shebang, tells the computer you are about to give it a program to use to interpret the code you are providing. i.e. #!/bin/bash
: - used to separate directory locations in PATH variable.
> - used to capture output to screen and put it into a file.

Hints:

Tab complete - will complete names to the next point of ambiguity
Hit the up key to go back through your previous commands
You can chain together multiple directories for commands. Ex. cd ../../.. - goes up 3 directories
Ctrl-c kills a running program
Add directories to your PATH with "declare PATH=$PATH:
.bashrc is your set up for your terminal in bash. You can add any code in this file that you wish to be executed upon entering bash (such as adding things to your path, aliases, variable declarations, etc).

Bioinformatic Programs:

Fastqc - Java program that gives you information about fastq sequence files, such as number of reads, quality, and repetative sequences.
muscle - multiple alignment program, called: muscle -in -out
hmmr - builds a profile based on a training set (ie: known examples of a gene) and uses the profile to search for matches in another file.
...hmmbuild - hmmbuild OUTFILE.hmm INFILE.alignment : builds a profile based off a multiple

alignment
...hmmsearch - hmmsearch PROFILE.hmm FILETOSEARCH.fasta : searches a file for matches to the hmm profile
BLAST - searches a database for matches to sequence(s).
...makeblastdb - makeblastdb -in DATABASE.fasta -out DATABASE_NAME.db -dbtype
...blastn/blastp/blastx/etc i.e.: blastn -db DATABASE.db -query QUERY.fasta -out OUTFILE.out
Grep - grabs lines that match a pattern
Sed - does many things (stream editor), but mainly used to find and replace text
Awk – also does many things, powerful for math operations in command line

Directories:

~ or $HOME - your home directory
cd ~ or cd (followed by nothing) - go $HOME
. - current directory
.. - one directory up
cd - - previous directory

Access to machines

Here's a list of some of Notre Dame's machines (running Red Hat Linux) that you can ssh into. This list is not comprehensive, and doesn't include any CRC machines.

darrow.cc.nd.edu
student01.cse.nd.edu*
student02.cse.nd.edu*
student03.cse.nd.edu*
student04.cse.nd.edu*

For details on how to access these machines, see first unit of notes.
* only work if you have enrolled in a CSE course previously.

Please note that there are often issues with darrow. Darrow is one of the few machines that can be accessed off campus, but also appears to lack certain software (i.e. java at times, R library function) and can be a bit of a pain to use. If you do not have cse access, consider using the CRC machines. There is a section on CRC usage in the Special Topics section at the end of this document.

Unit I: Introduction to Unix – Basic Navigation

Optional Pre-Course Navigation Tutorial

http://www.ee.surrey.ac.uk/Teaching/Unix/

Unix Background and Echo

Welcome to the course!

What is this course and why are we here?

******Goal******

Essentially, to get you used to using Unix and command line so that you can perform your own bioinformatic analyses. Also, even if you decide you hate doing this kind of thing (it isn't for everyone!), you will still be able to clearly convey to a bioinformatician what it is that you need done as well as understand the assumptions of each analysis.

******Quick vocabulary******

Operating System – The software that controls the computer's main functions, such as communicating with hardware, controlling input/output, etc.

GUI – Graphical user interface that allows for point and click, touch screen, and otherwise visual interaction, i.e.: Windows, iOS

Command Line (a.k.a. CLI or shell) – OS interface with mainly text based input/output, i.e. Unix

******Intro to Unix******

Unix is an operating system, just like Windows or MacOS, developed at Bell Labs in the early 1970s. It was a command line program with only text input. The first graphical user interfaces (GUI) came out around 1985, and were popular for home use because it was easier to navigate and interact with a mouse and visual input.

However, Unix was great at the time because it was a "programmer's delight". It has "simple, elegant underpinnings". It was driven by a philosophy that made sense in a programming community:

Write programs that do one thing and do it well.
Write programs to work together.
Write programs to handle text streams because that is the universal interface.

Because of this, it is designed to be:

flexible
processing centric
stable

It was also freely handed out to universities at the beginning of computing, which also required it to be:

portable
multi-user
secure

So it was great back then, what about now?
There is a major cost to running a graphical system. My windows machine requires 2GB of RAM just to exist in all its pretty glory, and eats ~4% of my processing capacity. Unix is a lot less resource intensive, which frees up much more of the same system for computational/analytical use.

Why do we want to use a Unix-based system for bioinformatics?
- it was built to be easy to program
- a building-block approach to programming - makes it easier to tailor existing code
- it is stable and powerful
- simple input/output (I/O) - adds to simplicity
- well-suited to text manipulation/text parsing (which we do all the time)
- lots of control over the system
- most bioinformatics tools are written for these systems (BLAST, ClustalW, PHRAP, MUSCLE)

tl;dr
Unix is an operating system based in a commandline interface instead of a graphic interface as you are used to using on home computers. Graphic interfaces allow you to navigate by point and click and provide you with a pretty, visually oriented environment where as in Unix you will be using mainly the keyboard.

NOTE:
Linux is a variant of Unix (thus it's a Unix-based system) used on personal computers. It offers the flexibility and simplicity of Unix with a user-friendly interface. Mac's OSX is also a Unix-based system, and thus has similarities with Linux. Windows isn't, and so it's pretty different. Some common distributions of Linux are Red Hat (which we have on our machines), Ubuntu, Fedora, and Debian.

Where do I start?
****Getting onto a Notre Dame Machine****
Most people tend to use either Mac OSX or Windows operating systems as opposed to some flavor of Linux. However, much of the software we use, and most of the university's computing resources, run on Linux (we have Red Hat Linux).

To save us the trouble of installing our own versions of Linux on our personal computers, and to gain access to the powerful machine and bioinformatics software used on ND's campus, we will learn how to do a remote login.

Remote login is the method of signing into and using a computer other than the one you are physically working on.

There are two basic methods for doing remote login: Putty and ssh.

1. PuTTY - if you are using Windows this is how you'll remotely log in. You'll need to install this free program called PuTTY. It's a "client" program that connects over a network to a "server," the machine we're trying to access.

http://www.putty.org/, download the putty.exe file.

To sign in, type your netID, @, and the hostname is the name of the machine we're logging into (i.e. ssander5@darrow.cc.nd.edu), and set the connection type is ssh. Set Port to 22. Click open, and you do this you will be prompted for your password. Your username and password are your netid and password.

2. ssh - "secure shell" - if you have a Mac, this is the command you will want to use to connect to a remote machine.

First you'll need to open up Terminal (you can get to it through Spotlight; I'd put it on your dashboard). Second you'll need to issue the ssh command. The format is as follows:

ssh username@machine.nd.edu

******Machines You Can Access******

student00.cse.nd.edu
student01.cse.nd.edu
student02.cse.nd.edu
student03.cse.nd.edu
darrow.cc.nd.edu – only accessible one from off campus

******In The Shell*******
What we mostly use every day on our computers is a graphical user interface (GUI) - we can click, highlight, drag, and in various other ways interact with graphical components that represent the contents of our computer.

There's a different way to interact, and that's via the shell (a command line interface, or CLI), where we can textually navigate the computer we're on. Everything you can do in the GUI you can do in the shell, though it's obviously not always as easy since you have to issue commands instead of just using a mouse. Think of the difference between a touch screen and a mouse; there's an extra level of distance between you and what you can do.

We're going to be working with a shell called bash. Notre Dame defaults to C-shell, which is pretty similar but not technically the same thing.

Some of you might see the similarities between this terminal and MS-DOS, if you remember that far back. As you can see I just have a little box with some stuff up here at the front and a little blinking cursor. As you'd expect, we interact with this by typing stuff since that's really all I can do. So let's try something. There's a command called echo that does pretty much what you'd expect - it echos what you tell it.

-->>echo hello class!

Sweet, it echoed it for us. But why is that useful? Well, among other things, we can use it to show us the values of variables in the shell. A variable is just like a variable in math. We name it something, and assign a value to it, like x = 4. You can assign variables, but the computer also has some variables it has already assigned. Let's see what my current name is:

-->>echo $USER

Shell variables start with a $ - so echo USER just gives me USER.

I can see where my home folder is with $HOME.

-->>echo $HOME

I mentioned that there are different shells. How do you know which one you're in? Well, you can check with a shell variable. $0 tells us the name of the program we're running which, in this case, is the shell we're running.

-->>echo $0

So, I told you guys that it's standard on ND computers to run C-shell instead of bash. So how do we get into bash? Simple!

-->>bash

And now you're running bash.

-->>echo $0

Remember how I said you can declare your own shell variables? Let's see how to do that.

-->>declare myvar=hello

Notice that there's no $ in there when we declare it. However, when we want to echo it we need to tell echo that this is a variable, not just the word myvar. Also, DON'T put a space between myvar and =, and = and hello. If you do you'll get this:

-bash: declare: `=': not a valid identifier

But let's check what we have:

-->>echo $myvar

Now we have a variable! Not everything can be that easy though. If we want to do something like:

-->>declare myvar=hello class

We run into a problem. Only the first word was put into the variable. I said that you need to make sure not have a space between the myvar and the =, and that's because spaces aren't just white space in the shell all the time - they can mean things. There are other characters too like slashes, dollar signs, and a host of others that mean things to the computer, so we need a way to tell it that we want the literal thing we typed. We can do this using the escape character, the \.

-->>declare myvar=hello\ class

And now it works! Use the escape character BEFORE the character you want to use literally (such as before the space). In some cases we can also put quotes around things to make sure that spaces are taken literally. The escape character is still VERY important to know, though.

You can also make it work with single or double quotes:

```
-->>declare myvar=”hello all”

-->>declare myvar='hello everybody'
```

Now, why do we have three ways of doing this? Because the rules on each are slightly different. For example, if we wanted to have the terminal echo $4 (four dollars), we can try the most common double quotes:

```
-->>declare myvar="$4"
-->>echo $myvar
```

But it’s empty... That's because characters inside double quotes are interpreted, as in Unix takes the special characters, like spaces and variable defining $ to mean something. It is reporting what is stored in $4, which is currently nothing. This can be useful in situations when you want to report a variable in context:

```
-->>declare myvar="My name is $USER and I am using $0"
-->>echo $myvar
```

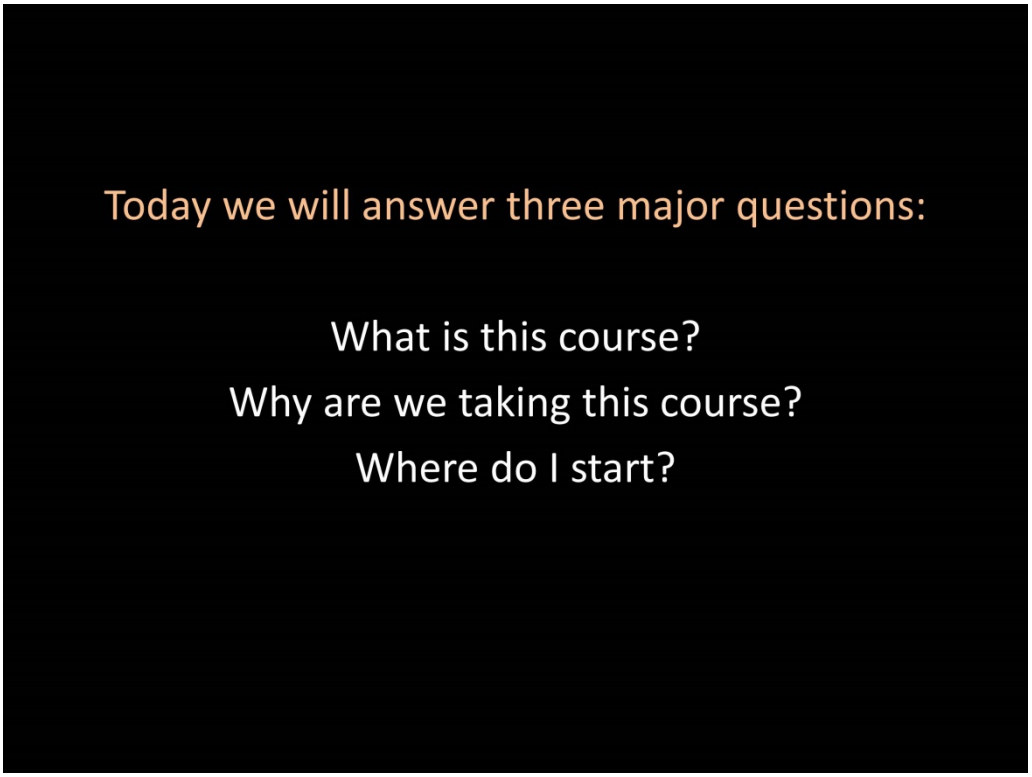
If you want it to be read literally, use single quotes:

```
-->>declare myvar='echo $USER will report your username'
-->>echo $myvar
```

If you look under Resources on the webpage or Sakai, there’s a special character list (updated as we encounter them), and you’ll see here the difference between double and single quotes. I typically use double quotes because you can use variables (like $USER or $HOME) inside of them. Single quotes, however, will allow you to use an exclamation point or dollar sign without problem because it doesn’t let anything inside the quotes be interpreted. Since you probably won’t be using an exclamation point in your shell scripts that often, it’s pretty safe to use double quotes...but like I said before, just keep in mind that sometimes you have to fiddle with things to get them to work.

.

PPT Notes for Above Information:



What is this course?

Quick Intro

Welcome!

Bringing computers to class is a good idea, I'll let you know if its required.

Actually registering for the class is also a good idea.

I'll talk more about the discussion board and projects later in the course.

"What is this course?" Syllabus Quiz

1. How much is homework worth?
2. Where is my office?
3. Where can you find course information?
4. When is the final?

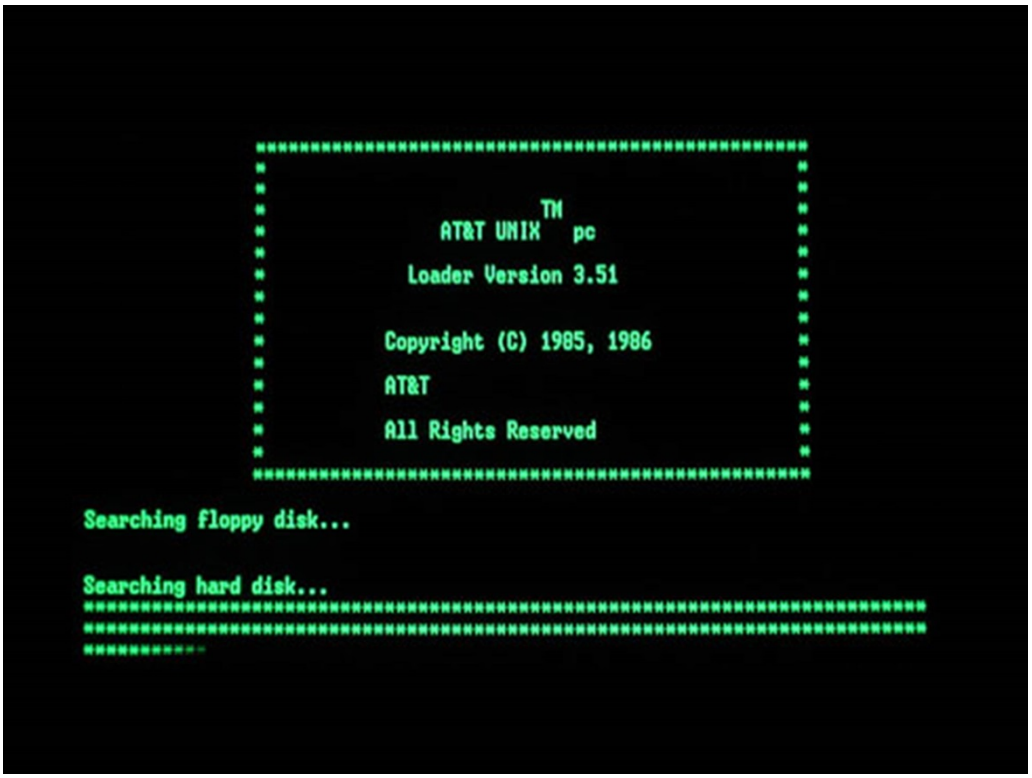
Why are we taking this course?

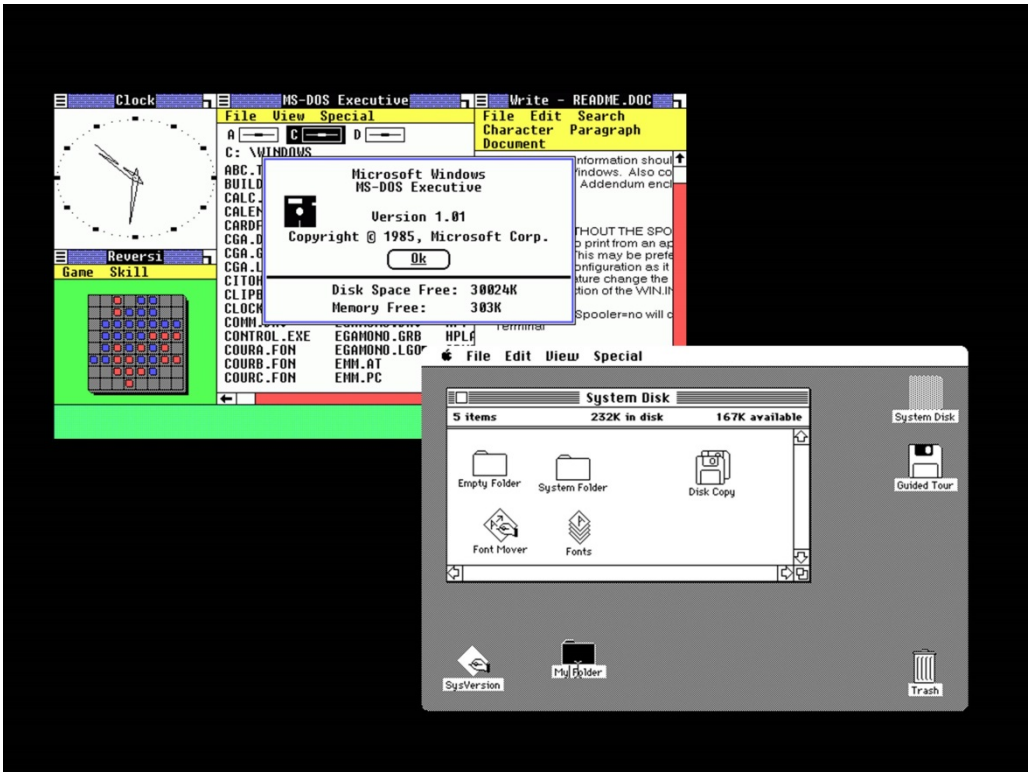
Essentially, to get you used to using Unix and command line so that you can perform your own bioinformatic analyses.

Operating System – The software that controls the computer's main functions, such as communicating with hardware, controlling input/output, etc.

GUI – Graphical user interface that allows for point and click, touch screen, and otherwise visual interaction, i.e.: Windows, iOS

Command Line – OS interface with mainly text based input/output, i.e. Unix


AT&T UNIX TM pc
Loader Version 3.51
Copyright (C) 1985, 1986
AT&T
All Rights Reserved
Searching floppy disk...
Searching hard disk...


Clock
MS-DOS Executive
File View Special
Write - README.DOC
File Edit Search
Character Paragraph
Document
Microsoft Windows
MS-DOS Executive
Version 1.01
Copyright @ 1985, Microsoft Corp.
Ok
Disk Space Free: 30024K
Memory Free: 303K
Reversi
Game Skill
File Edit View Special
System Disk
5 items
232K in disk
167K available
Empty Folder
System Folder
Disk Copy
Font Mover
Fonts
SysVersion
My Folder
Trash
Guided Tour

Unix was great at the time because it was a **"programmer's delight"**. It had "**simple**, elegant underpinnings". It was driven by a philosophy that made sense in a programming **community**:

Write programs that do *one thing* and do it well.
Write programs to work *together*.
Write programs to handle *text* streams.

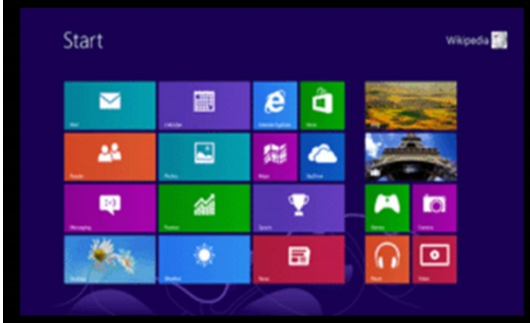
Because of this philosophy, it is designed to be:

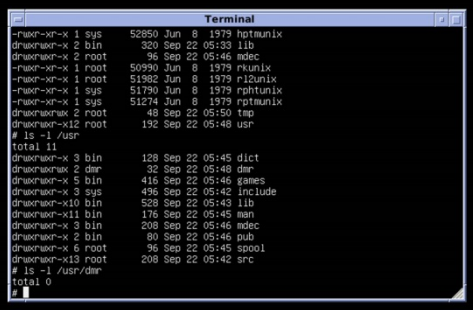
- flexible
- processing centric
- stable

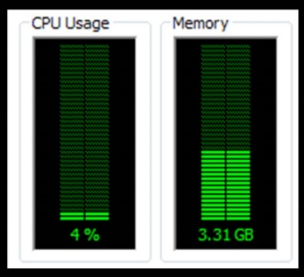
It was also freely handed out to universities at the beginning of computing, which also required it to be:

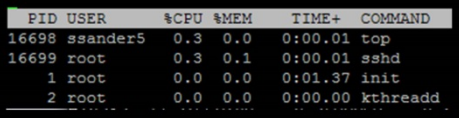
- portable
- multi-user
- secure

But why use it today?









^ Better for processing

< Better for visual tasks

Why do we want to use Unix for bioinformatics?

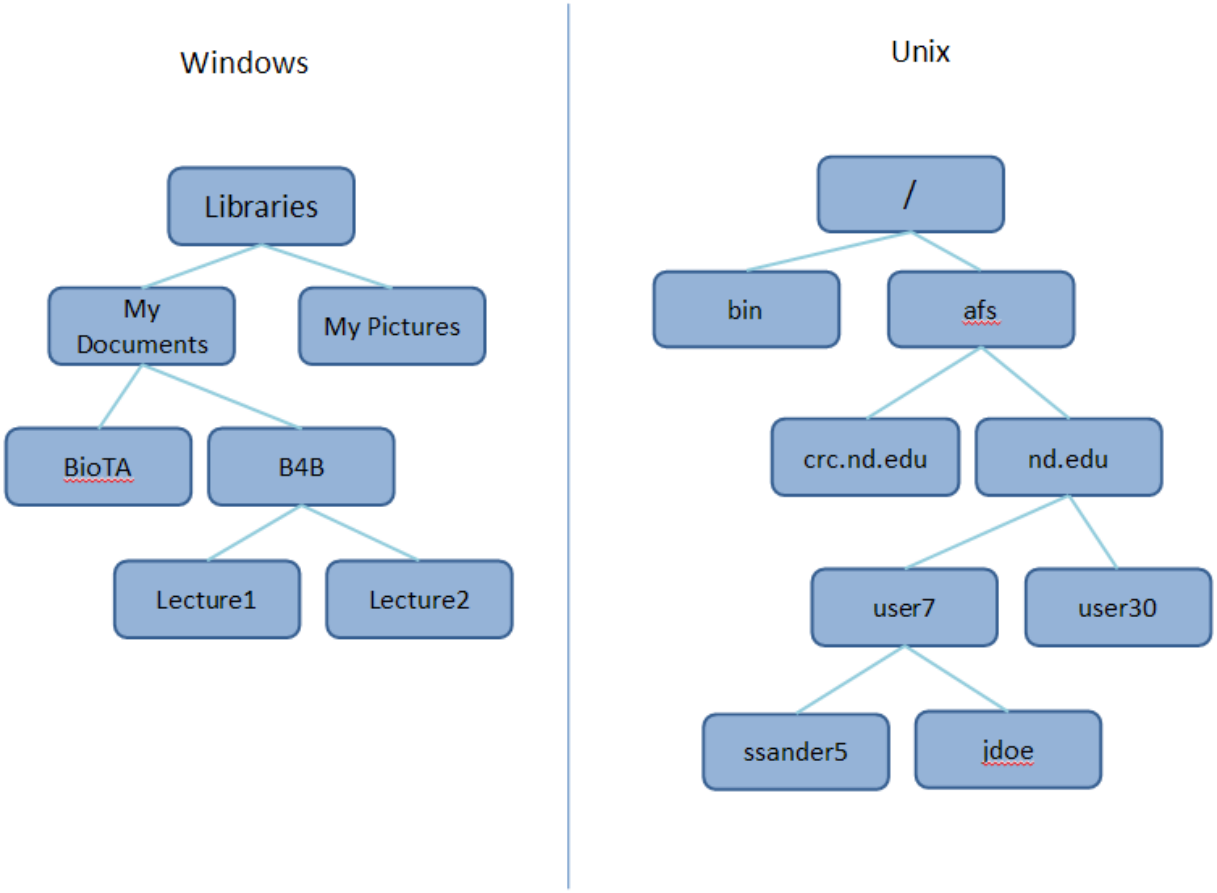
- it was built to be easy to program
- a building-block approach to programming
- its stable and powerful
- simple input/output (I/O)
- well-suited to text manipulation/text parsing
- lots of control over the system
- most bioinformatics tools are written for these systems (BLAST, ClustalW, MUSCLE)

File System Background and Introduction to Navigation

Intro to File Systems and Navigation

******Intro to the File Systems******

Most file systems are hierarchical, like a tree. Below is how Windows and Unix are structured. Note Unix has a defined root of the tree (the most inclusive folder, the parent) is /.



Unlike Windows, Unix uses forward slashes to denote separations in folders in the file system. For example /afs/nd.edu/user7/ssander5 is my home directory on the ND computers, whereas ...My Documents\B4B\Lecture2 is a directory on my Windows machine.

NOTE: Where are things like CD-ROM drives, or portable hard drives we plug in, though? On Windows, we assign each of these things (including the disk drive of the machine itself) a letter, like the C: drive and the D: drive, and they have their own hierarchical tree. On Unix, these devices can be mounted anywhere in the main tree. For example, CD-ROM files could be found in /media/cdrom0. Where they end up in the tree is controlled by the administrator (root), so we won't worry about it. The point is it all looks like one big seamless file system, despite the fact that we have external things attached to it. It's kind of a Unix paradigm that everything can be treated like a file, including your entire CD drive.

******Flags and File Viewing******

If you want to look at your home directory, recall $HOME.

-->>echo $HOME

We can also do this quicker with
-->> echo ~

Ok, great. Let's look at something slightly different, and see how you see where you are at any given point.

NOTE: My bash will look different from your bash for a little bit. In another week or so, we will adjust that for you!

You might want to see the entire path of your location (like the when we echo $HOME) even if you aren't at your home folder. You can view your current location by simply typing:

-->>pwd

This command gives us our current working directory (**pwd stands for print working directory**. A directory is the same as a folder). You can see the full path now (which matches $HOME since that's where we are currently located).

When I echo $HOME you can see that this is my afs space. AFS is the distributed file system that ND uses. I'm not going to go into the particulars at all, but this is my afs space so it's not really on the individual machine I'm on, but on the network, and I can access it from any ND machine. This is why it doesn't matter which machine you log into.

Ok...but what's in my current directory? We can list the contents using what I think of as "let's see":

-->>ls

Wonderful. I can see files, but I can't see all the wonderful information that I get on my Windows GUI, such as date of file creation, etc.

-->>ls -l

There it is (sort of). The -l is called a "flag", it's an option in the command that we ran. It stands for "list" or "long", which lists all the file information. We will talk about weird dxr-'s next week.

Ok, so now I know I have options, but what other options do I have?? Enter: the man(ual) pages!

-->>man ls

Look at all that wonderful information. Man pages give us lots of good info about the ls command, what it is, and its options. You can scroll down with the arrow keys (most programs will be like this), and you can exit just by typing q (this is also something of a standard so it's a good thing to try if you need to get out of something and you don't know how, Ctrl C is another thing to try). One important thing to note is that case does matter. While this isn't always the case, it's important to typically try to treat things as case-sensitive, because, for example, -a and -A are not the same flag.

Notice that there are shortened options with one - and longer flag names with --. This is a legacy from when (incorrectly) typing command names was costly because terminals were very slow.

Anyway, let's try another common flag, -h, for human readable.

-->>ls -l -h

Oh look, the file size is much more logical. We can also combine flags, if they are of the - variety.

-->>ls -lh

Finally, the last really common "let's see" option -a, for all. Let's see a list of all the files in human readable format:

-->>ls -lah

Look at all those files! Any file that starts with a "." is hidden from normal lists. This is akin to hidden administrative files on mac and windows and refers to administration files as well. A notable one that we will work with later is .bashrc, which is your bash profile that customizes your bash experience.

NOTE: Notice my pretty colors? That is a flag I have permanently turned on in my bash profile:

-->> ls --color

Some more to check out, look at the man file if it's not obvious:

-->>ls -r
-->>ls -R
-->>ls -t

******Navigating in the Shell******

Well, I don't want to stay in my home directory forever, so let's go adventuring. You can go somewhere else by **changing directories**. Let's go somewhere important, like... your dropbox, where all your homework and such will go.

It's located:

/afs/nd.edu/coursesp.16/cse/cse60132.01/dropbox/YOURID

How do we get there?
Syntax:-->> cd LOCATION

-->cd /afs/nd.edu/coursesp.16/cse/cse60132.01/dropboxes/YOURID

But let's get into a folder that is shared to the whole class.

-->>cd /afs/nd.edu/coursesp.16/cse/cse60132.01/

Ok, now we are in the course folder. Let's see what's in here!

-->> ls -lah –color

Ok, let's make our own folder, with the make directory command:
Syntax: -->> mkdir NAME

-->>mkdir NewData
-->>ls

A quick aside about directory naming: Earlier we talked about how spaces mean things, and thus we need to escape them when we want to use them. For this reason, I try to avoid giving folders names that have spaces in them. It's super straightforward and easy to mess with in the GUI, but when you're using the command line it gets annoying having to type stuff like cd This\ Folder when ThisFolder is just as readable and much less annoying to type.

Another reason not to do spaces? If you use :
-->> mkdir All the things
-->> ls –lah

You will see that there are three new folders. Because spaces separate commands in Unix. You are telling it to make three folders. If you instead use:

-->> mkdir "All the things" or
-->> mkdir All\ the\ things

You will get a folder and enjoy the very annoying experience of having to escape those spaces every time you refer to the folder.

You can see from the blue text, as well as the d all the way to the left, that Example, ExtraExample are directories (also apparent if you use –F). We also see two more at the top, "." and "..". Let's investigate...

-->> cd .

-->> pwd

...huh. That did nothing... ok, let's try the other one...

-->> cd ..
-->> pwd

...Ok, that brought us up one folder. These are two special references (like ~ was for HOME), that are shorthand for the folder you are in and the folder above. Can we use ~ the same way?

-->> cd ~
-->> pwd

Yup. And there is one more special character for cd, "-".

-->> cd -
-->> pwd

Review:
~ -> HOME
. -> present folder
.. -> parent folder
- -> previous folder

So... why do we want these symbols confusing us? Because you can type in locations one of two ways:

1) from root (absolute): cd /afs/nd.edu/courses/cse/cse60132.01/ExtraExample
2) from where you are(relative): ../ExtraExample

It is often much easier to do the latter.

There's one quick trick I'll teach you too, and that's tab complete. If we look back in our home directory there are two folders starting with E, Example and ExtraExample. If you type cd E, it will fill in the rest of the characters until there is some ambiguity. You can then type "a" and tab, and autocomplete Example, or you can type "t" tab, and it will autocomplete ExtraExample. You can either have the trailing slash or not for folders, but tab complete will always stick it there. Using tab complete will ensure that you are spelling things correctly and getting the path correct. If you hit tab and it doesn't complete words you think it should… you did something wrong ^_^.

You can also change multiple directories at once. So in Example we saw there's a folder called Awesome. So from home we can type cd Example/Awesome. This can also be tab completed.

******Interacting with Files and Directories******
Now we can move around in the shell and see where we are, but we also need to be able to look at file contents, make directories, and move files around.

Let's start with a basic way to look at the contents of a file. I have a file in here called This.txt, so let's look at what's inside.

-->>cd ExtraExample
-->>ls
-->>less Woobly.txt

Note that you can tab complete the name. If this were longer I could scroll through like I did on ls's man page. Like with man, you exit by typing q.

We can also move files:
Syntax: mv <SOURCE> <DESTINATION>

-->>mv Woobly.txt ../Example

-->>ls

We can also look in folders without moving:

-->>ls ../Example

Mv also renames things, renaming something is essentially 'moving' something to a new filename.

-->>cd ../Example
-->>mv Woobly.txt Zippity.txt
-->>ls

And now Woobly.txt is renamed Zippity.txt.

NOTE: Please for the sanity of all, use file extensions. It's hard to know what something is if you don't manually label it.

We can also copy or remove things. Copying and removing directories is a little different since there's stuff in them.
Syntax: cp <SOURCE> <DESTINATION>

-->>cp Woobly.txt Wobbly.txt
-->>ls

Sometimes directories (folder act different). If we try to move a folder, we get an error:
-->>mkdir Test
-->>cp Test Testing

How do you solve this?
-->>man cp

-r is a flag that tells cp to be recursive (to access everything in the directory).

-->>cd ..
-->>cp -r Test Testing

Onto removing files:

-->>rm Copy.txt

Similar to cp, directories need a -r flag

-->>rm -r Testing

Depending on permissions of the files, it might ask if you're sure you want to delete things. To make it not do this, you can force your remove to happen with the -f flag.

-->>rm -rf Testing

If you're using rm be VERY certain that's what you want to do - there's no trash can. I've accidentally deleted things before and have been very, very sad. On AFS there is a YESTRDAY folder where they back-up your AFS from the day before every day, so if you had the file yesterday you can pull it back from there, but if not... you're out of luck.

One last skill that you'll probably want is the ability to create and edit files. There are various editors for command line - the most simplistic is nano, so I'll show you how to use it. If you want to get adventurous you can look into emacs or vim (I don't), but they're full of lots of commands and weird key combinations, so we'll stick to the easy stuff.

To open an existing file or create a new one is the same. We'll make a new file called tester.txt, but if tester.txt already existed this command would open the file and display its contents for editing:

-->>nano tester.txt

Now we have a new file! I'm going to type some random stuff in it and then close it. To exit nano, hold down control and hit the x key (at the bottom the ^ is for control). It'll ask if you want to save (hit y for yes), and then it'll ask you the filename to write. Unless you want to save to a new file just hit enter here, as your filename that you opened is already in this space. If we run less on that file we can see that what I put in there is now saved to this file.

If you are going to create a bunch of files, we should probably know how to make new folders as well. This is done with the mkdir command:

-->>mkdir Stuff

A good thing to do, starting right now, is to get in the habit of making a file called README whenever you make a new major folder. Why this README file? README, in all caps, is a conventional file you will find in many folders, particularly if you downloaded it or it's a shared use folder. A VERY good habit to get into is to make one for any folder you use for the sanity of 1) your PI –who will be thrilled to know what these files are when you graduate, 2) your future self – who will be thrilled to know what AvB.redone.txt.filtered is and how it was redone and filtered, and 3) your colleagues/coauthors – who will be thrilled to know what any new "redone" files are floating around in the common space. You don't have to do it for every folder, but it is much better to overdo it than under do it.

-->>nano README

******Moving files across computers******

There are many times when you will want to move a file to your computer from your afs space or vice versa. There are two ways to do this, one that is command line and one that is just easier.

1) command line - secure copy
Secure copy, scp, allows you to copy files from one machine to another. It works much like cp and move, but you have to give it a little more information on where things are, because the assumption is that it's on the same computer unless told otherwise. This command is very similar to copy, but with the additional <DESTINATION COMPUTER> before a file that exists elsewhere.

Syntax: scp <Source> <Destination Computer>:<File Location>

-->>scp myfile.txt ssander5@opteron.crc.nd.edu:

Note the : with nothing after it in the example. The default is to copy files into your home folder. If you wish to copy it elsewhere, you can also do that by putting the location after the colon.

-->>scp myfile.txt ssander5@opteron.crc.nd.edu:samples/

Additionally, you can copy from opteron to your current machine:

-->>scp ssander5@opteron.crc.nd.edu:samples/samplefile.txt .

It's still the same syntax - you are specifying where the file is and where it is going. And by the way, you cannot use you have to log into the other computer (which is why you use the <username>@<computer> and have to put in a password). If you've watched the sign in video, you know you can't sign into a computer from itself.

2) FTP - Filezilla
Unfortunately, this is not always easy to do, especially if you are on a Windows machine or if you have weird firewall/security issues going on. Most biologists I know prefer to instead use FTP or Filezilla.

Obtain and install Filezilla from: https://filezilla-project.org/

WARNING: DO NOT DO THE DEFAULT INSTALLATION. It's not a great idea for anything really. Make sure you do the custom installation and unselect anything that is not Filezilla.

Filezilla connects to the remote computers in a similar fashion to Putty or ssh - you specify your username, host computer, password, and port and it will connect to the remote computer. You can then access the files on the right side (remote site), or left side (local site).

******A Few Last Useful Things******
I'll give you two more quick tips to help you navigate on the command line. First, if you want to reissue a command you've already done, like using less again. If you hit the up arrow key you can go back through your previous commands in order. Second, you can use wildcards in commands. I'll show you with ls.

ls file*.txt
ls file?.txt
ls file??.txt

* and ? are both special characters (like space), so if you want them taken literally you'll need to escape them with \. * says there can be any number of characters between file and .txt (including none). Each ? represents a single wildcard character, so the second will find something called files.txt but not files1.txt. The third will find files1.txt but not files.txt.

It is helpful to use ls to list what you are planning to remove, to make sure you don't accidentally delete the wrong thing!

Some other useful (or fun) commands are:
top - see who's doing what on the system (q to quit). You can look at a specific users processes on our machines by typing u, and then typing the username and hitting enter
info - like man, but more comprehensive (q to quit)
date - get the date
head - get the top few lines of a file
tail - get the last few lines of a file

More Optional Practice for Navigation

1. Do the following:
    a. Create a file called practice.txt in your home directory
    b. Create a directory in your home folder called temp
    c. Make a copy of the practice.txt file in temp with the name prac.txt
    d. Change the current working directory to temp
    e. List all the files in the directory temp
    f. Find the contents of the file prac.txt and practice.txt, without moving out of temp.
    g. Change the current working directory to you home directory.
    h. Execute the following commands and compare how they work with respect to files, directories and the contents of directories:
        ls
        ls practice.txt
        ls temp
        ls *
    j. Clean up by removing practice.txt, the temp directory, and all files within

2. Let's do some more simple commands but with more files:
    a. Set your current directory to your home directory
    b. Make a new directory called lots
    c. List out all the files in the Shared/lots/ folder in the class folder. There are almost 600 files. The list will probably scroll by too quickly you can scroll back in your terminal window to view them.
    d. Each one is a small file. These files have only a little text inside each one but we can pretend they are the real sequence files generated with the sequencing machines. They have names like jep12.c1.b1 or jep12.c1.g1. These names reflect the plate "jep12" the well "c1" and if they are forward "b1" or reverse "g1" reads.
    e. How would you copy all the files in the Shared/lots/ directory to the lots/ directory you made in step b above?
    f. Make a subdirectory call some within lots/
    g. How would you copy only those files that are reverse reads from lots/ into some/?
    h. Let's say you find that the sequencing machine had a problem and well c5 reads were defective. How would you delete all the files that came from this well?
    i. How would you delete all the reactions from plate jep10?
    j. Delete all files inside your lots folder and delete the lots/ folder itself.

Checkpoint: Basic Unix Understanding

Check your level of understanding from the first section of the class.

Match the command with its purpose:

1. less	A. Soft link
2. nano	B. Remove a file.
3. mv	C. Move or rename a file.
4. cp	D. Open a file for editing.
5. rm	E. Open a file without editing it.
6. top	F. List the resources in use on the computer cluster.
7. head	G. Get information
8. tail	H. Find a program's home folder
9. info	I. Copy a file
10. date	J. List last ten lines of file to terminal.
11. which	K. Output date and time
12. ln -s	L. List top ten lines of file to terminal.

Match the special character to its function:

1. Root or separate folder	A. /
2. Escape a special character to be read literally	B. "
3. Group text to be read as literal (but variables are still interpreted)	C. \
4. Group text to be read as literal (bash variables are not interpreted)	D. *
5. Indicates a variable	E. ?
6. Used to match any number of characters	F. '
7. Used to match any single character	G. :
8. Used to separate directory locations in PATH	J.$

____ will define the variable myvar as Hello class
____ will print "Hello class" to the terminal
____ ssander5@darrow.cc.nd.edu
____ removes all previous commands from the screen.
____ starts bash
____ will print the current directory's location
____ will just list the files in a folder
____ will list all the files in a directory
____ will list all the files and the information about them in a folder
____ will make a folder called HW2
____ will change the working directory to HW2

Pick the correct commands:
A. Echo Hello class
B. echo Hello class
C. echo "Hello class"
D. Echo "Hello class"

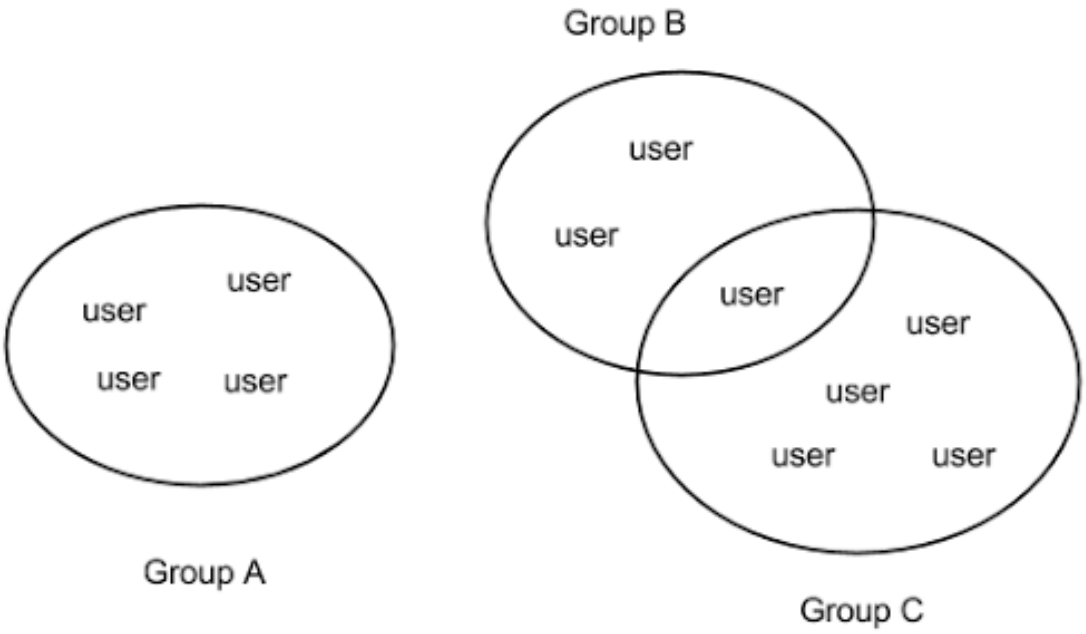
Pick the correct format of the command:
A. ls -lacolor
B. ls -l -a -color
C. ls -la --color
D. A and B
E. B and C

Unit 2: Introduction to Unix – Profiles, Programs, and Data

Permissions and Executable Files

I mentioned earlier that the root administrator of a system can control where things like the CD-ROM and AFS are mounted. The administrator is often called root, because they're the only person who can usually make changes to the root directory (/). Root is a special, all powerful user - a super user. Sadly we're all just regular users in Notre Dame's systems.

Root owns some files that we can't see or change. We can also own files that others can't see or change by setting up file and directory permissions. Root (and to some degree, we) can put users into groups, which get their own set of permissions. The convenience here is that we can add a person to a group (such as the class) and they will inherit all the access that have been set up for the group. Otherwise, we would have to set each folder manually.



