Phoethe rocks
Welcome to the TGen Software Carpenty Workshop!

June 19-20, 2017
Instructors: Nick Banovich and Emily Davenport
Helpers: Christophe Legendre, Elizabeth Hutchins, Eric Alsop, Ryan Richholt

Helpful links:
  Workshop website: https://erdavenport.github.io/2017-06-19-tgen/

Hiiiiiii everyon
California
Phoenix
Where you'r
Tempe Subic Bay, PI
Phoenix
Georgialsfor
Washington
San Antonio
Switzerland
Phoenix
where you from?
Wisconsin
pain
Scottsdale
Scottsdale
Peoria
Phoenix
Long Island
Alabama SLCma
Pakistan
ScottsdaleNJ
Gilbert
Everywhere
Toronto
Phoenix
Irvine, CA
PShoenixcottsdale
Utah
NY
North Carolina
Thunder Bay, Canada
Chandler
Ahwatukee
5th Floor
hellfire

Phoenix
Downstairs
az
what's an imputation?
You can "Impute" what a genotype might be based on a dataset, usually if you're missing data for a certain sample/patient
what does pwd actually do?

# The Unix Shell (Monday morning)

# Go to the the unix shell lessons http://swcarpentry.github.io/shell-novice/ -> Setup tab -> download the shell-novice-data.zip file to your desktop and unzip.
# You should see a folder called data-shell on your desktop. Orange sticky if you don't

# We're going to learn about shell using this mock example of Nelle Nemo

# To make the command prompt all look the same between people type the following in your terminal/gitbash
PS1='$ '

What's a directory?
-its a folder that can contain files and other folders

Can cd only be used to move into folders one step below the folder you're already in ?
cd can be used to move to any directory on the system. There are two strategies you can use: Relative paths or absolute paths.
If we have this hypothetical file structure:
    /Users/rrichholt/Desktop/code/script.sh
A relative path is a path in relation to your current directory, for example, if you are already working in /Users/rrichholt/Desktop/
"cd code" will bring you into the code folder.
Or you can use an absolute path: cd /Users/rrichholt/Desktop/code/ will bring you to the folder from anywhere on the system.
- Do you mean change to the previous directory simply use this command 'cd ..' the two periods represents previous directory

On my pc, I'm opening a notepad file - should I be in a different text editor or is this okay?
- notepad is great

How do you copy text in the terminal? I can't use ctrl + C on sections of text.

Is pwd a general way to know where you are or which directory you are in?
- Yes, pwd will list where in your file system you are (which directory you are in)

```
# Some of the commands we've used so far:

# What is your user name on your computer?
whoami

# To change directories:
cd

# What is my current location?
pwd

# ~ is a shortcut for your home directory. If you type "cd ~" it will take you back to your
home dirctory
cd ~

# Let's go into the data-shell folder (replace username with your username, might be slightly
different on a PC):
cd /Users/nbanovich/Desktop/data-shell

# Another way to do this:
cd ~/Desktop/data-shell

# How can we list out the contents of the folder we're in?
ls

# Let's list out the contents of the north-pacific-gyre/2012-07-03 folder (you should be in the
data-shell folder already):
ls north-pacific-gyre/2012-07-13/

# tab completion!! If you start typing the name of a file, try hitting tab. If there is only one
file that matches what you've typed so far it will automagically complete the filename.
# If you hit the tab once and nothing happens. Hit the tab again and it will show what files
match what you've started to type so far.

# Try tab completion. Start to type nor and then hit tab (and you should see north-pacific-
gyre fill in).
# Then start typing 2012- (and you should see 2012-07-03).

# We've finished the first part of the shell lesson. Woot! Let's do a few challenges.
# Do the absolute vs. relative path's challenge at the bottom of this page (http://
swcarpentry.github.io/shell-novice/02-filedir/)

# There's a difference between absolute vs. relative paths. Absolute paths give the specific
location of a file on that specific filesystem. Relative paths are the path to a file from the
location you're in. Absolute paths will work no matter where you are on your filesystem, but
there are times when relative paths can be more useful (for example, if you work on multiple
different computers)
```

# You can list out more information about files by using the -l flag
ls -l

# You can list out the file size as human readible (bytes) by including the -h flag
ls -l -h

# You don't need to have two separate flags separated by a space, you can include both after the dash
ls -lh

# If you include -G, it will include color coding for the files
ls -G

BREAK! Be back at 10:30

```
Emilys-Pro-v2:~ erdavenport$ ls -l
total 16
drwxr-xr-x   6 erdavenport  staff   204 May  5 19:28 Applications
drwx------+ 11 erdavenport  staff   374 Jun 19 12:54 Desktop
```