Each file and directory has an associated user and group, based on who created them. Using ls -l (remember, -l for long), we can see permissions for files and directories.

```
-rw-r--r-- 1 ssander5   dip      62 Jan 15 16:10 class0.txt
drwxr-xr-x 2 ssander5 dip 2048 Jan 15 17:10 Example
```

Now to discuss those letters at the left - I already mentioned that "d" means directory (folder), but what about r, w, and x?

Permission	Files	Directories
r	can read the file	can ls the directory
w	can write the file	can modify the directory's contents
x	can execute the file	can cd to the directory

Ok that makes sense, but why are there multiple r's, w's, and x's? These are for the different levels of users. There's the owner of the file, the other members of the owner's group, and all others. The first three (for class0.txt that's rw-) are for the user, the second three (r--) for the group, and the last 3 (r--) for others. As you can see, the order is always rwx, and a - means that that permission is not given. So no one can execute class0.txt, but everyone can read it, and only the owner can write to it. Similarly, Example has permissions set for the user (me) to be able to read the contents (ls,cp), write to the folder (mv, rm, etc.) and execute (cd into the folder).

Everyone else can see what is in there and copy the files, but not modify them. To change file and directory permissions, we use the command chmod. We can change permissions for the user (u), the group (g), or others(o), and we can either add(+) or subtract(-) the three kinds of permissions (r,w,x).

NOTE:** if you are going to follow along, you should make your own file in your home space (using nano). That way you are the user, not me, and can manipulate these options.

-->> chmod u+x class0.txt

would add execute permissions for the user to class0.txt. You can tack them all together:

-->> chmod ugo-rx class0.txt

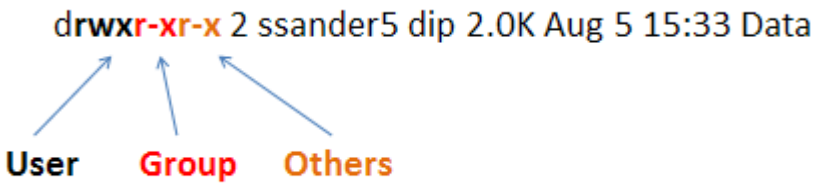
This command removes read and execute permissions for user, group, and others. One important flag for chmod is -R, which executes recursively on the contents of a folder. So if you want to add read permissions to every file and subdirectories (and their files and subdirectories, etc.) to execute for the user to the directory permissions/, you'd do:

-->> chmod -R u+x Example

Another way to do this that you will likely see and may prefer is through a numeric system. For example, if we have:

```
drwxr-xr-x 2 ssander5 dip 2.0K Aug 5 15:33 Data
```

We know that there are three parts to the permissions, each with the three permission levels:



If we attribute a number to each permission level, using binary counting, we get the following:

R W X
4 2 1

If we want to give permission to read, write, and execute to the user, we can add the number for each permission setting (4+2+1) and get 7. We have now coded an entire set of permissions (for one level) in one number. We can do the same for the group and others. Say we only want to give them read and execute permission, that would be coded as a 5. Because we are using binary increases, there is no two combinations that result in the same number (this numbers system doesn't work with 1,2,3). Now we can:

-->> chmod 755 Example

Why would we do this? The reason comes down to convenience of typing and convenience of memory. It is much easier to remember 755 (executable) or 644 (read enabled but protected file) than chmod ugo+xr <File>, chmod u+w <file>. It is also convenient to not have to know the default permissions, the numeric system defines all three groups, with all three parameters. And finally, it's much easier to define different levels of access in one line of code. I won't require you to know this system, but it is very useful.

******Too Bad AFS Doesn't Work Like That******
Not that chmod will never be useful for you (its actually really useful!), but changing permissions in AFS, where you'll primarily be doing your work, is a little different. AFS has users and groups, but **users can define their own groups**. This set up makes sense when you think about the number of labs/groups/departments that use and manage space but how few people have root privileges (i.e. no one in the biology department).

Access control lists (ACLs) can be set on a per-user and per-group basis instead of just one generalized owner group and other users. In Unix permissions, I could only define my permissions, "dip" permissions, and global permissions. I couldn't be in different groups – only "dip". ACLs make the situation a bit more complicated, but they allow users to belong to several groups.

The ACL system is just a fancy, more complex version of our typical rwx permissions, and they are set only on directories, replacing Unix directory permissions (**so the Unix-based permissions on directories are ignored**). Basically the Unix permissions are only meaningful for files, and only for users. Group and global settings are overridden by ACLs.

Permission	**Allowed in the Directory**
R	read files
L	list files (through ls, this is an important ability)
I	insert files
D	delete files
K	lock files
W	write to files
A	administer files (like changing ACLs)

ACLs are accessed through the fs (file system) command. For example, to look at the ACLs for a directory we can call:

-->>fs listacl .

to list ACLs for the current directory (.), though you can substitute whatever directory you want.

Setting ACLs is a bit more involved:
Syntax: fs setacl -dir <directory> -acl <user/group> rlidkwa

fs setacl -dir DirectoryExample -acl <user/group> rlidkwa

This is the long form of the command, and I recommend using this one if you have to look it up later (highly likely). You may see shorter versions of the command if you google around. There will be a permissions short guide posted for convenience if you have to change these things later (sometimes, you end up doing this for your lab group if you are the only one who knows AFS permissions).

There are two help commands for working with ACLs as well (man fs isn't great. I also recommend google):

-->>fs help
-->>fs help setacl

NOTE: The ACL really only affect directories. Technically, AFS a file inherits the ACLs for the directory, and everything other than user permissions are ignored. The user permissions are then more or less control the file ACL.

For example: If you have a directory with full open access, and a file within that has a Unix user group permission of –w, anyone with an ACL of "wl" the ability to write. If that file has a Unix user group permission of ---, even if they directory has full ACL permissions (rlidkwa), no one can write to that file. **This is kind of complicated, but you can more or less think about ACLs as folder permissions and Unix user permissions as file permissions.**

Quick review:
If you had a data folder, what commands would you execute in order to
1) give read and listing access in Unix, but protect your data from alteration?
2) give read and listing access in afs, but protect your data from alteration?

******Making your first program!******
Sometimes we want to use a set of commands frequently together, and not have to remember them or write them out each time. We can arrow up, but only really during that session. Instead, we can write a tiny program.

Remember the philosophy of Unix? Write programs that do one thing and do it well. Write programs that work together. Write programs to deal with text.

We are going to write a tiny program, using our bash commands that we learned last week, that deals with text. We are also going to learn a couple good practices for coding.

Let's first decide the purpose of the program, which is going to be admittedly a little silly at this point. Let's make a program that tells us who we are and how much space is being occupied by our current folder. First we have to figure out what commands we will use. We know echo will output to the screen so, let's start with that:

-->>echo "Hello, $USER. This folder is using:"

And here's a handy command that is listed on the website and the notes, but that we didn't explicitly go over in class.

-->> du –hsc

Now that we know our commands, we can put them together in a file.

-->>nano diskusage.ba

.ba is the extension for bash programs.

-->>echo "Hello, $USER. This folder is using:"
-->> du –hsc

Save and quit. Now let's change this to an executable program, rather than a run of the mill file.

-->>chmod 755 diskusage.ba

Let's try it! The ./ is required in some shells and it is simply saying that the program is in the current directory.

-->>./diskusage.ba

Hello, ssander5. This folder is using:
11K .
11K total

Fantastic. However, we're not done. We've made a little program that does one thing, and it handles text. However, there are a couple of things that make programs easier to share and use with other programs (make it do its job well) - the first one is commenting!

It seems silly here, but we're doing this for practice. To comment in a bash program, you use the # character. We can do this in the command line too.

-->>echo $HOME #This will print the home directory

So let's add a comment to the top of our little program that says what it does, and how to call it.

-->>nano diskusage.ba
-->>#This program greets the user and displays the directory size.
-->>#It is called by: ./diskusage.ba

One final consideration on that part - our user may not be using bash. We're going to add another line that won't work outside of bash to demonstrate this.

-->>nano diskusage.ba
-->>declare myvar="Enjoy your day, $USER" #Declare doesn't work in csh!!
-->>echo $myvar
(save and exit)
-->>csh
-->>./diskusage.ba

Hello, ssander5. This folder is using:
11K .

11K	total
declare: Command not found.
myvar: Undefined variable.

Let's add one little line that makes sure the computer knows this is to be run in bash.

-->>nano diskusage.ba
-->>#!/bin/bash
(save and exit)
-->>./diskusage.ba

Hello, ssander5. This folder is using:
11K	.
11K	total
Enjoy your day, ssander5

Great! Why didn't we just put the "bash" command in there first? Because we don't want to force our users to be in bash after the program is complete. We can check this by making sure:

-->>echo $0

Notes about commenting:
"Am I commenting enough?" It's a question I got a lot last year, so I figured I'd elaborate here for everyone!

Commenting style is one of those weird things people are weirdly picky and judgy about but it's much more critical to worry about that in large scale coding projects (i.e. Trinity) than in anything you will likely write for a while.

Situations in which you will often see comments:
Comments by classically trained programmers will often include a bunch of stuff at the top about how to run the program – kind of a README embedded in the program. The comments throughout the code are usually more on the sparse side. Most people who are digging around in code are people who can read it. Programmers don't need to spell out every line, because their usual end user doesn't need it. Functions may be labeled, major steps delineated, clarification on any really common things that need it, and sometimes things people intended to remove before it became public ^_^.

Situations in which YOU should comment:
Your audience is much different: future you and people at around your skill level (…and me).

- You should ABSOLUTELY put information at the top about
    - how to call the program
    - who wrote it

- the purpose if not obvious
- the input and output it needs.

What I try to do (make it pretty):

```
################################################################################
#
#       This program does whatever I needed it to do.  It is called by
#               ./thisprogram < input.fasta
#       and will output something I wanted at the time.
#
#       Author: Sheri Sanders 2016
#
################################################################################
```

- Everywhere else is a matter of choice.
  - Is it convoluted enough that future you could use a map through your calculations? Explain it now.
  - Is it something that may need to change in the future? Make a note.
  - Are you going to be looking for this specific place in the code? Make it easy to see while scrolling or hashtag it with a logical search term.
  - Is it something that you've been asked or had to think through more than once? Never have to do it again.

Examples:

```
###########DECONTAMINATED, ANNOTATED TRANSCRIPTOME COMPLETE################

#The following will fail if the input file is not sorted. (I should probably make this print something if it fails here...)

#In the following bash magic, we make an array of library names, then calling bowtie on each set and convert it into a more compact output (bam).
```

- Resist the urge to comment every line. It's almost never needed. Trust me, I can probably figure out what you are doing, since I likely asked you to do it ^_~.

Tl;dr: Write useful comments for your future self and people with similar skills to you, whatever that level may be. This of it as a line by line README.

Checkpoint: Permissions

____ will change the permissions of test.txt from rwx to r—
____ will change the nd_campus groups to be able to delete and insert files in the current AFS directory.
____ will list the options for the setacl command in AFS

Given:

```
drwxr-wr-w Files/
dr-xr--r-- Backups/
-rwxr-xr-- Text.txt
```

What permissions does the group have for Text.txt
A. Read, write, execute
B. Read, execute
C. Read only
D. Read, write

Given:

```
drwxr-xr-x Files/
dr-xr--r-- Backups/
-rwxr-xr-x Text.txt
```

What permission do all users have for Files/?
A. Read
B. Read, write
C. Read, write, execute
D. Not defined

Given:

```
course rlidkw
admins rlidkwa
nd_campus rl
instructor rlidkwa
```

What can the admins do that the course group cannot?
A. List all files
B. Remove files
C. Change permissions
D. Access the directory

Given:

```
course rlidkw
admins rlidkwa
nd_campus rl
instructor rlidkwa
```

What Unix permissions are similar to nd_campus AFS permissions?
A. read B. write
C. execute D. None

PATH and .bashrc

Programs, Commands, and $PATH

Less, nano, and our little greeter.ba are all programs, but ls, man, cp, etc are commands. What is the difference? Commands are built into the shell, and some programs are as well. However, where they exist is slightly different and very important.

A program is a executable file, that the operating system can read and run. We've run echo, but what IS echo?

```
cd /bin
less echo
```

If we try to look at the file, it tells us it's in binary. We've looked at it anyway and its pretty much gibberish. This is because it has been translated into something that the machine can read (but we can't).

Sometimes we'll want to run programs with their full path name (from root) to make sure that our shell knows where to find it.

```
/bin/echo hello class!
```

How come we don't have to always define the full path if the file isn't in our folder? This phenomenon is the result of a very important shell variable $PATH.

```
echo $PATH
```

As you can see, /bin is in there. I have a lot of other stuff in here by my own doing, which you will learn how to do shortly. Also note that . is in PATH as well, which makes it so we don't technically need to call our greeter.ba program with ./greeter.ba, we could just use greeter.ba. However, it is generally safer to make sure you are calling the right program (who knows what is in all those folders!). All the folders are separated by a :. It is important to note that bash takes the FIRST instance of a program it comes across in the order of the folders in the PATH variable. If you have two copies of a program, it will run only the first it comes across. This is again why ./ is helpful, since it is usually one of the last folders listed.

Because of this order rigmarole, sometimes we want to know where a program is, to make sure it's the correct one.

```
which less
```

This tells you from where less is being executed. Let's try looking for cd. If we call cd though… it says there is no cd in our path anywhere. That's because cd isn't a program, it's a command built into bash. Doesn't change anything, just interesting to note.

If I wanted greeter.ba to run anywhere I am, I would have to add the folder it lives in to my PATH variable, or more conveniently, put it in a folder that is in my path and holds all my programs. This is the /bin. We don't have permission to add things to /bin, so we will make our own.

Go to your dropbox
mkdir local
cd local
mkdir bin
cp ../../Shared/permissions/greeter.ba /bin

Local/bin is a very common folder for installations, and some programs will look for it when installing. No let's add it to the path, so Unix can find programs we write.

declare PATH=$PATH:<NEW DIRECTORY>

Remember the declare commands we carried out earlier? By saying PATH=$PATH:newstuff, we're appending the new stuff to what is already stored in PATH. Remember that : separates directories in your path. You can add several at a time if you'd like:

declare PATH=$PATH:<NEW DIRECTORY>:<NEW DIRECTORY>: (etc)

However, if we sign out, this variable is lost. Variable definitions are cleared every time you leave bash. How do we add these permanently? By using that .bashrc file I mentioned the other day. It is an administrative file, with our profile information in it.

Modifying .bashrc
First, let's get to the file:
cd ~
nano .bashrc

Find the line that says 'export PATH=' – if it doesn't exist, add it! Export is another way to declare variables in the shell (see the discussion board for more). Add a colon after the last directory listed, and insert the ABSOLUTE path to your bin directory. Save and exit.

You have to restart bash to affect change (it's read at log in). You can either exit out to csh and back in, or you can type:

source .bashrc

This command tells the shell to run the current .bashrc without having to log out. Now let's check to see if it worked:

echo $PATH

In Class Activity 1: Make it your own.

Here are a couple more changes to .bashrc to make your life better. Work in groups and be sure to ask each other if something seems weird or if you have any random questions about the environment. I will be around to chat with groups.

1) Open .bashrc again, and find the PS1 variable, which determines the text for the prompt (if it's not there add a line that says PS1=)
There are a few special instructions you can give the prompt to display variables:

\u = user
\h = hostname
\w = current working directory

You can replace \w with $(pwd) to give you the entire path instead of just the current directory (which is how I have it sometimes).

declare PS1="\u@\h \W$ "

But… that's boring. Here's how to make a really snazzy prompt:
http://bashrcgenerator.com/
Make it however you want, in a point and click manner, and then paste that PS1 variable command that it gives you into your .bashrc file. Save, exit, and source.

2) Adding in some aliases:
Find where aliases are in the .bashrc file, so we can add below them to keep our file organized. To make your own, add a line like: "alias ls='ls –-color' ". Another is to make nano start without word wrapping. Word wrapping is nice for human eyes, but can mess up your coding. "alias nano='nano -w' ".

3) While we're customizing, let's make bash come up automatically!
I normally have my shell automatically log into bash when I open up a new connection to an ND computer, instead of going to the default C-shell. You can do this by adding 'exec bash' (no quotes) to the .login file above 'if ("`tty`" != "/dev/console") exit' (mine is commented out for the class)

4) Two more convenient things to save from having to cd all over the place.
a) We can add a soft-link to our dropbox in our HOME directory. This is like adding a shortcut on your desktop. In terminal in our HOME directory, type:
Syntax: ln -s {target-filename} {symbolic-name}

-->>ln -s {target-filename} {symbolic-name}

Where target-filename will be your ABSOLUTE dropbox path and symbolic-name can be whatever you want (though calling it dropbox might be the smartest).

If you want to remove the link you can just call a rm on it without the trailing slash - this

won't remove the original, just the symbolic link. **NOTE:** Try removing this file. There is a difference between rm dropbox/ and rm dropbox. Can be very confusing if you are using tab complete!

b) Add the following line to your .bashrc
Declare course="/afs/nd.edu/coursesp.16/cse/cse60132.01/"

This allows you to go to the course directory by simply typing cd $course.

Checkpoint: Getting Data and Moving Around, Install Pre-activity (HW1)

Part 1: New Data

****

Email from Dr. I.M. YourBoss

Hey,
Your data just came back. Can you download it and tell me if it looks okay?
-IMY

****

1. Get the data onto the machine:
I put the data file on the website and Sakai. Download it.

2. Move data:
In your dropbox folder, make a new folder called "HW1" and move the data into this folder.

3. Make a README:
In your dropbox folder, make a file called “README” with a quick description of the purpose of the folder and origin of the files.

4. Can you open the data?
Try to open the file and see what is in there.

Bonus: Can you figure out how to make this file unzipped? We will talk about it soon, but feel free to try to solve it on your own! (+1 point)

REQUIREMENTS: HW1 folder in you dropbox, with the data file and README in it. (3 points) Bonus point if you get it to be unzipped.

Part 2: Troubleshooting (you will do this a lot):
For each of these "quick emails" from fictitious coworkers, write out the corrected line and a quick explanation as to why their original code didn't work.

1. I cannot seem to get this variable to work! Help! This is what I'm getting:
-->>declare FILE = text.txt

it gives me:
bash: declare: `=': not a valid identifier
bash: declare: `text.txt': not a valid identifier

2. Ok, I have that squared, thanks! But when I try to call the file, it just tells me the file name!

-->>echo FILE

it gives me:
FILE

3. I cannot remember how to remove a directory. Where do I look that up? It keeps telling me that I cannot rm a folder because it's a directory...

4. I cannot get ls to tell me the file size. When I try ls lah, it just complains that it cannot access lah: No such file or directory? What's going on??

5. Why can't I declare a variable?!

-->>declare $FILE=text.txt
bash: declare: `=text.txt': not a valid identifier

REQUIREMENTS: Create a text file called "Troubleshooting.txt" in your HW1 folder that has the corrected commands and a short explanation why it was wrong (5 points).

Part 3: Installing programs preactivity:
****
Hey again,

Dr. Soandso mentioned that hmmer-3.1 was a good program to use for analyzing our data. Could you find it and install it in your afs space? Thanks.

-IMY
****
We will be talking about how to install programs in Unix soon. Before class on Tuesday, January 26th, try to install hmmer-3.1 on your own. This is good practice for when you have to install programs after this class. If you don't know where to start - try reading the README and the INSTALL files. Google any issues you have. Post a minimum of two MEANINGFUL replies on the "HW1 Part 3" discussion thread. These can be asking help with the process, links to pages that were helpful, or anything else that strikes you as useful to the group (but avoid just posting how to do it! Suggest things to google, hints, etc.). This exercise is meant to introduce you to installing Unix programs on your own (which you will always do henceforth).

Try it. Don't worry if you get stuck, it's a pre-activity! Success is not required ^_^.

REQUIREMENTS: Two posts on discussion board (2 points)

Required:
In HW1 folder:

- Data file, either in the format it is given to you or in a readable format (3 points)
- Troubleshooting.txt (5 points)
- README.txt (2 points)

Compression and Sequence Formats

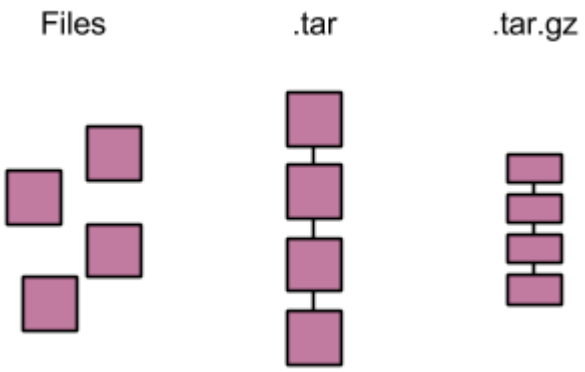
NOTE: Keep in mind that all these commands that I am teaching you work in Linux, but not all will work on a Mac. Navigation commands like cd and mv will, but programs won't always exist. One such example is "wget", which we will learn shortly. It doesn't exist on Macs, instead you will have to use "curl". I don't use a Mac and don't know most of the conversions. As such, it's your best bet to use afs/ND computers with bash in this class. There are a myriad of tiny differences in different environments (csh, using your own machine, etc.) which you can learn later, once you grasp the main concepts. Additionally, you likely have root access on your home machines, which means you aren't in as controlled (safe) of an environment compared to the ND computers.

NOTE 2: It is generally a good idea to type in the commands I am giving you, not just copy and paste. The manual entry of commands will get them ingrained into your brain much faster!

******Tarballs and Gzip******
We've learned how to get into and move around in the Unix shell, but we are going to need to start using programs to actually do some things. We will cover how to install programs shortly, but let's explore some data first.

Often files are compressed to make it easier to transfer them. Compression is very common with Illumina files and often with program downloads. In most familiar systems like Windows, these are .zip files. In Unix, these are gzipped, or .gz files. You may see them as .tar files as well, or often, .tar.gz.

Tarring links files together into a group, like a folder that is a single object to easily move. They are stuck together with tar (I warned you there are puns everywhere). Gzipping compresses the files, making the tarball (real term) easier to push around. These two things are very often used in conjunction.



So how do we untar and ungzip?

Because the file extension on the end is gzip, we have to undo that first. We will use the program gzip.

-->>man gzip
It looks like we will need the -d flag.

-->> gzip -d file.tar.gz

NOTE: This program will not run on files without the .gz extension.

-->> mv file.tar.gz file.tar.haha
-->> gzip -d file.tar.haha
gzip: file.tar.haha: unknown suffix -- ignored

Once we have decompressed the file, we are left with a tarball. Notice it is a bit bigger:

-->>ls -l

We can tar or untar with the program tar.

-->> man tar

That third example sounds like what we need. We will need the x and f flag (x is for extract, and f tells it that you are taking it from the file you are giving it).

-->> tar -xf file.tar.gz

You can actually get tar to ungzip for you as well, since these two processes are so commonly used together

-->> tar -zxf file.tar.gz #z for gzip

There are two other common compressions, bzip2 (.bz2) and zip (.zip), which we won't go through, but are on the cheat sheet. Additionally, if you ever see a file extension that you don't know, google it!

You can also tar and gzip a file as follows (note that the default is to create the compressed tarballs, not decompress them; hence the flags):

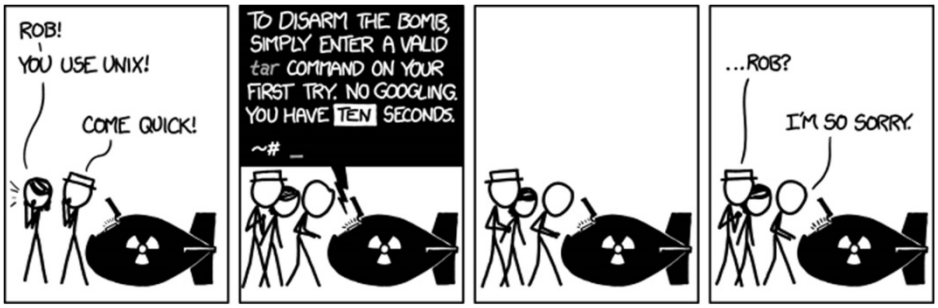
-->> tar file.txt
-->> gzip file.txt

NOTE: The defaults for both tar and gzip are to CREATE compressed folders, yet the most common usage (and therefore what we just went over) is to use these commands to unzip and untar files. Why is the default to create and compress? BECAUSE BACKING UP FILES

IS IMPORTANT AND SPACE IS LIMITED. Remember how we said rm is forever? It's generally a good idea to have a backups folder of anything important. As space is limited, it's also important to use space frugally - hence tar and zip.

tl;dr - USE THESE COMMANDS TO BACKUP COMPRESSED VERSIONS OF YOUR DATA ^_~

Bonus info on tar:



It turns out that tar has three flag syntaxes:
- long options (--file)
- short options (-f)
- old options (f)

Old options are to be clumped and parameters for them follow. With short options, parameters much follow the flag they belong to:

i.e. tar cfz archive.tar.gz file

tar -cf archive.tar.gz –z file

tar –czf archive.tar.gz file

Some flags (-f) are looking for input (archive file). If you use short options, that input is required directly after the flag. It is not required if you use old options.

******NewData.tar.gz******

In the homework, I had you download a file and transfer it to your dropbox space. Some of you ran into disk space concerns (make sure you get that bumped via the link in your email), so let's talk revisit checking our disk space.

To check disk space before you start moving files into your directory, you can use the two different commands:

-->>du -hcs #for use in Unix
-->>fs lq #for use in AFS (lq stands for "list quota")

I showed you Filezilla and scp earlier, but there is another command when you are getting files from the internet, which is common for source code. It eliminates the requirement of downloading the files to your machine, and instead download them directly to AFS space:

-->>wget <url>

NOTE: This command does not exist in iOS, instead it is curl. If you are logged into the machines on campus, it will work however.

We can go to the course website, copy the URL and wget the data right to our folder.

-->>wget
https://www3.nd.edu/courses/cse/cse60132.01/www/resources/Newdata.tar.gz

Now we have transported the file, which we can unzip and untar at the same time:

--> tar -zxf NewData.tar.gz

Or

--> gzip -d NewData.tar.gz
--> tar -zx NewData.tar

...and look at the Illumina file:

-->> less NewData.fq

******Fasta/Fastq Format******

Being able to generally understand a flat file is useful, so that you can generally scan though the file and see the quality, know when things are being cropped incorrectly, and just for curiosity ^_^.

Fasta Format:
Fasta files have two line types - sample headings (designated by a >) and sequences (lack a > designation):

>SAMPLE_NAME
gtcagatctcagtagc #sequence
>SAMPLE_NAME
gtcagatctcagtagc #sequence

Because the headings are designated by a special character, sequences can be on multiple lines without confusing the machine or ourselves. Anything after a heading that doesn't begin with a > is assumed to be part of the same sample:

>SAMPLE_NAME
gtcagatctcagtagc #sequence
gtcagatctcagtagc #sequence line 2
gtcagatctcagtagc #sequence line 3
gtcagatctcagtagc #sequence line 4

>SAMPLE_NAME
gtcagatctcagtagc #sequence
gtcagatctcagtagc #sequence line 2

>SAMPLE_NAME
gtcagatctcagtagc #sequence

Illumina Raw Data - Fastq format (see PPT for more information):
Fastq files have a bit more detail to them, particularly when they come from an illumina machine. The first line is a header (designated by an @), the next line is the sequence (similar to fasta), then there is a comments line (+), and a base call quality line (of the same length of the sequence line).

For example:
@FCC1PFHACXX:2:2316:19544:100787#ATCACGA/1
CGGTCCAACAATTCTGTGCAAAAAAAAAAAATTTTCAAGAGGGGAGGGGGGGGAGAGATGGCGG
+
\^^\``^acYbcchhddBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB

Each part of this has information coded into it. In parentheses, I have labeled each of these parts with the information they refer to:

@ (start) FCC1PFHACXX (machine name):2(flow cell):2316(individual tile):19544(x coordination):100787 (t coordinate) #ATCACGA(tag)/1(pair number)
CGGTCCAACAATTCTGTGCAAAAAAAAAAAATTTTCAAGAGGGGAGGGGGGGGAGAGATGGCGG
GGCCTAGGCACGAACAAAACACCAAGCCAGCGAACA(seq)
+ (optional info - missing in this case)
\^^``^acYbcchhddBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB (quality score for each position in the sequence - required to be the same length).

Quality Scores (see PPT for more information)

One of the first things you will notice in these quality scores is a lot of "B"'s everywhere. These are related to the quality score, and are in what is called phred format, which was developed for the human genome project. It is based on the original Sanger style sequencing, where chromatograph peak shapes were used to determine how accurate the read for any one base pair was (picture). This information was encoded in a log scale from 1 (terrible) - 50 (the best - approximately a 1 in 100000 chance it's an error).

This scale is all well and great, until you want the same number of characters for your quality scores as you do for your sequence - meaning the two digit phred score has to be translated into a one character code. If quality control scripts or whatever you are doing results in different a different number of characters in the sequence than in the quality scores, upstream programs will not be able to read this information (important to consider when writing your own scripts and trouble shooting in general).

This problem is solved by using something called ASCII codes, which are numbers that map to a single characters. However, to avoid weird characters in the lower range of ASCII codes that might not do good things for text based files, 65 is added to the phred score (65 is 'A' in ASCII). This means all those 'B's are a phred score of 1 (66 - 65), or are complete junk, as is basically anything that is a capital (phred score of <26). This is good to know, since you should see substantially less 'B's and capital letters in your quality controlled data as compared to your raw reads - a quick check for quality control scripts!

There are different versions of quality score are as follows:

S - Sanger Phred+33, raw reads typically (0, 40)
X - Solexa Solexa+64, raw reads typically (-5, 40)
I - Illumina 1.3+ Phred+64, raw reads typically (0, 40)
J - Illumina 1.5+ Phred+64, raw reads typically (3, 40)
with 0=unused, 1=unused, 2=Read Segment Quality Control Indicator (bold)
(Note: See discussion above).
L - Illumina 1.8+ Phred+33, raw reads typically (0, 41)

This is generally not a huge problem, with the exception of Illumina 1.3 and Illumina 1.5 scores... because they do not appear any different, they are just coded differently. I will show you how to tell, because some programs (such as the Genomics Analysis Toolkit, vastly used in RAD-tag analysis and RNA-seq) crash if you do not adjust the scores. Determining your formatting is difficult because Illumina started as 1.3, then went to 1.5, then went BACK to 1.3. Just keep this in mind - if a program complains that the quality scores are weird and crashes, it means you will have to convert the scoring (with your new magical bioinformatic skillz) or tell the program to read it correctly (more common).

****Fastqc *(see PPT for more information)*****

-->> less NewData1.fq

Not a lot of capitals or Bs! Good news!! But we don't want to scroll through all of this and we don't know what format it is in.

There is a wonderful program called Fastqc (another pun). It is easy to call:

-->> fastqc NewData1.fq

NOTE: This program requires java to run.

It produces a compressed folder for you (.zip). The output is a html document (ls to see), so it will be easier to just upload it to windows with filezilla and open.

Download file with Filezilla.

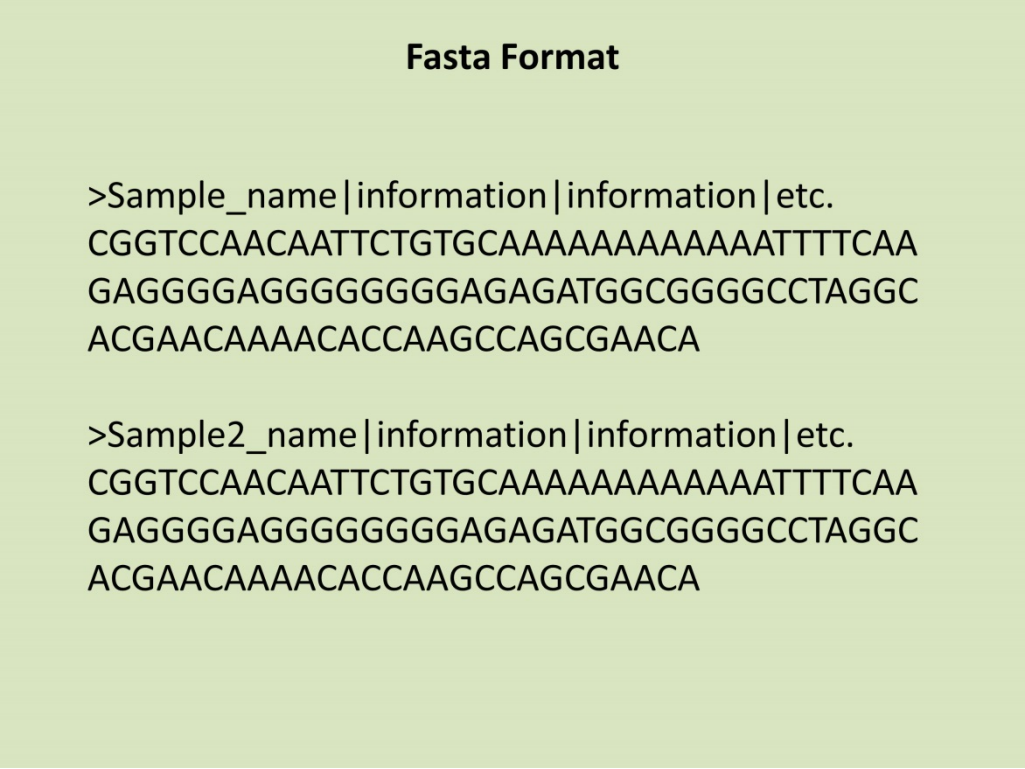
NOTE: This is a great video that talks about all the information in the FastQC output: https://www.youtube.com/watch?v=bz93ReOv87Y

Fantastic. We now know our data is in Illumina 1.9 coding, is high quality, and seems to be reasonable. You have completed the first step that you will likely do in any initial bioinformatics!

You can now tell your boss that you have your data, where it lives, and if its quality or not.

But there is a little more to calling these programs. If you try calling fastqc, you will get an error because it is not in your path. You should add /afs/nd.edu/coursesp.16/cse/cse60132.01/Shared/bin to your PATH!

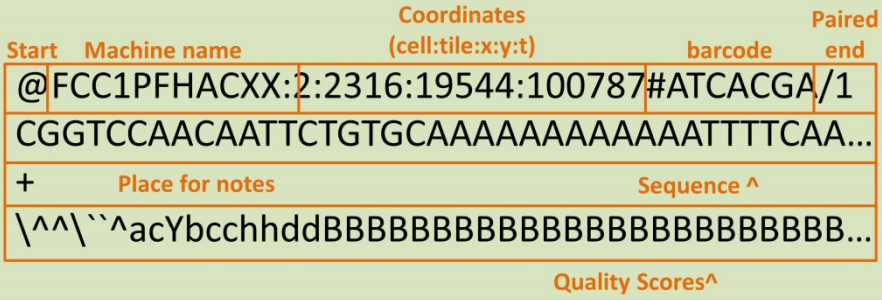
PPT that accompanies above notes



Illumina Raw Data (Fastq)

@FCC1PFHACXX:2:2316:19544:100787#ATCACGA/1
CGGTCCAACAATTCTGTGCAAAAAAAAAAATTTTCAA…
+
\^^\``^acYbcchhddBBBBBBBBBBBBBBBBBBBBB…

Illumina Raw Data (Fastq)



What's with all the B's?

Phred quality scores are **logarithmically** linked to error probabilities		
Phred Quality Score	Probability of incorrect base call	Base call accuracy
10	1 in 10	90%
20	1 in 100	99%
30	1 in 1000	99.9%
40	1 in 10000	99.99%
50	1 in 100000	99.999%

We want 99% accuracy or better, meaning scores of 20+. However, if we put these in the same order as the sequence, the spacing gets offset quickly and much harder to parse:

CGGTCCAACAATTCTGTGCAAAAAAAAAAAATTTTCAA

20**30**22**9**….

What's with all the B's?

Phred quality scores are **logarithmically** linked to error probabilities		
Phred Quality Score	Probability of incorrect base call	Base call accuracy
10	1 in 10	90%
20	1 in 100	99%
30	1 in 1000	99.9%
40	1 in 10000	99.99%
50	1 in 100000	99.999%

To avoid this offset, these scores are coded as single letters, using an old numbering system in computer science (ASCII). All characters have a number associated with them, with A=65 (a bunch of punctuation starts before letters).

ASCII Table

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

What's with all the B's?

Phred quality scores are **logarithmically** linked to error probabilities		
Phred Quality Score	Probability of incorrect base call	Base call accuracy
10	1 in 10	90%
20	1 in 100	99%
30	1 in 1000	99.9%
40	1 in 10000	99.99%
50	1 in 100000	99.999%

To avoid this offset, these scores are coded as single letters, using an old numbering system in computer science (ASCII). All characters have a number associated with them, with A=65 (a bunch of punctuation starts before letters).

We want our scores to start with A because a bunch of punctuation is coded 0-64, and we don't want to confuse the computer with special characters.

Letter code = 65 (starts at A) + Phred Quality Score

What's with all the B's?

Phred quality scores are **logarithmically** linked to error probabilities		
Phred Quality Score	Probability of incorrect base call	Base call accuracy
10	1 in 10	90%
20	1 in 100	99%
30	1 in 1000	99.9%
40	1 in 10000	99.99%
50	1 in 100000	99.999%

For example: A quality score of 20 is coded as:

Letter code = 65 + 20 = 85

Letter code =U

Inversely:

Letter code = B (66) = 65 + Score

Score = 1

Basically, all those B's are very poor quality sites (1 in 1 probability of incorrect)…

The scale is subject to change…

To make matters worse, the scaling has changed slightly through time. Illumina has at least two different coding systems which have been used intermittently across different years.

Why do we care?

You might not, but the analysis programs sure do. They will error out if they are getting letter codes outside of their range. Knowing what this means makes fixing the issue much easier.

Installing and Using Programs – Part I

******REVIEW OF PATH******

Where does fastqc live?

-->>which fastqc

I've made a Shared/bin folder in our class directory, so we don't have to have twenty iterations of the same installed programs. You can install what you wish in your home folders, but do keep in mind that you have semi-limited space.

Let's investigate this folder:

-->>ls -lah /afs/nd.edu/coursesp.16/bios/bios60132.01/Shared/bin/

Huh. Looks like a soft link, let's look in the directory that it points to. We will return to why this is softlinked in a moment.

-->>ls -lah /afs/nd.edu/coursesp.16/bios/bios60132.01/Shared/bin/FastQC/

There it is, the program fastqc. Notice the x's in the permissions? That means it's an executable. Also, notice how much other stuff is in here? These are all needed to run the program. But… instead of adding every new directory to the bin, we can simply softlink it to a folder in our path (in this case our Shared/bin).

If you call /afs/nd.edu/coursesp.16/bios/bios60132.01/Shared/bin/fastqc file.fq it will now run for you! But that's annoying to type out... so make sure to add the path to this directory to your $PATH variable.

Refresher:

The temporary way:

-->>declare PATH=$PATH:newdirectory/

By saying PATH=$PATH:newdirectory/, we're saying we want to append the other stuff onto the end of the existing PATH (which is $PATH). Remember that : separate directories in your path, so always put that in there - you can also add multiple directories this way by putting multiple colons followed by directories. This will last until you exit the shell you're in, and will not be present in any new shells you open.

The "permanent" way:

When you look at my shell you see that I have a personalized prompt. I also already have a huge PATH when I open up a new window. This is all due to a little file called .bashrc. This is like a recipe for bash when it starts up, telling the shell what to set variables to and how to set up my environment to my liking. This includes things like my prompt, aliases (where you can tell bash to run something with a command you specify - you can check yours by simply typing alias into your terminal), and PATH information.

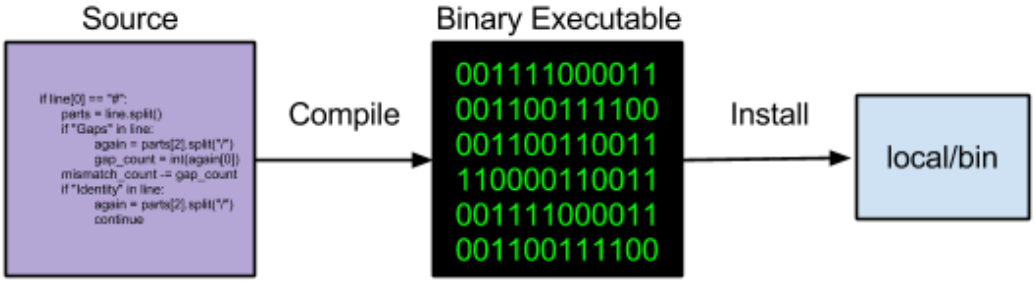
******Installing Software******

Installing software is kind of a rite of passage for bioinformatics. This is why I had you attempt it first. To give you a chance to figure it out on your own, since that is the majority of what you will be doing in the future. What is the best resource? The README and INSTALL files. ALWAYS READ THE README. It's there for a reason, and it's called Read Me for a reason. It'll often tell you exactly how to install the software, but we'll go through HMMER's source installation, which is pretty standard. On occasion, there's also a file called INSTALL which, not surprisingly, will tell you how to install the program.

Let's download it (google hmmer-3.1 to find):

-->>wget http://selab.janelia.org/software/hmmer3/3.1b1/hmmer-3.1b1-linux-intel-x86_64.tar.gz

NOTE: How do you tell which file? "arch" will give us the architecture (which chip hardware we have running in the computer). We know we are on Redhat linux from the first week. No worries though, if you do it wrong, it will just tell you it won't work and you can rm it and try again. Also, you want source files (these are converted into binary, which the computer then runs). :



You can also download binary versions, but they are harder to customize (not human readable) and troubleshoot at times (cannot change directory locations, etc. as easily – more on that later).

Now let's decompress the file:

-->> gzip -d FILE
-->> tar -xf FILE

Ok. Now we have a folder with a bunch of files. We are going to install this into our shared bin on the course space (which is in my path and should be in yours!). First let's look in the folder.

Oh look, a readme. Which tells us to go to the Install...

And it tells us the basic recipe:

-->>./configure
-->>make
-->>make install # will not work! See below

Now, I'm going to post a more in depth explanation as to what the point of each of these step is really doing (look for a link on the Lesson 6 tab), but for now (and for the purposes of this class) all you need to know is the following.

./configure sets up environmental variables, checks your system requirements, and well, configures things. Basically, checking your set up and program lists (in $PATH) to determine how to install things on YOUR SPECIFIC set up. It completes the Makefile to match your file locations, system settings, etc.

make follows the instructions of the Makefile and converts source code into binary for the computer to read.

make install installs the program by copying the binaries into the correct places as defined by ./configure and the Makefile.

But if you try this, you will get a very common error: you don't have permission to install things in the default location!! How many of you got this error? So what do you do?

-->>./configure --prefix=<new location>

This flag will tell the configure program to set up the Makefile to put everything where you would like it. Remember this command, you will need it often!!

Where do you put things? More (extensively) on that later.

******HMMER******
We've installed HMMER now, so let's talk about the actual program itself. If you were to read the user's guide, you'd see that the first step is to create an HMM "profile" (using hmmbuild included in HMMER) from a multiple alignment. What does that mean?

Given a set of sequences (we'll say protein sequences) that are all aligned to each other:

TLARVKLAQ...
T-ARTKLAQ...

TLVR-KTAQ...

With a profile:

We create a "profile" of what the protein sequences "tend" to look like. But in order for hmmbuild to do this, we need to create the multiple alignment for the set of sequences first. Enter our next program to install. Let's look for "multiple alignment linux" on Google. We'll use the program called MUSCLE - if you look at the first link though you'll see there are about a bazillion options. Ask your colleagues about software they use! We will also talk about alignment later when we get to tree building.

So we're going to download and open it up like we did before, with tar and gzip. However, when this file untars, there is only one executable file. If we try to open this file, the computer tells us that it is a binary file. This is an example of a binary installation – notice the lack of customization we got to do?

And now we have an executable file called muscle! That seemed easy. We can copy that into our bin directory.

NOTE: Sometimes, installations will be a little weird. If you see a Makefile, you can just run make and then look around for an executable binary that was created (which might end up in subfolders), and then just copy those into your bin directory. The readme (or google) will help you out in these situations.

Let's do a quick recap before we do some awesome bioinformatics:
1) We can often install software using standard configure/make/make install instructions
2) We can install software by downloading binaries and copying them into our bin directories (or wherever in our $PATH), like MUSCLE
3) We can install software by "other" means. These can vary a lot, so look at the README!

NOTE: The question of version control was brought up today. Programs will change through time, especially large software suites (GATK, Trinity, etc.). Usually, you have to specify what version of a program you ran in a publication, so it's a good idea to keep track of what version you download. One way to do this is to keep the version in the file name. Another is to keep a README in your ~/local/ that stores version information. May programs have a --version flag also built in.

******Obtaining Data******
How do we get data to work on? We can:

- Get it from a colleague (i.e. me)
- Produce it yourself
- Simulate it ("fake" data)

- Find some on the Internet and download it (what we will do today)

We're going to be operating on our own copies of data sets since we're running programs locally, though some things (like NCBI BLAST) let you run programs via a web interface, and you can use their data (like using their copy of the nr database, which we'll learn about later).

Let's use HMMER to find Toll-like receptor (TLR) genes in the *Xenopus laevis* gene set.

Again, we want to download our own, full data sets; knowing where to find them (and what you want) can be super tricky. It requires:

1) Good colleague resources (ask questions!)
2) Good Google (your search skills)
3) Practice

So, let's download the Xenopus frog proteins. We'll use a resource called XenBase. Often genome resource cites are names for the organisms that they detail. Flybase for Drosophila, FleaBase for Daphnia (water fleas), etc. I like amphibians, so we'll use Xenopus.

XenBase->Genomes->Download Xenopus Genomes -> X. laevis

Since genomes and genetic information we have is constantly being made more accurate, every once in a while groups will make a new official "release" of the data, which represents the state of the data at that time. This is important so that papers can refer to which release they worked with.

We will want the fasta files, typically (why are these not fastq files?). We want the protein files, so let's grab the peptide(FASTA) file. We are actually going to use the previous release because as of testing, the "peptide" file isn't amino acids for version 9.1!

--> wget
ftp://ftp.xenbase.org/pub/Genomics/JGI/Xenla8.0/Xla_8.0.primaryTrs.PEP.Named.fa.gz
-->gzip –d Xla_8.0.primaryTrs.PEP.Named.fa.gz

What happens if you try to use tar to unzip this file?

NOTE: Sometimes when you are looking for a file on a genome browser, you will have to dig a little. Don't despair. A lot of times when you're looking for data you'll run into things like this - a lot of files, with no clear indication as to which one you need. Sometimes you can ask, and sometimes you just need to hunt. Just be ready to look, and come in with a clear idea of what you're looking for!

Now, suppose we want some examples of Toll like receptor genes (since we want to look for TLR genes in the Xenopus gene set).

Enter uniprot.org: a gene database, some of the genes are "reviewed," i.e. a human confirmed them to actually look like their annotations. Others aren't. SwissProt is the database/set of reviewed proteins, while UniProt is all proteins.

******Running MUSCLE and HMMER******

So, we want to search for "TLR2", since maybe we're interested in the TLR2 variant, and we select "reviewed." Since this still gives us a lot of other things, we can scroll along the side bar to "Search terms" and filter by gene name. This forces only TLR2 genes to come up. Then we can select the ones we want, and click "download", "GO".

It may make a temporary page for us, which you can copy and paste into a new file called "TLR2.fasta"; or it may automatically download the file, which you would need to transfer and rename.

NOTE: How you download things on Uniprot may change through time!

So, we've installed our bioinformatics software, and we've gotten our datasets (Xenopus gene model and TLR2 genes). How do we do bioinformatics?

1) Ask our colleagues
2) Read the manuals
3) Try it!

Again, we want a profile of what a TLR2 looks like, so we can search for that profile in the Xenopus gene model. To do that, we need a multiple alignment (via muscle), and then we'll generate a profile (via hmmbuild), and then, finally, we can search for TLR2 in Xenopus (via hmmsearch). We can check out the help for each of these programs using the -h flag.

First, the multiple alignment.

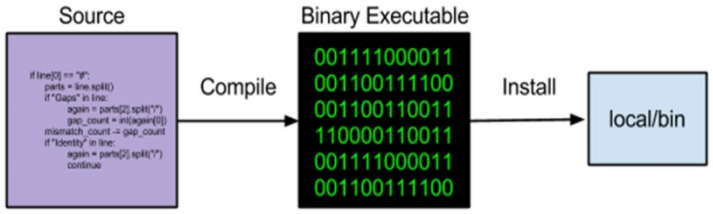
-->>muscle -h
-->> muscle -in TLR2.fasta -out TLR2.aln
-->>less -S TLR2.aln

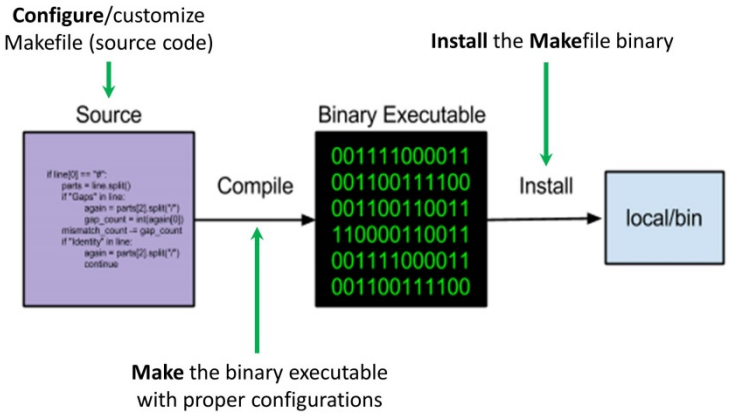
Now, we can build the HMM profile:

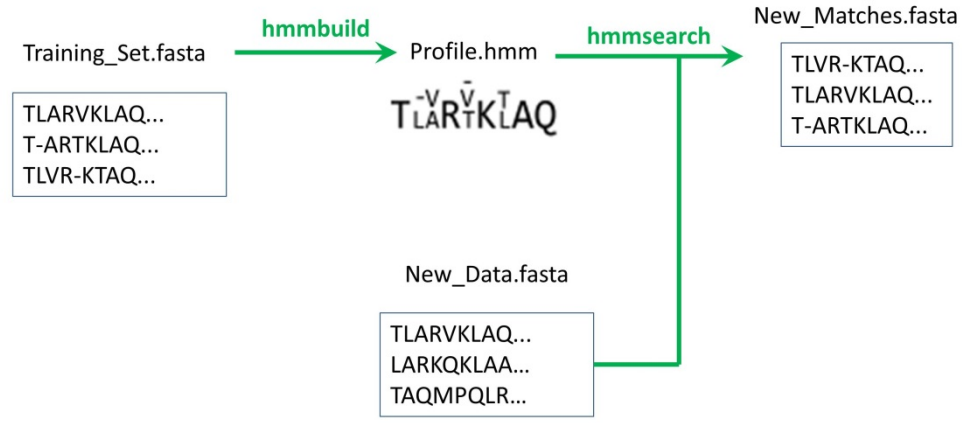
-->>hmmbuild -h
-->> hmmbuild TLR2.hmm TLR2.aln
-->>less -S TLR2.hmm

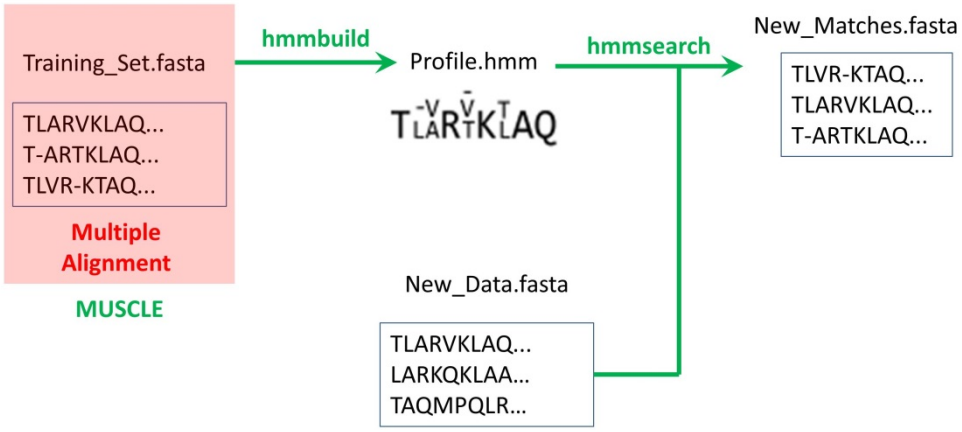
We will move on to hmmsearch Tueday!

PPT that accompanies above notes:

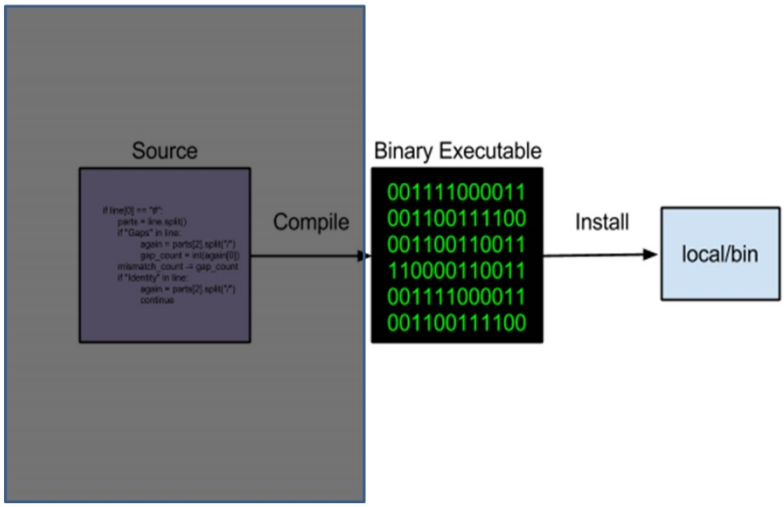





Training_Set.fasta
hmmbuild
Profile.hmm
hmmsearch
New_Matches.fasta
TLARVKLAQ...
T-ARTKLAQ...
TLVR-KTAQ...
TLVR-KTAQ...
TLARVKLAQ...
T-ARTKLAQ...
New_Data.fasta
TLARVKLAQ...
LARKQKLAA...
TAQMPQLR...


Training_Set.fasta
hmmbuild
Profile.hmm
hmmsearch
New_Matches.fasta
TLARVKLAQ...
T-ARTKLAQ...
TLVR-KTAQ...
Multiple Alignment
MUSCLE
TLVR-KTAQ...
TLARVKLAQ...
T-ARTKLAQ...
New_Data.fasta
TLARVKLAQ...
LARKQKLAA...
TAQMPQLR...

You can download binaries (MUSCLE), but there is no configuration step!



Installing and Using Programs – Part II

******HMMer Continued******

Thus far we have gotten the known TLR2 genes (in fasta) from Uniprot, made a multiple alignment (.alignment, similar to fasta), and used this to make a model of what TLR2 genes look like in general (.hmm). We then grabbed the Xenopus gene model from Xenbase, so we can search in that file for genes that match the .hmm model.

A review of the commands we have used:

Retrieved data:

-->>wget <fasta output from Uniprot database> #called TLR2.fasta
-->>wget <Xenopus gene models from Xenobase> #if you are having problems with this, just grab it from the Shared/data folder

First, the multiple alignment.

-->>muscle -h
-->>muscle -in TLR2.fasta -out TLR2.aln
-->>less -S TLR2.aln

Now, as per the tutorial, we can build the HMM profile:

-->>hmmbuild -h
-->>hmmbuild TLR2.hmm TLR2.aln
-->>less -S TLR2.hmm

Continuing on

Now we can search our proteins:

-->>hmmsearch -h
-->>hmmsearch TLR2.hmm Xla_8.0_aa.fasta

That just showed us all the output, and didn't make any file! This is because some programs write to standard out (stdout), which is our console screen. We need to redirect the output to a file, which we can do with a **new special character: >**.

Note: Every time you capture STDOUT with the >, it will overwrite your file. If you want to add the output to the end of a file, >> will append.

-->>hmmsearch TLR2.hmm Xla_8.0_aa.fasta > Xla_TLR.hmmout

And now we have a file called hmmer_Xla_TLR2.hmmout!

NOTE: This would be a good time to update your README list of files – so you don't forget where this file came from. I named it something obvious, but if we just called it Xla_TLR2.txt, it might not be obvious that it was the result of hmmer.

When we look at it, we've got lines and columns of numbers, which is what we've been working towards this whole time. However, we don't know much about this kind of data yet (we'll come back to it). Let's get a fasta file with the genes that matched the query.

GOOGLE: hmmer to fasta → That Cryptogenomicon link is promising!

Ok, we have to use an option in the hmmsearch program:

-->>hmmsearch –A Xla_TLR2.sto TLR2.hmm Xla_8.0_aa.fasta

This –A flag lets us designate that we would like to output what is called a Stockholm format of alignment (there are so many versions of alignment formats. This is an older one, but it's one that this program will output). This format can then be converted to fasta with a built in, but somewhat hidden, HMMER program called **easel**. It wasn't dumped into our bin when we installed Hmmer, instead it is in a subfolder. We will have to install it just like anything else – you will install this into your home bin:

-->>cd $COURSE/Shared/hmmer-3.1b1-linux-intel-x86_64/easel
-->>./configure prefix=path/to/home/local #SEE NOTE BELOW
-->>make
-->>make install

NOTE: Remember that these commands will install the program in your local/bin. Why do we not have to put the full path into the bin folder? Because the program is looking for $prefix/bin/ to add the executables to (and $prefix/src, and $prefix/inc, etc.). See the installing programs supplement on Lesson 6 or see me if you are confused!

Now let's go back to where our data is and run the conversion:

-->>cd $COURSE/Shared/data
-->>which esl-reformat #check if you have the program in your path!
-->>esl-reformat -h
-->>esl-reformat fasta Xla_TLR2_hmm.sto > Xla_TLR_hmm.fasta

Now, finally, we have a fasta file of the results from hmmer, giving us all the TLR2 genes from *X.laevis*. We'll use this file in our next analysis – BLAST.

******BLAST: How it works******

BLAST stands for Basic Local Alignment Search Tool, and is likely going to be a program that you use a lot. Given one or more query sequences, it looks for matches between the

query/queries and sequences in a database file. In theory, it is similar to hmmer; you are searching for something (profile/query) in something else (genome/database).

BLAST relies on a well-established alignment algorithm called Smith-Waterman (the Needleman-Wunsch algorithm is for global alignment). You don't need to know these for this class, but it's helpful to know the name when talking to computer scientists. These are very basic algorithms they all learn and they may simply mention them casually in conversation, expecting you to know what they are. The algorithm uses a scoring system and a matrix to determine the best alignment by building a "path" through the two sequences being aligned. It finds the optimal alignment for two sequences, with the ability to adjust for gaps and mismatches.

BLAST can be used for things like:
1) Determining orthologs and paralogs for a particular nucleic sequence, e.g. a new bacterial genome is sequenced. What proteins are paralogous? Which have no significantly related matches in GenBank?
2) Finding what genes are present in a particular organism, e.g. new transcriptome is assembled. Which of the 100,000 potential sequences look like known sequences?
3) Discovering new genes, e.g. there are no matches in nr ("The nr database is compiled by the NCBI (National Center for Biotechnology Information) as a protein database for Blast searches. It contains non-identical sequences from GenBank CDS translations, PDB, Swiss-Prot, PIR, and PRF.")
4) Investigating ESTs that may exhibit alternative splicing. There are specialized EST databases that can be searched using BLAST.
5) Finding the close orthologs for a set of sequences in a different species.
6) Searching for a group of diverse sequences all at once (antimicrobial sequences, transposons, etc.). You wouldn't want to make a hmm profile based on them, as they are lack large homologous, conserved regions. You instead would want a list of which sequences (from say, a transcriptome) look like any one of the sequences in the set (different antimicrobials).

Note: In most of these cases, you are comparing an UNKNOWN SEARCH QUERY (new genome sequences, genes with unknown annotations, unknown splicing) to a KNOWN DATABASE (paralogs, orthologs, all documented genes, all documented splices). When we used hmmer, we did the opposite – A known profile (.hmm) made from known annotations to search in an unknown database. Also, think about the difference between searching with a profile versus a single string. A profile will catch more than a single string (its more flexible in its matching) and therefore is more useful to use to search unknowns. If you are starting with unknowns or a large set of data (genome, transcriptome), you really just want the best match (BLAST) – which is faster.

BLAST essentially takes a sequence and looks for a match to that sequence. Deviations are scored based on how similar the matches are. You can customize this scoring with different scoring matrices.

Tl;dr: BLAST looks for matches to a query in a database. Usually, the query is unknown (new data) and the database is known (genbank), but not always. Sometimes the query is known (group of genes) and the database is large (genome). Either way, BLAST requires a query, a database, and an output name.

A nice little summary: http://bitesizebio.com/21223/how-does-blast-work/

******Running BLAST******

NOTE: BLAST is preinstalled in our Shared/bin. It's a bit larger, as are databases.

GENERAL REMINDER: If we're going to run stuff that's going to take a really long time and use up lots of CPUs, and use up boatloads of RAM, we should talk to other people who use the machine, and maybe see what other people are using the machine when we hop on. Top is a command that gives us all sorts of info about who is running what, what their CPU usage is, what their memory (RAM) usage is, etc. On our Linux machines, with top open, we can also type u, and then enter in a user name to filter the results, showing only processes run by the particular user. We can exit top by typing q.

Step 1: Make a database:
As stated above, BLAST needs a query and a database to run. The query is a normal FASTA file and the database is a FASTA file that has been processed for use as a database, allowing for quick lookup. We can create this database using makeblastdb.

makeblastdb -in <FASTA> -out <DBNAME> -dbtype <prot/nucl>

The -dbtype flag is to tell makeblastdb if this is a protein (prot) database or a nucleotide (nucl) database.

Step 2: Search the database
There are actually at least 6 different types of BLAST we can run against this database, depending on whether the query and the target database are nucleotide or protein in content.

blastn - Nucleotide query, nucleotide database
blastp - Protein query, protein database
blastx - Nucleotide query, protein database
tblastn - Protein query, nucleotide database
tblastx - Nucleotide query, nucleotide database (with translation)
psiblast - Protein query, protein database for distant proteins

The most used in general is blastn or blastp. Why do you think we would not want to always run tblastx? The basic format for running a blastn (nucleotide vs nucleotide) command is as follows:

-->>blastn -db <DATABASE> -query <QUERY> -out <OUTFILE>

For each match, BLAST gives the following information:

queryID, matchID, "description" (length of matching region, number of bases that are exact matches, etc.), e-value ("roughly," the likelihood of finding such a match "by chance"), alignment of the match.

Options:
All the BLAST programs (blastn, blastx, etc.) also take a number of other important options (if you want):

-num_threads 4 - try to use 4 cpus on a multi-cpu machine
-num_alignments 10 - show only the top 10 database matches per query sequence
-num_descriptions 10 - should equal the above number - gives top 10 descriptions
-evalue - 1e-6 - only show matches with e-value equal to this or smaller
-outfmt 6 - use output format 6 (described in -help).
Useful ones are typically:
0 - pairwise: human readable (default – not really useful)
5 - xml: specialized computer readable, programs will ask for this
6/7 - tabular (w/o and w/ comments): rows and columns

For practice, try the following:

1. See if you can grab the *Xenopus tropicalis* gene model (protein fasta) from the internet and put it in your home directory.
2. Try blasting the results against the $COURSE/Shared/data/uniprot_sprot.fasta.db. This file is the downloaded uniprot database. How would you blast Xla_TLR2.fasta against a nucleotide database?
3. Try finding the orthologs to the sequences in Xla_TLR2.fasta (in our Shared/data) using BLAST.

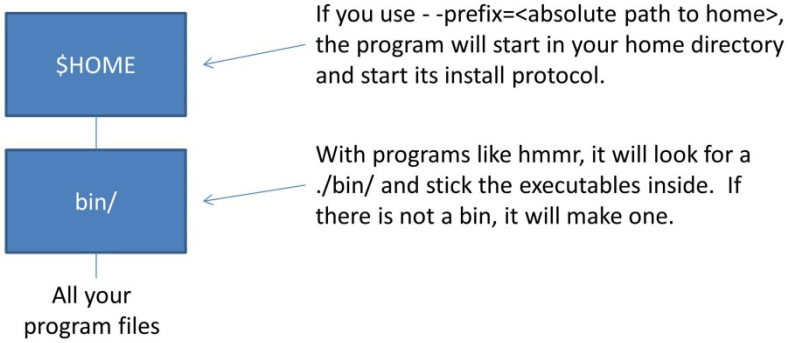
Note: Using blast can take time. If you wish to cancel a command you can use Ctrl-C, if you wish to pause a command (or make it a zombie to come back from dead later), you can use Ctrl-Z. Technically what a zombie is in computing is slightly more complicated than this (but still a very real thing!: https://zombieprocess.wordpress.com/what-is-a-zombie-process), but it is an easy way to remember the command.

For the curious – algorithms:
Smith-waterman (alignment):
http://docencia.ac.upc.edu/master/AMPP/slides/ampp_sw_presentation.pdf

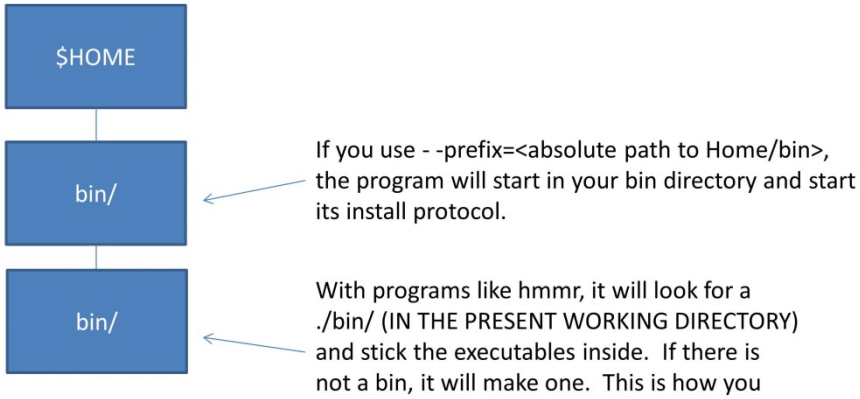
Hidden Markov Models Analogy as Applied to Grad Students (HMMER) – Thanks Chissa!:
http://www.ece.drexel.edu/gailr/ECE-S690-503/markov_models.ppt.pdf

BLAST:
https://en.wikipedia.org/wiki/BLAST#Algorithm

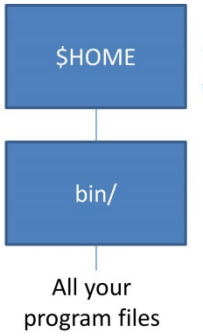
When you go to install a program, you use the - -prefix command:



When you go to install a program you use the - -prefix command:



But what about programs like muscle?

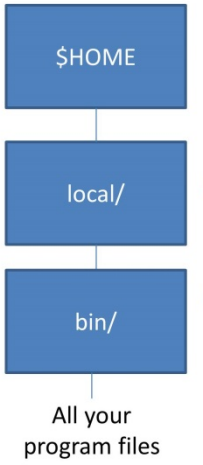


If you use - -prefix=<absolute path to Home>, the program will start in your bin directory and start its install protocol.

But some programs, like muscle, just install in the folder you just directed them to (no configure or --prefix option)… they won't make it into your bin, because of the way the install protocol is written.

Now you will have files in your home directory that will be hard to find and not in your PATH!

We solve these issues with variable program install protocols with a local/bin:

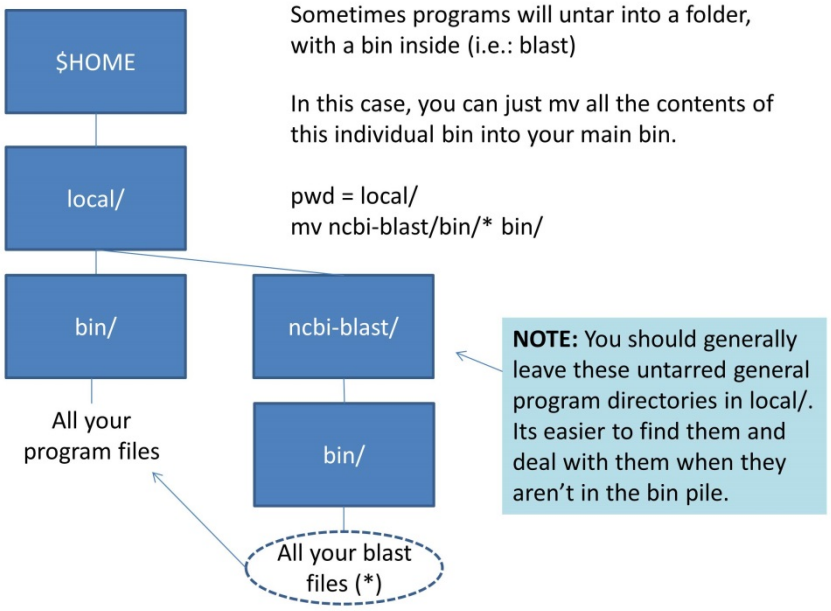


If you use - -prefix=<absolute path to Home/local>, the program will start in your bin directory and start its install protocol.

With programs like hmmr, it will look for a ./bin/ and stick the executables inside. If there is not a bin, it will make one.

With programs like muscle, they will just install into the local/ which makes them easier to find, since they will not be in the pile of all your home stuff. You can then easily move them to the local/bin/.

We solve these issues with variable program install protocols with a local/bin:



Sometimes programs will untar into a folder, with a bin inside (i.e.: blast)

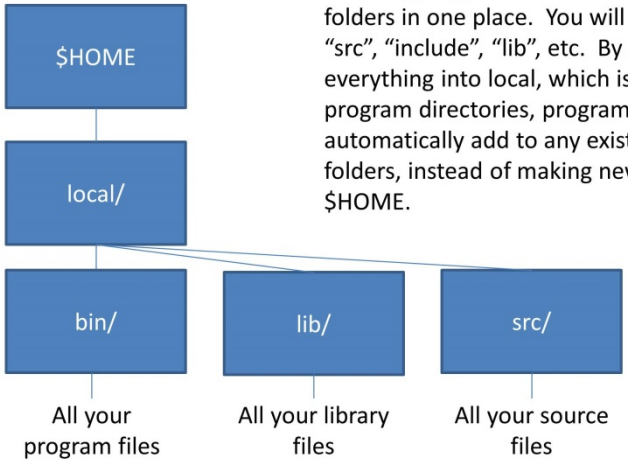
In this case, you can just mv all the contents of this individual bin into your main bin.

pwd = local/
mv ncbi-blast/bin/* bin/

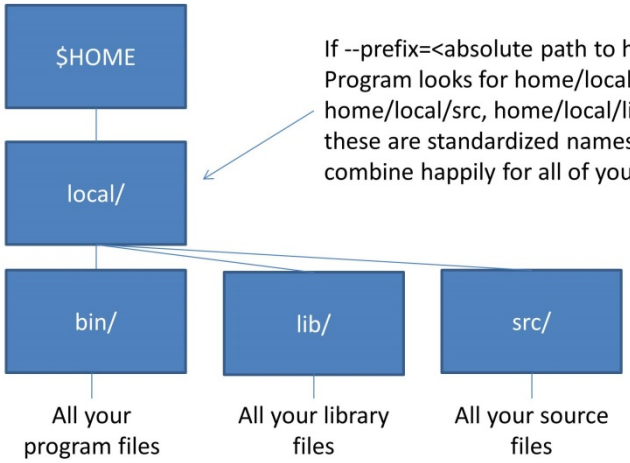
NOTE: You should generally leave these untarred general program directories in local/. Its easier to find them and deal with them when they aren't in the bin pile.

We solve these issues with variable program install protocols with a local/bin:

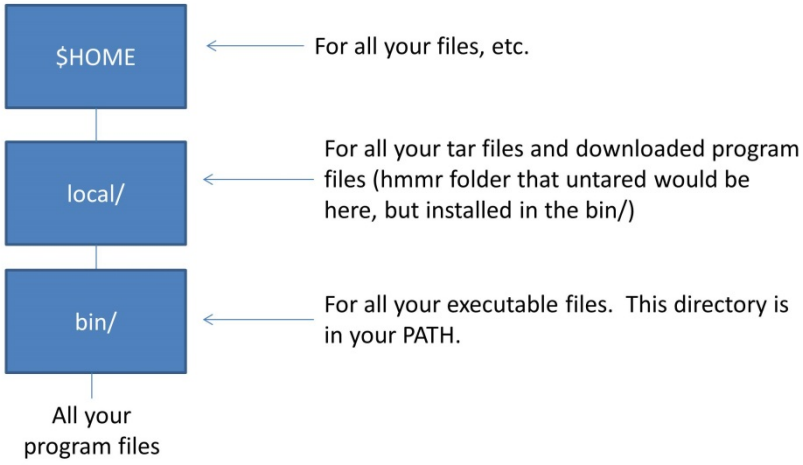
The local/bin set up is also helpful with allowing programs to make other standard folders in one place. You will sometimes see "src", "include", "lib", etc. By directing everything into local, which is above the program directories, programs will automatically add to any existing program folders, instead of making new ones in your $HOME.



We solve these issues with variable program install protocols with a local/bin:



The end result is organized program files, and all executables in one bin, meaning you only have to have ONE directory added to your PATH!



Recap:

We learned different ways to install programs:

1 - ./configure-make-make install:
- Programs that have dependency on other stuff on the computer and require configuration – very typical!
- --prefix=<path to home/local> will install it in your bin
- i.e.: hmmr, easel

2 – simply untar
- Programs that are pretty stand-alone and don't require custom information from your computer – common.
- Decompressing it in your Home/local/ will "install" it in your local/, but you can easily move it into /local/bin, because it usually one file and it will be separate from all your main files.
- Sometimes decompression will result in a folder with a bin in it – just copy all the bin files into your bin (ie: mv ncbi-blast/bin/* bin/).
- i.e.: muscle, blast

3 – weird ones
- Sometimes programs have weird install requirements – not so common.
- Follow the README

Checkpoint: .bashrc, executables, tree pre-activity (HW2)

Part 1: Set up your directory and .bashrc

1. Make a new alias in your .bashrc. It can be as simple as making the word "hi" echo "Hello!" to the screen but good ones to try might be making cp and mv default to using the -i flag (I wonder what these do…).

2. Permanently add the following directories on the course space to your PATH variable:
/afs/nd.edu/coursesp.16/cse/cse60132.01/Shared/bin

REQUIRED: copy your .bashrc into your HW2 directory. (Points: 1)

Part 2: Permissions

1. Make a file called permissions.txt with the answers to the following questions:

    a. What will “chmod 600 file.txt” do?
    b. How would you set file.txt to be readable by you without using the number code?
    c. What is the major difference between AFS and Unix permissions?
    d. What should the first line of a program called example.ba be?

REQUIRED: Answers to the four questions in a file called permissions.txt in your HW2 directory. (Points: 4)

Part 3: Make and executable program

1. Write a command to make the screen print: "my home dir is <your home directory>."
with <your home directory> replaced by your actual home directory path.

2. Write a command to make the screen list the contents of your home directory.

3. Create a text file called "myprogram.ba", and have it contain the two commands you wrote above.

4. Make the file executable.

5. Execute the program. I will run this myself, so don't worry about a screenshot.

6. Change the permissions to the file to give everyone (user, group, other) the ability to write to the file.

REQUIRED: myprogram.ba, as an executable file, in your HW2 directory. (Points: 3)

Part 4: Tree Pre-activity

Let's try to make a tree! Just like last time, you don’t have to complete this part of the homework. I simply want you to try. You learn a lot more by troubleshooting and thinking about the problem than my simply telling you how to do it! Think of it as a challenge to make the best tree!

1. Download data from the course site using wget.

2. Go to www.phylogeny.fr.

3. Check out the site. See if you can figure out how to make a tree and what the different options mean. Hint: Phylogeny Analysis -> a la cart, then mouse over any options for more details!

4. If you succeed in making a tree, save the nexus file (outputs are listed at the bottom of results pages) as Hw2/tree.nexus or take a screenshot and save it as HW2/tree.jpg.

REQUIRED: Two posts on the forum (HW2, Part 4) thread on Sakai. (Points: 2)

SUMMARY OF REQUIREMENTS:
In HW2 directory:

- README for HW2 files
- copy your .bashrc
- a file called permissions.txt
- myprogram.ba, as an executable file

Two posts on the forum (HW2, Part 4)

Checkpoint: Analysis and trees (HW3)

Part 1: Analysis practice

1. Blast the Xla_TLR2.fasta (in our Shared/data) against the $COURSE/Shared/data/uniprot_sprot.fasta.db. This file is the downloaded uniprot database. Use an evalue cut off of 1e-5, output7; and only one output per sequence. (This will take a while – if it takes too long, end it and give me what you have).
   How would you blast Xla_TLR2.fasta against a nucleotide database?
   How would you pause and run this in the background?
2. Grab the *Xenopus tropicalis* genes (protein fasta) from the internet and put it in your home directory.
3. Find the Xtrop orthologs to the sequences in Xla_TLR2.fasta using BLAST. Output this file in outfmt 7 and use an evalue cut off of 1e-5.
4. Use the TLR2.hmm to generate a file, Xtrop_TLR2.fasta, by searching the Xtrop genome.

Part 2: Troubleshooting

Respond to the following troubleshooting requests:

1. I don't have permission to install hmmr. How do I install it if I'm not root?
2. I downloaded hmmr-3.1b1-macos-intel.tar.gz into my home folder on Darrow. I installed it, but it getting an error when I run it. What did I do wrong?
3. This program I grabbed doesn't have a configure file… what do I do?
4. The following command is erroring out. What am I doing wrong?! **Tar –xfz file.tar.gzip**

REQUIRED: HW3Troubleshooting.txt in HW3 directory (Points: 2).

Part 3: Unix Tools Pre-activity

Read through the following tutorials and try the new commands (| and grep). Don't worry about the regular expressions part – we'll get to that later!
http://www.codecoffee.com/tipsforlinux/articles/24.html
http://www.uccs.edu/~ahitchco/grep/

REQUIRED: Screenshot of you performing a grep command using a pipe (POINTS 2).

REQUIRED Summary – your $DROPBOX/HW3 directory should have:

README
Xtrop_TLR2.outfmt7 -3 points
Xtrop_TLR2.fasta -3 points
HW3Troubleshooting.txt- 2 points
Screenshot of grep command – 2 points

Checkpoint: Review to this point

Please note that extensively long answers are not required for any of these questions. In fact, each can be answered with less than two sentences.

1. What purpose does the Bash variable PATH serve?

2. What does the -l option of ls do?

3. How might you find out what the command sleep does (assuming you don't have access to the internet).

4. In my ~/local/bin directory, if I do an ls -lh and I see the following line of output:

```
-rwxr-xr-- 1 ssander5 staff 611B June 16 10:42 median
```

What do sssander5 signify?

Who can execute this file?

Who can write to it?

5. What command would I use to change the permissions of a file?

6. What is the purpose of the .bashrc file (assuming it occurs in a user's home directory).

7. What command do you use to create a directory?

8. What command do you use to download a file from the internet?

9. What are the two typical ways to install software that we saw. Hint, one is a set of 3 commands, one can be described in a sentence.

10. If I wanted to search for all occurrences of a word in a file, what tool would I use?

11. What command would I use to count the number of lines in a file?

12. Write a pipeline command that does the following: - Uses cat - Finds every occurrence of the AC in file.txt - Sorts them based on the 1st column in reverse order - Pipes the result into less with no word wrapping

Unit 3: Introduction to Unix – Basic Unix Tools

Part I: I/O, Cat, Grep, and Pipes

******I/O Intro******

Before we get into how to manipulate files, we need to talk about where data is going in the Unix system. There are three major flows of data streams:

Standard In (stdin) - reads in data (if needed).

- less takes data from the file indicated, which is read in through stdin. When you scroll, more lines are being fed into less (which is great for huge files).
- ls does not require stdin, as it is not reading a data stream. Data is not being directed into it, it is reading system information instead.
- **NOTE:** Options/arguments/parameters are different than data stream. "-" tells the computer that you are about to give it an option (-h) and it needs to do something special. less –S file.txt will read the file without line wrapping, but less S file.txt will trick the computer into trying to open a file called "S" (and likely complain that it cannot find it). This is why options all come first in most commands!

Standard Out (stdout) - writes out data (if needed), often to the terminal

- Some commands do not have a stdout, mv for example.
- We learned how to capture the output of hmmer with the ">" character. This takes stdout and dumps it into a file.

Standard Error (stderr) - writes out errors, also often to the terminal

- This isn't terribly useful to us, but in the interest of completeness, I included it!

Ok, so why do we care about these? Well, we already learned one means of manipulating where data goes, with the ">" character. There are others:

> - directs data from stout to a file
< - directs data into stin (i.e. less < file.txt)
>> - directs data from stout to the end of a file, that may or may not exist. Basically, appends to a file.
| "pipe" - directs stout into stin for the next command. This is how you make "pipelines".
ie: hmmsearch -h | less --> way easier to read! Essentially how man pages work

******GREP INTRO******

Now, we get to use our first cool "data analysis" tool: grep (globally search a regular expression and print). We will talk about the regular expressions part later.

grep: extracts lines that match a pattern (regular expression).
e.g. grep 'pattern' file.txt
NOTE: it does more than that too: see "man grep"

So let's explore some of the utility of grep:

******GREP WITH FASTA******
One of the features of fasta is that the non-sequence lines start with ">". Let's use grep to pull out all the sample information lines:

-->>grep ">" TLR2.fasta

Nice! But it printed it to stdout, which isn't super useful. We could redirect the results to another file:

-->>grep ">"TLR2.fasta > grepresults.txt

But, let's instead pipe it to less:

-->>grep ">" TLR2.fasta | less

Notice I didn't tell less which file to read, instead I had it read on its stdin by redirecting the stdout of grep. We can also do the same thing the following way:

-->>less TLR2.fasta | grep ">" | less

Usually we don't use less as the input, because there is an extra step involved in piping it to stdin. It has to go from less, to stdout to stdin. Instead, we usually use cat:
cat: read from one or more files, print to stdout

-->>cat TLR2.fasta

Cat reads the file directly to stout. Basically file -> stdout, rather than file -> less -> stdout.

Lots of basic tools can read from stdin, and grep is one of them:

-->>cat TLR2.fasta | grep ">"

But maybe we want to look at it:

-->>cat TLR2.fasta | grep ">" | less

So you can chain command together in a line with the pipe character. This is a pipeline!

####Do we all get this? This is a crucial point so let's make sure everyone is on board!####

******GREP WITH HMMER******
Let's look at a more complex file - the original hmmer output. We couldn't do much with it before, because we lacked the tools. Let's see what we can do now that we know a bit about grep.

-->>cat Dmel_p450.hmmout | less

There is a bunch of stuff after the table of numbers. Let's see if we can get grep to pull out this table and work with it a little bit. What's a good line to find the top of that section?

Let's try Query:

-->>cat Dmel_p450.hmmout | grep "Query"

Well, not quite. Let's fix that:

-->>cat Dmel_p450.hmmout | grep "Query:"

Now to get the rest of the lines. **-A** allows us to specify how many lines after the match we would like output:

-->>cat Dmel_p450.hmmout | less -NS
-->>cat Dmel_p450.hmmout | grep -A 131 "Query:" | less -S

It's important to know that there are different ways to do the same thing:

-->>cat Dmel_p450.hmmout | grep -A 130 "Scores" | less -S

Great. Now we have the table. What else can we do? Just like before, we can pull out the lines referring to genes on the X chromosome:

-->>cat Dmel_p450.hmmout | grep -A 131 "Query:" | grep "loc=X" | less -S

Or ones everywhere other than the X chromosome:

-->>cat Dmel_p450.hmmout | grep -A 131 "Query:" | grep –v "loc=X" | less -S

What can we actually do with this tool? Let's look a bit deeper at our fasta file output from hmmer. We will use another quick tool:
wc: counts words, lines, characters, and bytes.
e.g. wc file.txt
OR cat file.txt | wc
Use -l to just count lines

-->> cat Dmel_p450.hmmout | grep -A 131 "Query:" | grep –c "loc=X"
-->> cat Dmel_p450.hmmout | grep -A 131 "Query:" | grep "loc=X" | wc

We can also combine grep lines to filter out other information as well. Let's find all the p450 genes that fall on the X chromosome:

-->> cat Dmel_p450.hmmout | grep -A 131 "Query:" | grep "loc=X" | wc -l

We can also find all the genes that fall everywhere other than the x chromosome with the -v command:

-->> cat Dmel_p450.hmmout | grep -A 131 "Query:" | grep –cv "loc=X"
-->> cat Dmel_p450.hmmout | grep -A 131 "Query:" | grep –v "loc=X" | wc -l

Why would you ever use wc –l if you have grep –c? Most often you will be building these pipelines iteratively, meaning you will have less –S at the end of most of them. It's pretty easy to change the end of a line:

-->> cat Dmel_p450.hmmout | grep -A 131 "Query:" | grep "loc=X" | less -S
-->> cat Dmel_p450.hmmout | grep -A 131 "Query:" | grep "loc=X" | wc -l

I HIGHLY recommend checking everything with less before using wc –l. You may not be counting what you think you are. What's wrong with this command?

-->> cat Dmel_p450.hmmout | grep -A 131 "Query:" | grep –v "loc=X" | wc -l
-->> cat Dmel_p450.hmmout | grep -A 131 "Query:" | grep –v "loc=X" | less -S

NOTE: Grep –c will be faster. Why? There are less conversions between stdout and stdin, meaning you loop through the file one less time. Also, there is no reason to have something hanging out in RAM when you really just want the number in the end anyway. This concern about time can be important while doing large jobs.

If we just want the table and no header to cause us problems, we can use tail:
head/tail: print the first/last N lines of a file (default is 10).
-n = number of lines
-->> cat Dmel_p450.hmmout | grep -A 131 "Query:" | less -S
-->> cat Dmel_p450.hmmout | grep -A 131 "Query:" | tail –n 126 | less -S

We can also sort the listings using another tool, sort:
sort has lots of different sorting options:
-g = general numeric sort (can understand scientific notation)
-d = dictionary sort (like you'd see in a dictionary)
-n = numeric sort (faster than general numeric sort)
-k = fields to consider (i.e. only refer to the first field –k1,1)

-->> cat Dmel_p450.hmmout | grep -A 131 "Query:" | tail –n 126 | sort | less -S

Well, that's not working right! Scientific notation requires a -g:

-->> cat Dmel_p450.hmmout | grep -A 131 "Query:" | tail –n 126 | sort –g | less -S

We can sort by different columns as well, using the -k command. Say we wanted to sort by the best domain hit, rather than the best overall sequence hit (check header):

-->> cat Dmel_p450.hmmout | grep -A 131 "Query:" | less -S
-->> cat Dmel_p450.hmmout | grep -A 131 "Query:" | tail –n 126 | sort –gk 4 | less -S

We can also sort by the full name of the gene, using dictionary sort:

-->> cat Dmel_p450.hmmout | grep -A 131 "Query:" | tail –n 126 | sort –dk 9 | less -S

NOTE: Pipelines can get crazy, especially when we get to regular expressions (the re in grep). A good thing to have in a README is code that you found useful in this directory. Or having a project file code bin in your home is helpful as well. Easy way to dump pipelines? Echo.

-->> echo 'cat Dmel_p450.hmmout | grep -A 131 "Query:" | tail –n 126 | sort –dk 9 | less –S' >> README
-->> echo 'dictionary sort hmmout table' >> ~/Dmelcode
-->> echo 'cat Dmel_p450.hmmout | grep -A 131 "Query:" | tail –n 126 | sort –dk 9 | less –S' >> ~/Dmelcode

******When do we use this stuff?******

How many sequences did I find?

grep –c ">" file.fasta

Is this gene in the pile?

grep "gene_name" file.fasta

Make a list of genes in a file!

grep ">" file.fasta >> genes.list

Find that list in a table?

grep –f genes.list file.tbl

Oops, I didn't want that sequence, remove it from the list!

grep –v "bad_seq" genes.list > genes.better.list

Give me this list of genes from a fasta!

grep –A 1 –f genes.better.list file.fasta #only works if all of the sequences are on one line!
grep --no-group-separator –A 1 –f genes.better.list file.fasta # no -- between lines

Combine files (I do this while grading):

grep "" *.fasta

Program fails and tells me "There is something wrong here" or other such non-standard errors??

grep –C "There is something wrong here" fake.ba

******Troubleshooting******

The command fails. What do you do?

General tips:

<u>Don't enter full pipelines at a time. Check EACH STEP:</u>

- Check each input and output of everything you do, it makes catching errors so much faster. Think about what you expect and what you get.
- This goes for wc –l command in particular – use less first! Make sure what you are counting is what you want!!!

<u>Check inputs when a program fails:</u>

- Make sure the format of the input is correct, that there aren't weird characters (such as –'s in a fasta file that isn't being considered an alignment!), or truncations.
- Try the program on demo data (usually included)
- Run program –h or man program to make sure it is installed correctly.

Check that your call is correct:

- Are you using the exact syntax the program wants? Are things in the correct order? Are you missing options you need?
- Read the man pages and really think about what you are typing in. Think about each part of the command in sequence and determine if it makes sense.

Check your output:

- You can check your output as it is happening by causing the program to pause (^Z – for "zombie", make the command dead, but able to be resurrected), and running it in the background (bg). Once it is in the background, you can less the output file and see if there is something happening and if it is what you expect. You can also see if anything is happening or if the job is just hanging (usually a result of looking for an input you forgot!!).
- You can view jobs in progress with "jobs", and kill jobs with "kill %#" where # is the number next to it when you call "jobs".
- Check the output after it finishes, is it in the right format? Is it ready for the next step? Are there weird characters that will cause you issues?

Part II: awk

******Fun with Awk******

Up until now, we have been using the general syntax of:

program -options inputs

i.e. : hmmsearch -A file.sto file.hmm file.fasta
blastn -query query.fasta -db db.db -out file.out
grep -v "#" file

Basically, telling the computer what program to run, how to run it, and what to run it on.

Our next program is a bit different. Awk, which gets its name from the people who wrote it in the 70s, is kind of complex. It's a bit of a mini programming language. Let's look at an example call and then we will start to break it down.

-->>cat file.txt | **awk '{if ($1!=$2) print $0}'** | less -S
or
-->>**awk '{if ($1!=$2) print $0}'** < file.txt

Right off the bat, this looks quite a bit different. There are familiar things; we can clearly see that we start the command with the program name, and it appears there are some variables being used. But we've never seen weird single quotes or brackets.

Let's break this down and fill in the concepts we haven't covered yet.

1) awk
2) '{ }'
3) $1, $2, $0
4) !=
5) if () print

1) awk

awk is a great program for doing calculations and some text manipulation. We start the command with awk to tell the computer that the following information will mean something to that program, just like we have done with everything else. Awk is a bit complex and can be a bit hard to get used to at first; but thankfully, a lot of the complexity can be ignored in day-to-day use and it is a super useful basic unix tool.

2) '{ }'

As we said, awk is kind of a mini language. Everything inside these brackets is code that will be run by awk. There is a distinct structure to awk:

awk 'BEGIN {things to do before first line} {things to do for each line} END {things to do after last line}'

Remember how we said grep was reading in a line at a time, then identifying if it had the search pattern, and printing it? Grep prints the whole line if it matches. That is because many unix tools are line based - they analyze each line as a unit and then output the results after each line. Grep will output each line as

it finds matches, awk will output each line if you tell it to. We can see the progress as each line is analyzed by running jobs in the background and using less to read the current output.

Awk works in a similar way, except you can use the BEGIN block to define variables before the computer starts reading in lines. So we could do something like (this is not real code but an English version of it - called *pseudocode*):

'BEGIN {sum = 0} {add value from first column to sum for each line} END {after all lines are read, print sum}'

Notice our example doesn't have a BEGIN or END block - they are optional! If you don't have things you wish to do at the beginning or end, you can skip them. You can technically skip the middle line by line code block, but I cannot really think of a case where you would want to do that...

3) $1, $2, $0

Awk has variables built into it, just like unix. They, like in unix, are designated by the $. And just like unix, we have built in, predefined variables - just like $HOME.

Predefined awk variables:

$0 - the full current line
$1 - first field (column) in current line
$2 - second field (column) in current line
...

NF - number of fields in current line
NR - number of lines (records) seen so far including the current line (so first line NR = 1)
FS - field separator. Set this in BEGIN to define your field delimiter

So we can see in our example that we are looking at the the first and second columns and the whole line variables.

4) !=

This is read as "not equal to". In our example then, ($1 != $2) means the first column in the line does not equal the second column in the line.

This is called a BOOLEAN (true/false) logic operator. Basically, these are points of evaluation that result in a TRUE or a FALSE. Different options are as follows:

== - equal to
!= - not equal to
> - greater than
< - less than
<= - less than or equal to
>= - greater than or equal to
|| - or
&& - and

Note: The equal always comes second. X = Y defines X as equal to the value Y. X == Y is either true or false. X =< Y would make X equal to "< Y". By putting the equal sign second, the computer can differentiate between definitions and boolean logic operators.

So in our example, in the line by line block, the computer is evaluating if the first column is identical to the second column.

5) if () print

Awk has built in functions too, similar to built in bash commands like cd. if is one of them.

In the example:
if (it is true that our first column does not equal our second column) then print the whole line.

The part after the if statement is only completed if the boolean operation in the if statement is true. If $1 == $2, it would not print the whole line ($0).

There are other functions, here are a couple useful ones:

length($1) - length (in characters (chars)) of field 1
$1**2 - field 1 to the power of 2
log($1) - natural log of $1
int($1) - integer part of numeric field 1 (e.g. 3.5 -> 3, -3.6 -> -3)
exp($1) - e to the power of $1
rand() - random number between 0 and 1 (including 0 but not 1).
srand() - reseed random numbers

Note: You have to call srand() in the BEGIN block to get a new set of random numbers each time awk is run, or you won't have randomized output - you will get the same output each time. This is because the computer doesn't have real random numbers, it has a very large list of unordered numbers that it then uses. If you don't ask it to start in a different spot (use a different seed number), you will start in the same point in the unsorted list and use the same "random" numbers each time. Many programs will ask you for a seed (Mr Bayes is one) for random numbers. This allows you to rerun the same numbers ans check consistency if you wish.

Let's put it all together:

-->>**awk '{if ($1!=$2) print $0}'** < file.txt

We are calling awk and telling it to run the following code: Nothing needs to be done before lines are read in. For each line, evaluate if the first column equals the second column, if it's true, print the whole line. Nothing needs to be down after all the lines are read. Do all this using file.txt.

******Examples******

Let's look at some other examples using our hmm output. Good thing I stored the code to get the table from the Dmel file in Dmelcode.txt

-->> less Dmelcode.txt
-->> cat Dmel_p450.hmmout | grep -A 131 "Query:" | tail -n 126 | sort -dk 9 | less –S

-->> cat Dmel_p450.hmmout | grep -A 131 "Query:" | tail -n 126 | sort –dk 9 >
Dmel_p450_hmmtable.tab

HMMER EXAMPLES:

1) Print first two columns of each line:

-->>cat Dmel_p450_hmmtable.tab | awk '{print $1, $2}' | less –S

Compare to:

-->>cat Dmel_p450_hmmtable.tab | awk '{print $1 $2}' | less –S

2) Print not very good hits:

-->>cat Dmel_p450_hmmtable.tab | awk '{if ($1 > 1e-10) print $0}' | less -S

3) Categorize hits into low, med, high quality:

```
-->>cat Dmel_p450_hmmtable.tab |
awk '{
if ($1 > 1e-10) print "high " $0;
else if ($1 > 1e-30) print "med " $0;
else print "low " $0
}' | less -S
```

Note that loops will only continue as long as nothing true has been found. So if $1 = 5e-3, which is bigger than 1e-10, the first part of the if statement is true. The computer will print "low" and the line, and then the next line will be read – none of the other conditions matter. If/elseif are order dependent!

NOTE: You can have several conditions in if statements as long as they are strung together with else ifs. You can also have newlines in the middle of your command. The program will wait until you give it all that it needs.

BLAST EXAMPLES:

4) Print number of bases matched: Xla_TLR2.outfmt7

-->>less Xla_TLR2.outfmt7

Let's get rid of the commented lines:

-->>cat Xla_TLR2.outfmt7 | grep -v "#" | less -S

And now let's look for the number of bases matched:

-->>cat Xla_TLR2.outfmt7 | grep -v "#" | **awk '{print $4 * $3/100}'** | less -S
#Where $4 is the hit alignment length and $3 is the percent identity.

Well that doesn't give us nicely formatted information, so let's make it tell us what it's comparing:

-->>cat Xla_TLR2.outfmt7 | grep -v "#" | awk '{**print $1 " to " $2 " :** " $4 * $3/100}' | less -S

Hmm... all those numbers are close to integers, but not rounding quite right. Let's use the int() function.

-->>cat Xla_TLR2.outfmt7 | grep -v "#" | awk '{print $1 " to " $2 " : " **int(**$4 * $3/100**)**}' | less -S

Buuut... that just rounds numbers down! Let's fix that with a typical rounding trick:

-->>cat Xla_TLR2.outfmt7 | grep -v "#" | awk '{print $1 " to " $2 " : " int($4 * $3/100 **+ 0.5**)}' | less -S

If you add 0.5, and take the integer, it will round properly (i.e.: 3.4 + 0.5 = 3.4 -> 3. 3.6 + 0.5 = 4.1 -> 4).

Note: Strings which can't be parsed as numbers are treated as 0 in math computations. So 1e-4 = 0.0004, but bob = 0.

5) Average hit length:
-->>cat Xla_TLR2.outfmt7 | grep -v "#" | awk '{print $1 " to " $2 " : " int($4 * $3/100 + 0.5)}' | **awk 'BEGIN{sum = 0} { sum = sum + $5 } END {print "average hit length: " sum/NR}'** | less -S

******See Reciprocal Best Hit PPT for Extended Example******

PPT that accompanies the above notes:

Up until now, we have been using the general syntax of:

program -options inputs (outputs)

i.e. : hmmsearch -A file.sto file.hmm file.fasta → STDOUT

blastn -query query.fasta -db db.db -out file.out

grep -v "#" file → STDOUT

cat file.txt | **awk '{if ($1!=$2) print $0}'** | less -S
(Or **awk '{if ($1!=$2) print $0}'** < file.txt)

Let's break this down and fill in the concepts we haven't covered yet:

1) awk
2) '{ }'
3) $1, $2, $0
4) !=
5) if () print

cat file.txt | **awk** **'{if ($1!=$2) print $0}'** | less -S

1) awk
awk is a great program for doing calculations and some text manipulation.

We start the command with awk to tell the computer that the following information will mean something to that program, just like we have done with everything else.

Awk is a bit complex and can be a bit hard to get used to at first; but thankfully, a lot of the complexity can be ignored in day-to-day use and it is a super useful basic unix tool.

```
cat file.txt | awk '{if ($1!=$2) print $0}' | less -S
```

2) '{ }'

There is a distinct structure to awk:

awk 'BEGIN {things to do before first line} {things to do for each line} END {things to do after last line}'

Awk works in line by line, like grep, except you can use the BEGIN block to define variables before the computer starts reading in lines. We could do:

PSEUDOCODE

'BEGIN {sum = 0} {add value from first column to sum for each line} END {after all lines are read, print sum}'

Like a pseudogene, pseudocode is similar to code but not functional

Notice our example doesn't have a BEGIN or END block - they are optional! If you don't have things you wish to do at the beginning or end, you can skip them.


```
cat file.txt | awk '{if ($1!=$2) print $0}' | less -S
```


3) $1, $2, $0

Just like unix, we have built in, predefined variables - just like $HOME.

Predefined awk variables:

$0 - the full current line
$1 - first field (column) in current line
$2 - second field (column) in current line
...

NF - number of fields in current line
NR - number of lines (records) seen so far including the current line (so first line NR = 1)
FS - field separator. Set this in BEGIN to define your field delimiter

In our example, we are looking at the first and second columns and the whole line variables.

```
cat file.txt | awk '{if ($1!=$2) print $0}' | less -S
```

4) !=

This is called a BOOLEAN (true/false) logic operator. Basically, these are points of evaluation that result in a TRUE or a FALSE.

== - equal to
!= - not equal to
> - greater than
< - less than
<= - less than or equal to
>= - greater than or equal to
|| - or
&& - and

Note: The equal always comes second.
X = Y defines X as equal to the value Y.
X == Y is either true or false.
X =< Y tries to defines X equal to "< Y".
X<= Y is either true or false

By putting the equal sign second, the computer can differentiate between definitions and boolean logic operators.


```
cat file.txt | awk '{if ($1!=$2) print $0}' | less -S
```

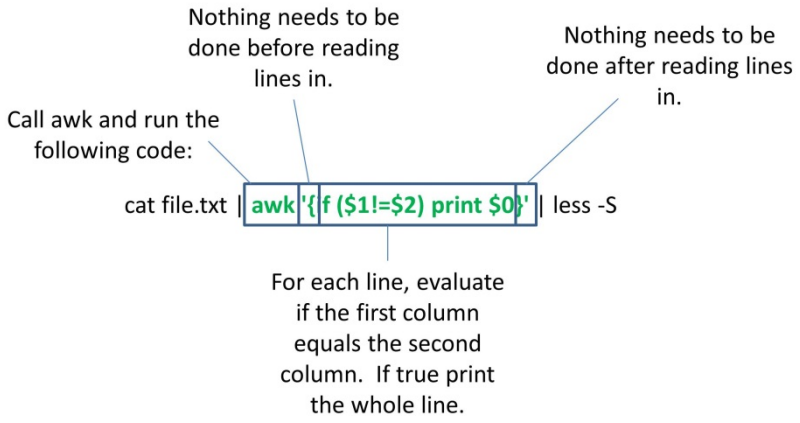

5) if () print

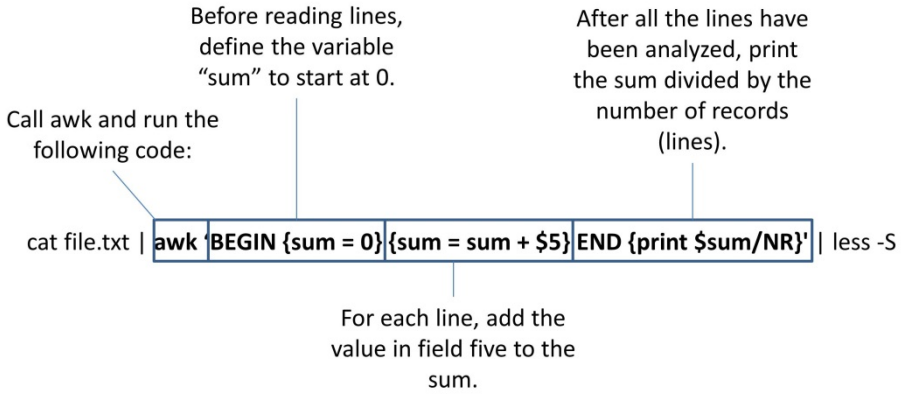
Awk has built in functions too, similar to built in bash commands like cd. "if" is one of them. The part after the if statement is only completed if the boolean operation in the if statement is true.

If $1 == $2, it would not print the whole line ($0).

There are other functions, here are a couple useful ones:

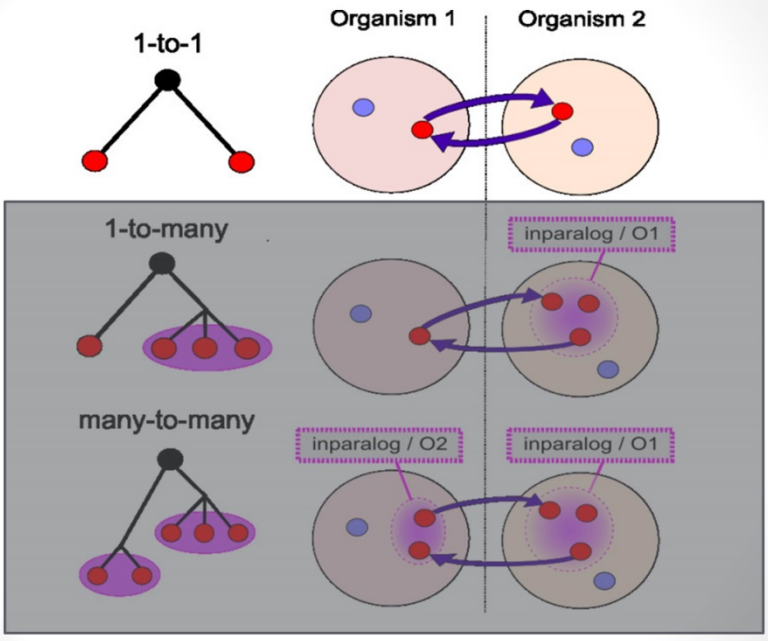
length($1) - length (in characters (chars)) of field 1
$1**2 - field 1 to the power of 2
log($1) - natural log of $1
int($1) - integer part of numeric field 1 (e.g. 3.5 -> 3, -3.6 -> -3)
exp($1) - e to the power of $1
rand() - random number between 0 and 1 (including 0 but not 1).
srand() - reseed random numbers





What is it really doing?

Reciprocal Best Hit PPT:



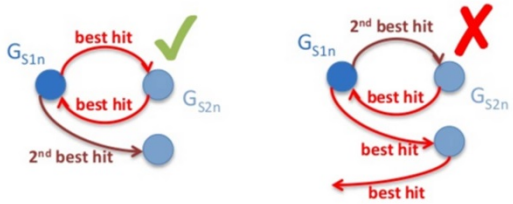
RBH Approach

Set1.fasta and Set2.fasta are gene sequences from two organisms. We want to find the one-to-one orthologues.

BLAST:

Query: Set1.fasta Subject: Set2.fasta
Query: Set2.fasta Subject: Set1.fasta

Optionally filter BLAST hits (~30% identity, 80% coverage)
Find all cases where gene1 is gene2's best hit and vice versa.



Genome Level

One-to-one orthologs for all genes in *X. tropicalis* and *X. laevis*?

Get genomes
Blast Q:Xtrop_9.0_aa.fasta S:Xla_8.0_aa.fasta
Q:Xla_8.0_aa.fasta S:Xtrop_9.0_aa.fasta

Family Level

One-to-one orthologs for all genes **(or TLR2)** in *X. tropicalis* and *X. laevis*?

Get genomes **and get TLR2 genes**.
Make hmmer model for TLR2 gene.
Find TLR2 genes in Xtrop
Find TLR2 genes in Xla
Blast Q:Xtrop_**TLR2**.fasta S:Xla_**TLR2**.fasta
Q:Xla_**TLR2**.fasta S:Xtrop_**TLR2**.fasta
Blast output7…

Set2toset1.outfmt7

Fields: query id, subject id, % identity, alignment length, mismatches, gap opens, q. start, q. end, s. start, s. end, evalue, bit score
1 hits found

Contig9	c8460_g1_i1	77.22	1238	262	17	247	1474	1797	570	0.0	706

Set2toset1.outfmt7:

Fields: query id, subject id, % identity, alignment length, mismatches, gap opens, q. start, q. end, s. start, s. end, evalue, bit score
2 hits found

c8460_g1_i1	Contig9	84.36	1413	212	9	553	1961	509	1916	0.0	1376

We want to get to a file that has the information from both files in one spot, with one column for each species (use Awk!):

Contig9	c8460_g1_i1	#From file 1
Contig9	c8460_g1_i1	#From file 2
Contig45	c23563_g1_i1	#From file 1
Contig45	c34632_g1_i1	#From file 2
Contig83	c56356_g1_i1	#From file 1
Contig83	c56356_g1_i1	#From file 2
Contig83	c56389_g4_i1	#From file 2

Then we will collapse and count identical lines…

While you wait…

We'll pick this up on Thursday, but while you wait –

- The following files are in the $COURSE/Shared/bin
  - set1.fasta are amino acid TLRs from Xla_8.0_aa.fasta
  - set2.fasta are amino acid TLRs from Xtrop_9.0_aa.fasta
  - set1toset2.outfmt7 is the output from Q: set1.fasta S: set2.fasta
  - set2toset1.outfmt7 is the output from Q: set2.fasta S: set1.fasta

- Try your hand at the following (in your space!):
  1. output the first two columns from set1toset2.outfmt7 to a file (RBH.list) – deal with those headers first!
  2. output the first two columns from set2toset1.outfmt7 to the same file (RBH.list), but reverse the order (print column2 column1).
  3. check to see that the columns have only one individual each

Set1toset2.outfmt7

Contig9	c8460_g1_i1	77.22	1238	262	17	247	1474	1797	570	0.0	706

Set2toset1.outfmt7:

c8460_g1_i1	Contig9	84.36	1413	212	9	553	1961	509	1916	0.0	1376

Print column 1 and column 2 of Set2toset1.outfmt7 ~~to out_file~~

collapse all lines sharing the same query then output

Print column 2 and column 1 of Set2toset1.outfmt7 ~~to out_file~~

collapse all lines sharing the same query then output

Sort lines to stack lines

But what about the multiple hits?

Hint: BLAST lists best first!!

Contig9	c8460_g1_i1	#From file 1
Contig9	c8460_g1_i1	#From file 2
Contig45	c23563_g1_i1	#From file 1
Contig45	c34632_g1_i1	#From file 2
Contig83	c56356_g1_i1	#From file 1
Contig83	c56356_g1_i1	#From file 2
Contig83	c56389_g4_i1	#From file 2

Count identical lines/Look for lines that occur twice (once from each file)

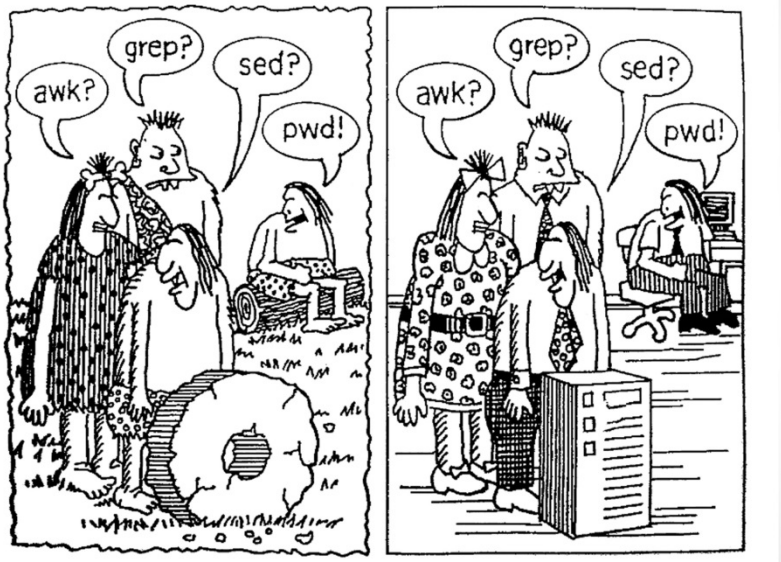
Read Set1toset2.outfmt7	cat	
Collapse all lines sharing the same query	**?**	**sort –u –k1,1**
Print column 1 and column 2	awk	
Output to file	>	
Read Set2toset1.outfmt7	cat	
Collapse all lines sharing the same query	**?**	**sort – u –k1,1**
Print column 2 and column 1	awk	
Output to file	>	
Sort lines to stack lines	sort	
Count identical lines/Look for lines that occur twice	**?**	**uniq -c**

So… what's the difference between sort –u and uniq?

These two cases actually!

- Sort –u allows for collapsing based on a single column, uniq will not.
- Uniq will count uniq lines with the –c option, sort lacks a similar option to my knowledge.
- Sort –u and uniq without options are identical.

And now… sed.



Part III: Sed and Regular Expressions

******Sed******

Sed stands for "stream editor," and in its most common usage is a line-by-line search and replace. It was first written in 1973.

First, I'll give you the command to do what we want, which is to "this" with "that". Then we'll cover sed in detail for a while. Here's what the command we want is going to look like:

-->>sed 's/this/that/g'

Let's deconstruct it:

s - tell sed we're running a substitute operation
/this/that / - replace the first pattern (this) with the second pattern (that)
g - global, do the replacement for every match on the line, not just the first

Sed can do a lot; we're only going to cover maybe 5% of what sed can do. But even that 5% is about 80% of what biologists use sed for, so its pretty useful. …And that's pretty much the basic syntax:

's/something/somethingtoreplace/g'

where s stands for substitute, and g for global. You can change the g to a number to define a number of times to make the substitution, but you really never have a need to do this.

Okay, let's try some things!

****** Toy Examples******

Given a file salamanders.list
Spotted
Blue Spotted
Jefferson's
Tiger
Red-backed
Slimy
Small Mouth
Marbled

If I wanted to replace Small Mouth with Texas (another name for it), I could write:

-->> cat salamanders.list | sed 's/Small Mouth/Texas/g' | less –S

And my output would have the changed name.
Additionally, I could do:
-->>sed –i.bak 's/Small Mouth/Texas/g' salamander.list

And edit the file in place, while creating a backup. Note that the following two commands will not work:

cat salamander.list | sed –i.bak 's/Small Mouth/Texas/g'
sed –i.bak 's/Small Mouth/Texas/g' < salamander.list

You will get an error that sed cannot find the input. This is because you are trying to read from STDIN (which is part of the stream flow) and not a file. Without a file name attached (as in it comes from STDIN) , it cannot make a file_name.bak.

Note: You can use ANY extension you want. sed –i.plzletthiswork 's/Small Mouth/Texas/g' salamander.list is a legitimate unix command ☺.

You can also edit a file in place without a backup simply by using –i.

Finally, I can delete matches as well:

-->>cat salamanders.list | sed 's/Small Mouth//g'

******Real Examples******
Trinity assembly files have the naming convention of:

c10000_g1_i2
c10001_g1_i2
c10001_g2_i1
c10001_g2_i2
…

These represent groupings of genes (c values), isoforms (g values), and alleles (i values). We also looked at how to pull a subset list out of a fasta file. Imagine the following:

I have a list of genes that are identified as well supported transcripts (real assemblies). For example:

c1000_g1_i2
c10000_g2_i1
….

But for each of these great identified transcripts, I want to pull all the forms of the gene, not just the best identified ones. How would I change my search query from:

c1000_g1_i2
c10000_g2_i1
….

To also include c1000_g1_i1 and c10000_g1_i1, etc? By reducing the pattern to the point where they match (the c term)!

We want to convert c####_g#_i# →c####_ (why do we want that last underscore?)

But how?

******Regular Expressions (or Regexs)******

We've been talking about pattern matching in grep and now sed (you can use it in awk too). Thus far we have been talking about absolute, fixed patterns (called literal – ATC matches ATC exactly). We can do a bit more with a more flexible pattern matching concept called regular expressions, or regex. Regex often look like your cat walked on the keyboard, and take some getting used to (I HIGHLY suggest the tutorial I posted – it's interactive and will help a ton. The rubular link will help you test regex interactively before sticking it in your code).

Let's start with some basic special characters of regex, but first, let's talk about the two different versions of regex. There are the POSIX (portable operating system in unix) standards and the extended regex. The POSIX characters will work with everything. The extended are very useful in many cases, but require you to tell the program to run the extended version. You have to use sed –E and egrep to have your regex work more consistently (because it will run all the regex options). I recommend using aliases so you don't have to remember (i.e.: sed='sed –E'; grep='egrep' in .bashrc). You may also see sed –r as a flag – it means essentially the same thing as –E. From what I can find, awk runs with extended regular expressions by default.

Some basic characters:
. - matches any single character, including whitespace (e.g. GC. matches GCA/GCC/GCG/GCT)
*** - match the preceding thing 0 or more times. So ATG*TGA matches both ATGTGA (there's no stuff in between) and ATGGGGTGA**

Just like when we were matching any character in unix with ?(single character) and *(0 or more), we can match some number of characters in regex syntax.

Let's try some things with just these two options. We can actually solve our list problem pretty ineloquently:

-->> cat bestorf.list | sed -E 's/_g._i.//g' | less -S

Hmm… but that misses anything with a two digit number (i.e. g1_i10)… We need to learn something new here…

-->> cat bestorf.list | sed -E 's/_g.*_i.*//g' | less -S

What makes this ineloquent is the fact that it's not terribly specific and can cause problems if we aren't careful.

What about matching a set of things, like nucleotides or in this case numbers?

[ ] - matches a single character named in the brackets (e.g. [ATCG] matches a SINGLE base. [A-Z0-9a-z] matches any SINGLE lowercase letter, uppercase letter, or digit)

Ok, so let's try this:

-->> cat bestorf.list | sed -E 's/_g[0-9]_i[0-9]//g' | less –S

Or better yet:

-->> cat bestorf.list | sed -E 's/_g[0-9]*_i[0-9]*//g' | less -S

What if we want to match something 1 or more times? Say we want to know how many sequences have a base was called as totally ambiguous (N)?

+ - matches one or more characters (NOT POSIX)

-->> cat Newdata.fasta | grep –v “>” | egrep N+ | wc -l

Egrep N+ is functionally equivalent to egrep NN* - which tells it to match one N, then match N again 0 or more times.

What about a certain number of times? Say we want to print any line that has a sequence longer than 500 bases?

{x} - matches the preceding thing y times (NOT POSIX)

-->> cat RNA.fasta | egrep -B 1 [ATGC]{1000} | less -S

How do we reverse the set, like we did with –v in grep?

[^] - matches a single character NOT named in the brackets

-->> cat RNA.fasta | sed -E 's/[^ATCG]//g' | less -S

A more complex set? Like finding a stop codon?
| - the ‘or’ operator - matching thing A or B (NOT POSIX)

-->> cat RNA.fasta | sed -E 's/TAA|TAG|TGA/stop/g' | less -S

A specific location? A start codon, but only if it is at the beginning (and therefore not just methionine)?

$ - matches the end of a line
^ - matches the start of the line (like we talked about before) - no change in behavior if in [ ] (NOT POSIX)

-->> cat RNA.fasta | sed -E 's/^ATG/start/g' | sed -E 's/TAA|TAG|TGA/stop/g' | egrep -B 1 "stop" | less

NOTE: Running sed commands in serial like this (one after another) requires the computer to loop through lines several times. You can make your pipes faster by using the –e option (which stands for expression). You put a –e before each sed statement as follows:

-->> cat RNA.fasta | sed -E –e 's/ATG/start/g' -e's/TAA|TAG|TGA/stop/g' | egrep -B 1 "stop" | less

See how weird these can end up being? It helps a lot to test and check them on rubular…
http://rubular.com/
Also see they grymore for an AWESOME handbook to sed (search for anything you need):
http://www.grymoire.com/Unix/Sed.html

Checkpoint: Unix Tools (HW 4)

Given the HW4.fasta file:

Grep:

Given the HW4.fasta file:
How many sequences are there in HW4.fasta?
How many sequences are there in the + strand?
How many sequences are there with at least two versions (i.e.: c####_g1_i# and c####_g2_i# exist)?
REQUIRED: Grep.answers

Awk:

Given the HW4_awk.txt
What is the average e-value of the MHC full sequence hits?
What is the average e-value of the MHC domain hits?
Hint: how did we pull out the table in class? How could you eliminate the header lines from the STOUT?
REQUIRED: Awk.answers

You choose:

Make an executable program that converts fastq files into a fasta file. Use NewData.fq (from HW1) for reference and testing!
Hint: Only the first two lines are needed, how do you print just those to STOUT? What needs to change on the first line? How do you get rid of the --'s?
REQUIRED: fastq2fasta.ba

Regex:

Complete this tutorial: http://regexone.com/
This site is a great place to test your regex: http://rubular.com/

We can use regular expression (regex) in place of any pattern in grep or sed. Using regular expressions, complete the following:

Shorten the names in HW4.fasta to just the c###_g_#_i# part of the name (use sed).
Hint: look for where you want to start cutting off, match to that and then match to the end of the line. Use sed to replace that match with nothing.

REQUIRED: shortenedNames.fasta

Summary of Required Files to be in your HW4 dropbox folder:
1 - README (Points: 1)
2 - Grep.answers (Points: 2)
3 - Awk.answers (Points: 2)
4 - fastq2fasta.ba (Points: 2)
5 - shortenedNames.fasta (Points: 3)

Checkpoint: Unix Tools In-class Practice

Answers are in the Answers section of this document, at the end.

1. Given $COURSE/Shared/data/Greppractice.txt, write the grep command for printing lines with (use the man pages to figure this out if you need to):

a. "this" regardless of case:

b. only the word "is", so you would return "is" but not "this":

c. display 1 lines after finding "this" (case-insenstive):

d. display 4 lines before "this" (case-insenstive):

e. print 2 lines around "this" (case-insenstive):

f. print every line without "this" (case-insenstive):

g. display line number of all lines with "this" (case-insenstive):

2. sed practice:

a. Given $COURSE/Shared/data/Sedpractice_a.txt, convert all "can't" to "can"
HINT: if escaping doesn't work, try putting single quotes around it!

b. Given $COURSE/Shared/data/Sedpractice_b.txt, convert all numbers to have two decimal places.

c. Given $COURSE/Shared/data/Sedpractice_c.txt, swap the first two letters of each word.

d. Given $COURSE/Shared/data/Sedpractice_d.txt, swap the first two words in each sentence.

3. Regex golf (use rubular or the other regex interactive I sent you). The goal is to match all off the targets and none of the "but not's". Each target counts for 10 points, each non target is -10 points, each character in your regex is -1 point. Try to get the best score you can in your group! Each list is available as Hole 1-7 in the Shared/data section!

Hole 1. How would you match:

	But not:
caaba	intransigeantly
aface	unthematic
dace	autolithography
bedad	haemostasia
decede	alite
feed	bigmouthed
decaf	ash-free
affa	Marcelle
fab	switchtail
abaff	aftermilk
faced	argenol
adead	grillade
cadee	CASU
accede	half-wittedly

Hole 2. How would you match:

	But not:
	exercent
auriscalpium	nonfreeman
blood	ignorances
countersuggestion	intersegmental
conformator	fourth-rate
Causey	players
besieger	regathers
biseriately	nonsurvival
cyanoacetate	unsanctitude
cioppino	decurrence
Crashaw	thanato-
	coto

Hole 3. How would you match:

	But not:
allochirally	anticker
anticovenanting	corundum
barbary	crabcatcher
calelectrical	damnably
entablement	foxtailed
ethanethiol	galvanotactic
froufrou	gummage
	gurniad

Hole 4. How would you match:

uninstructedly	But not:
reinstituting	unbenign
carryings-on	misstops
forgings	unsusceptible
insipiently	cauterizations
institutionalisation	prerogatives
inseminations	about-faced
leggins	rattlesnakes
instruments	tempest-loving
antivenins	Mayday
tinsmiths	drown
well-inspected	dolus
peninsularism	royalistic
patins	chitting
recordings	Borman

Hole 5. How would you match:

Ultraparallel	But not:
suprastapedial	fork-tongued
naphthanthracene	polyclinic
Walachian	foolscap
cachaemia	sacrosciatic
bathomania	plotlib
Craniata	self-direct
bahamians	ladderway
tarantula	serologic
achromaturia	whaling
aguavina	hurlpit
aquaria	LHS
abaxial	dentition

Hole 6. How would you match:

damager	But not:
venins	polarization
piner	Ovangangela
unsimpleness	life-guard
humbuggism	zoographically
meacon	reundulate
mythist	Globe
lycanthropist	well-solved
decreases	dumpling
directing	zoophorus
Munchhausen	tartly

4. Functional example:
To make a multiline fasta:
>sample1
ACTGATGATGGGACTGAT
ACTGATGATGGGACTGAT
ACTGATGATGGGACTGAT
ACTGATGATGGGACTGAT

Into a single line fasta:
>sample1
ACTGATGATGGGACTGATACTGATGATGGGACTGATACTGATGATGGGACTGATACTGATGATGGGACTGAT

We need to get rid of all the newlines in the sequence. There are two problems here:

1) Most of our programs only read one line at a time.
2) We don't need to change the lines starting with a ">"

One way to do this (in bash) is the following:
Mark the end of all info lines with "###"
Remove all newlines
Add newlines where we want:

a) At end of info lines (where ### are)
b) Before the info lines (where > are)

Some necessary information:

You can write pipelines on multiple lines, as long as they have the pipe at the end of the line:

```
i.e. cat info |
sed 's/this/that/g' |
sed 's/that/this/g' |
less
```

sed cannot read across lines and will not see the newline (\n) character at the end of a line.
Instead we can use translate (tr) to replace the newlines with a character (&), follow with a |. Note that tr can only replace one character with one character. \n counts as one character.

```
i.e.:   tr 's' 'z'
        tr '^M' '\n'      #^M is a newline sometimes used in Windows
```

Why have one tool that does everything when you can have two that do sort of the same thing (like: sort –u/uniq)!

```
#####################################################################
###                                                               ###
###                fasta2oneline.ba - pseudocode                  ###
###      Fill in the real code to make this program work!         ###
###                                                               ###
#####################################################################


#HOW TO CALL PROGRAM: cat input.fa > fasta2online.ba > output.fasta
#How to troubleshoot while you are building: cat input.fa > fasta2oneline.ba | less

#Add “###” to the end of any info line.  Remember to end with a |!


#Replace all new lines (\n) with & using tr (general usage: tr 'this' 'that')
#Why didn't we just remove the \n with tr?  What happens if you try to do: tr '\n' ''?


#Delete &'s


#Replace all ### with new lines (\n), end with a |


#Replace all > with \n>, end with a |


#Remove first line, as it will be blank
sed ‘1d’
```

MIDTERM REVIEW (HW5):

Major topics:

Navigation commands
Permissions
PATH and .bashrc
Installing programs
Options
BLAST and HMMer
File formats
Tree building
Unix tools – grep/awk/sed
Regular expression

Matching:

Match the command with its purpose:

A. Move or rename a file.
B. Get information
C. Remove a file.
D. Copy a file
E. Find a program's home folder
F. Output date and time
G. List last ten lines of file to terminal.
H. Soft link
I. Open a file without editing it.
J. Open a file for editing.
K. List top ten lines of file to terminal.
L. List the resources in use on the computer cluster.

__ less
__ nano
__ mv
__ cp
__ rm
__ top
__ head
__ tail
__ info
__ date
__ which
__ ln –s

A. / __ Root or separates directory names
B. * __ Escapes a special character to make it literal (i.e:$ means $, not variable)
C. $ __ Indicates a Unix variable
D. # __ Used to match any number of characters (in Unix)
E. : __ Used to match any single character (in Unix)
F. ; __ Begins a comment in bash
G. #! __ Separates commands on a single line
H. ? __ Indicates the location of the program that the code is written for
I. \ __ Used to separate directory locations in PATH

Match the program with its function:

A. Blast __ performs math functions, prints columns individually
B. HMMer __ text reader
C. grep __ counts characters, lines, and bytes
D. sed __ finds lines that have matches to a pattern
E. less __ lists the resources in use for a machine
F. nano __ uses an unknown string to search a known database
G. wc __ uses a known set of strings to search an unknown set of data
H. top __ tells you the quality information of a sequence file
I. fastqc __ find and replace
J. awk __ text editor

Multiple Choice:

1. Who has permission for the following file?
-rw-rwx--x MMacTaggert acethisexam.txt
A. user, group, and others can execute
B. user and others but not group can read
C. only the group has all permissions
D. user can read and execute the file

2. Which command would you use if you wanted to find an average e-value from a .txt file output from a hmmer search?
A. Use grep to find the average e-value
B. Use sed to isolate the table and then awk to find the average e-value
C. use grep to isolate the table and then awk to find the average e-value
D. Use awk to find the average e-value

3. Which of the following tar options will work correctly?
A. tar –xfz file.tar.gz
B. tar -xzd file.tar.gz
C. tar xdf file.tar.gz
D. tar -xzf file.tar.gz

4. Which of the following things would you NOT expect to see in a .bashrc file?
A. aliases
B. automatic login to bash
C. PATH declarations
D. customization of your prompt
E. Variable declarations

5. The difference between fastq and fasta is:
A. fastq has amino acids, fasta has nucleotides
B. fastq has quality scores, fasta does not
C. fastq has two lines per read, fasta has four
D. fastq has ">" to denote samples, fasta has "@"

6. How can you fix a broken .bashrc?
A. give the absolute path to .bashrc
B. close terminal and reopen
C. use nano in cshell and edit
D. less .bashrc

7. How would you replace all instances of “abba” in a file with “bob”?
A. sed ‘g/abba/bob/s’
B. sed s/abba/bob/g
C. sed ‘s/abba/bob/g’
D. sed ‘s/bob/abba/s’
E. grep –r abba bob

Short answer:

1. What does the “which” command do?

2. What are two considerations when picking an alignment program?

3. What are two ways to direct input into a program?

4. What is an advantage of regular expressions? What is an advantage of extended regular expressions?

5. What is the purpose of PATH?

6. Which are the two most commonly used blast types?

7. If you already have both a set of known genes and a file with unknown sequences, what are the two steps of Hmmer and what does each one do?

8. What is the difference between a source and a binary file?

9. Where would you look if you don't know how to install a program?

10. What is the difference between AFS ACL's and Unix permissions?

11. What are two things wrong with the following command? Grep –A –v 1 ">"

12. What is the purpose of a |? What are STDIN and STDOUT?

13. Explain what this awk command does in plain English:
awk 'BEGIN{sum=0}{if ($1 > $2) {sum = sum + $1}END{print sum}'

14. Explain what this sed command is doing:
sed ‘s/(sample1).*/\1/g’

There will be a longer answer section…
It would be wise to review RBH as a bridge of a lot of our concepts...

Foobar Trivia Review

Rules and Organization

- Form groups of ~4.
- There will be three rounds, three questions each
- You can “bet” 5, 3, or 1 point per question (only once each though!)
- For each question, give me the points bet, group name, and answer.

You just sourced your .bashrc with “declare PATH=$HOME/local/bin”… how do you fix it?

Exit to csh to edit .bashrc
Use absolute path to nano (/bin/nano) to edit .bashrc

Remember, the computer only does what you tell it…

How would I know if something is a command or a program?

If it runs without a path (cd file.txt) and isn’t found in $PATH (which cd), it’s a command.

DISARM THE BOMB!

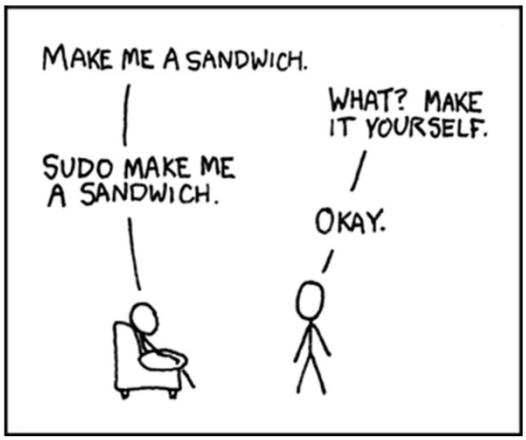
Good or Bad?

Is this a quality sequence?

```
@FCC1PFHACXX:2:2316:19544:100787#ATCACGA/1
CGGTCCAACAATTCTGTGCAAAAAAAAAAAATTTTCAAA…
+
\^^\c^acYbcchhddZXY\\\^^^acdasalwejlrwe…
```

What are the two meanings of root?

Root can mean either the super admin or the top of the file tree.



Name a special character with two meanings we have discussed in class (I can think of 3).

/ - root or directory separator
> - directs STDOUT into file, starts a sequence line in fasta
- comment or first half of #!

BLAST or HMMER

1) "L  or this set of unrelated genes in our new genome"

2) "We found this alien ar  u to tell us if it has hemc "

3) "I  to know if these random strings etters are real proteins."

Unix or AFS

1) Refers to files

Unix

2) Refers to directories

AFS (our system and all large systems

3) Is changed by chmod

Unix

Bonus: Allows you to have more than one group

Both!

Source or binary

1) Human readable

Source

2) Has a makefile

Source

3) Not customizable

Binary

Unix Tools

“How many sequences are there?”

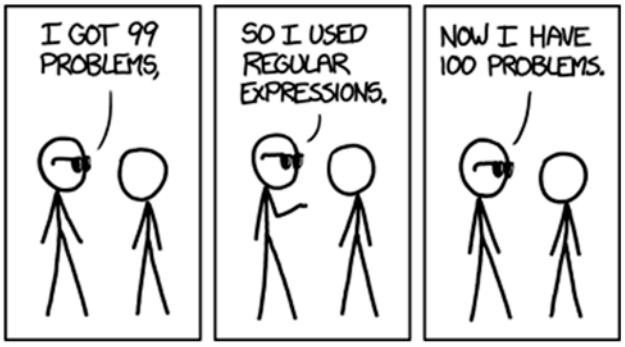
Grep

“Let’s shorten those sequence names!”

Sed

“What’s the average length?”

Awk



level [0-9]+

Tree Tangle

Regex error scrambled my file - Unscramble and put in the steps in order!

- repdonecsdo dendroscope (4)
- hlmpy phyml (3)
- elmusc muscle (1)
- bcsglok gblocks (2)

>Please enter three valid passwords for points.
>
>Valid passwords are: ^([a-zA-Z0-9@*#]{8,15})$
>

Any 8-15 letter entry containing a-z, A-Z, 0-9, @, or #.

"bu|[rn]t|[coy]e|[mtg]a|j|iso|n[hl]|[ae]d|lev|sh|[lnd]i|[po]o|ls" matches the last names of elected US presidents but not their opponents.

Editing for the lazy

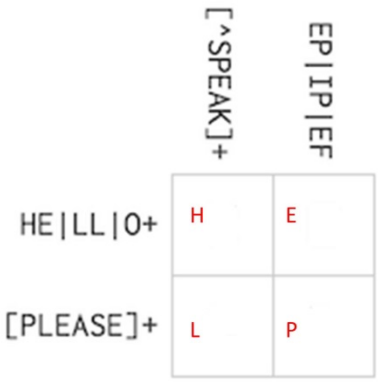
How would you change the following file to have absolute paths instead of relative paths, without manually editing each line?

Assume all references are to .. and pwd is ~/local/data/run1

```
bowtie2 -x Ref -1 ../J_1.sample1.fq -2 ../J_2.sample1.fq -S J.sample1.sam
bowtie2 -x Ref -1 ../J_1.sample2.fq -2 ../J_2.sample2.fq -S J.sample2.sam
bowtie2 -x Ref -1 ../J_1.sample3.fq -2 ../J_2.sample3.fq -S J.sample3.sam
bowtie2 -x Ref -1 ../J_1.sample4.fq -2 ../J_2.sample4.fq -S J.sample4.sam
```

Must consider: escaping .., / in sed and in file name, and that /run1 is not included.

BOSS LEVEL!!!!



Unit 4: Introduction to R

Basic R Commands and Philosophy

Introduction to R – a second programming language

Programming concept

We're going to start learning the statistics programming language R. There are other classes here that offer more experience with R, so we are going to focus on the programming aspects of the language. To start, we'll go over basics like accessing the environment, declaring variables, moving around, use commands, load data, how to get help, and install new modules. Sound familiar? It should - they are the same as the ones we covered in Unix! It is easier to learn a new language when you already know some basic programming concepts (like what a variable is and how it works). Much the same way learning French helps you understand English better, learning new languages helps solidify unifying concepts. And in turn, learning what concepts are similar will help you pick up additional languages faster – as we will see in Python!

How to get into R

Before we can start learning R, we will actually have to gain access to R. There are several ways:

1) download it to your personal computer (http://cran.mtu.edu/)

This is easy, it's not a big program, and I generally prefer to be able to work on stuff offline. You should be able to figure this one out.

2) use Rstudio

Also a potentially useful program to look into, especially if you are a fan of Windows is RStudio. RStudio is a slightly smaller version of R that makes viewing graphs, etc much easier. It installs on your machine (not requiring unix, crc, etc) but requires you to first have R installed (see 1). I tend to use this version for everything I do, but we will work on the unix version for now.

3) R commandline

Because everything is more fun on the commandline. I have install R into our shared bin. It should work. Check this with ... you guessed it - which R.

Launch R - just type R.
Quit R - q()

Please make sure you have access to R before you come to class!

Declare Variables

Ok, now that we are in the program, let's learn to print to the terminal and declare some variables.

There are lots of ways to print things in R; one that is super familiar is the cat() function, which simply prints to stdout. Another is simply "print".

```
name = "class";
cat("Hi", name, "\n");
print(name);
```

Note that comma separates the things to be printed, and it adds in spaces all on its own.

Also note the syntax here is a bit different. We don't have to "declare" anything – we can just define a variable with whatever we'd like. However, we don't use cat the same – it requires ()'s around what we want to cat. We will talk more about why this is later. Similar to unix, the ; defines the end of a command, and just like unix, it's not required. Finally, cat does not automatically put new lines at the end of commands like we saw in unix (its less of a line based language), but print does.

Things that are printed go to terminal by default (same old, same old), but R likes to print a lot (which can be annoying). For example, any value that is calculated but not stored somewhere gets printed. We will get more into this next week.

R Functions are like Commands, and they have options too.
In unix, we're mostly used to:

```
Command –option file
```

Functions in R are similar – the name of the function starts things off, followed by options, but in a slightly different format.

```
command(filename, option);
cat("word","are","here","\n", sep=",")
```

The major difference is that R functions have the options in a defined order and within parentheses. You can find out what the options are and what order you are to enter in the information by typing ?command:

```
?cat
```

One reason I love RStudio is it automatically gives you a heads up on what is expected and prints out the information in its own little frame.

Also, just like in unix, there are defaults to the functions. For example the cat function is as follows:

```
cat(... , file = "", sep = " ", fill = FALSE, labels = NULL,
    append = FALSE)
```

Any time you see something defined, such as sep = " " – it's a default! You can skip this part of the call if you want, which is why our first cat call (^_~) was so much shorter than this!

Where am I?

We used pwd to determine our present working directory. R has a similar concept of working directories – where it expects files you are referring to directly to be found.

getwd() – reports what the working directory is currently (ie: pwd for R)
setwd(dir) – allows you to define a new working directory (ie: cd dir for R)

```
getwd()
setwd("/afs/nd.edu/courses/cse/cse60132.01/Shared/data")
```

A Few Data Types in R

This is a topic we didn't get too far into in Unix, we happily used numbers and words in variables and thought little about it. We will build on this topic because R does care about the different types of variables, because each has different functions and rules attached to them.

Scalars

Scalars are numbers, strings, and boolean values. This is what we generally used in unix.

```
myvar = "hi";
myvar = 5.2;
myvar = FALSE;
```

Again, note that R does not require us to declare variables before we put data in them. However, if we try to pull data out of a variable that doesn't exist yet, R will give an error (just like unix).

```
print(mychar)
Error in print(mychar) : object 'mychar' not found
```

Vectors

Vectors are ordered lists of data, basically a group of scalars with distinct places in line:

```
vect = c("A", "C", "T","G","C");
```

c() is a function that takes its arguments and returns a vector. It stands for catenate.

Since the vector has ordered lists, we can access elements by asking for a specific index (position) in the vector. **NOTE: R is 1-indexed!**

vect[1]; -> "A"

We can also define a new, ordered vector using the seq function:
seq(start, end, increment);

And print the length of the vector:
length(vect); -> gives you the length of your vector

Or read a vector in from a file (read documentation on this one if you use it!)
vect.txt:
1 2 3 5 6
6
7
vect = scan(file="vect.txt");
vect[2]; -> 2

Matrices
Matrices are 2-dimensional grids of data. All cells must be of the same 'type' of scalar.

y = matrix(1:20, nrow=5, ncol=4); <--This generates a matrix from a vector (1:20), with a given number of rows and columns.

We can also fill a matrix by rows instead of columns, using an option called byrow:
a = matrix(c(1,2,3,4), nrow=2, ncol=2, byrow=TRUE);

We can also read it in from a file:
a = as.matrix(read.table("matrix.dat"));
a = as.matrix(read.table("matrixheader.data", header = TRUE, row.names = 1));

The standard separator for read.table is whitespace. To do a csv file (comma separate values), you can use read.csv, or in your read.table function you can set sep = ",".

We can also subset matrices:
y[3,] -> 3rd row
y[,4] -> 4th column
y[3,4] -> element at row 3, col 4
y[1:3,2:4] -> submatrix of rows 1-4 and cols 2-5
nrow(y) -> number of rows in y
ncol(y) -> number of cols in y

Arrays

There are also arrays in R, but they're not what we typically think of as arrays (which are vectors in R). These are multi-dimensional matrices. I'm not going to really go over these, but I just wanted to let you guys know they exist!

Data Frames
Data frames are like matrices, but each column can be a different type. This is likely the data type you will use most, since one column can be gene names (strings), one can be e-values of hits (numbers), another can be bit scores, etc. Columns can also be named, which really comes in handy.

We pretty much do not initiate data frames, we load them in from files.

Reading a Data Frame from a File
If the first line is a header describing columns:
```
mydata = read.table("filename.txt", header = TRUE);
```

If there is no header line and you want to name columns:
```
mydata = read.table("filename.txt", col.names = c("GeneID", "e-value",...));
```

We can also set colnames and rownames after the fact, using:
```
col.names(mydata) = c("GeneID", "e-value"...);
row.names(mydata) = c("a", "b"...);
```

For example:
```
mydata = read.table("hmmR.txt", header = TRUE, row.names = 1);
```

Subsetting Data Frames
Extracting columns:
```
idnbitscore = mydata[c("bitscore")];
```

Extracting rows:
```
goodhits = subset(mydata, e-value < 1e-35);
nrow(mydata);
nrow(goodhits);
```

Getting a vector of column names:
```
colNames = names(mydata);
```

To get the row names of a dataframe just call:
```
row.names(mydata);
```

General Syntax Note:

Do we use <- or = to assign? Technically these are the same thing, so you can use either. I tend to use = for everything but function creation, simply because = is a common assigner across most languages.

Getting more help:
Other than ?command, we can look at the function itself by calling
function; <- NO parentheses!

Or if that doesn't work:

methods(function);

Packages – installing new stuff into R
Packages are things you can download that add functionality to R. They contain functions written by other people, and packaged up all nicely so you can just load them, and then call the functions within.

To download a package:
install.packages("packagename");
Load a package:
library(packagename);

However, just like before, lack of root access makes things fail if we don't direct the program to a location we have access to!

install.packages("ggplot2", lib="/afs/nd.edu/courses/cse/cse60132.01/Shared/Rpackages/");
library(ggplot2, lib.loc="/afs/nd.edu/courses/cse/cse60132.01/Shared/Rpackages/");

Packages only need to be installed once, but they need to be loaded every time you have a new R instance. I recommend having a text file saved on your computer so you can copy and paste the text (it's long). This is much easier if you are working on your own machine, or in Rstudio.

Some potentially useful packages are:
vegan - principal components analysis, bray-curtis, multivariate statistics
ggplot2 - pretty pretty graphs
wgcna - pretty microarray heat maps, network visualizations
gplots - more graphing stuff
RColorBrewer - more colors
Rcmdr - GUI for statistical tests

Checkpoint: Bioinformatics in R I (In-class/HW6)

This is an introduction to bioinformatics in R, that I semi-stole from a great book I will link you to next week. The purpose of this activity is to get you working in R and getting a feel for some of the syntax. This is definitely a case of "the more you put in, the more you get out". Think about the syntax you are using, and what it's doing. It's all well and good to be able to type the commands and answer some questions, but thinking about commands and options and how vectors work will help you get more familiar with R.

Install your packages

Some well-known bioinformatics packages for R are the Bioconductor set of R packages, which contains several packages with many R functions for analyzing biological data sets such as microarray data; and the SeqinR package, which contains R functions for obtaining sequences from DNA and protein sequence databases, and for analyzing DNA and protein sequences.

To use function from the SeqinR package, we first need to install the SeqinR package. Try this in your own space (but if you are having a ton of trouble, flag me, and skip it and use the one I installed in Shared/Rpackages until I can help you). Once you have installed the "SeqinR" R package, you can load the "SeqinR" R package by typing:

```
> library("seqinr", lib.loc = "/wherever/you/put/it");
```

NOTE: This library requires a second package called ade4. You should install and load that one the same way before seqinr if you get an error saying it is missing. It is also preloaded in the Shared/Rpackages file.

Retrieving genome sequence data using SeqinR

You can retrieve sequence data from NCBI directly from R, by using the SeqinR R package. To retrieve a sequence with a particular NCBI accession, you can use R function "getncbiseq()" on the following page, which you will first need to copy and paste into R. This function just takes a several step process and converts it into one. I commented the code so you could see what it is doing. All functions (i.e. choosebank) are in the help files. If you have time after getting through the activity, I suggest digging into this a bit, since we will be writing functions next week!

```
getncbiseq <- function(accession)
 {
   require("seqinr") # this function requires the SeqinR R package
   # first find which ACNUC database the accession is stored in:
   dbs <- c("genbank","refseq","refseqViruses","bacterial")     #list of common databases to search
   numdbs <- length(dbs)
   for (i in 1:numdbs)                                          #for each database
   {
     db <- dbs[i]                                               #define db as current database
     choosebank(db)                                            #choose that database
     # check if the sequence is in ACNUC database 'db':
   resquery <- try(query(".tmpquery", paste("AC=", accession)), silent = TRUE)
   if (!(inherits(resquery, "try-error")))                      #if assession found
     {
       queryname <- "query2"                                    #give the query a name
       thequery <- paste("AC=",accession,sep="")       #define the search criteria in the correct form
       query2  = query(`queryname`,`thequery`)         #search and save
       # see if a sequence was retrieved:
       seq <- getSequence(query2$req[[1]])             #get the sequence
       closebank()
       return(seq)
     }
     closebank()
   }
   print(paste("ERROR: accession",accession,"was not found"))   #error if not found in any database
 }
```

Once you have copied and pasted the function getncbiseq() into R, you can use it to retrieve a sequence from the NCBI Nucleotide database, such as the sequence for the DEN-1 Dengue virus (accession NC_001477):

```
> dengueseq  = getncbiseq("NC_001477");        #This may take a minute!
```

The variable dengueseq is now a vector containing the nucleotide sequence (print to screen to see!). Each element of the vector contains one nucleotide of the sequence. Therefore, to print out a certain subsequence of the sequence, just like any vector. *Why do we used square brackets here?*

```
> dengueseq[1:50];
```

Writing sequence data out as a FASTA file
You can write out a sequence to a FASTA-format file in R by using the "write.fasta()" function from the SeqinR R package. To complete a FASTA entry, you will need 1) the name you wish to give the function (what follows the >), 2) the sequence, and 3) the file to export it too.

```
> write.fasta(names="DEN-1", sequences=dengueseq, file.out="YOUR_NID_HERE.fasta");
```

If your pwd is currently the shared folder, you won't be about to write the file. *How do you check? How do you change this?*

Reading sequence data into R
Using the SeqinR package in R, you can easily read a DNA sequence from a FASTA file into R.

You can read this FASTA format file into R using the read.fasta() function from the SeqinR R package:

```
> dengue = read.fasta(file = "YOUR_NID_HERE.fasta");
>dengue;
```

Note that R expects the files that you read in to be in the working directory, so if you stored your file somewhere else, you will have to either move your working directory or route to the file (i.e. relative or absolute path to file).

When we read out the new variable, dengue, we see that it has the sequence, but also some "attr"s. These are attributes of the sequence (which is an "object"). In this case, it has a name (which we gave it when we created the file), an annotation (first line of the sequence), and a class (type of data). We will talk more about objects shortly.

Length of a DNA sequence
Once you have retrieved a DNA sequence (dengueseq), we can obtain some simple statistics to describe that sequence, such as the sequence's total length in nucleotides. In the above example, we retrieved the DEN-1 Dengue virus genome sequence, and stored it in the vector variable dengueseq.

*How would you output the length of the sequence in dengueseq (**store the output as seqlength**)?*

Base composition of a DNA sequence
An easy first analysis of any DNA sequence is to count the number of occurrences of the four different nucleotides ("A", "C", "G", and "T") in the sequence. This can be done using the the table() function. For

example, to find the number of As, Cs, Gs, and Ts in the DEN-1 Dengue virus sequence (which you have put into vector variable dengueseq, using the commands above), you would type:

```
> table(dengueseq);
```

Alternatively, you can find the value of the element of the table in column "g" by typing:

```
> table(dengueseq)["g"];
```

GC Content of DNA

How would you manually calculate GC content given the above information? Can you do it using variables instead of the raw numbers? ***Store this value as GCcontent.***

Can you find a function in the seqinr package that would do this for you?

DNA words

As well as the frequency of each of the individual nucleotides ("A", "G", "T", "C") in a DNA sequence, it is also interesting to know the frequency of longer DNA "words". The individual nucleotides are DNA words that are 1 nucleotide long, but we may also want to find out the frequency of DNA words that are 2 nucleotides long (ie. "AA", "AG", "AC", "AT", "CA", "CG", "CC", "CT", "GA", "GG", "GC", "GT", "TA", "TG", "TC", and "TT"), 3 nucleotides long (eg. "AAA", "AAT", "ACG", etc.), 4 nucleotides long, etc.

We already have the breakdown of 1 letter words in the genes ("a", "t", "c", "g"). We can also perform this with the count function. *Use the help options to* ***count*** *the two letter words in the gene.* ***Save the output as Count2words.***

Over-represented and under-represented DNA words

It is interesting to identify DNA words that are two nucleotides long ("dinucleotides", ie. "AT", "AC", etc.) that are over-represented or under-represented in a DNA sequence. If a particular DNA word is over-represented in a sequence, it means that it occurs many more times in the sequence than you would have expected by chance. Similarly, if a particular DNA word is under-represented in a sequence, it means it occurs far fewer times in the sequence than you would have expected.

A statistic called ρ (**Rho**) is used to measure how over- or under-represented a particular DNA word is. For a 2-nucleotide (dinucleotide) DNA word ρ is calculated as:

$\rho(xy) = fxy/(fx*fy)$

where fxy and fx are the frequencies (percent occurrence) of the DNA words xy and x in the DNA sequence under study. For example, the value of ρ for the DNA word "TA" can be calculated as:

$\rho(TA) = fTA/(fT* fA)$

where fTA, fT and fA are the frequencies of the DNA words "TA", "T" and "A" in the DNA sequence.

The idea behind the ρ statistic is that, if a DNA sequence had a frequency fx of a 1-nucleotide DNA word x, and a frequency fy of a 1-nucleotide DNA word y, then we expect the frequency of the 2-nucleotide DNA word xy to be fx* fy. That is, the frequencies of the 2-nucleotide DNA words in a sequence are expected to be equal the products of the specific frequencies of the two nucleotides that compose them. If this were true, then ρ would be equal to 1. If we find that ρ is much greater than 1 for a particular 2-nucleotide word in a sequence, it indicates that that 2-nucleotide word is much more common in that sequence than expected (ie. it is over-represented).

A) Output the number of occurrences of each 1-nucleotide word.
B) Calculate the frequency of G (define as fG) and C (define as fC).
C) Output the number of occurrences of each 2-nucleotide word.
D) Calculate the frequency of GC (define as fGC).
*E) Calculate rho for "GC" and save as **pGC**.*

*Repeat this for rho for "CG" and save as **pCG**.*

Can you find a function that would calculate this for you (do it by hand first to practice R syntax)?

Note that if the ratio of the observed to expected frequency of a particular DNA word is very low or very high, then we would suspect that there is a statistical under-representation or over-representation of that DNA word. However, to be sure that this over- or under-representation is statistically significant, we would need to do a statistical test.

In this case, we can use a **zscore** (standard score, a measure of the deviation from mean). See https://en.wikipedia.org/wiki/Standard_score if you are not familiar with zscores. Basically, you are looking at the relationship between an observed value of a "word" occurring versus the expected value of a word occurring if you shuffled the sequence randomly (null). *Using the command **zscore**, are the "GC" or "CG" words significantly over- or underrepresented?*

Required: Paste the values to the following variables into a document called HW6.txt in your HW6 folder of your dropbox.

seqlength **pGC**
GCcontent **pCG**
Count2words

Functions and Graphing in R

Introduction to R continued – writing custom scripts

Running R as an Executable Script

R can be run on the command line in 'interactive' mode simply by calling the program 'R'. However, we can also make executable programs in R, just like we did in unix. In fact, the process is very similar.

In nano (or whatever text editor you prefer), you can write lines of R code:

```
cat ("This is R!\n");
```

Ok, that was simple. But remember, we have to tell the computer that this is R code – with the #!. What goes after the #!? The output from which R:

```
which R
```

Now, my executable script will look like this:

```
#!/afs/nd.edu/coursesp.15/cse/cse60132.01/Shared/bin/Rscript
cat ("This is R!\n");
```

You can load this into R or RStudio via copy and paste (frequently used while troubleshooting) or with:

```
source("test.R");
```

We saw this with the function I reposted for you last week after the trials and tribulations of Thursday's activity. One use of these executables is to have a file, much like your .bashrc, that will load your commonly used libraries, run code to the point you were at yesterday, etc.

Best Practices in R

We will be going over writing programs in R, rather than just using basic commands. We will write functions and make them executable.

One rule to try to follow is that a function should encapsulate a single "idea" - it shouldn't be too long, or try to do too much – just like unix programs. Further, it should only assume the existence of data given to it as a parameter ("option") - don't refer to any outside variables; they may not exist! You should be able to copy and paste an entire function to use in another program.

Make sure to write comments to document what functions do, what parameters are, and what they return. Basically, everything you rely on in the ?command information should be commented. Comment lines start with # in R, just like in unix.

Functions in R

Ok, now that we know some basic rules, let's talk about how to write our own functions. First, note that function and variable names can have a . as part of the name in R; in most languages, . has other meanings. I wouldn't get used to naming functions with .'s in them.

R functions can take a number of required parameters ("options") and/or a number of optional parameters. When declaring your own function, optional parameters (like sep, or header), you must specify default values for them if they are not given (not followed by an =). Required parameters should be specified first.

```
myFunction <- function(reqParam1, reqParam2, optParam1 = 0, optParam2 = FALSE)
        {
                thingToReturn = reqParam1 * reqParam2 + optParam1; #code goes here
                return(thingToReturn);
        }
```

Note that I used "<-". This is required for function definition.

Now our function, myFunction, can be called in several different ways (even if all the parameters aren't used):

```
result = myFunction(5, 6, 2, TRUE);
result = myFunction(5,6);
result = myFunction(5,6, optParam2 = TRUE);
```

This sort of calling allows us to give the required parameters, and then give non-default values for some parameters while accepting the defaults for others. This is common in R, as many built-in functions have lots of optional parameters. We can even just see that in our read.table function.

if Statements and for loops

We saw if statements in unix – mostly used in awk in our case. Conveniently, an if loop is similar across all languages, because it is a basic logical function. The major differences are typically in syntax:

```
toyFunction <- function(var)
{
    if (var < 10) {
       cat("var is less than 10\n");
    } else if (var == 10) {
       cat("var is 10\n")
    } else {
       cat ("var is greater than 10\n");
```

```
    }
```
}**Note that the "else if" and "else" are on the same line as the closing bracket**, but not contained within the preceding clause. Also, else if can be a number of different variations in different languages (elseif, else if, elsif, etc.).

We can make this if loop a function so we don't have to keep typing it:

```
toyFunction <- function(var)
{
    if (var < 10) {
       cat("var is less than 10\n");
    } else if (var == 10) {
       cat("var is 10\n")
    } else {
       cat ("var is greater than 10\n");
    }
}
```

We can also add ifs to our previous function:

```
myFunction <- function(reqParam1, reqParam2, optParam1 = 0, optParam2 = FALSE)
        {
                thingToReturn = reqParam1 * reqParam2 + optParam1; #code goes here
                if (optParam2 == TRUE) {
                        thingToReturn = thingToReturn * (-2);
                } else {
                        thingToReturn = thingToReturn * (-1);
                }
                return(thingToReturn);
        }
```

For loops are new vocabulary, but for the most part, you've already used them! For loops allow us to, say, do something for each element in a vector… or each line of a file. Sound familiar? This is how awk works - it does something for each line - it's pretty much like a huge for loop.

```
vector = c(0,1,2,3,4);
for (element in vector) {
   cat("Element is",element,"\n");
}
```

Computing the stdev of each column in dataframe:

PSEUDOCODE:

```
for X in range, use i to denote point in range
        column i = column that matches point in range
        mean column i = calculate mean for that column
        print to screen
end
```

```
table=read.table(file="hmmR.txt", header=TRUE, row.names=1);

for (i in 1:ncol(mydata)) {
        coli = mydata[,i];
        sdcoli = sd(coli);
        cat(sdcoli, "\t");
}
```

Some Graphing Functions

Graphing is a pretty convenient use for R, especially in Rstudio. plot() is the most generalized graphing function. If you give it all numeric data (numeric vs. numeric) it'll do a scatter plot. You can tell it different plot types with type = "".

```
plot(seq(0,10,1), seq(0,10,1));
plot(seq(0,10,1), seq(0,10,1), type = "l");
plot(seq(0,10,1), seq(0,10,1), type = "p");
plot(seq(0,10,1), seq(0,10,1), type = "b");
```

A few other useful parameters are:

main="The Title"
xlim=c(1,20)
ylim=c(1,20)
col="darkblue"

```
plot(seq(0,10,1), seq(0,10,1), main = "the Title", xlim=c(1,20), ylim=c(1,20), col="darkblue");
```

One additional useful function for graphing is abline. This function allows you to give R a function to plot, so you can add a regression line to a plot.

```
abline(a=0, b=1, col="blue");
```

We can also use explicit plotting functions like hist.

```
table=read.table(file="hmmR.txt", header=TRUE, row.names=1);
hist(table$bitscore);
hist(table$bitscore, breaks=seq(0,350,1));
hist(table$bitscore, breaks=seq(0,350,10));
```

Writing to file

To direct output to a file, you use the sink function (opposite of source)!

```
sink("myfile", append=FALSE, split=FALSE)
sink()   # return output to the terminal
```

sink( ) will not redirect graphic output. To redirect graphic output use one of the following functions. Use dev.off( ) to return output to the terminal.

Function	Output to
pdf("mygraph.pdf")	pdf file
win.metafile("mygraph.wmf")	windows metafile
png("mygraph.png")	png file
jpeg("mygraph.jpg")	jpeg file
bmp("mygraph.bmp")	bmp file
postscript("mygraph.ps")	postscript file

Use a full path in the file name to save the graph outside of the current working directory.

```
# example - output graph to jpeg file
pdf("myplots.pdf")
table=read.table(file="hmmR.txt", header=TRUE, row.names=1);
hist(table$bitscore);
hist(table$bitscore, breaks=seq(0,350,1));
hist(table$bitscore, breaks=seq(0,350,10));
dev.off()
```

If there are multiple graphs, they will each get their own page.

Useful Statistical Functions

Some useful functions you may want, that we will not really go into – but it helps to know their names!

lm() - used to fit linear models
aov(deptvar~indept1*indept2*indept3) - ANOVA (use + signs if you don't want interaction of factors).
t.test() - t-test
hclust() - hierarchical clustering

And that is the basics of R – I included some things below that may be of use to you, such as examples of loops, regex (yikes!), and

BONUS: Regex – if you dare!

http://www.regular-expressions.info/rlanguage.html
https://stat.ethz.ch/R-manual/R-devel/library/base/html/regex.html

Checkpoint: Bioinformatics in R II (In-class/HW7)

A. Make an executable that will load your libraries for you! (Optional but useful!)

B. A sliding window analysis of GC content

First, you will have to get the sequence again:

```
dengueseq  = getncbiseq("NC_001477");
```

In order to study local variation in GC content within a genome sequence, we could calculate the GC content for small chunks of the genome sequence. The DEN-1 Dengue virus genome sequence is 10735 nucleotides long. To study variation in GC content within the genome sequence, we could calculate the GC content of chunks of the DEN-1 Dengue virus genome, for example, for each 2000-nucleotide chunk of the genome sequence:

```
GC(dengueseq[1:2000]);     # Calculate the GC content of nucleotides 1-2000 of the Dengue genome
GC(dengueseq[2001:4000]);  # Calculate the GC content of nucleotides 2001-4000 of the Dengue genome
```

Calculate the rest of the windows manually to get practice using ranges of vectors. How do you know when to stop? Do we know a function to identify the length of a sequence?

From the output of the above calculations, we see that the region of the DEN-1 Dengue virus genome from nucleotides 1-2000 has a GC content of 46.5%, while the region of the Dengue genome from nucleotides 2001-4000 has a GC content of about 45.3%. Thus, there seems to be some local variation in GC content within the Dengue genome sequence.

Instead of typing in the commands above to tell R to calculate the GC content for each 2000-nucleotide chunk of the DEN-1 Dengue genome, we can use a **for loop** to carry out the same calculations, but by typing far fewer commands. That is, we can use a for loop to take each 2000-nucleotide chunk of the DEN-1 Dengue virus genome, and to calculate the GC content of each 2000-nucleotide chunk.

First let's make a vector that contains all the starting points we wish to use. We could do this by hand (i.e.: c(1, 2001, 4001, etc)) but that isn't very cs savvy! Let's instead use the seq command.

*Make a variable, named "**starts**", that starts at 1, goes through (the length of dengueseq – 2000), and increments by 2000. Why the -2000? Because these are START points ^_~.*

Make a variable, named "n", that contains the length of starts.

We're going to make a for loop to calculate this for us. Let's start with the part that is easiest to troubleshoot – the code we want to run. We want this code to calculate the GC content for each chunk of the sequence. *How would we do that for the first chunk – 1:2000?*

Your answer probably looks similar to the code from above:
```
GC(dengueseq[1:2000]);     # Calculate the GC content of nucleotides 1-2000 of the Dengue genome
```

If you did the next chunk, you'd also get something similar:
```
GC(dengueseq[2001:4000]);  # Calculate the GC content of nucleotides 2001-4000 of the Dengue genome
```

If you look at your starts vector, you will see that the first two elements (starts[1] and starts[2] are the beginning of each of those commands above… so we could really do:

```
GC(dengueseq[starts[1]:2000]);
GC(dengueseq[starts[2]:4000]);
```

The second part of the call also increases in a logical manner…

```
GC(dengueseq[starts[1]:(starts[1]+1999)];
GC(dengueseq[starts[2]:(starts[2]+1999)];
```

Note that you can use a variable for the beginning and end of ranges. This is saying calculate the GC content of the subset of dengueseq that starts at starts[1], which the computer reads as 1, and ends at (start[1] + 1999), which the computer reads as 1+1999.

So there is a pattern we could use *for each* calculation. We want to calculate the above code for each i in the range of 1 through the end of the starts vector (which we called n!).

Ok, now for the fun part – let's make a for loop that runs **for each start point**. For loops have a structure we talked about last time:

```
for (i in some_range) {        #gives the conditions for the continued looping of the code
   #code that runs each time goes here!
   #usually want to print or return something!
}
```

Complete this for loop:

```
for (i in some_range) {        #"some_range" here needs to be 1 through the end of the starts vector!
   content =                   #code that calculates the GC content for each start point i
   print(content);              #print our findings!
}
```

The above analysis of local variation in GC content is what is known as a sliding window analysis of GC content. By calculating the GC content in each 2000-nucleotide chunk of the Dengue virus genome, you are effectively sliding a 2000-nucleotide window along the DNA sequence from start to end, and calculating the GC content in each non-overlapping window (chunk of DNA).

Note that this sliding window analysis of GC content is a slightly simplified version of the method usually carried out by bioinformaticians. In this simplified version, we have calculated the GC content in non-overlapping windows along a DNA sequence. However, it is more usual to calculate GC content in overlapping windows along a sequence, although that makes the code slightly more complicated.

C. A sliding window plot of GC content

It is common to use the data generated from a sliding window analysis to create a sliding window plot of GC content. To create a sliding window plot of GC content, you plot the local GC content in each window of the genome, versus the nucleotide position of the start of each window. We can create a sliding window plot of GC content by typing:

```
#Stuff we already used!
starts = seq(1, length(dengueseq)-2000, by = 2000);
n = length(starts);    # Find the length of the vector "starts"

#We need to make a vector with the same number of elements as starts (which will be our x) to
hold our GC content (which will be our y).  Check ?vector to see what this "mode" option is!
chunkGCs = vector(mode = "numeric", length=length(starts));

#Our loop from before, with one change!  This is non-working code – use your loop!
for (i in some_range) {         #some range here would be 1 through the end of the starts vector!
   content =                    #code that calculates the GC content for each start point i
   print(content);               #usually want to print something!
   chunksGCs[i] = content;    #stores our GC in a vector!
}

#Plot
```

Using plot(), write a line of code that plots starts verus chunkGCs, that shows both points and lines, has an x label of "Nucleotide start position", and a y label of "GC content"

D. Making a flexible function

You may want to use the code above to create sliding window plots of GC content of different species' genomes, using different windowsizes. Therefore, it makes sense to write a function to do the sliding window plot, that can take the windowsize that the user wants to use and the sequence that the user wants to study as arguments (inputs).

Let's review how functions look:

```
NAME <- function(reqparam1, reqparam2, optparam)
{
#CODE
}
```

We the code already, we just need to tweak it very slightly – just like we did when we went from troubleshooting the for loop code.

```
slidingwindowplot <- function(windowsize, inputseq)
{
   # all code from C goes here – with change described below!
}
```

Alter your code so that instead of using "dengueseq" it uses "inputseq", and instead of hard coding 2000 in as our window size, use "windowsize"). I recommend doing this in a text editor first. Beware of quotation weirdness.

This function will make a sliding window plot of GC content for a particular input sequence specified by the user, using a particular window size specified by the user. Once you have typed in this function once, you can use it again and again to make sliding window plots of GC contents for different input DNA sequences, with different window sizes. For example, you could create two different sliding window plots of the DEN-1 Dengue virus genome sequence, using window sizes of 3000 and 300 nucleotides, respectively:

```
slidingwindowplot(3000, dengueseq);
slidingwindowplot(300, dengueseq);
```

NOTE: It is very typical to build code this way – starting with a finite case, then building up to a more flexible function. We build the code we wanted to run in the loop, then made it flexible with changing the hardcoded values to the ones defined in the for loop. We then made the graphing code using a finite window size and an example sequence, then made the function more flexible with changing the hardcoded sequence name and window size to flexible variables. It may seem slow, but it's MUCH easier to troubleshoot and approach than starting with "gee, I'm going to write a function to make sliding window plots"!!!!!

E. Python Pre-activity
Please complete the **first four sections** of the Python Tutorials on CodeAcademy.com (free and brilliant site). These are the Python Syntax, Tip Calculator, Strings and Console Output, and Date and Time activities. They are not long, and are very straight forward.

Required in your HW7 folder:
slidingwindowplot.R - contains your complete function (6 points)
Plot from slidingwindowplot(300, dengueseq) (2 points)
Screenshot of your codeacademy progress bar! (2 points)

Unit 5: Introduction to Python

Basic Python

Introduction to Python – a third programming language!

Programming concept:
There are obviously many different programming languages, and they all do things a little differently. With that in mind, don't get too hung up on syntax, you'll see that the ideas and constructs will be the same (they all have variables, loops, functions), but how we represent them in our code will be different. Unix wanted everything to be a string, R wanted everything packaged in brackets, and now Python wants to be as close to English as possible (including a kind of paragraph style)!

Python was designed to be approachable (which is why I'm teaching it) and as a result Python can be a little lazy sometimes, so a lot of times your python lines will look way more simplistic (by design) than in another language.

For example, *the parenthesis in the if statement for Python are not necessary*, but I recommend using them for consistency (and ease of transition for other languages). You'll also notice the semi-colon is optional again - in other languages semi-colons after single statements (so not an if statement or a loop, etc.) are necessary. *In Python they aren't*. You can put them in, but the interpreter doesn't need them. I again recommend trying to use them, since it will help you in the future, but don't worry about them too much.

So what does all this mean in practice? Let's look at a brief example of differences in R and Python:

R:
```
if (age < 18) {
        cat("You're young!")  #This would run even if the line was not indented!
}
```

Python:
```
if age < 18:                        #Can also be if (age < 18):
        print "You're young!"       #This code will crash if the line isn't indented.
```

Let's look at that syntax a bit more… The if statement is familiar, but we have a colon at the end (this is necessary – and obnoxious), and then the next block of text is indented. In most languages, even awk, we define blocks of code (what falls under the if statement in this case) by curly brackets. Python is much like unix in that it is heavily white space dependent – but mostly in the case of indenting. And, unlike the semi-colon and the parenthesis, you can't just use curly braces and the interpreter will ignore them - they'll break your code. So just keep that in mind!

For example:

```
if age < 18:
   print “You’re young!”
   print “You cannot vote!”
```

is different than:

```
if age < 18:
   print “You’re young!”
print “You cannot vote!”
```

The first code will only print “You cannot vote” if you are under 18… the second block will print the line no matter what the age, because it is not inside the if statement. How do you know? By the indent!

Take home: Python wants to be as close to English as possible and relies on indenting to structure code. Most other things (like semi colons and parentheses) are more often suggestions.

Running Python

There are two ways we can run Python - making it executable or calling it with the interpreter. Just like before, we can make a file that contains python code, and make it executable with chmod, as long as it has the hash-bang line at the top:

```
#!/usr/bin/python
```

So, a very basic Python script would look like:

```
#!/usr/bin/python
print “Hello world!”
```

(Hello world is the classic first program everyone writes in a new language).

The print command prints to stdout, meaning what? That we can use this output and pipe it to other programs! Print, by default, appends a newline (\n) to the end of every print statement.

Alternatively, we can simply run a .py file with:

```
python <pythonscriptname>
```

And bam, it runs.

The unfortunate thing with programming languages (and one reason I love Codecademy) is that you cannot see the output and errors immediately like you do in R. One way we are going to get around that in class is to use: http://repl.it/languages/Python.

Print to Terminal
Printing in Python is amazingly easy:

```
print “Hello Class!”;
print(“Hello Class!”);
```

Commands have similar structure as they did in R, and to a degree unix. The command leads, usually followed by parentheses containing information needed by the command (options and files and such).

```
Bash:     echo “Hello Class!”
R:        cat(“Hello Class!”);
Python:   print “Hello Class!”     #or
          print(“Hello Class!”);
```

Variables
Just like unix and R, Python has variables, which hold data. In Python there are different types of variables:

1) Scalars: variables that hold strings or numbers.
2) Lists: just like vectors in R – hold ordered sets of scalars.
3) Dictionaries (commonly known as hashes/hash tables): an unordered list of variables, indexed by strings or numbers. Dictionaries have to have a scalar for a key rather than just the position number, but they can be different kinds of scalars

Declaring Variables
Oftentimes, we need to set a base value for a variable (like 0 for the start of a sum, or an empty list to add things to), and this is how we declare a variable for use. Scalars are straightforward - we just assign a value and get going:

```
sum = 0;                # if the term is naked, it tells Python it’s a number or a variable (if letters)
my_dog = "Elinora";     #”” tells Python this is a string
```

Declaring lists and dictionaries, we either need to make empty versions of them (otherwise how will our program know what they are?). Notice that lists(vectors) use square brackets – just like they did in R. This is very very common practice for referencing vectors (or whatever the language decides to call them). Dictionaries use curly brackets. Python, just like R, wants to make assumptions about your data type right away and will decide what you can do with that data type in response!

```
my_list = [];              #[] tells Python this is a list/vector
my_dict = {};              #{} tells Python this is a dictionary
print my_list;
print my_dict;
```

Or we need to make versions that already are somewhat populated:

```
my_critters = ["Elinora", "Metzi", "Momo"];
my_dict = {"a":3, "b":20, "c":49};
print my_list;
print my_dict;
```

Notice the terminology for dictionaries - the first value is the 'key', then there's a colon, then another value which is the 'value' of the key. When declaring a filled list or dictionary, use commas between values. After it has been declared, objects can still be added.

If you want to make a matrix in Python, you will have to use a list of lists:

```
row1 = [0,1,2,3,4];
row2 = [5,6,7,8,9];

matrix = [row1, row2];
print matrix;
```

For this reason, R is usually more popular for working with matrices.

Modifying Variables

Once again, scalars are simplistic - simply assign a new value, and you're good to go.

For both lists and dictionaries, we can once again do two things: modify an existing value, or add a new one. **Note:** Comments are the same in all three languages! Also, "" won't work. They have to be vertical just like in R! Also, note that **Python is 0-indexed**!

<u>For lists:</u>

Modify an existing value (index MUST already exist):

```
my_list = ["Ellinora", "Metzi", "Momo"];          #[] = list!
my_list[0] = "Ellie";                             #list[index] = item at that location in the list
print my_list    #["Ellie", "Metzi", "Momo"];
```

Add a new value at index x:

```
my_list = ["Ellie", "Metzi", "Momo"];
my_list.insert(1, "JanitorFish");                 #list.insert(index, value);
print my_list                                     #["Ellie", "JanitorFish", "Metzi", "Momo"];
```

Add a new value (to the end):

```
my_list = ["Ellie", "Metzi", "Momo"];
my_list.append("JanitorFish");
print my_list                                   #["Ellie", “JanitorFish”, "Metzi", "Momo"];
```

For dictionaries:

Modify an existing value:

```
my_dict =  {"a":3, "b":20, "c":49};             #~”a” means 3; “b” means 20;
my_dict["a"] = 18;                              #dictionary[key] = returns “definition”
print my_dict  #{“a”:18, “b”:20, “c”:49};
```

Add a new value:

```
my_dict = {"a":3, "b":20, "c":49};
my_dict["r"] = 18;                              #add new “definition” to “key”
print my_dict  #{"a":3, "b":20, "c":49, "r", 18};
```

As you can see, it’s the same for both in a dictionary. Keep in mind, also, that dictionaries are unordered so there’s no sense in saying ‘add this at index x,’ or ‘add this at the end of my_dict’ because there is no index number or “end”.

Accessing Variables

Once again, scalars are easy - simply use their variable name without quotes and we’re good to go! There are some cool things we can do with strings, but I’ll cover that after we go over basic access.

Getting at a list value and a dictionary value are pretty similar.

```
print my_list[0];        # This is almost identical to R
```

This gives us the item at index 0 in our list. If there’s no element (that is, you’ve gone past the end of the array), you’ll get an index out of bound error. We can also get a subset of a list quite easily:

```
print my_list[0:2];      # This is almost identical to R
```

This says I want the 0th element to the 2nd element (**non-inclusive on the end**). So this would give us element 0 and element 1 in another list. You can only get a single range doing this (so you can’t do my_list[0:2:3:4] to get 0, 1, and 3).

Two things to note

1) indexes start with 0:

```
my_list = [1, 2, 3, 4, 5]          #index :    0 , 1, 2, 3, 4!
```

2) ranges are not inclusive to the end:

```
print range(0,3)                #0,1,2 are used!
```

For dictionaries:

```
print my_dict["a"];
```

This gives us the item with a key “a” from our dictionary. If there’s no element with that key we get a key error. Notice that “a” is in quotes, because this is a string. If the dictionary translated numbers to other numbers, you could do my_dict[0].

```
my_dict= {
   0 : 3,
   4 : 6
   };

print my_dict;
print my_dict[0];       #Note, you can only look up by key – just like a real dictionary!
```

Working with Strings

Strings are a really big data type you’ll be using because, well, pretty much everything at its core is a string, especially in unix. When we read stuff in, whether from stdin or a file, it will be a string. So, what are some things we might want to do?

Well, one of the simplest things is we might want to stick two strings together. **Note:** just like lists and dictionaries, strings have a special character wrapped around them, in this case “’s.

```
a = "a";
b = "b";
c = a + b;
print c;
```

This makes c = ab! We can do this with any combination, even setting a = a + b, overwriting the value of a with “ab.” Notice that if we wanted a space in between “a” and “b” we’d either have to put it on the end of a, the start of b… or we could just add it in ourselves like so!

```
c = a + " " + b;
print c;
```

And this gets the job done. We can concatenate as many strings together as we want! **Note:** I said we can concatenate as many *strings* together as we want. You cannot concatenate an integer (or a float) to a string, you’ll have to cast it as a string, something else I’ll cover a little later. **Another exciting note:** We can concatenate lists this way too, but not dictionaries because there is no guide to order!

Another thing that is super useful is being able to look at substrings of a string, including a single character. This is really useful if we want to look at the first character of a line, or the end of a line, or a certain loci in a string, or maybe even move through a few characters at a time (like a sliding window)! Python allows us to access strings like lists/vectors – making this much easier than many other languages! This means that something like this:

```
print my_string[0];              #character at the first location of my_string
```

Gives us the first character of my_string and something like:

```
print my_string[0:3];            #characters at the first – third location of my_string
```

Gives us the first 3 characters of my_string. Sweet! If we had an alignment, we could pull the same loci from each sample, without a special package like seqinr.

Also a kind of cool feature is that we can use negative indices to access a list backwards - so my_list[-1] gives me the last element of my_list, and so forth; rather than doing something along the messy lines of my_list[len(my_list)-1] in R.

Useful String Functions
Let's get a quick introduction to a few important string functions:

len() - this gives you the length of a given string (or list) - ex. string_length = len(my_string);
strip() - this removes whitespace (including new lines) from the ends of strings - ex. new_string = my_string.strip(); If you put a string inside the parenthesis it will remove trailing instances of these (similar to split below).
split() - splits a string into a list based on the delimiter you provide. If no delimiter is provided the default is whitespace - ex. my_list = my_string.split(","); Useful when pulling in tables.
join() - joins a list into a string with the given separator - ex. " ".join(my_list); joins all the values in my_list into a string separated by spaces (the " ")

Control Structures/Logic (ifs, elses, fors, and whiles)
We've already been introduced to the concept of control structures (like the if/else statements and for loops), so a lot of this is syntax differences (surprise), but I'll also teach you guys about a new one as well (while loops).

If and else statements are also pretty much the same, except we use elif instead of elseif and we have : instead of {}. **Note:** there is no closing brackets, or ends, or anything. This is a bit weird, but anything at the same level of indentation is considered part of the same loop.

```
x = 5
if (x >= 10):
        print "Greater than or equal to 10!";
elif (x <= 4):
        print "Less than or equal to 4!";
else:
        print "Between 4 and 10";
        print "When does this print?";
```

Again, this code is different:

```
if (x >= 10):
        print "Greater than or equal to 10!";
elif (x <= 4):
        print "Less than or equal to 4!";
else:
        print "Between 4 and 10";
print "How many times does this print?";
```

For loops are pretty much the same, but for some reason cannot have the condition in parentheses:

```
for value in my_list:    # think about reading this as “for (each) value in my_list:”
        print value;
```

We can also do this for a range:

```
for value in range(1, 30, 2):    # will not include 30!
        print value;

for value in range(3):           # 0,1,2 – defaults start at 0 and increment by 1
        print value;
```

While loops are the third major control structure (if and for were the other two). Can anyone guess how they work? They execute as long as the conditional is true (so like the x >= 10 in if).

```
count = 0;
while (count < 10):
        print count;
        count = count + 1;
```

This is exactly the same as:

```
for count in range(0,10):
        print count;
```

A Note on Looping through Dictionaries
Dictionaries aren't really sorted, so we can't loop through them like we do a list. Let's say I have a dict that has strings as keys, and lists as values:

```
my_dict = {"seq1":["a","c","t"], "seq2":["a","t", "t"]};
print my_dict.keys()
```

To access the first base in seq1:

```
print my_dict["seq1"][0]  #for the key "seq1", the first index of the value.
```

This also works on our "matrix":

```
for row in matrix:
        print row;
print matrix[1][0]              #instead of print(matrix[2,1]) in R
```

We can also check to see if a key is in our dictionary, we can use the keyword 'in':

```
if "seq1" in my_dict:
        print "FOUND"
```

To see if something is not in a dictionary, we can do:

```
if "seq3" not in my_dict:
        print "NOT FOUND"
```

Libraries (also called Packages in R)
Libraries are just like what we talked about in R - they're a package of code that someone else has written that you can take advantage of. One of the main ones you might be interested in would be BioPython, which has a lot of functions for dealing with bio filetypes (like BLAST output!), and other things. Unlike R we don't just tell the program to download a library - we have to go out ourselves and get it, which I'll show you guys later.

There are also default libraries that come with Python, but aren't automatically loaded (just like R). So we need to load them, and then we can use them.

We load libraries at the beginning of our code after our hashbang with the import call:

```
import sys;
```

This imports the sys package, which has all the stuff we want to read from stdin!

You can get information on what the function requires by using:

```
dir(function);
```

or google the documentation: https://docs.python.org/

I/O (Input/Output)
Input and output are obviously very important to us, right? Hardcoding (that means, putting constants in your data, which we'll discuss in a sec) isn't really good, because that means every single time you want to run it, you might have to go in and change the code.

So, as we know, there are two ways we can get input - we can either do stdin by catting a file or piping info into the program, or we can give the program a filename (so cat this.txt | grep "cat" vs. grep "cat" this.txt). **Note:** you cannot do this on the website interpreter – you have to be on the console in order to pipe things in!

Let's look at the first. How to we read from stdin with Python? First, we need to use our importing skills and import sys, like we did above. Then we need to tell it to read. Maybe we want to do something to each line… let's say we want to count them (instead of using wc -l).

```
import sys;
count = 0;

for line in sys.stdin:
        count = count + 1;

print count;
```

Sweet! This works like a charm, and we used our nice little for loop to loop over the input. What if I want to prompt the user for input? This is less common, but you might need it.

```
user_input = raw_input("Give me a number please!\n");
print "The number you provided is " + user_input;
```

Cool! Sometimes we want to read directly from a file though.

```
count = 0;
my_in_file = open("mytestfile.txt");
for line in my_in_file:
        count = count + 1;
my_in_file.close();
print count;
```

You'll notice I did two things here - I opened a file, and I closed it. **Closing is pretty important, so remember to do it!**

Outputting to a file is pretty straight-forward, but we need to make sure that we tell python when we open this file that we want it to be writable.

```
my_out_file = open("myoutfile.txt", "w");
my_out_file.write("Some stuff\n");
```

Note that I added a new line after my write - this is because, unlike print, it doesn't automatically add a new line. In this vein, we can also write to stdout the same way (and it won't add the \n automatically):

```
sys.stdout.write("Some stuff\n");
```

BONUS INFO: In that for statement, we can see that it automatically reads a line every loop...but what if we want to read two lines? We can use our while loop to give us this power.

```
line = my_in_file.readline();

while line != "":
        print line;
        line = my_in_file.readline();
        print "The next line is: " + line;
```

At the end of a file in python, line = "" (an empty string). Also be careful when doing this. For loops automatically move you onto the next line, but while loops don't. In this example we'll print every line but the first and last twice (once on their iteration and once after 'The next line is". How would I have every line only printed once (so in a file with lines a b c d I'd print:

a
The next line is: b
c
The next line is: c
)?

```
line = my_in_file.readline();

while line != "":
        print line;
        line = my_in_file.readline();
        print "The next line is: " + line;
        line = my_in_file.readline();
```

Now, how do we do output? We're super cool and already know how to output to stdout - what was it? Print!

Casting

Going back to our counting program, what if I want it to say:

There are this many lines: X

Let's try:

```
print "There are this many lines: " + count;
```

Whoops! It says it can't concatenate a string and an int. So what do we do? Well, we want to do something called casting. Casting says take this object, and try to turn it into this other object. Strings are straight-forward since anything can be a string.

```
print "There are this many lines: " + str(count);
```

And now it works! What if I had a file full of numbers and I wanted to add all of them? We know python automatically reads in everything as a string, so we'd need to cast them as integers.

```
sum = 0;

for line in file:
        sum = sum + int(line);

print "The sum is: " + str(sum);
```

What if I try to cast something as an integer that clearly isn't one? Well, python gets mad.

A Note About Floats

So I mentioned before that an int is really an integer - that is, there is no 2.3, or even a 2.0. There is only 2. Floats are what we want for decimal places - this is where 2.3 and 2.0 (but not 2) exist. I warned about integer arithmetic too, where if we divide 2 by 3, we get… 0. So how do I get around this? A few ways! We can use our newfound casting -

```
print float(2)/3;
```

(As long as one number is a float we're good to go).

We can default one to a float:

```
print 2.0/3;
```

Or we can even just do something a little sneaky:

```
print (2*1.0)/3;
```

Since we've multiplied 2 by a float, 2 becomes a float (and since it's one it doesn't change its value), and then we get our lovely float division.

Mathematical Operators
+, -, *, / - as expected
% - modulus, or the remainder - so num%2 -> 0 if even, 1 if odd
** - power, like before

If we import the math library we can do more cool stuff:
math.log(x,y) - x to log base y (default is e)
math.sqrt(x) - square root of x
math.exp(x) - e**x

Boolean Operators
Things you put in if statements, for loops, etc. Most of these are familiar:

== - equals (for both strings and numbers)
<= - less than or equal
>= - greater than or equal
!= - not equal
> - greater than
< - less than

In other languages we used && and || for and and or. In python, we just use the words!

```
if x > 10 and x < 20:
...

if x > 10 or x < 3:
...
```

Bioinformatics in Python I (In-class/HW8)

Part 1:

One thing that is commonly done upon the receipt of sequences is to trim the adaptor sequences off each sequence. There are several programs that do this, but we are going to write our own!

In the file "seqfile.fa" in the Shared/bin, each sequence starts with the same 14 base pair fragment – a sequencing adapter that needs to be removed.

Goal: Write a program that will

(a) trim this adapter and write the cleaned sequences to a new file
(b) allow users to identify the file and length they wish to remove.

Step 1: read in the file.
Step 2: store the sequence names with the sequences
Step 3: subsample the sequence
Step 4: print the sequence name and sequence to a file
Step 5: make it customizable

Step 1: read in the file:

We will be writing this program in python, using nano (or whatever you wish), on the terminal. Start a file called trimmer.py.

We need to first read in our file from Unix into our program. Let's start this with hardcoding the file name into the program (so it only works on this one file), and we can adjust that later to make it possible to add input.

First we are going to need to load our library that deals with input/output in the system: sys.
Second, we will need to open our file, and close it at the end.

Let's get that working first.

Line 1 – write a line that imports the "sys" library.
Line 2 – write a line that opens "seqfile.fa" and saves it as my_in_file.
Line 3 – write a line that closes my_in_file.

Make this file executable and run it (either with python trimmer.py or include a #! line at the top of the file). Correct any errors – it won't do anything yet, so if it gives you nothing when you run it, you should be golden!

Checkpoint: ./trimmer.py (or python ./trimmer.py) runs without error.

Step 2: store the sequence names with the sequences

Beautiful. Now we can start reading and storing the sequences. We are going to link the name of the sequence with the sequence itself in a dictionary. So eventually we will have a data structure like the following:

```
seq_dict = { ">seq1": ["a","t","c",…],
             ">seq2": ["a","t","c",…],
```

“>seq3”: [“a”,”t”,”c”,…]};

Before we can read lines into the program, we will have to define an empty dictionary to store the sequences. Your program will look like this:

Line 1 –line that imports the “sys” library.
Line 2 –write a line that defines an empty dictionary and call it “seq_dict”.
Line 3 –line that opens “seqfile.fa” and saves it as my_in_file.
Line 4 –line that closes my_in_file.

Now for a bit of a trickiness. We are going to have to read lines in from the fasta file (which has the sequence on a single line) and store them. But we have two lines per sequence, hm…

Let’s first make sure we can read in a line correctly. Add a line to your program to read in a single line another to print it. Your program will look like this:

Line 1 –line that imports the “sys” library.
Line 2 –line that defines an empty dictionary and call it “seq_dict”.
Line 3 –line that opens “seqfile.fa” and saves it as my_in_file.
Line 4 – write a line that defines “line” as a read in line from my_in_file. (see the bonus section in the notes for help!)
Line 5 – print line
Line 6 –line that closes my_in_file.

Checkpoint: python ./trimmer.py should output the first sequence name!

Wonderful, we can read in a line. Since we know the first line is going to be a sequence name, let’s save it in a variable called “name”. We can also read in another line, which we know is going to be the sequence itself, which we will store in a variable called “seq”. These will be added to the seq_dict dictionary, which we can then print to see if we did this correctly. Add the lines as follows:

Line 1 –line that imports the “sys” library.
Line 2 –line that defines an empty dictionary and call it “seq_dict”.
Line 3 –line that opens “seqfile.fa” and saves it as my_in_file.
Line 4 –line that defines “line” as a read in line from my_in_file.
Line 5 – write a line that defines “name” as the current “line”.
Line 6 – write a line that reads in another line and saves it as “seq”.
Line 7 – print name + seq.
Line 8 –line that closes my_in_file.

Checkpoint: python ./trimmer.py should output the first sequence name and sequence!

This is the tricky part – we need to loop through the file to get more than just our first sequence! We don’t know the length of the file (we could look in unix, but that doesn’t help us if the file changes), so we will have to use a while loop. What will our condition be? Well, all files end with “” when read into python! So, we can add the following:

Line 1 –line that imports the “sys” library.

Line 2 –line that defines an empty dictionary and call it “seq_dict”.
Line 3 –line that opens “seqfile.fa” and saves it as my_in_file.
Line 4 –line that defines “line” as a read in line from my_in_file.
Line 5 – while line != “”:
*Line 6 –line that defines “name” as the current “line”.
*Line 7 –line that reads in another line and saves it as “seq”.
*Line 8 – print name + seq.
**Line 9 – line = my_in_file.readline(); #NO INFINITE LOOPS HERE FOLKS!*
Line 10 –line that closes my_in_file.

* - these lines will be indented.

Why is do we need to have another line read in at the end? Well, we had to read in the first line above the while loop so we had a variable called “line” (which is referenced in the while loop – it’s part of the condition). Remember how we had to increase count in our while loop example in the notes? We have to do the same here. The first line is read in, saved as “name”. The next line is read in, saved as “seq”. Then we read in a new line, the loop starts over. It then saves this line as “name”, etc.

Why not read in the line at the beginning of the loop? We would miss the first name (because we read the first line in before the loop begins), and also we want the loop to end when the line “” is read in – which it checks when the loop starts. This is kind of tricky, so try moving the line and seeing what happens if you don’t follow. Experiment away – just save it as a different file name!

Checkpoint: python ./trimmer.py should output the each sequence name and sequence and print it to the terminal!

<u>Step 3: subsample the sequence</u>
Okay, now let’s chop off that adapter! One of the cool things about python is that strings are just like lists/vectors. We can subsample them exactly the same way:

```
string = “testing”;
print string[0:4];          # outputs “tes”
```

So now we can alter line 8 to just print the substring we want – which is the 15th position onward.

Line 1 –line that imports the “sys” library.
Line 2 –line that defines an empty dictionary and call it “seq_dict”.
Line 3 –line that opens “seqfile.fa” and saves it as my_in_file.
Line 4 –line that defines “line” as a read in line from my_in_file.
Line 5 – while line != “”:
*Line 6 –line that defines “name” as the current “line”.
*Line 7 –line that reads in another line and saves it as “seq”.
**Line 8 – print name + seq. ← ALTER THIS LINE*
*Line 9 – line = my_in_file.readline();
Line 10 –line that closes my_in_file.

Checkpoint: python ./trimmer.py should output the each sequence name and the shortened sequence and print it to the terminal! The last sequence should start with “ATGTGAA”.

Step 4: print the sequence name and sequence to a file

Spectacular. Now we just have to write this to a file instead of dumping it to the terminal. We need to open an output file and write to it:

Line 1 –line that imports the "sys" library.
Line 2 –line that defines an empty dictionary and call it "seq_dict".
Line 3 –line that opens "seqfile.fa" and saves it as my_in_file. **Remember to make it writable**!
Line 4 – write a line that opens "seqfile.fa.trim" and saves it as my_output_file.
Line 5 –line that defines "line" as a read in line from my_in_file.
Line 6 – while line != "":
*Line 7 –line that defines "name" as the current "line".
*Line 8 –line that reads in another line and saves it as "seq".
**Line 9 – print name + seq (trimmed!) ←ALTER THIS TO WRITE TO my_output_file RATHER THAN TERMINAL*
*Line 10 – line = my_in_file.readline();
Line 11 –line that closes my_in_file.
Line 12 – write a line that closes my_output_file.

Checkpoint: python ./trimmer.py should return nothing in the terminal, but create a file called seqfile.fa.trim, containing all our sequences.

Step 5: make it customizable

One more step! Let's ask the user to tell us which file and how much they would like trimmed! Remember that input is always a string – even if it looks like a number!!!

Line 1 –line that imports the "sys" library.
Line 2 – write a line that asks the user "Which file would you like to trim? ", save answer as "infile"
Line 3 – write a line that asks the user "How much would you like to trim off? ", save the answer as "trim"
Line 4 –line that defines an empty dictionary and call it "seq_dict".
*Line 5 –line that opens **infile** and saves it as my_in_file. Remember to make it writable!*
Line 6 – write a line that adds ".trim" to the infile string, save it as "outfile"
Line 7 –line that opens outfile and saves it as my_output_file.
Line 8 –line that defines "line" as a read in line from my_in_file.
Line 9 – while line != "":
*Line 10 –line that defines "name" as the current "line".
*Line 11 –line that reads in another line and saves it as "seq".
Line 12 – line that prints name + seq (trimmed!) to **my_output_file ←alter this to subset from **int(trim)** on.*
*Line 13 – line = my_in_file.readline();
Line 14 –line that closes my_in_file.
Line 15 –line that closes my_output_file.

Checkpoint: python ./trimmer.py should ask you for the file name and trim length, then return nothing in the terminal, but create a file called seqfile.fa.trim, containing all our sequences, trimmed as defined.

Part 2: Codecademy
Complete through “A day at the supermarket”.

REQUIRED in HW8 folder:
trimmer.py (2 points for each step, 10 total)
screenshot of progress in Codecademy (2 points)

Less Basic Python – Functions and Classes and Objects

Functions

We now know how to write python code...but what if I want code that's easy to reuse? We can always copy-paste, but that leads to sloppy and bloated code that we don't want. So we'll write a function instead, and then we can call it from anywhere in our code!

Generally, functions should be declared near the top of your code. I normally declare everything up top all together.

We declare a function using the def keyword. Let's do a simplistic example. Let's say I want a function to calculate the GC content of a DNA string I give my function:

Basic concept:

1) Name our function
2) Define our current base to start at 0
3) Define our gc content to start at 0
4) Loop through our string from 0 to end of the DNA string
    5) if the current base is a C or a G
        6) increase gc count
7)Return gc count

In R it looks like this:

```
gc_content <- function(dnaarray)          #Note: this cannot take a string – R doesn't index strings!
{
   currbase = 1;
   gccount = 0;
   while (currbase < length(dnaarray)) {
      if (dnaarray [currbase] == "C" || dnaarray [currbase] == "G") {
         gccount = gccount + 1;
      }
     currbase = currbase +1;
   }

   return(gccount/length(dnaarray));

 }
```

In Python it looks like this (it takes strings!):

```
def gc_content(dnastring):
        currbase = 0;
        gccount = 0;
        while currbase < len(dnastring):
            if dnastring[currbase] == "C" or dnastring[currbase] == "G":
                gccount = gccount + 1;
            currbase = currbase+1;

        return float(gccount)/len(dnastring);
```

Just like in R, we define the input parameters (dnastring) when we define the function, it's just a little different in the actual syntax. It is apparent in both cases that the function call needs to be gc_content(string). The return is what is given back after the function ends. How is return different than print? Return allows you to do more! You can store the output in a variable, you can use it in loops to call the function over and over, whatever you want.

Once we've declared the function, we can call it just like any other function wherever we want in the code, as long as it comes after the definition of the function. So now I can do something like this:

```
mydna = "AGTGCGTATTTCAG";
mygccontent = gc_content(mydna);
```

And mygccountent will be set to the gc content of mydna!

One important thing to remember, simply for code safety's sake - functions should only assume the existence of parameters it's given. You don't want to reference anything that is not defined inside your function!

Also, functions should also not be unnecessarily long - if it's huge, maybe think about splitting it into multiple functions. Having mini modular functions makes it easier to customize and change what you would like the program to do later. People who use your code are very happy with this!

Finally, just like in R, we can "hardcode" optional information into the function - that way there are defined values, but they can be changed:

```
def output_stuff(name, greeting = "Hi"):
        print greeting, name

output_stuff("class");
output_stuff("class", greeting = "Hey");
output_stuff("class", "Hey");
```

Classes

I've sort of used the term 'object' off and on throughout us learning R and Python, and we've used them, but let's actually talk about what is going on under the hood. Don't feel bad if you have a hard time understanding this - this took me a while to pick up when I was first learning (like... years - mainly because we didn't cover this part in my first course)! I'm going to teach you the bare bones of objects and classes, so just be aware there's more stuff out there (like inheritance!) that we're not going to go over. All of this is covered in codecademy (including inheritance)!

Like I said before, an object is an instance of a class of things - like my Rav4 is an instance of a class of things (cars) and my pup Ellie is a class of thing (dog). Can we guess what a class is, then? It's a blueprint for making an object; it defines what an object is and what we can do with an object.

Let's look at a real world example and try to tie that into objects and classes in programming. We'll go back to my car.

I have a car - it has lots of things particular to it:

My car has a color: custom
My car has a make: Toyota
My car has a model: Rav4
And my car has a year: 2008

Let's make a car class. We'll first start by hardcoding information for this one instance into the class.

```
class Car(object):
        def __init__(self):
                self.color = "Custom";
                self.make = "Toyota";
                self.model = "Rav4";
                self.year = "2008";
                self.gear = "P";
                self.brakestate = True;
```

When I make a new object of type Car (we'll go over how to do this in a sec), it will be initialized (see the init?) with all those values set. The init function is called a constructor. It's customary for class names to start with a capital letter and be named with staggered caps, but it's not required.

Let's make a car now. It's as simple as this:

```
my_car = Car();
```

Notice that this doesn't take any parameters, even though _init_ lists (self)? We'll come back to this in a second.

We can now grab bits of information about my_car with "dot notation":

```
print my_car.color;
```

Ok, so let's make this a little more flexible. The init function here tells python what to do when we make a new Car object. We can add parameters to the init call to make the user supply information (just like adding more required variables to R functions changes the required call).

```
class Car(object):
        def __init__(self, color, make, model, year):
                self.color = color;
                self.make = make;
                self.model = model;
                self.year = year;
                self.gear = "P";
                self.brakestate = True;
```

Similar to our function definitions (because all init is is a function), we can do something like this to make model and year optional, by adding defaults:

```
class Car(object):
	def __init__(self, color, make, model="undefined", year="undefined"):
		self.color = color;
		self.make = make;
		self.model = model;
		self.year = year;
		self.gear = "P";
		self.brakestate = True;
```

Let's make an object again. This time we need to give it the required parameters, just like a function!

```
my_car = Car("Custom", "Toyota");
```

Self isn't used the definition... odd. Why is this? It's not terribly important, but it's useful to know. An object is a thing that we create. We want a means of linking all this information, so python links it all to the variable name - which gets defined as "self" as a placeholder in the class definition. Consider this:

```
my_car = Car("Custom", "Toyota", "Rav4", "2008");
print my_car.color;
```

The initialization function takes the information passed to it, and links it to the variable name. This is why dot notation works. Also, it has a second implication in a minute!

Now we've created a car, but cars aren't just existing objects... they do things!! So let's give our car some functions! We will do this just like defining functions before, but now we're doing it inside a class (and therefore indented).

```
class Car(object):
	def __init__(self, color, make, model="undefined", year="undefined"):
		self.color = color;
		self.make = make;
		self.model = model;
		self.year = year;
		self.gear = "P";
		self.brakestate = True;

	def change_gear(self, new_gear):
		self.gear = new_gear;
		print "You've changed gears to " + new_gear + "!";
```

And now when we check to see our gear, we can see that it has actually changed!

```
my_car.change_gear("D");
```

Notice functions that "belong" to the class/object are also referenced with dot notation.

Functions can also lack parameters. Let's change the emergency brake's state. If the brake is on and we want to take it off, and if it's off we want to put it on.

```
class Car(object):
	def __init__(self, color, make, model="undefined", year="undefined"):
		self.color = color;
		self.make = make;
		self.model = model;
		self.year = year;
		self.gear = "P";
		self.brakestate = True;

	def change_gear(self, new_gear):
		self.gear = new_gear;
		print "You've changed gears to " + new_gear + "!";

	def change_brake(self):
		if self.brakestate == True:
			self.brakestate = False;
			print "The brake is now off";
		else:
			self.brakestate = True;
			print "The brake is now on";

my_car = Car("Custom", "Toyota", "Rav4", "2008");

print my_car.brakestate;
my_car.change_brake();
print my_car.brakestate;
my_car.change_brake();
print my_car.brakestate;
```

Voila! We can change the brake. Notice there is still a () after the function. Why? Because the function is still passing "self", you just don't see it. Also, it helps python realize you are refering to a function, not a characteristic, of the object.

You likely won't end up writing many classes in your life, but it is useful to know how they work so that you can read and troubleshoot code easier! You can see what a function is requiring, you can see what the defaults are, you can see what things you can do with an object. All useful information!

Regex

Regex in Python is a little weird, and … kind of annoying in some ways. The symbols will pretty much be the same, so that's nice, but we can't just plop them into the middle of code as we did in unix and as I'm used to in Ruby. Python makes us use another default library, called "re", to handle them. So, first thing is first - import re and make a string to do a match on:

```
import re;
mydna = "AUGUGUGGAGTGCGTATUAGTTCAG";
```

Now, let's look at how to define a regular expression. Let's do a regular expression for A.+A, which is A followed by anything one or more times, followed by an A.

```
my_regex = re.compile("AUG.*UAG");
```

Now to match that regex in a string (let's look for the first match only to start):

```
my_result = my_regex.search(mydna);
```

This returns a match object. We can also do this in one step:

```
my_result = re.search(r"AUG.*UAG", mydna);
```

But here's something that may be a bit surprising… What outputs from:

```
print my_result;
```

Absolute nonsense, because it's a weird object of a regex class. We have to call another function to get it to give us something logical. There are a few functions to call on it:

my_result.group() - the string matched
my_result.start() - the start index of the original string
my_result.end() - the end index of the original string
my_result.span() - the start, end index of the original string

If we want all the matches we can do two things - just get the resulting strings, or get all the match objects (like we get with match):

```
my_result = my_regex.findall(mydna);
print my_result;
my_result = re.findall(r"AUG.*UAG", mydna);
print my_result;
```

This is usually what you need, and Python's nice in that there isn't greediness to deal with – it will give you every match. One day you will realize this saves you a HUGE mental strain (it did for me yesterday).

Python doesn't support the character classes we learned about earlier (like [:punct:]) but there are a few that it does recognize - the ones you were all so desperate to use!

\d - any digit ([0-9])
\D - any non-digit ([^0-9])
\s - any whitespace
\S - any non-whitespace
\w - any alphanumeric (plus _)
\W - any non-alphanumeric (plus not _)

To do a find/replace with Python we use the function sub in re. Let's replace all A\.+A with .!.:

```
my_regex = re.compile("AUG.*UAG");
mydna = "AUGUGUGGAGTGCGTATUAGTTCAG";
new_string = re.sub(my_regex, "cds", mydna);
print new_string;
```

Sub also takes an optional parameter count, which tells it how many occurrences to replace (from left to right).

PPT that accompanies the above notes:

Functions

We now know how to write python code...but what if I want code that's easy to reuse?

- We can always copy-paste, but that leads to sloppy and bloated code.
- We can write a function instead, and then we can call it from anywhere in our code!

Structure of Programs

Generally, functions should be declared near the top of your code. I normally declare everything up top all together.

```
#!/usr/bin/python
#import libraries
#define functions
#define "global" variables
#open input (close it when you are done using it)

#Cooooooode
```

GC content Example

Let's do a simplistic example. Let's say I want a function to calculate the GC content of a DNA string I give my function:

Basic concept:

1) Name our function
2) Define our current base to start of string
3) Define our gc content to start at 0
4) Loop through our string from 0 to end of the DNA string
5) if the current base is a C or a G
6) increase gc count
7) Return gc count after loop completes

GC content Example

Let's do a simplistic example. Let's say I want a function to calculate the GC content of a DNA string I give my function:

Basic concept:

1) Name our function
2) Define our current base to start of string
3) Define our gc content to start at 0
4) Loop through our string from 0 to end of the DNA string
5) if the current base is a C or a G
6) increase gc count
7) Return gc count after loop completes

GC content Example – in R

```
gc_content <- function(dnaarray)        #Note: this cannot take a string – R doesn't index strings!
{
   currbase = 1;     #Starts at 1 because R is 1 indexed!
   gccount = 0;
   while (currbase < length(dnaarray)) {
      if (dnaarray [currbase] == "C" || dnaarray [currbase] == "G") {
         gccount = gccount + 1;
      }
     currbase = currbase +1;   #No infinite loops!
   }

   return(gccount/length(dnaarray));   #return the value after we've looped through it all
```

GC content Example – in R

```
def gc_content(dnastring):   #Python can take a string straight from Unix.
    currbase = 0;            #Starts at 0 because python is 0 indexed!
    gccount = 0;
    while currbase < len(dnastring):
        if dnastring[currbase] == "C" or dnastring[currbase] == "G":
            gccount = gccount + 1;
        currbase = currbase+1;   #No infinite loops!

    return float(gccount)/len(dnastring);   #return the value after we've looped through it all
```

Optional Parameters

Just like in R, we can "hardcode" optional information into the function - that way there are defined values, but they can be changed:

```
def output_stuff(name, greeting = "Hi"):
        print greeting, name

output_stuff("class");                          #prints “Hi class”
output_stuff("class", greeting = "Hey");        #prints “Hey class”
output_stuff("class", "Hey");                   #prints “Hey class”
```

Some tips

- Functions should only assume the existence of parameters it’s given. You don't want to reference anything that is not defined inside your function!
- Functions should not be unnecessarily long - if it’s huge, maybe think about splitting it into multiple functions.
  - Having mini modular functions makes it easier to customize and change what you would like the program to do later.
  - People who use your code are very happy with this!

Objects

An object is an instance of a class of things

my Rav4 is an instance of “cars”

my pup Ellie is an instance of a “dog”

A **class** is a blueprint for making an **object**; it defines what an object is and what we can do with an object.

Car Class

I have a car - it has lots of things particular to it:

My car has a color: custom

My car has a make: Toyota

My car has a model: Rav4

And my car has a year: 2008

Car Class

```
class Car(object):
        def __init__(self):
                self.color = "Custom";
                self.make = "Toyota";
                self.model = "Rav4";
                self.year = "2008";
                self.gear = "P";
                self.brakestate = True;
```

Car Class

When I make a new object of type Car it will be **init**ialized with all those values set.

The **init** function is called a **constructor**. It's customary for class names to start with a capital letter and be named with staggered caps, but it's not required.

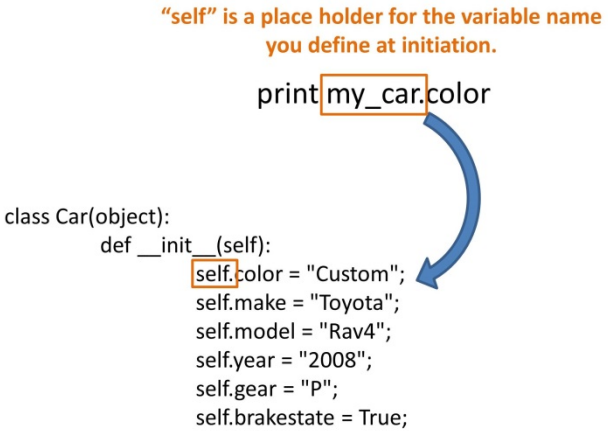
Let's make a car now. It's as simple as this:

```
my_car = Car();
```

And we can look at some of the values as such:

```
print my_car.color;    #prints "Custom"
```

Dot Notation



Car Class

```
class Car(object):
        def __init__(self, color, make, model, year):
                self.color = color;
                self.make = make;
                self.model = model;
                self.year = year;
                self.gear = "P";
                self.brakestate = True;

my_car = Car("Custom", "Toyota");
print my_car.color   #prints "Custom"
print my_car.gear    #prints "P"

old_car = Car("Purple", "Saturn");
print old_car.color   #prints "Purple"
print old_car.gear    #prints "P"
```

Bioinformatics in Python II – (In-Class/HW9)

Part 1: Trimmer2.py

Last week, we made a little program that took in a file, trimmed off a bit of the file and then output it to the screen and a file. That is a lot for one program to do without any functions! Let's rework the trimmer.py program to be a bit more modular, making it more powerful in the process! I recommend copying the file to a new one – trimmer2.py

Step 1: Building Data Structure

We made a seq_dict dictionary last time, and we didn't use it! That's because I decided to cut the assignment into two parts ^_~. Let's use that variable now, since its bad to just have variables sucking up space with no point!

Your current program should look something like this:
Line 1 –line that imports the "sys" library.
Line 2 –line that asks the user "Which file would you like to trim? ", save answer as **infile**
Line 3 –line that asks the user "How much would you like to trim off? ", save the answer as **trim**
Line 4 –line that defines an empty dictionary and call it **seq_dict**.
Line 5 –line that opens **infile** and saves it as **my_in_file**.
Line 6 – write a line that adds ".trim" to the **infile** string, save it as **outfile**
Line 7 –line that opens **outfile** and saves it as **my_output_file**.
Line 8 –line that defines **line** as a read in line from **my_in_file**.
Line 9 – while line != "":
*Line 10 –line that defines **name** as the current **line**.
*Line 11 –line that reads in another line and saves it as **seq**.
*Line 12 – line that prints name + the trimmed seq to **my_output_file**
*Line 13 – line = my_in_file.readline();
Line 14 –line that closes **my_in_file**.
Line 15 –line that closes **my_output_file**.

Let's add a line that saves the name and sequence into the dictionary:
…
Line 9 – while line != "":
*Line 10 –line that defines **name** as the current **line**.
*Line 11 –line that reads in another line and saves it as **seq**.
*Line 12 – line that prints name + the trimmed seq to **my_output_file**
Line 13 – line that saves seq[int(trim)] in seq_dict[name]
Line 14 – line that prints seq_dict[name]
*Line 15 – line = my_in_file.readline();
…

Checkpoint: If you run your program – you should see only the trimmed sequences printed to the screen. It will also make you a file with the newly trimmed sequences.

Great, now we have our data in a data structure – in this case, a dictionary. Right now our while loop is doing a lot of things – reading in, printing out, and saving the structure. This might not be a great idea if we want to build onto this code – which we do! All our sequences are stored in the dictionary, so there is no reason to print them in the while loop. Let's instead print them after the while loop (we also no longer need line 14):

…
Line 9 – while line != “”:
*Line 10 –line that defines **name** as the current **line**.
*Line 11 –line that reads in another line and saves it as **seq**.
*Line 12 – line that prints name + the trimmed seq to **my_output_file** #moved to Line 17
*Line 13 – line that saves seq[int(trim)] in seq_dict[name]
*Line 14 – line that prints seq_dict[name]
*Line 15 – line = my_in_file.readline();
Line 17 – for name in seq_dict:
**Line 18 – former Line 12*
…

Your code should now look similar to this (renumbered):

#LOAD LIBRARIES
Line 1 –line that imports the “sys” library.
#GET INFO FROM USER
Line 2 –line that asks the user “Which file would you like to trim? “, save answer as **infile**
Line 3 –line that asks the user “How much would you like to trim off? “, save the answer as **trim**

#DEFINE VARIABLES AND OPEN FILES
Line 4 –line that defines an empty dictionary and call it **seq_dict**.
Line 5 –line that opens **infile** and saves it as **my_in_file**.
Line 6 – write a line that adds “.trim” to the **infile** string, save it as **outfile**
Line 7 –line that opens **outfile** and saves it as **my_output_file**.

#LINE 8-13 READ IN FILE AND SAVE IT IN A DATA STRUCTURE
Line 8 –line that defines **line** as a read in line from **my_in_file**.
Line 9 – while line != “”:
*Line 10 –line that defines **name** as the current **line**.
*Line 11 –line that reads in another line and saves it as **seq**.
*Line 12 – line that saves seq[int(trim):] as seq_dict[name];
*Line 13 – line = my_in_file.readline();

#LINE 14-15 PRINT THE SEQUENCES TO THE OUTFILE
Line 14 – for name in seq_dict:
*Line 15 – line that prints name + the trimmed seq to **my_output_file**

#CLOSE FILES
Line 16 –line that closes **my_in_file**.
Line 17 –line that closes **my_output_file**.

Checkpoint – Running this file should output the trimmed sequences to a file with but the questions to the terminal.

Step 2: Building Functions

Hey, those blocks of code that I just delineated with comments look a LOT like little functional groups… Let's make them into functions!

A. Let's start with the easy one – printing the sequences:

#LINE 14-15 PRINT THE SEQUENCES TO THE OUTFILE
Line 14 – for name in seq_dict:
*Line 15 – line that prints name + the trimmed seq to **my_output_file**

Let's call our function **print_seq** and have it take in a parameter (let's call it **fasta_dict**).

```
def print_seq(fasta_dict):
   for name in fasta_dict:
      my_output.write(name + fasta_dict[name]);
```

Notice how we changed where it used to say **seq_dict** to **fasta_dict**? That's because anything passed to the function is now called **fasta_dict** inside of the function. If we called print_seq(seq_dict) – it would print out that dictionary! In fact, now we have to!

…
Line 14 - def print_seq(fasta_dict):
*Line 15 - for name in fasta_dict:
**Line 16 - my_output.write(name + fasta_dict[name]);
Line 17 – print_seq(seq_dict);
…

Checkpoint – your code should still run as before!

Hmm… but this function is using variables (**my_output_file**) that aren't local to that function (they are defined elsewhere)! Ok, let's move the part where we ask the user what they want to save the file as!

…
Line 14 - def print_seq(fasta_dict):
*Line 15 – line that asks the user "What would you like to save the output as? ") and save it as **output**
*Line 16 – line that opens **output**, as a writable file and saves it as **my_output_file**
**Line 17 - for name in fasta_dict:
***Line 18 - my_output_file.write(name + fasta_dict[name]);
*Line 19 – line that closes **my_output_file**
*Line 20 – print_seq(seq_dict);
…

Wow – look at that. We don't have to leave our output file open for the whole program. We can now remove the Lines 6 and 7 (naming and opening our my_output_file) and the last line (which closed our my_output_file). This function handles all that for us!

Checkpoint – your code should still run as before, but ask you for an output file name.

B. Ok, now the beefier one - reading in the file. Our current segment looks like this:

#LINE 9-13 READ IN FILE AND SAVE IT IN A DATA STRUCTURE
Line 8 –line that defines **line** as a read in line from **my_in_file**.
Line 9 – while line != “”:
*Line 10 –line that defines **name** as the current **line**.
*Line 11 –line that reads in another line and saves it as **seq**.
*Line 12 – line that saves **seq[int(trim):]** as **seq_dict[name]**;
*Line 13 – line = my_in_file.readline();

Let’s call the function **load_seq** and have it take no parameters.

Line 8 – def load_seq():
*Line 8 –line that defines **line** as a read in line from **my_in_file**.
*Line 9 – while line “”:
Line 10 –line that defines **name as the current **line**.
Line 11 –line that reads in another line and saves it as **seq.
**Line 12 – line that saves seq[int(trim):] as seq_dict[name];
**Line 13 – line = my_in_file.readline();
Line 14 – seq_dict = load_seq();

Ok, great – we have another function, but this one is also using variables that come from outside(**my_in_file**, **seq_dict**, **trim**). Let’s fix that.

We need to move the line where we define **seq_dict()** (Line 4), ask for the **infile** and **trim** (Line 2-3), opens **my_in_file** (Line 5), and the one that closes **my_in_file** (last line).

Line 8 – def load_seq():
*Line 2 –line that asks the user “Which file would you like to trim? “, save answer as **infile**
*Line 3 –line that asks the user “How much would you like to trim off? “, save the answer as **trim**
*Line 4 –line that defines an empty dictionary and call it **seq_dict**.
*Line 5 –line that opens **infile** and saves it as **my_in_file**.
*Line 8 –line that defines **line** as a read in line from **my_in_file**.
*Line 9 – while line “”:
Line 10 –line that defines **name as the current **line**.
Line 11 –line that reads in another line and saves it as **seq.
**Line 12 – line that saves seq[int(trim):] as seq_dict[name];
**Line 13 – line = my_in_file.readline();
*Line 14 - line that closes **my_in_file**
*Line 15 – return(seq_dict); #needed because we aren’t printing – we’re calculating!

Line 16 – seq_dict =load_seq();

Checkpoint – your code should run exactly as before.

Move your functions to the top, after import sys; you will only have TWO LINES that aren't in functions! Your code should look like this (order of your functions does not matter at all):

#IMPORT LIBRARIES
Line 1 - import sys;

#DEFINE FUNCTIONS
Line 2 – def load_seq():
*Line 3 –line that asks the user "Which file would you like to trim? ", save answer as **infile**
*Line 4 –line that asks the user "How much would you like to trim off? ", save the answer as **trim**
*Line 5 –line that defines an empty dictionary and call it **seq_dict**.
*Line 6 –line that opens **infile** and saves it as **my_in_file**.
*Line 7 –line that defines **line** as a read in line from **my_in_file**.
*Line 8 – while line "":
Line 9 –line that defines **name as the current **line**.
Line 10 –line that reads in another line and saves it as **seq.
**Line 11 – line that saves seq[int(trim):] as seq_dict[name];
**Line 12 – line = my_in_file.readline();
*Line 13 - line that closes **my_in_file**
*Line 14 – return(seq_dict); #needed because we aren't printing – we're calculating!

Line 15 - def print_seq(fasta_dict):
*Line 16 – line that asks the user "What would you like to save the output as? ") and save it as **output**
*Line 17 – line that opens **output**, as a writable file and saves it as **my_output_file**
*Line 18 - for name in fasta_dict:
**Line 19 - my_output_file.write(name + fasta_dict[name]);
*Line 20 – line that closes **my_output_file**

##########MAIN###############
Line 21 - seq_dict = load_seq();
Line 22 - print_seq(seq_dict);

Step 3: Why the heck did we do all that?
Five pages of coding and … nothing really changed? What? Why did we do this? Here's why. You can now call several fasta files, and trim them to different lengths. Try putting these at the end of your code:

my_seq = load_seq();
more_seq = load_seq();
print_seq(my_seq);
print_seq(more_seq);

You can also easily change your program and make it more friendly - to drive this home, I wrote you some simple code (you should be able to read almost all of it) that allows you to load multiple files and trim them to different points, and print them to the same file. It most of this code is user input control (did the user actually type what we asked, etc.). The rest is just using the same functions we encapsulated above. Functions allow you to make programs much more powerful by being modular in this way.

Paste the following code under your functions (directly under where I marked MAIN). Try trimming seqfile.fa to 14 and seqfile2.fa to 3:

```
######MAIN#####
print "Welcome to Trimmer!" #everyone likes friendly programs!

my_seq = load_seq(); #load the first file
more = raw_input("Would you like to load more (Y|N)? ");

while more == "Y": #As long as they want more – give it to them!
   more_seq = load_seq(); #load another file!
   my_seq.update(more_seq); #.update allows you to stick dictionaries together
   more = raw_input("Would you like to load more (Y|N)? "); #if this wasn't here, infinite loops!

if more != "Y" and more !="N": #what if they decide to not listen?
   print "You did not enter in Y or N - defaulting to N" #decide for them ^_~

print_seq(my_seq); #print your files if you want!
```

Part 2: Class (don't worry it is very short!)

Given the following:

```
class Seq(object):
   def __init__(self, name, seq):
      self.name = name;
      self.seq = seq;
      self.GC = "Uncalculated";
      self.length = len(seq);

   def GCcal(self):
      currbase = 0;
      gccount = 0;
      dnastring = self.seq

      #print dnastring[currbase];
      while currbase < len(dnastring):
         if dnastring[currbase] == "C" or dnastring[currbase] == "G":
            gccount = gccount + 1;
            currbase = currbase+1;
      return float(gccount)/len(dnastring);

name = "seq1"
seq = "ATCGGCATTATGATC"
```

How would you output (using the CLASS) – should be ~ 3 lines of code:

```
NAME Length GC
seq1 15 0.4
```

Required in HW8 folder:

Trimmer2.py (10 points) – your new modular trimmer program!
Commands.py (2 points) – your answer to Part 2

Optional Comparison between languages:

	Unix	R	Python
Declaring variables:	declare my_var="string"	my_var="string"; my_vect=c("a","b","c"); my_matrix=(1:20, nrow=5, ncol=4);	string="string"; list = ["a", "b", "c"]; dict={"a":1, "b":3, "c":4}
Printing to Screen:	echo $my_var; cat $my_var?;	print(my_var); cat(my_var);	print myvar;
Getting Help:	man COMMAND; COMMAND --help;	?function; function #no parentheses; methods(function);	dir(function); google documentation
Where am I?:	pwd; echo $PWD	getwd(); setwd(directory_name);	Wherever you launch from
Commands/options:	command -option file	function(file, option)	function(option, option)
Reading from a file:	> or |	read.table(file, options)	import sys; my_in_file.readline(); pipe in
stdin	default input	no default set - can specify stdin by using "stdin" as filename	default input, have to capture with my_in_file
Installing new things:	./configure, make, make install	install.packages("package_name"); load with library(package_name)	import library
Making it executable:	chmod 755 file	same!	same!
Comments:	#	#	#
BOOL	==, !=, <=, >=, &&, ||	same!	same!
Loops:	if() , else if(), else	if () {}, else if() {}, else{}; for (){} loops - check syntax!	if:, elif:, else:, check sytax - indenting is critical!
Useful Tools:	grep, sed, sort, uniq, awk	graphing functions, grep/sub	Biopython, classes
Regex:	slightly painful	really pretty similar...	same, even defaults to extended!

Additional Tutorials

Unix Navigation
Chmod explanation
Bash Variables explanation
PATH and bashrc explanation (note: .bashrc and .bash_profile functionally very similar. .bash_profile only executes the contained commands when you log in initially, .bashrc will execute all commands each time you open a new terminal - which is generally preferable).
Difference between .bash_profile and .bashrc - Mac specific differences listed here (because of the terminal built in to Mac).
Terminal Customization for Mac - getting colors to work if you can't with ls --color
Generates code to customize your terminal prompt (with colors!) in a drag and drop interface!
Source file install instructions tutorial

Bioinformatics in R (excercises adapted from this site)
Python book recommended by David
Python book recommended by Warren – Practical Computing for Biologists
Python resource recommended by James
Codecademy

Special Topics

CRC Quick Guide

CRC Quick Guide:

Computers you can log into

crcfe01.crc.nd.edu (4 16 core 2.4 GHz AMD Opteron processors with 256 GB RAM)
crcfe02.crc.nd.edu (4 16 core 2.3 GHz AMD Opteron processors with 256 GB RAM)
crcfeIB01.crc.nd.edu* (4 16 core 2.4 GHz AMD Opteron processors with 256 GB RAM)

If you need more RAM, you may be able to get permissions to use rosalind.crc.nd.edu. You would have to contact Scott Emrich in this case.

What is nice on the CRC is that you start with 100GB of home space. That isn't really enough to do much with, but your lab gets space as well. You will have to contact your lab group to get access to the lab space (permissions will lock you out until they add you).

Additionally, crc is backed up much longer than AFS space normally is.

How much space do I have?
You can check it with:
quota

Modules
Modules act kind of like .bashrc in that they will make certain programs available to you in your path without having to find/install them.

View module list:
module avail

Load a module (for example "bio" which contains useful bioinformatics programs like trinity):
module load bio

Unload a module (if you for some reason need to)
module unload bio

List the modules you have loaded
module list

Job Queues
Another wonderful part of the CRC is you can run jobs without having to be signed in. They are submitted to the "queue". There are a couple queues:

long - jobs run for a maximum of 15 days, but you may have to wait longer to have your job run.
short - jobs run for 4 hours, but your jobs will move through this que longer.

Job Submission

In order to submit to the que, you need to make a job file. They work kind of like an executable.ba would - they run the code in the file. However, additional information is needed in the header. It looks like this (all #comments are added):

```
#!/bin/csh          #NOTE this is NOT bash - anything you have in .bashrc will not work in this script. I don't think qsub can work in bash
#$ -M ssander5@nd.edu   #email where you want it to notify you
#$ -m abe           #sends an email to specified address above if job is aborted (a), begins (b), or ends (e); use any or all
#$ -r y             #or n; Indicates whether job is "rerunnable". If y, job will restart from beginning if killed for some reason
#$ -q long          #"short" times out after 4 hours I believe, where "long" times out after a couple days
#$ -N RunTrimmerMagna1  #Unique, recognizable name the process runs under

#you must tell it where to look for the program you are going to call - I usually place it in my personal bin (notice this command is slightly different than in bash).
set path = ( $path /afs/crc.nd.edu/user/s/ssander5/local/bin )

#you must enter the directory where your data is, if different from directory where job is launched
cd /afs/crc.nd.edu/group/genomics/beta/Daphnia/MAGNA_QTL1_04-23-13/130307_I1135_FCC1PFHACXX_L2_CCAPEI13010018/Trimmer

#call the program as you would normally - see Trimmer tutorial for this example]
radtag0_trimmer.py -p ../130307_I1135_FCC1PFHACXX_L2_CCAPEI13010018_2.fq ../Magnaplate1index.txt ../130307_I1135_FCC1PFHACXX_L2_CCAPEI13010018_1.fq
```

Managing your jobs

What is the status of my job (for example if you are me/ssander5?

```
qstat -u ssander5
```

NOTE: qw - waiting in the que, r - running

Delete a job (the number is listed when you use qstat

```
qdel -j 60123
```

What's going on with my job?

Your job will create a file, entitled whatever you used in the -N flag above (i.e. RunTrimmerMagna1) followed by a .o and the job number (i.e. RunTrimmerMagna1.o23453). You may also see a .po file of the same sort (i.e. RunTrimmerMagna1.po23453). These are your STDOUT and STDERR outputs, respectively.

Tips

- Try running commands before putting them in a job file. This ensure they will run correctly. You can always stop them with ctrl c.
- Create a blank job file and just copy it to another file name when you are creating a new job submission
- Have some sort of naming paradigm for jobs. I call all mine RunWhatever. Some people give them a .job extension. It doesn't matter what you do, but it helps when you are looking for them in your folders.

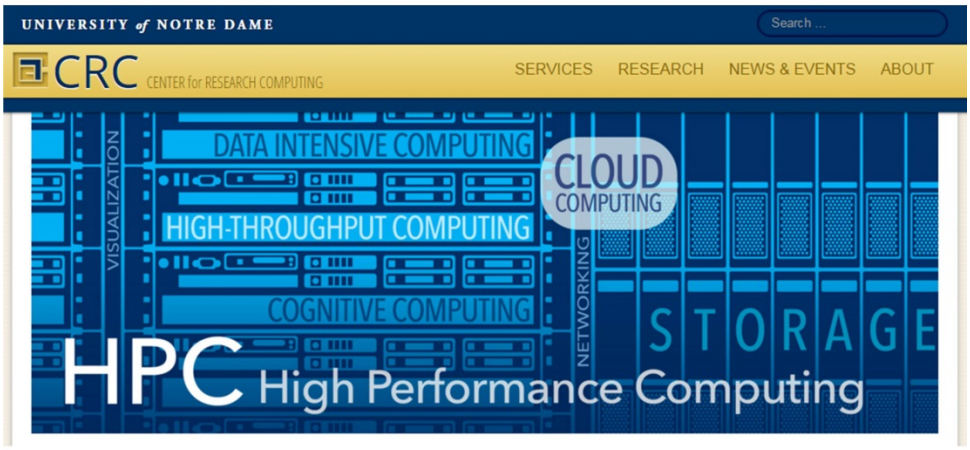
- If you are having issues with a job running, comment out some of the lines and check the outputs of each step. You can submit several jobs with the same name, it won't confuse the system.

Contact the CRC
Email:
crcsupport@nd.edu

Troubleshooting:
http://wiki.crc.nd.edu/wiki/index.php/Trouble_Shooting

For more in depth information:
http://wiki.crc.nd.edu/wiki/index.php/CRC_Quick_Start_Guide

CRC introduction PPT



Introduction to the CRC

Is it connected?

Student machines - CSE	**Darrow – common use**	**CRC – research**
Share an afs server...	but different access.	Different afs server
Access to all the same	files and programs.	No files shared*

Darrow/student machines are not connected to the CRC.

AFS is essentially a server that holds a set up that multiple machines can access. All the nd.edu machines share a set up and access the /afs/nd.edu/ server. All the crc.nd.edu machines share a set up and access the /afs/crc.nd.edu server.

What is the CRC?

- The Center for Research Computing
  - High Performance Computing
    - 23,000 cores of computation power
    - 3 PB data storage
  - Accessible to anyone with a faculty sponsor

Way more power

4 16 core 2.4 GHz AMD Opteron processors with 256 GB RAM

- crcfe01.crc.nd.edu
- crcfe02.crc.nd.edu
- crcfeIB01.crc.nd.edu*
- Run jobs on ~5000 cores with ~2GB per processor

4 16 core 2.4 GHz AMD Opteron processors with 512GB RAM

- rosalind.crc.nd.edu
- Runs jobs on 384 cores with 1.5 TB RAM

Way more space

- Onthe CRC is that you start with 100GB of home space.
- Lab gets space as well. You will have to contact your lab group to get access to the lab space.
- Scratch space
  - /scratch365/
  - /scratch30/

Way more programs

Modules act kind of like .bashrc in that they will make certain programs available to you in your path without having to find/install them.

View module list:
-->>module avail
-->>module avail bio

Load a module (for example "bio" which contains useful bioinformatics programs like trinity):
-->>module load bio

List the modules you have loaded
-->>module list

Unload a module (if you for some reason need to)
-->>module unload bio

There is also a link to the bio modules (Thanks Mike)!
http://wiki.crc.nd.edu/wiki/index.php/Installed_Applications#Biology

Way more convenient

******Job Queues******

Another wonderful part of the CRC is you can run jobs without having to be signed in. They are submitted to the "queue". There are a couple queues:

long - jobs run for a maximum of 15 days, but you may have to wait longer to have your job run.

short - jobs run for 4 hours, but your jobs will move through this que longer.

******Job Submission******

In order to submit to the queue, you need to make a job file. They work kind of like an executable.ba would - they run the code in the file. However, additional information is needed in the header.

How to make a job file

```
#!/bin/csh                  #NOTE this is NOT bash
#$ -M ssander5@nd.edu       #Email where you want it to notify you
#$ -m abe                   #Sends an email if job is aborted (a), begins (b), or ends (e)
#$ -r y                     #Do you want the job to restart if it crashes (y/n)
#$ -q long                  #"short" times out after 4 hours, "long" times out after a couple days
#$ -N RunTrimmer            #Unique, recognizable name the process runs under

#you must tell it where to look for the program you are going to call (notice this command is
slightly different than in bash).
set path = ( $path /afs/crc.nd.edu/user/s/ssander5/local/bin )

#you must enter the directory where your data is, if different from directory where job is
launched
cd /afs/crc.nd.edu/group/genomics/beta/Daphnia/data/

#call the program as you would normally
radtag0_trimmer.py -p read2.fq read1.fq
```

Managing jobs

Submit a job:
-->>qsub <job title>

Check the status of your jobs:
-->>qstat –u <username> #I have this aliased
-->>qstat –j <jobID>

Delete a job:
-->> qdel –j <jobID>

Read output from a job:
STDOUT → Job_title.o<jobID>
STDERR → Job_title.po<jobID>

Tips for Job Files

Try running commands before putting them in a job file. This ensure they will run correctly. You can always stop them with ctrl c.

Create a blank job file and just copy it to another file name when you are creating a new job submission

Have some sort of naming paradigm for jobs. I call all mine RunWhatever. Some people give them a .job extension. It doesn't matter what you do, but it helps when you are looking for them in your folders.

If you are having issues with a job running, comment out some of the lines and check the outputs of each step. You can submit several jobs with the same name, it won't confuse the system.

How to get an account

https://crc.nd.edu/index.php/services/usersupport

Where to get information

****Contact the CRC****

Email:

crcsupport@nd.edu

Troubleshooting:

http://wiki.crc.nd.edu/wiki/index.php/Trouble_Shooting

For more in depth information:

http://wiki.crc.nd.edu/wiki/index.php/CRC_Quick_Start_Guide

Most FAQ: transferring files

http://wiki.crc.nd.edu/wiki/index.php/Setup_CRC_AFS_Cell_Access

Tree Building Guide

******Trees******

Tree building is kind of an art. Remember how we said that there were MANY different multiple alignment programs? There are many different options for every step in making a tree (phylogeny.fr is great because it describes these options and allows you to swap between them). This variability stems from the problem that making a perfect tree by trying all the options is computationally impossible. All programs must reduce the problem to being feasible, requiring assumptions and choices to be made. Each one makes different assumptions in its algorithms, and each performs well in different situations.

Below is a guide for making trees.

What alignment to use?

Goal: Align sequences so that homologies are in line and all sequences are the same length with gaps.
Purpose: Allow for future analyses, gather information about the sequences as a whole (mutation points, level of divergence, etc.)
Challenges: Variable input, variable divergence, how do we know what is "correct" when alignments aren't a real thing (they are a concept we use for analyses, they don't exist in nature)?

Alignment requires a lot of decisions!

Pairwise alignments are easier, but still…
Which is closer to "graffe"? Graft? Giraffe? Graf? Raffle?

String matching is a computer science problem that is usually approached by finding the minimal edit distance between two strings.

```
Giraffe          Graf--          Graft-
| |||||          ||||            ||||:
G-raffe          Graffe          Graffe
```

In biological systems:

```
ATCG-ATACGAGGTA
|||  |||| |  ||
ATCCAATACTA--TA
```

However, decisions are still needed to determine how much each change is "worth". Are insertions and deletions weighted the same as substitutions? Are all substitutions equal (transition versus translation)? What about protein changes (Dayhoff versus BLOSSUM versus PAM)? Is one gap better than two (gap open penalties)? Do the sequences have to be the same length (global versus local)? Are substitutions weighted differently if they are close together versus further apart (gamma)? These are all addressed to different degrees in different programs and usually adjustable with options.

Additionally, multiple alignment makes this problem even more difficult. Typically, only conserved sequences will produce a good multiple alignment. Multiple alignment is also very much based on pairwise alignments, meaning the order of consideration of sequences has a heavy impact on the alignment output. It is impossible to calculate all options and calculate the score for each one, as the number of calculations scales exponentially.

Because of this, different heuristics and algorithms are used in computer science to make these problems solvable. Because these heuristics are built on different assumptions about the data (size, amount, variability), different programs are used for different data types. We will continue talking about the most common (sequence alignment), but know that different approaches are used for mapping short reads to a reference (we will talk about this near the end of the semester).

Different ways to deal with challenges:
Basic, string based solution (Clustal, MUSCLE):
These programs seek to maximize alignment score, more or less reducing the number of gaps and mismatches between strings, without biological guidance for the most part. These are fast algorithms, but basic solutions and not always accurate in the face of horizontal transfer, variable domains, etc.

Incorporate biological information to guide alignments (T-coffee):
These programs seek to maximize phylogenetic information by penalizing mismatches between chemically different amino acids or nucleotides, decreasing number of gaps that are individual to a sequence, among other things (if you are curious for the details, look up the T-coffee paper!). These algorithms are more accurate, but take longer to compute.

Probabilistic Concordance (ProbCons):
This program looks for the best model based on the consistency between all algorithms, including MUSCLE and T-coffee. Basically, it is assuming that multiple sources of independent evidence is more likely to occur in good alignments. It is currently no longer an option in Phylogeny.fr. You can use it by searching for "Prob-Cons server" and then pasting the alignment other websites give you into any of the tree builders on phylogeny.fr. This is a slow program.

All are explained by scrolling over the names on the phylogeny.fr page.

For more on specific software:
https://en.wikipedia.org/wiki/List_of_sequence_alignment_software

For more on algorithms:
http://www.amazon.com/Sequence-Alignment-Methods-Concepts-strategies/dp/0520271319

What alignment curation to use (optional but important step)?
Goal: Improve the alignment by removing loci that are uninformative (random noise) and sequences that are missing too much information.

Purpose: Decrease computational time in tree building, improve final tree.
Challenge: Replicability due to variation in alignments, variation in sequence similarity, biological information such as shared domains.

Gblocks is a common option. It is a tool I like to use since it makes trimming and cropping sequences consistent every time. It reads the alignment produced and determines poorly aligned and highly divergent regions (i.e. regions with little or confusing phylogenetic signal) and removes them. However, if the sequences are too divergent, it will fail out. At this point, I would reconsider the dataset or choose not to use curation and move on.
Information
here: http://molevol.cmima.csic.es/castresana/Gblocks/Gblocks_documentation.html

Additional manual curation may be required, to remove sequences with too much missing information or adjust anything that is really weird. Be sure you can justify anything you do and that you are consistent across all sequences and loci – needs to be replicable!

NOTE: Some tree building programs request or offer to take a model. JModelTest will look at your alignment and determine the likely pattern of evolution that it has (are there a lot of gaps, are there a lot of transversions, etc). These parameters help tree builders make decisions about where to put things. Phylogeny.fr can do this for you.

How do I know an alignment looks okay?
This is a hard question. Generally, before attempting to make a tree based on an alignment, you want to look at the alignment. Think about what you expect from your data: do you expect to see large insertions (exon shuffling for instance?), fragments, etc? If not, a large number of –'s may indicate a poor alignment and that more thought is needed before moving on. Removing poorly aligning sequences is often a requirement!

What software to use to build tree?
Goal: Build a tree, summarize the alignment information, add statistical support to our findings.
Purpose: Needs to be done before we can make visualization
Challenges: Tree space is too huge and complex as to search everything. Different algorithms do different things to reduce the search space. For reference:

Given n sequences | # rooted tree for n >2: (2n-5)!/(2n-3(n-3)!)
i.e.; n = 5 → 105 rooted trees, which is searchable.
n = 10 → 34,459,425 trees, which is not really searchable.

Different solutions to heuristic tree search
(tree search with assumptions that make it computationally possible):

Distance based: Neighbor joining – i.e.: BioNJ, Trex
These algorithms assume that sequences with lower edit distance are closer related. Basically, you join the two closest related sequences with a node, and calculated the distance between that node and the next closest sequence. It doesn't take into account

homologies or really much in the way of biological information – just sequence similarity. Generally, this solution performs poorly for anything with complex evolution or horizontal gene transfer. Statistical support is gleaned from subsampling the sequences and rebuilding the tree, then comparing it to the original. A significant use of these algorithms is non-sequence data (any distance measure can be made into a tree).

Character based I: Maximum Likelihood – i.e: see below
These algorithms assume sequences with more shared homologies are closer related. These algorithms generally start with two sequences connected with a node and iteratively add sequences to the tree and attempt to optimize its best location. As a result, ML is sensitive to order of sequences added. To get around this issue, trees are rebuilt hundreds to thousands of times with randomized input order. The branches that appear frequently are considered to have highest likelihood of being the correct relationship. By adding branches one at a time, the program limits the number of searches it has to do each time. It is still pretty slow with huge data sets.

RAxML
Randomized Axelerated Maximum Likelihood
Derived from Phylip
Starts a basic tree, adds a sequence. Each time it adds a sequence from the alignment, it attempts to optimize its location in the tree. Randomizes the start set and the order of addition of sequences to obtain bootstraps.
Benefits: Constantly updated, pretty good with optimization
Cons: Limited in model of evolution used.
Requires .phy format
Tutorial: http://bodegaphylo.wikispot.org/RAxML_Tutorial

Garli
Genetic Algorithm for Rapid Likelihood Inference
Similar to RAxML, but a little simpler to use.
Benefits: More flexibility in options, similar run time to RAxML
Cons: Not terribly common use, requires editing a runfile each time
Requires Nexux format
Tutorial: http://bodegaphylo.wikispot.org/Maximum_Likelihood_%28GARLI%29

*PhyML #*This is primarily what I use, for a reason I will elaborate on below.
Phylogenetic estimation using Maximum Likelihood
Benefits: Large number of substitution models, online execution - fast and easy(Phylogeny.fr), allows for quick "SH-like" branch support as a first pass look to troubleshoot.
Cons: Bit harder to run on command line.

Character Based II: Baysian – i.e.: MrBayes
These programs are similar to ML tree building in that they seek to join sequences that have shared homologies. They constrain space by determining a prior constraints, either by inputting a species tree to guide it, or calculating one based on thousands of calculations

(monte carlo markov chains). This is a newer algorithm, and the exact parameters aren't well established, leading to debate about the outputs. Most commonly, you will see these values mapped onto a ML tree. Statistical support is in the form of posterior probabilities, which are a Bayesian statistical concept.

NOTE: Many of these programs will test for a model of evolution before running the tree. However, some may ask you to choose a model. To find which best fits your data, you can input your alignment into programs such as JModelTest or MrModelTest.

Output from Tree Building: The Newick File
Here is a short introduction to the format:
http://www.megasoftware.net/mega4/WebHelp/glossary/rh_newick_format.htm

I will never require you to write these out. I simply want you to be aware that a flat text file output exists, which makes for easy transfer of trees from person to person and between programs. If you edit the tree in the below visualization programs, they will output an adjusted newick file.

Tree Visualization
Goal: Make a pretty tree.
Purpose: Visualize Phylogenies for further analysis, make it easier to interpret data.
Challenge: Many options exist for this step because editing and outputting options vary between programs.

General options: FigTree, TreeDyn, Dendroscope

Further analyses:

PAML - Phylogenetic Analysis Using Maximum Likelihood
PAML is a program that intakes PHYLIP 4.0 format which can include tree data. This program can be used to make trees, but it's older and clunkier, but it does allow for the analysis of selection parameters, omega, etc.
Here is a good
overview/tutorial: http://abacus.gene.ucl.ac.uk/ziheng/data/pamlDEMO.pdf

Mega – These same analyses can also be done in mega, which can be used to
import trees and alignments, and will calculate selection, neutrality, etc. It is the more user friendly option of these two, and then one I have preferred.
Tutorial: http://www.megasoftware.net/tutorial.php

CompPhy – New program that allows you to compare trees and easily co-edit them across the internet with collaborators. I have not used it, but the publication and website were very pretty.

In summary:
What I do: Align with probcons (or whatever you choose), Gblock to remove noise

consistently (using default parameters), model test, use PhyML to build a tree based on that model. These are all done online using phylogeny.fr. Then I use Mega to import my alignment and tree, and test for selection. If I need to make a huge tree than phylogeny.fr cannot handle, I use CYPRESS (online job submissions) or RAxML on the CRC.

Scaling to Large Jobs – Parameters and Hacks

Making flexibility more friendly

How many times did you have to answer:
"What file would you like to open?"

How frustrated did you get not being able to tab complete the file name?

Let's fix that.

Parameterizing

Adding flags and options (parameters) to your command calls is a useful skill.

We learned raw input first, because it is easy.

```
user_input = raw_input("Type something");
```

However, this is a low level hack – think about doing this for blast.

Sometimes we want to run a small program a lot of times:

```
trimmer.py infile.fa outfile.fa 15
trimmer.py infile2.fa outfile2.fa 12
trimmer.py infile2.fa outfile2.fa 10
…
```

Parameter lists

Generally how parameters work:

The first thing typed into Unix is the command, as we know. The full input is saved in a list/vector, with the first element as the command followed by user input.

python test.py arg1 arg2 arg3 ⟶ ["test.py", "arg1", "arg2", "arg3"]

```
#!/usr/bin/python

import sys

print 'Number of arguments:', len(sys.argv), 'arguments.'
print 'Argument List:', str(sys.argv)
```

Now run above script as follows:

```
python test.py arg1 arg2 arg3

Number of arguments: 4 arguments.
Argument List: ['test.py', 'arg1', 'arg2', 'arg3']
```

Parameter lists

We can assign members of this vector to variables:

```
#!/usr/bin/python

import sys

print 'Number of arguments:', len(sys.argv), 'arguments.'
print 'Argument List:', str(sys.argv)
array = sys.argv
first = array[1]
print first
```

Now it runs as follows:

```
python test.py arg1 arg2 arg3

Number of arguments: 4 arguments.
Argument List: ['test.py', 'arg1', 'arg2', 'arg3']
arg1
```

Back to Trimmer.py

Now you could do the following:

```
#!/usr/bin/python
import sys

array = sys.argv
infile = array[1]
outfile = array[2]
n = array[3]
…

trimmer.py infile.fa outfile.fa 15
```

But it will break if you don't have the right number of parameters.

More Advanced

Guess what? There exists a class for arguments! It allows a bit more flexibility and control in parameters and solves the previous problem.

```
import getopts
getopt.getopt(sys.args[1:], options, [long_options])
```

Here is the detail of the parameters -

args: The argument list

options: This allows you to use flagged options

long_options: This allows you to use long format options as well

Each option-and-value pair returned has the option as its first element, prefixed with a hyphen for short options (e.g., '-x') or two hyphens for long options (e.g., '--long-option').

sys.argv is what the class calls the saved vector taken in

```
args = getopt.getopt(sys.argv[1:],'hi:o:n:',["ifile=","ofile=","ntrim="])
```

h (for help), requires no input to follow

i, o, and n all require input that follows

user can also use long flags, --ifile, --ofile, --ntrim

Quick caveat

If you have a list:

```
my_list = [“q”, “w”, “e”, “r”]
```

You can parse it in one step:

```
a,b,c,d = my_list
print a          #prints “q”
print b          #prints “w”
print c          #prints “e”
print d          #prints “r”
```

If you have a list of lists:

```
my_list = [[“-i“, “in.fa”],[“-o”,”out.fa”]]
```

You can do the same:

```
For opts in my_list:
    o,a = opts
    print o
    print a
```

prints:

```
-i          #first loop
in.fa
-o          #second loop
out.fa
```

```
#!/usr/bin/python
import sys,getopts
opts, args = getopt.getopt(sys.argv[1:],"hi:o:n:",["ifile=","ofile=","ntrim="])
inputfile = ""
outputfile  = ""
n = 0
for o, a  in opts:
    if o == '-h':
      print 'trimmer.py -i <inputfile> -o <outputfile> -n <trim amount>'
      sys.exit()
    elif o in ("-i", "--ifile"):
      inputfile = a
    elif o in ("-o", "--ofile"):
      outputfile = a
    elif o in ("-n", "--ntrim"):
      n = a
print 'Input file is "', inputfile
print 'Output file is "', outputfile
print 'Trimming off ", str(n), "bases"
```

opts = [["h"],["-i","in.fa"],["-o","out.fa"],["-n",15]]

If we wanted to, we could add an if loop to define **outfile = infile + ".trim"** if no outfile was provided.

```
#!/usr/bin/python
import sys,getopt
inputfile = ""
outputfile = ""
n = 0
try:
    opts, args = getopt.getopt(sys.argv[1:],"hi:o:n:",["ifile=","ofile=","ntrim="])
except getopt.GetoptError as err:
    print(err)
    print 'trimmer.py -i <inputfile> -o <outputfile> -n <trim amount>'
    sys.exit()
for o, a  in opts:
    if o == '-h':
      print 'trimmer.py -i <inputfile> -o <outputfile> -n <trim amount>'
      sys.exit()
    elif o in ("-i", "--ifile"):
      inputfile = a
    elif o in ("-o", "--ofile"):
      outputfile = a
    elif o in ("-n", "--ntrim"):
      n = a
...
```

And now in bash…

If you type the following into bash:

```
some_program word1 word2 word3
```

It saves the command, separated by white space, into an array with length **$#**, that works kind of like awk:

$0 would contain "some_program"
$1 would contain "word1"
$2 would contain "word2"
$3 would contain "word3"

For example:

```
#!/bin/bash
echo "Positional Parameters“          #Positional Parameters
echo ‘$# = ‘ $#                       #$# = 3
echo '$0 = ' $0                       #$0 = some_program
echo '$1 = ' $1                       #$1 = word1
echo '$2 = ' $2                       #$2 = word2
echo '$3 = ' $3                       #$3 = word3
```

Flags in bash

We won’t cover this. Believe it or not, flags are easier in python and you should start there before trying it in bash. Then, eventually, you can look up tutorials on how to do this in bash if you ever need to!

You can exert some user control using if loops, i.e.

```
if [ $# -ne 3 ]; then
  cat “call the program properly!”;
  exit;
fi
```

Another trick: Parallelizing on CRC

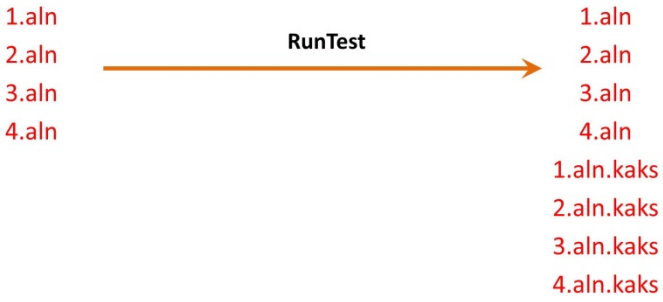
qsub reads line by line…
…but we can parallelize jobs by using a job number!

```
#!/bin/csh
#$ -M ssander5@nd.edu
#$ -r y
#$ -q long
#$ -N RunTest
#$ -pe smp 8
#$ -t 1-13:1
```

-t flag makes a set of tasks, each with a number. These numbers are defined as 1-13 by a count of 1 in this case.

```
/afs/crc.nd.edu/group/pfrenderlab/shared/bin/KaKs_Calculator -i
$SGE_TASK_ID\.aln -o $SGE_TASK_ID\.aln.kaks -c 5
```

We can then use $SGE_TASK_ID as a variable that holds those numbers.



But the files have to be numbered…

One great trick in bash to apply a command to a bunch of files (common!)

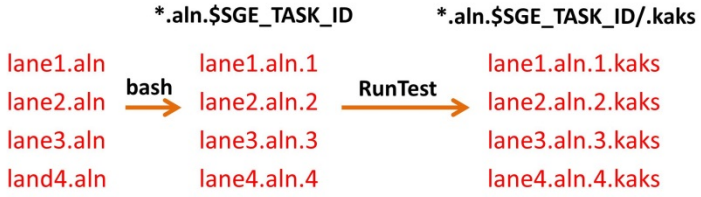
```
for f in *.fasta; do
  mv $f /fasta/;
done
```

We can rename things the same way:

```
for f in *.fasta; do
  mv $f ${f%.fasta}.fa;
done
```

Or add numbers to the front to leverage the $SGE_TASK_ID variable

```
i=1
for f in *.aln; do
   mv $f $f.$i;
   i=$((i+1));
done
```



Return files back to normal with:

```
i=1
for f in *; do
   mv $f ${f%.$i.kaks}.kaks;
   i=$((i+1));
done
```

Find and execute

```
find ~/ -name '*.txt'
```

All found items are subbed in for {}.

```
find ~/ -name '*.txt' –exec rm {} \;
```

-exec: takes a command as a string and executes it for each item found

\; tells the computer this is the end of the command string

Find all directories under the current folder:

```
find . –type d
```

Change permissions for all directories under the current folder

```
find . –type d –exec fs setacl {} ssander5 rlidkw \;
```

Galaxy

RAD-tag intro:
http://static1.squarespace.com/static/542f1ecee4b0a7afa8674662/t/544ab0fee4b007f4307ff885/1414181119821/

sources of error in RAD-tags
- adaptor ligation
- read through
- sequencing error
- PCR error

This analysis is more or less standardized now, but it still takes a lot of time and babysitting to get it through the initial stages of analysis. We are beginning to implement Galaxy as a means of automating this analysis in a much more friendly, efficient manner.

Why Galaxy? Because the CRC has limitations!
- wait time – especially bad during the end of semesters!
- allocation of resources - no scaling, hard to predict!
- sequential jobs, not time efficient

Galaxy (https://usegalaxy.org/)
- web-based resource to perform, reproduce, share data analyses

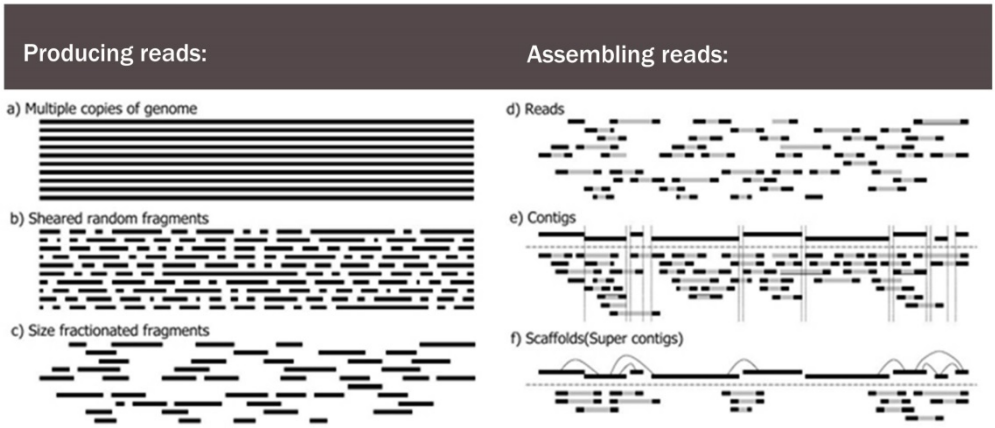
Goals of the project:
- accelerate analyses
- a nice user friendly GUI
- pre-installed programs
- create integrated pipelines
- reproduce tasks easily
- minimum user intervention

Galaxy demo – ND system as of 4/15
- upload files - 5G limit, very limiting but we're working on it!
- workflow maker – beautiful interactive GUI.
- history of runs are shown - failed, successful, etc.

See website for more details or contact Joe Sarro:

Joe Sarro – Senior Analyst
Office: 109 Galvin (by student lounge)
E-mail: jsarro [at] nd.edu
Office Hours: Wed 2-4 pm

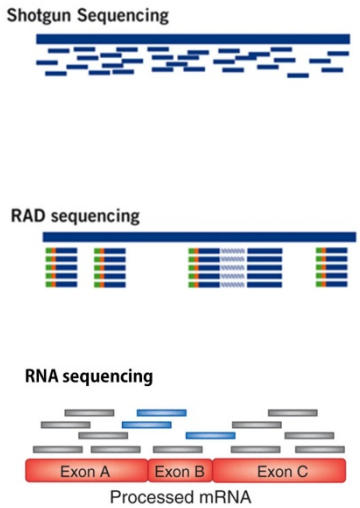
Assembly PPT



Reads – raw output from sequencer

Contigs – created by overlapping reads that flesh out the whole length

Super contigs/scaffolds – contigs connected with spanning paired end reads.



Different data leads to different assembly approaches.

How did length affect your assembly?

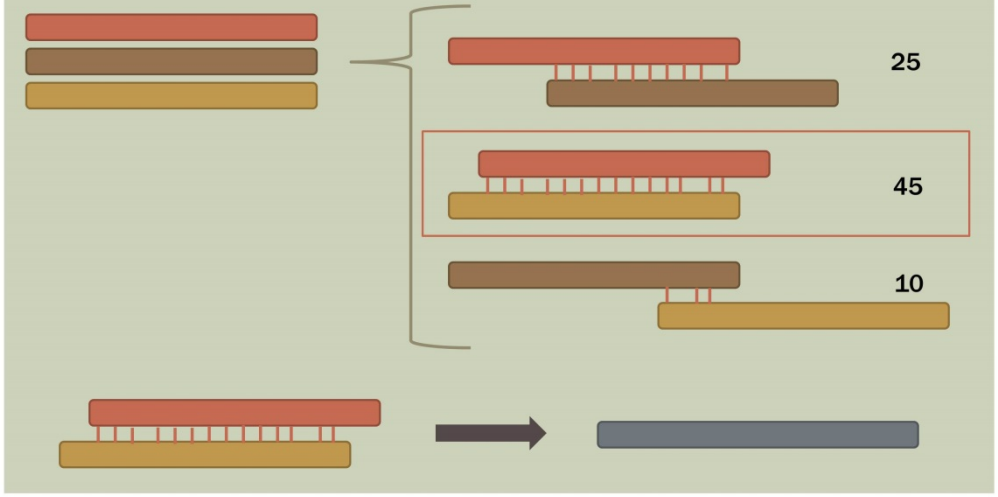
How did repeats affect your assembly?

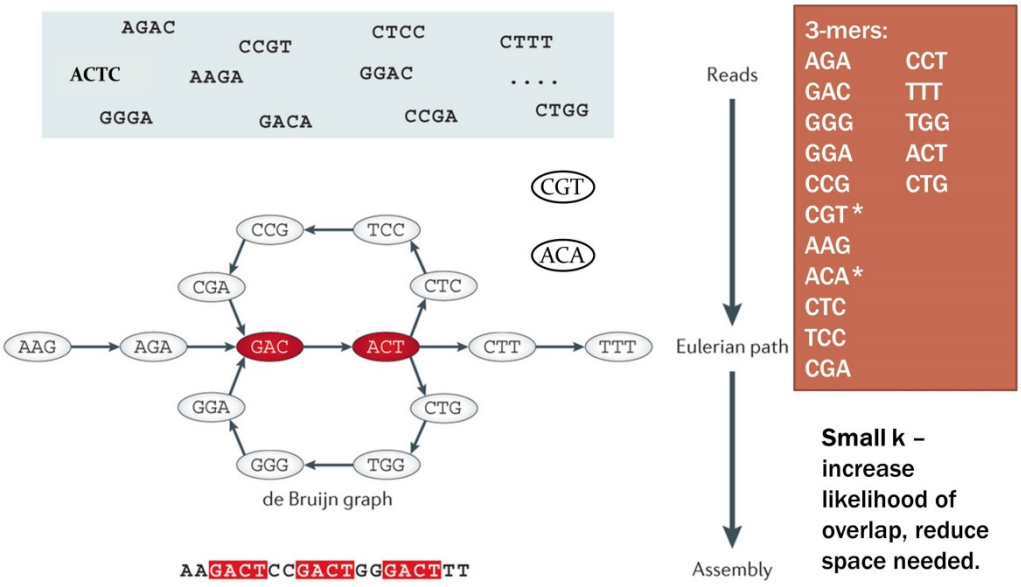
How did having a reference affect your assembly?

Requires pairwise comparison of EVERY sequence each time!!!

If you have three sequences:
1: 3x2x1 =6
2: 2x1 =2
3: 1 =1
Total: 9 comparisons

If you have five sequences:
1:5x4x3x2x1 = 120
2:4! = 24
3: 3! = 6
...
Total: 152 comparisons!



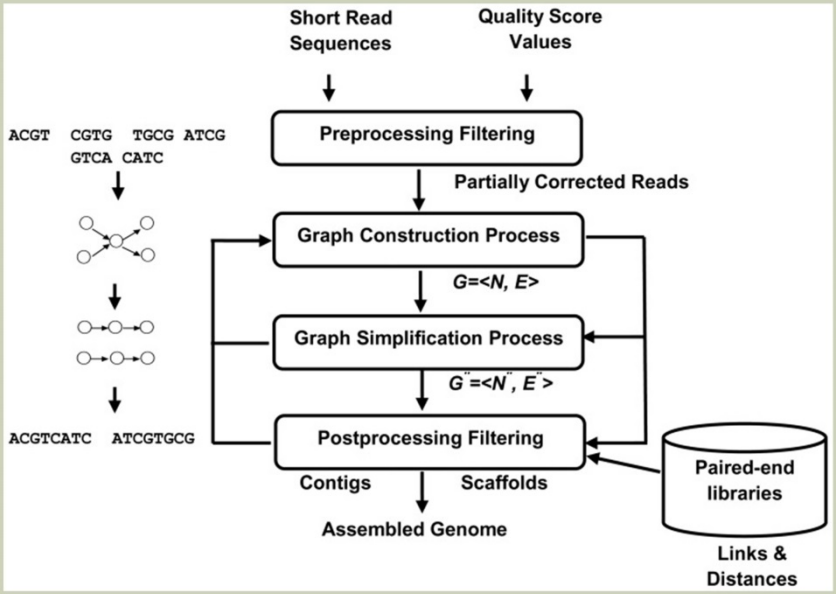


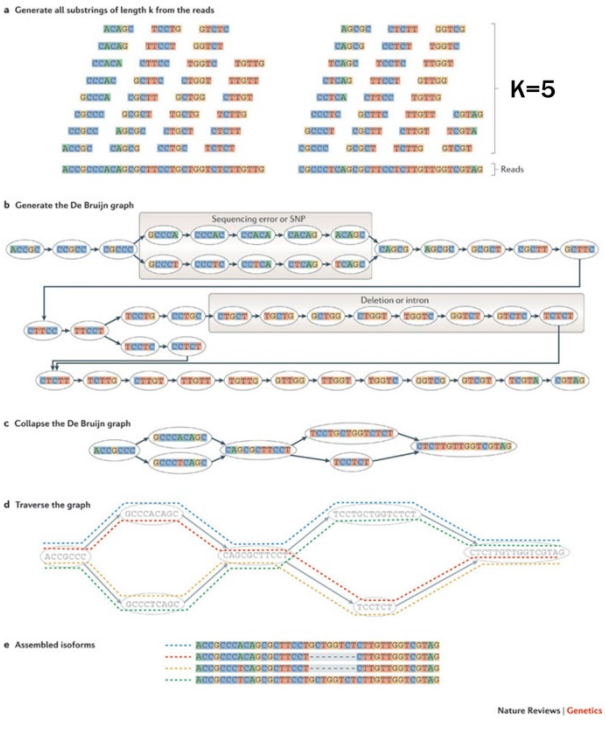
3-mers:

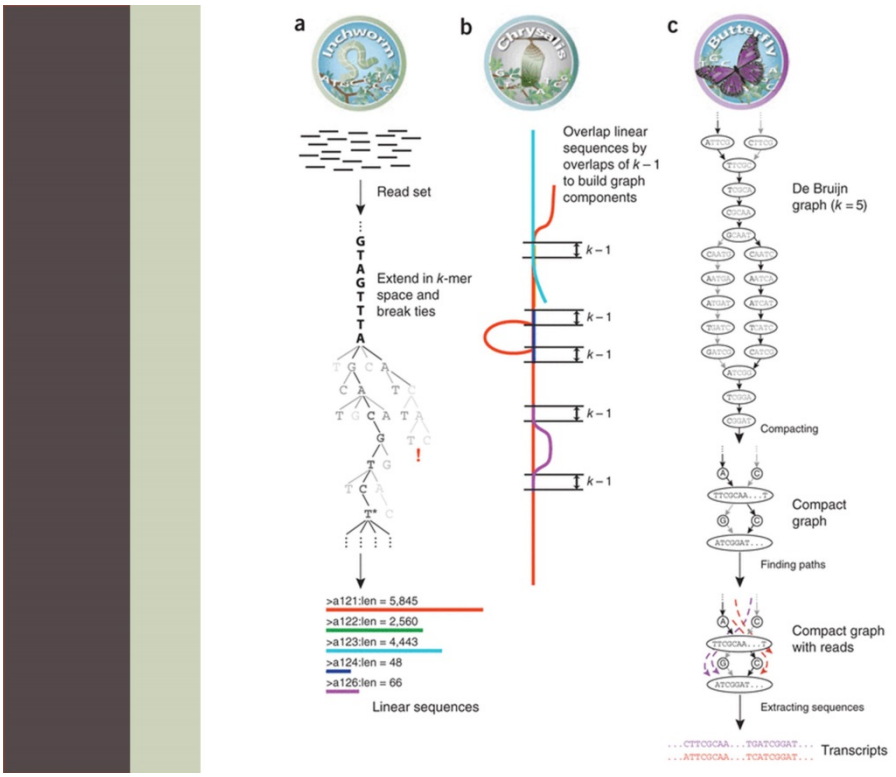
AGA	CCT
GAC	TTT
GGG	TGG
GGA	ACT
CCG	CTG
CGT *	
AAG	
ACA *	
CTC	
TCC	
CGA	

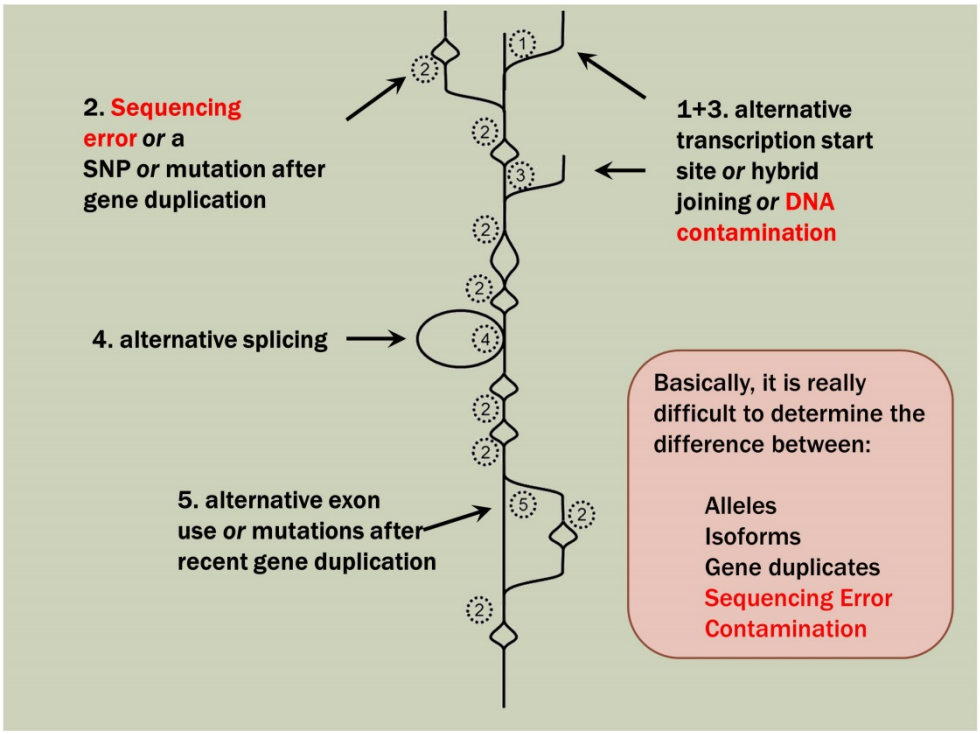
Small k – increase likelihood of overlap, reduce space needed.

Large k – bridges over repeats

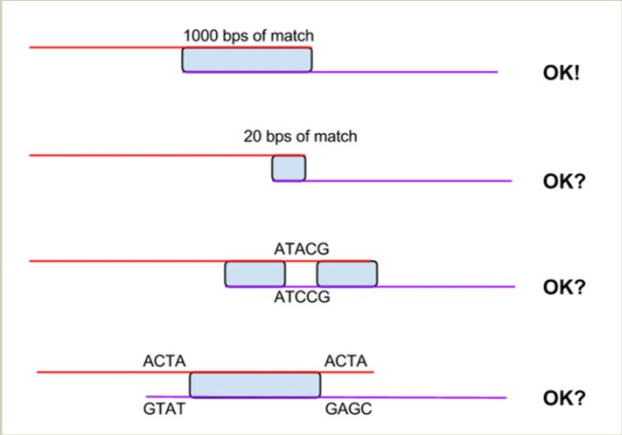

Short Read Sequences
Quality Score Values
ACGT CGTG TGCG ATCG
GTCA CATC
Preprocessing Filtering
Partially Corrected Reads
Graph Construction Process
G=<N, E>
Graph Simplification Process
G''=<N'', E''>
ACGTCATC ATCGTGCG
Postprocessing Filtering
Contigs
Scaffolds
Assembled Genome
Paired-end libraries
Links & Distances


a Generate all substrings of length k from the reads
K=5
Reads
b Generate the De Bruijn graph
Sequencing error or SNP
Deletion or intron
c Collapse the De Bruijn graph
d Traverse the graph
e Assembled isoforms
Nature Reviews | Genetics


a
b
c
Inchworm
Chrysalis
Butterfly
Read set
Extend in k-mer space and break ties
Overlap linear sequences by overlaps of k – 1 to build graph components
k – 1
De Bruijn graph (k = 5)
Compacting
Compact graph
Finding paths
Compact graph with reads
Extracting sequences
Transcripts
>a121:len = 5,845
>a122:len = 2,560
>a123:len = 4,443
>a124:len = 48
>a126:len = 66
Linear sequences


2. Sequencing error or a SNP or mutation after gene duplication
1+3. alternative transcription start site or hybrid joining or DNA contamination
4. alternative splicing
5. alternative exon use or mutations after recent gene duplication
Basically, it is really difficult to determine the difference between:
Alleles
Isoforms
Gene duplicates
Sequencing Error
Contamination

Other things to consider:



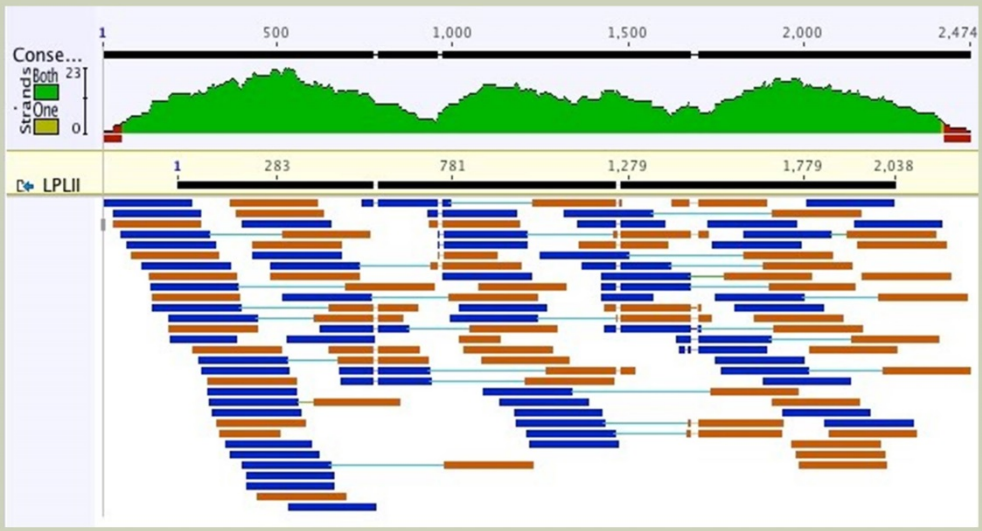
So how do we know how good it is??

- **ND50** – Length where fifty percent of contigs fall below (want this high!)
- **Max length** – Want this to be logical (RNA vs Genome, organism…)
- **Average length** – Want this to be logical (RNA vs Genome, organism…)
- **Number of scaffolds** – Want this to be low and logical!
- **Compare to closely related species** – Want to be similar!

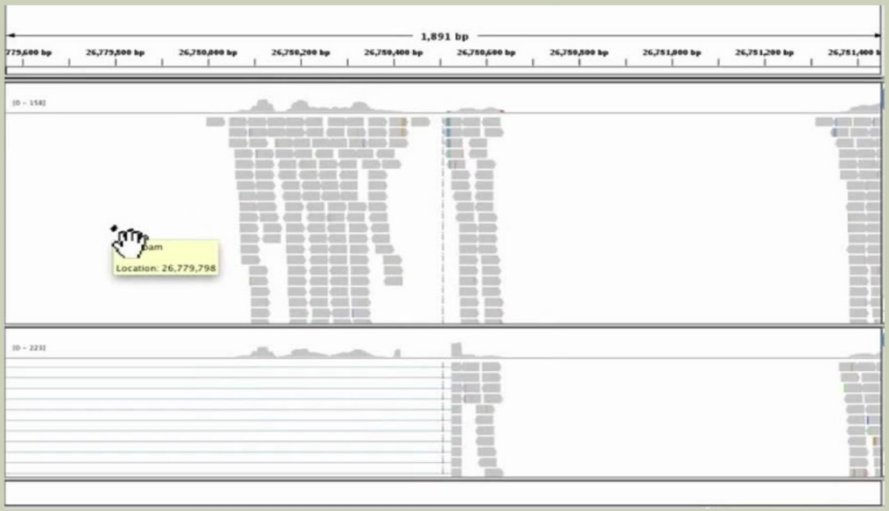
WHY SO MANY SCAFFOLDS?
WHY SO MANY PROGRAMS?

WHAT ABOUT MAPPING?

Coverage will vary. Some pairs won't be kept (poor quality).



RAD-seq does not have full coverage! It is a Reduced Representation Library

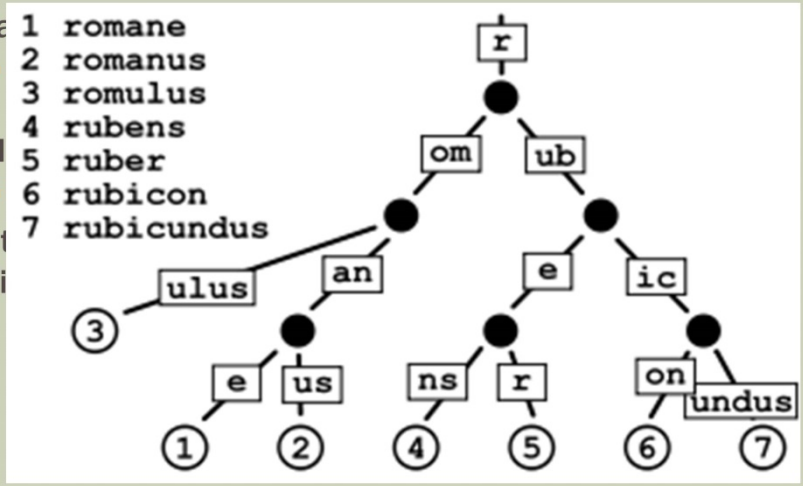


LET'S REVISIT BWA

- "Burrows Wheeler Alignment"
- Designed by Sanger Institute
- This is the program we are currently using in most of our pipelines at ND.
- Handles Sanger, Illumina, 454, and Solid
- Outputs to standard Sequance Alignment/Map (SAM) format

WHAT IS IT DOING?

- Makes a prefix tree (index function)
- Ta
  - 1 romane
  - 2 romanus
  - 3 romulus
  - 4 rubens
  - 5 ruber
  - 6 rubicon
  - 7 rubicundus
- M
- St
  di



DIFFERENT FLAVORS

Bwa-backtrace (traditional, what we use)

- Designed for Illumina sequence reads up to 100bp. Matches to prefix tree only based on seed.
- Repetitive read pairs will be placed randomly (true random due to prefix tree).

Bwa-SW(Smith Waterman, a bit obsolete)

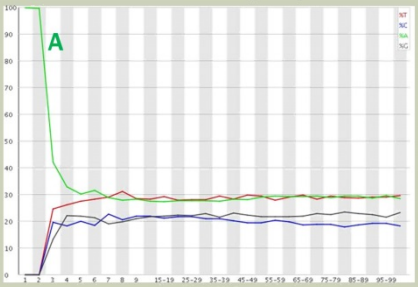
- Designed for longer sequences ranged from 70bp to 1Mbp.
- Matches seed alignments and extends using basic alignment with SW to find best match.
- In the paired-end mode, it may still output split alignments but they are all marked as not properly paired (more later); the mate positions will not be written if the mate has multiple local hits.

Bwa-MEM (Maximum exact matches)

- Designed for longer sequences ranged from 70bp to 1Mbpm. Matches seeds based on MEMs, then extends with SW to determine best match.
- Latest edition and generally recommended for high-quality queries as it is faster and more accurate. BWA-MEM has better performance than BWA-backtrack for 70-100bp Illumina reads.

CASE IN POINT THINKING ABOUT THIS IS IMPORTANT: BEN'S RAD DATA

Base Pair Frequency



GC content

EcoRI leaves this sticky end,

5' ---G AATTC--- 3'
3' ---CTTAA G--- 5'

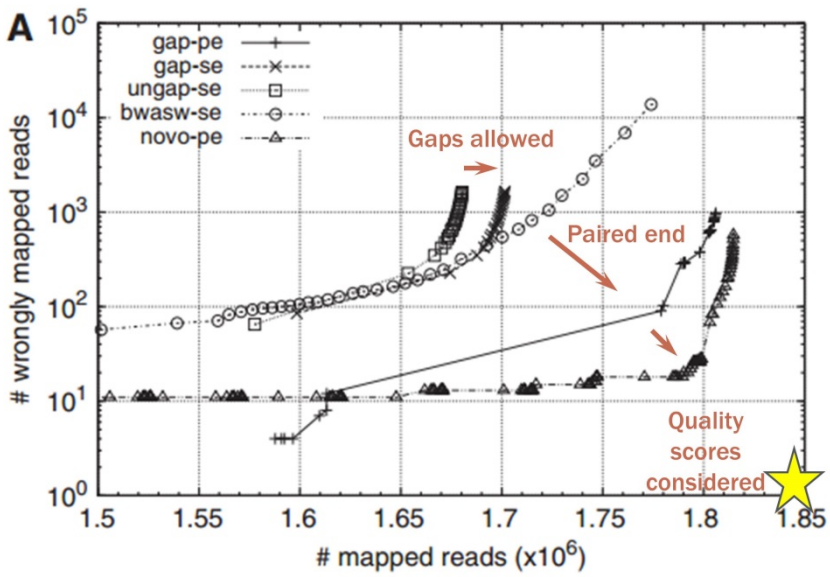
He was seeing low mapping success... might it be a result of this low variability in the beginning?

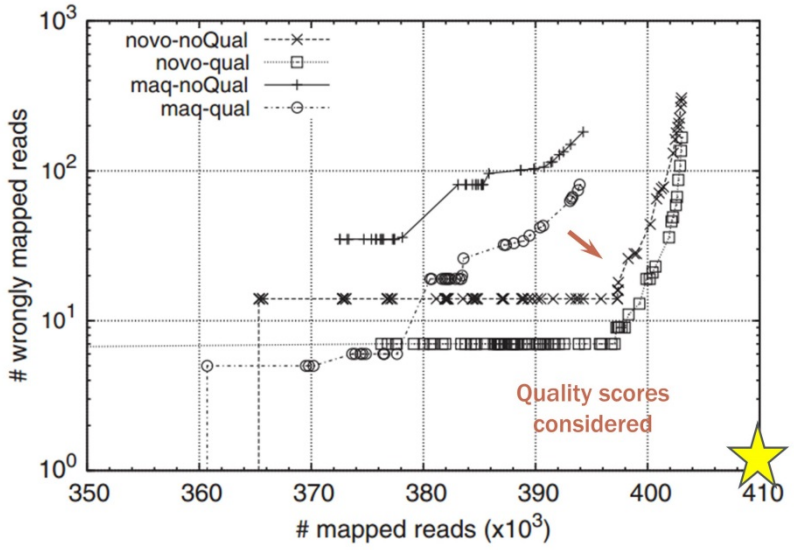
It does! Since mappers use prefix trees, the beginning of the sequence matters A LOT! Low diversity can cause issues, esp. if its not real.

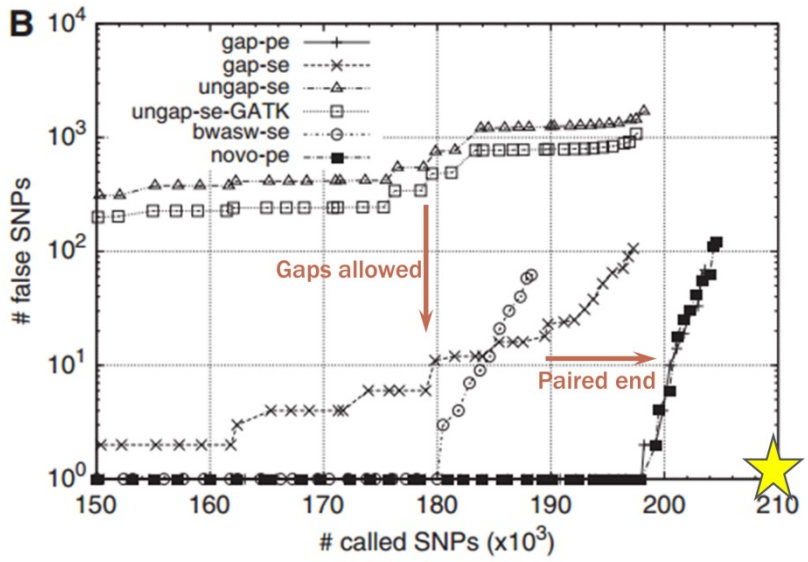
Out of a subset of 400,000 reads: 255,950 mapping reads after cropping the first 6bps versus 160,936 without (160% mapping improvement).

BEYOND BWA

There are ~20 different alignment program options


A
gap-pe
gap-se
ungap-se
bwasw-se
novo-pe
Gaps allowed
Paired end
Quality scores considered
wrongly mapped reads
mapped reads (x10^6)


novo-noQual
novo-qual
maq-noQual
maq-qual
Quality scores considered
wrongly mapped reads
mapped reads (x10^3)



What do you need to think about?

- Mapping vs de novo assembly
- What are you expecting?
  isoforms, alleles, recent duplications?
- Repetitive areas cause problems, think about removing them!!
- QC at the beginning is really important – trim, remove, etc. You will have PLENTY of data left and it will improve your output.
- Paired ends REALLY help.
- Using the appropriate aligner/assembler really helps.
  - What data is it designed for?
  - Consider using a variety of k-mers and merging after (lots of resources on how to do this)
  - Aligners that allow for gaps, paired end data, and de bruijn graph algorithms tend to perform best, even more so if they incorporate quality data.

Links to Useful Information

Unix Hater's Handbook A slightly out of date handbook that is pretty humorous and covers many topics. Definitely a for your interests link, but the first chapter has some fun content similar to what we covered in the first lecture.
Unix-like OS What it means to be "Unix-like"
Putty A link to download and set up Putty for the Windows users in the group!
Bash Variables Here is a nice explanation of bash variables and declarations.
Unix Tutorial We have gone through the material from the Intro, Tutorial 1, and Tutorial 2 from this site. This site is helpful if you would like a review of the material, but not using my direct notes
Codecademy This site is FANTASTIC. We will use it while learning python, but they just launched a command line learning tool as well. It is interactive and self-contained. I have not gone through it entirely myself (as I have with the Python and Ruby versions), but I figured it could only help you practice!
Chmod Tutorial Chmod tutorial that goes into more depth than we did, and also explains the binary number system (755 and 644).
Bash Variable Tutorial An overview and summary of bash variables.
PATH tutorial An overview and summary for setting PATH
Bash Profile vs Bashrc The difference between .bashrc and .bash_profile (which you may or may not have seen on this system).
MacOS terminal setup tips For those of you on MacOS who want to see colors like I do on putty.
Setup PS1 The PS1 variable set up that we used in the in class assignment.
Fastqc Video A nice walkthrough for the fastqc program output.
Compiling and Installing Programs Tutorial A summary of the ./configure, make, make install installation protocol
HMMER Tutorial A more in depth description of what all the programs of hmmer do. We will not go over all of them, but it is useful to visit if you are going to start HMMERing things regularly!
Blast Overview A nice summary about blast
Blast Algorithm A description of the steps that blast uses to search alignments
Smith-Waterman Overview A deeper explanation if you are curious about Smith Waterman
HMM metaphore A slightly dark metaphor about what hidden markov models are (Thanks to Chissa!).
BLAST stats A computational explanation of where the scores come from in BLAST (bit score, evalue, pvalue, etc).
Request a CRC account Link to requesting access to the CRC
CRC Quick Guide The handy quick guide on the CRC wiki
DISC REU Link to the Data Intensive Science Computing REU opportunity Dr. Pfrender mentioned
Edit distances An in-depth look at the more computational side of minimal edit distance.
Alignment Programs A pretty extensive link to different alignment programs, their uses, and where to obtain them.
Regex Tutorial
Wonderful website for help and depth of options for all three of our Unix power tools:
Awk Grymoir
Sed Grymoir
Grep Grymoir
Rubular A great resource for testing your regular expressions!
Download R
Download Rstudio
repl.it Python Online, instant feedback, python emulator we used in class.
Canopy The RStudio of Python - Canopy. Thanks Toby for sharing this!

Permanent Link to Course Documents

On AFS:
/afs/nd.edu/coursesp.16/cse/cse60132.01

Link to web space: TBA

Answers to the HW (from real submissions)

HW1 - troubleshooting:

1) correct command: -->>declare FILE=text.txt
There should be no spaces between "FILE" and "=" or "=" and "text.txt". The space is why you are getting the specific error.

2) correct command: -->> echo $FILE
There must be a $ before FILE to tell echo that it is a variable, not just the word FILE. The $ is essential to indicate a variable.

3) correct command to look up how to remove a directory: -->man rm
This gives you the manual for the remove/rm command which tells you a lot of information about the command, including how to remove a directory correct command to remove a directory: -->rm -r DirectoryName

Reading the manual for rm, you will discover that to remove a directory you need a -r, -R, or --recursive. This tells the rm command to be recursive and access and remove everything in the directory.

4) correct command: -->>ls -lah
Add a - to lah in order to see all file lists in human readable form. The -lah is necessary and without it, the command will not work! -lah (or even just -l) is a flag for the general ls command and it indicates an option in the ls command. The - (or -- for longer options) is essential for getting these flags to work and it will error out without them.

5) correct command: -->declare FILE=text.txt
When you declare a variable you DO NOT need $, that is absolutely necessary when echoing a variable but not to declare it.

HW2 - executable:

```
#!/bin/bash
#This program states your home directory and lists the content of your home directory
#It is called with: myprogram.ba

echo "my home dir is $HOME" #states home directory
ls --color ~ #lists contents of home directory with color coding
```

HW3 – troubleshooting:

1. If you are not root you will not have permission to install hmmr in the default installation location. Instead you will need to tell the program where to install. To do this use the command: --> ./configure --

prefix=<new location> and make the location your bin or another location that you have permission to install to!

2. You need to make sure you are downloading the program that matches the architecture of the computer you are running it on, otherwise even if you install the program, you will not be able to run it! The program you installed, hmmr-3.1b1-macos-intel.tar.gz, is for Mac. Even if you are on a Mac, the ND computers we are using are all running Linux so we can't use these Mac binaries/program. If you type the command -->>arch you can see what architecture your computer is running on and find the correct version of hmmer to install. For the machines we're using the architecture is x86_64, so you would want to download this version of hmmer-3.1b1-linux-intel-x86_64.tar.gz. If you install this version you should not get any errors when you attempt to run it.

3. If the program you grabbed doesn't have a configure file there are several things you can try! Some programs you download will not have a configure file because they have a binary installation (ex: MUSCLE). In this case when you open the program file with tar and gzip you will get an executable file. This is your program! You do not need to configure it, which is nice because you don't have extra steps, but is not customizable. Make sure you move this executable to your bin (or another location in your $PATH) so the computer will be able to find it when you try running it.

Other installations may be weird and just have a Makefile that you run and that gives you the executable binary, that again you can copy/move into your bin. If you ever have problems installing a program, the read me that comes with the program download is a good place to look for specific installation instructions. So is Google.

It's good to remember that when a program doesn't use the standard configure/make/make install instructions there is usually a README to help you figure out the specific installation instructions, and if there isn't it's time to check Google.

4. The first issue is that gzip files need to be labeled with .gz not .gzip. The programs will not run on files without the .gz extension. Your first step should be to change the file name from file.tar.gzip to file.tar.gz using the following command: -->>mv file.tar.gzip file.tar.gz.

However, if you try to use the command you used before (tar -xfz file.tar.gz) you will still get an error! The -f flag has to be the last one. This tells tar that the next thing you are typing is a file name. In the command that you have written, it is looking for a file named "z" since that is what directly follows the -f flag. -f should always be the last letter you list in any tar command flag line. Instead, you could use this command and not get an error: -->>tar -zxf file.tar.gz

Additionally, usually the -z flag should come before the -xf flags. If you are trying to unzip and tar in the same step the order of the flags matter. When using tar, the -x flag extracts the tar file, the -f flag tells tar that you are taking it from the file you are giving it to, and the -z flags unzips a gzip file. The -xf flags are looking for a .tar file NOT a .tar.gz file. which is why if you use the command tar -xfz file.tar.gz, it will error out. You are trying to extract the tar file before you unzip it from the gzip file! You need to unzip the gzip file first, before you try to extract the tar file. Thus, the following command is the correct one and will not error out: -->>tar -zxf file.tar.gz.

Note that additionally, if you use the old formating of tar, you can use the command without the - flag and then the order of your letter flags don't matter. For example the command -->>tar xfz file.tar.gz should work without any errors!

You should also note that some gzip files are made in such a way that even tar -zxf won't work on them. In this case, you will need to deal with the file in two steps. First unzip the gzip file using the command -->>gzip -d file.tar.gz. And then you can untar the file using the following command: -->>tar -xf file.tar. This is the most sure fire way to unzip and untag a .tar.gz file and should never error out!

HW3 – Tree Building

I spent a lot of time trying to create the best tree possible with the data we had. Our data was not the best alignment so certain programs could not be used. Below is the work through that I ultimately used to generate my .newick file and tree. Note that I also shortened the names of all of my sequences because phylogeny.fr said they were too long!

Curation:
I did two types of curation. I first applied some manual curation to the data before I went through alignment since certain sequences were messing up the alignment. I removed 5 sequences that were each 256 bp (labeled 165407) because they were much longer than the other sequences and this was making alignment difficult to handle.

After alignment, I tried to use Gblocks to curate the data, but the alignment was not solid enough to use Gblocks. Instead of skipping the curation step entirely, I decided to use the simple curation procedure that is built into phylogeny.fr, which is more simplistic than Gblocks and less stringent. It removes gaps and cleans up your alignment to some degree and I decided it was better to use this than do no further curated.

Alignment:
For the alignment, I really wanted to use Tcoffee due to the fact that it aligns the data while taking biology into account and optimizes phylogenetic information. The Tcoffee on phylogeny.fr was not working so I found the program elsewhere and generated the Tcoffee alignment and put this into the phylogeny.fr program, starting at the curation step. Note that my initial data was not good enough to be put into Tcoffee. Only after removing those five extra long sequences was I able to align via Tcoffee. Comparing the trees I generated using MUSCLE vs. the final one I generated using Tcoffee, the Tcoffee one seemed to group things better and provide a more biologically relevant tree. I'm glad I made the effort to get Tcoffee working.

Tree-building:
For the tree-building step I decided to use PhyML, a maximum likelihood/character based tree-building program. I used PhyML because it again focuses on biologically relevant relationships, in this case trying to optimize for shared characters and looking for homologies and shared domains. Because input order matters you need to randomize the order, and I chose to use bootstrapping for this step. Bootstrapping is not the default choice on PhyML because it takes a long time, but it is what is needed for a tree to be publishable. I decided the extra time was worth spending to get a tree that could theoretically be publishable.

Tree-viewing:
I used TreeDyn for the tree building step because it was mentioned as a good tree building software and it was easy to use on phylogeny.fr. I know that tree building is not an analytical step and since I am not using the tree figures for any specific purpose I did not feel the need to research and try out a bunch of different trees. If I actually needed the tree figures for another purpose I would have likely tried out different tree building programs to see which trees I liked best. What's nice is that I can continue to explore tree-viewing programs using the .newick file that I generated in the tree-building step if I want to revisit this data/tree at another time.

HW4 – Answers to Questions

Grep:
1) There are 91732 sequences in HW4.fasta.
2) There are 47323 sequences in the + strand.
3) There are 10900 sequences with two versions.

Awk:
1) The averag the MCHo of the MHC sequence hits is 0.00135955.
2) The average e-value of the MHC domain hits is 0.00402926.

Regex:
1) 20716 sequences have 100 or more AAs.

HW4 – fastq2fasta.ba

```
#!/bin/bash
#this program converts fastqc files to fasta files

#############
#first element of the pipeline finds every instance of @ in the .fq file
#first element also prints the first 2 lines of each entry in the .fq file starting with line that starts with @
#second element of the pipelne replaces every @ symbol with  >  symbol
#third element of the pipeline deletes every -- created in the 1st part of the pipeline
############

#call program as:  cat input.fq | ./fastq2fasta.ba > output.fasta

grep -A 1 "@" | sed 's/@/\>/g' | sed 's/--//g' #convert .fq file to .fasta file (outline in header)
```

Midterm Review – Answers

Major topics:

Navigation commands	Options	Unix tools – grep/awk/sed
Permissions	BLAST and HMMer	Regular expressions
PATH and .bashrc	File formats	Reciprocal Best Hit
Installing programs	Tree building	Compression

Matching:

Match the command with its purpose:

A. Move or rename a file.
B. Get information
C. Remove a file.
D. Copy a file
E. Find a program's home folder
F. Output date and time
G. List last ten lines of file to terminal.
H. Soft link
I. Open a file without editing it.
J. Open a file for editing.
K. List top ten lines of file to terminal.
L. List the resources in use on the computer cluster.

I less
J nano
A mv
D cp
C rm
L top
K head
G tail
B info
F date
E which
H ln –s

Match the special character to its function IN UNIX:

A. /	A Root or separates directory names
B. *	I Escapes a special character to make it literal (i.e:$ means $, not variable)
C. $	C Indicates a Unix variable
D. #	B Used to match any number of characters (in Unix)
E. :	H Used to match any single character (in Unix)
F. ;	D Begins a comment in bash
G. #!	F Separates commands on a single line
H. ?	G Indicates the location of the program that the code is written for
I. \	E Used to separate directory locations in PATH

Match the program with its function:

A. Blast	J performs math functions, prints columns individually
B. HMMer	E text reader
C. grep	G counts characters, lines, and bytes
D. sed	C finds lines that have matches to a pattern
E. less	H lists the resources in use for a machine
F. nano	A uses an unknown string to search a known database
G. wc	B uses a known set of strings to search an unknown set of data
H. top	I tells you the quality information of a sequence file
I. fastqc	D find and replace
J. awk	F text editor

Multiple Choice:

1. Who has permission for the following file?
-rw-rwx--x MMacTaggert acethisexam.txt
A. user, group, and others can execute
B. user and others but not group can read
C. only the group has all permissions
D. user can read and execute the file

2. Which command would you use if you wanted to find an average e-value from a .txt file output from a hmmer search?
A. Use grep to find the average e-value
B. Use sed to isolate the table and then awk to find the average e-value
C. use grep to isolate the table and then awk to find the average e-value
D. Use awk to find the average e-value

3. Which of the following tar options will work correctly?
A. tar –xfz file.tar.gz
B. tar -xzd file.tar.gz
C. tar xdf file.tar.gz
D. tar -xzf file.tar.gz

4. Which of the following things would you NOT expect to see in a .bashrc file?
A. aliases
B. automatic login to bash
C. PATH declarations
D. customization of your prompt
E. Variable declarations

5. The difference between fastq and fasta is:
A. fastq has amino acids, fasta has nucleotides
B. fastq has quality scores, fasta does not
C. fastq has two lines per read, fasta has four
D. fastq has ">" to denote samples, fasta has "@"

6. How can you fix a broken .bashrc?
A. give the absolute path to .bashrc
B. close terminal and reopen
C. use nano in cshell and edit
D. less .bashrc

7. How would you replace all instances of “abba” in a file with “bob”?
A. sed ‘g/abba/bob/s’
B. sed s/abba/bob/g
C. sed ‘s/abba/bob/g’
D. sed ‘s/bob/abba/s’
E. grep –r abba bob
Short answer:

1. What does the “which” command do?

Reports folder location of a program as long as it is in $PATH

2. What are two considerations when picking an alignment program?

Any number of things, including amino acids/nucleotides, if you wish to consider transversions/transitions or gamma, if you wish to consider string matching or probabilistic models, etc.

3. What are two ways to direct input into a program?

> , |

4. What is an advantage of regular expressions? What is an advantage of extended regular expressions?

Regex allow for pattern matching in searches, extended regular expression (non POSIX) allow for more options in patterns to match.

5. What is the purpose of PATH?

The PATH variable acts as a kind of map for the computer to find executables that you reference without giving an explicit path to. You can reference nano without typing /bin/nano because /bin/ is in you PATH.

6. Which are the two most commonly used blast types?

Blastn and blastp

7. If you already have both a set of known genes and a file with unknown sequences, what are the two steps of Hmmer and what does each one do?

Hmmbuild makes a profile based on known sequences you feed it, and hmmsearch uses that profile to search a database of unknowns to find matches to that sequence.

8. What is the difference between a source and a binary file?

A source file is a file that is human readable and customizable, usually installable by ./configure, make, make install. A binary file is only machine readable and is not as customizable.

9. Where would you look if you don't know how to install a program?
INSTALL, README, internet

10. What is the difference between AFS ACL's and Unix permissions?

AFS ACL's refer to directories and allow for much more convenient group handling, whereas Unix permissions are file based and only used (on our system and most large systems) to deal with individual files.

11. What are two things wrong with the following command? Grep –A –v 1 ">"

grep is capitalized, -A is not followed by the 1, there is no file to grep.

12. What is the purpose of a |? What are STDIN and STDOUT?

| redirects STDOUT into STDIN. STDIN is the default input stream that programs read from, either from the terminal input or from pipes. STDOUT is the default output stream that programs write to, either to the terminal screen or to pipe.

13. Explain what this awk command does in plain English:
awk 'BEGIN{sum=0}{if ($1 > $2) {sum = sum + $1}END{print sum}'

TYPO sorry!

awk defines a variable "sum" that starts as 0. It then adds the first column of a file to that variable for each line, if the first column is smaller than the second column. Then at the end, it prints the sum variable. Basically, we are reporting the sum of column 1 values that are smaller than column 2 values.

14. Explain what this sed command is doing:
sed ‘s/(sample1).*/\1/g’

sed is finding “sample1” at some point in a line, and matching the rest of that line (.*). Then sed is replacing that match with just “sample1”. Basically, this command is truncating all lines with “sample1” to end with “sample1”.

There will be a longer answer section…
It would be wise to review RBH as a bridge of a lot of our concepts...

HW6 – Answers

seqlength = 10735

GCcontent = .466977 which is around 46.7%

Count2words =
aa ac ag at ca cc cg ct ga gc gg gt ta tc tg tt
1108 720 890 708 901 523 261 555 976 500 787 507 440 497 832 529

pGC = .8651367

pCG = .4516013

HW7 – Slidingwindowplot.R

```
#!/afs/nd.edu/coursesp.15/cse/cse60132.01/Shared/bin/R

#function that creates a sliding window plot which plots nucleotide position vs. GC content

slidingwindowplot <- function(windowsize, inputseq)
{
starts = seq(1, length(inputseq) - (windowsize), by = windowsize); #defines starts
n = length(starts) #defines N

chunksGC=vector(mode="numeric", length=length(starts)); #defines chunksGC

for (i in 1:n) {
content = GC(dengueseq[starts[i]:(starts[i] + (windowsize-1))]);
chunksGC[i] = content;
} #creates a for loop to generate chunksGC vector

plot(starts, chunksGC, type="b", xlab="Nucleotide Start Position", ylab="GC content") #plots the starts vs. chunksGC
} #makes the final plot
```

HW8 – trimmer.py

```
#!/usr/bin/python

#LIBRARIES
import sys; #imports library needed

#INPUT
infile = raw_input("Which file would you like to trim?\n"); #prompts infile
trim = raw_input("How much would you like to trim off?\n"); #prompts number of nucleotides to trim

seq_dict = {}; #creates an empty dictionary to sue to store the sequence names with the sequences

#OPEN FILES
my_in_file = open(infile); #opens and renames infile
outfile = infile + ".trim" #generates outfile name
my_out_file = open(outfile, "w"); #opens, renames and makes outfile writable

#LOOP THROUGH FILE AND OUTPUT TRIMMED SEQUENCES
line = my_in_file.readline(); #defines line as a read in line from my_in_file
while line != "": #starts the loop
   name = line;  #defines name as current line
   line = my_in_file.readline(); #reads in the next line
   seq = line; #saves the next line as seq
   my_out_file.write(name + seq[int(trim):]); #prints name and trimmed sequence to my_out_file
   line = my_in_file.readline(); #ends the loop
#CLOSE FILES
my_in_file.close(); #closes my_in_file
my_out_file.close(); #closes my_out_file
```

HW9 – trimmer2.py

```
#!/usr/bin/python

#import libraries
import sys;

#define functions
def load_seq():
   infile = raw_input("Which file would you like to trim?\n"); #prompts infile
   trim = raw_input("How much would you like to trim off?\n"); #prompts number of nucleotides to trim
   seq_dict = {}; #creates an empty dictionary to use to store the sequence names with the sequences
   my_in_file = open(infile); #opens and renames infile
   line = my_in_file.readline(); #defines line as a read in line from my_in_file
   while line != "": #starts the loop
      name = line;  #defines name as current line
      seq = my_in_file.readline(); #reads in the next line
      seq_dict[name] = seq[int(trim):];
      line = my_in_file.readline(); #ends the loop
```

```
    my_in_file.close(); #closes my_in_file
    return(seq_dict); #returns function to seq_dict

def print_seq(fasta_dict):
    output = raw_input("What would you like to save the output as?\n")
    my_out_file = open(output, "a")
    for name in fasta_dict:
        my_out_file.write(name + fasta_dict[name]); #prints name and trimmed sequence to my_out_file
    my_out_file.close()

####MAIN####
seq_dict = load_seq();
print_seq(seq_dict);
```

HW9 – Commands.py

```
#!/usr/bin/python

class Seq(object):
    def __init__(self, name, seq):
        self.name = name;
        self.seq = seq;
        self.GC = "Uncalculated";
        self.length = len(seq);
    def GCcal(self):
        currbase = 0;
        gccount = 0;
        dnastring = self.seq
#print dnastring[currbase];
        while currbase < len(dnastring):
            if dnastring[currbase] == "C" or dnastring[currbase] == "G":
                gccount = gccount + 1;
            currbase = currbase+1;
        return float(gccount)/len(dnastring);

my_seq = Seq("seq1", "ATCGGCATTATGATC") #makes our sequence an object in the classSeq
print "NAME   " "Length   " "GC   " #prints a header
print my_seq.name,   "   ", my_seq.length, "  ", my_seq.GCcal();
```

Checkpoint: Unix Tools In-class Practice (with answers)

Answers are in the Answers section of this document, at the end.

1. Given $COURSE/Shared/data/Greppractice.txt, write the grep command for printing lines with (use the man pages to figure this out if you need to):

a. "this" regardless of case:
cat Greppractice.txt| egrep -i 'this' | less

b. only the word "is", so you would return "is" but not "this":
cat Greppractice.txt | egrep -w 'is' | less

c. display 1 lines after finding "this" (case-insenstive):
cat Greppractice.txt | egrep -A 1 -i 'this' | less

d. display 4 lines before "this" (case-insenstive):
cat Greppractice.txt | egrep -b 4 -i 'this' | less

e. print 2 lines around "this" (case-insenstive):
cat Greppractice.txt | egrep -C 2 -i 'this' | less

f. print every line without "this" (case-insenstive):
cat Greppractice.txt | egrep -vi 'this' | less

g. display line number of all lines with "this" (case-insenstive):
cat Greppractice.txt | egrep -ni 'this' | less

2. sed practice:

a. Given $COURSE/Shared/data/Sedpractice_a.txt, convert all "can't" to "can"
HINT: if escaping doesn't work, try putting single quotes around it!
cat Sedpractice_a.txt | sed 's/'\''t'//g' | less

b. Given $COURSE/Shared/data/Sedpractice_b.txt, convert all numbers to have two decimal places.
cat Sedpractice_b.txt | sed 's/\([0-9]*\)/\1.00/g' | less
or
cat Sedpractice_b.txt | sed 's/$/.00/g' | less

c. Given $COURSE/Shared/data/Sedpractice_c.txt, swap the first two letters of each word.
cat Sedpractice_c.txt | sed -E 's/^(.)(.)/\2\1/g' | less

d. Given $COURSE/Shared/data/Sedpractice_d.txt, swap the first two words in each sentence.
cat Sedpractice_d.txt | sed -E 's/^([A-Za-z]*) ([A-Za-z]*)/\2 \1/g' | less

3. Regex golf (use rubular or the other regex interactive I sent you). The goal is to match all off the targets and none of the "but not's". Each target counts for 10 points, each non target is -10 points, each character in your regex is -1 point. Try to get the best score you can in your group! Each list is available as Hole 1-7 in the Shared/data section!

Hole 1. How would you match:

[abcdef]*

caaba
aface
dace
bedad
decede
feed
decaf
affa
fab
abaff
faced
adead
cadee
accede

But not:

intransigeantly
unthematic
autolithography
haemostasia
alite
bigmouthed
ash-free
Marcelle
switchtail
aftermilk
argenol
grillade
CASU
half-wittedly

Hole 2. How would you match:

[abc][a-zA-Z]*

auriscalpium
blood
countersuggestion
conformator
Causey
besieger
biseriately
cyanoacetate
cioppino
Crashaw

But not:

exercent
nonfreeman
ignorances
intersegmental
fourth-rate
players
regathers
nonsurvival
unsanctitude
decurrence
thanato-
coto

Hole 3. How would you match:

(..).*(\1)

allochirally
anticovenanting
barbary
calelectrical
entablement
ethanethiol
froufrou

But not:

anticker
corundum
crabcatcher
damnably
foxtailed
galvanotactic
gummage
gurniad

Hole 4. How would you match:

ing?s

uninstructedly
reinstituting
carryings-on
forgings
insipiently
institutionalisation
inseminations
leggins
instruments
antivenins
tinsmiths
well-inspected
peninsularism
patins
recordings

But not:

unbenign
misstops
unsusceptible
cauterizations
prerogatives
about-faced
rattlesnakes
tempest-loving
Mayday
drown
dolus
royalistic
chitting
Borman

Hole 5. How would you match:

.*a.*a.*

Ultraparallel
suprastapedial
naphthanthracene
Walachian
cachaemia
bathomania
Craniata
bahamians
tarantula
achromaturia
aguavina
aquaria
abaxial

But not:

fork-tongued
polyclinic
foolscap
sacrosciatic
plotlib
self-direct
ladderway
serologic
whaling
hurlpit
LHS
dentition

Hole 6. How would you match:

[^qzl]

damager
venins
piner
unsimpleness
humbuggism
meacon
mythist
lycanthropist
decreases
directing
Munchhausen

But not:

polarization
Ovangangela
life-guard
zoographically
reundulate
Globe
well-solved
dumpling
zoophorus
tartly

4. Functional example:

```
#####################################################################
###                                                               ###
###              makeFastaOneLine.ba - pseudocode                 ###
###      Fill in the real code to make this program work!         ###
###                                                               ###
#####################################################################


##Basically, we want to remove all the newlines within a sequence.  It is
##difficult to reliably remove all the newlines just in the sequences, so
##we will remove them all, but only after making sure the ones we want to
##keep are marked!  Every command (except the last one) has to be followed
##by a | so they are connected (even if not on the same line).
## i.e. sed 's/this/that/g' |

#HOW TO CALL PROGRAM: cat input | ./makeFastaOneLine.ba > output.fasta

#Replace end of sample name with ### followed by a |
sed -E 's/(>.*)/\1###/g' |

#Sed cannot read across two lines and will not see the newline (\n) character
#at the end of a line.  Instead we can use translate (tr) to replace the
#newlines with a character (&), follow with a |.  Note that tr can only replace
#one character with one character.

#Replace all new lines (\n) with & using tr (general usage: tr 'this' 'that')
#Why didn't we just remove the \n with tr?  What happens if you try to do: tr '\n' ''?

tr '\n' '&' |

#Delete &'s
sed -E 's/&//g' |

#Replace all ### with new lines (\n), end with a |
sed -E 's/###/\n/g' |

#Replace all > with \n>, end with a |
sed -E 's/>/\n>/s' |

#Remove first line, as it will be blank
sed 1d
```