# Let's start the next lesson: http://swcarpentry.github.io/shell-novice/03-create/

# Make sure you're in the data-shell directory
cd ~/Desktop/data-shell

# Make sure everything looks right
ls -F

# Let's make a new directory for our analysis called "thesis"
mkdir thesis

# mkdir is short for "make directory"

# List out everything again and you'll see the new folder:
ls -F

# Let's list out everything in the directory thesis:
ls -F thesis

# We don't see any files, which is what we expect to see.

# Let's talk about file names
# 1. Use names that are descriptive, but not overlylong
# 2. AVOID SPACES. Use underscores or dashes instead. Bash interprets spaces as spaces between commands. Much easier when your file contains no spaces!

# Let's move into thesis
cd thesis

```
# Let's make a file using a basic text editor called nano (or notepad)
nano draft.txt        # If you're a mac
nano notepad       # If you're a PC and nano isn't working

# Whenever you write a script - use a plain text editor to write the script (versus something
like Word). There are fancier ones than the ones we are using today. Nick uses Sublime and
I use TextWranger -

# In your draft document, write a message to yourself:
Yay I finished my thesis!!!!

# Once you've written your text, hit control and the letter x at the same time from nano. Hit
"y" to save.

# Now if you type ls -F, you should see this one file in your directory
ls -F

# How can we delete files? rm (stands for remove)
rm draft.txt
```

**# CAUTION! rm does NOT put things in a trashcan. If you rm something, it's GONE.
Be careful when you rm!**

```
# Let's recreate this file
nano draft.txt

# Let's move one directory backwards:
cd ..

# Try to delete the directory thesis
rm thesis

# You should see an error. This is intentional - it saves you from deleting entire directories in
one swoop.
# If you DO want to delete a directory and everything inside it, use the -r flag (which stands
for recursively)
rm -r thesis/

# Now if you ls -F, you won't see the thesis directory

# What if you want a failsafe?

# Let's make another new directory called thesis
mkdir thesis

# cd into thesis
```

```
cd thesis

# Once again generate this file called draft.txt
nano draft.txt

# cd back up one directory
cd ..

# The -i flag is called the "interactive" flag
rm -r -i thesis

# Hit "y" when it prompts you to examine the files in your directory
# It will ask you if you're sure you want to remove each of those files
# For each file then you need to say yes.

# Once again make these files:
mkdir thesis
cd thesis
nano draft.txt

# Let's learn about moving and copying files

# To move a file, type mv (stands for move). Type mv, the name of the file you want to
move, and the place you want it to go (or the new name you want to give it).
# mv will move a file and it is also used for renaming iles.
mv draft.txt quotes.txt

# If you ls, you should only see quotes.txt
ls

# Go ahead and move back one directory
cd ..

# Type of the following (note that dot):
mv thesis/quotes.txt .

# Type ls
ls

# You should see that quotes.txt is now in the current directory, and the thesis directory is
empty
# remember that that dot . is a shortcut for the current directory

# Note that the syntax for moving files is this:
mv file-you-wanna-move place-or-name-you-wanna-move-it-to

# What if you want to make a copy? use cp. Let's make a copy of quotes and save that in the
thesis folder with the name quotations.txt
```

cp quotes.txt thesis/quotations.txt

# List out the contents of the directory
ls

# You should see quotes.txt in your current directory.
# If you list out the contents of thesis, you'll see quotations.txt
ls thesis

# Difference between mv and cp: mv will delete the original file whereas copy will create a copy of the file

# You can use ls on specific files as well as directories
ls -l quotes.txt

# You might want to do this to see the size of the file or when it was generated

# Let's remove quotes.txt
rm quotes.txt

# If you look inside thesis though, you'll see quotations.txt is still there
ls thesis

# File endings: .pdf, .txt, and .doc are all file endings that help your computer interpret what a file type is and what program to use to open it.

# Challenge: Work through the "Copy with Multiple Filenames" exercise at the bottom of this page http://swcarpentry.github.io/shell-novice/03-create/.
# Blue sticky when you're done, orange if you need help!

# If you copy multiple files at the same time, the final argument must be the name of a directory.

# Make sure you're in the data-shell folder

# If you want to create an empty file, you can use the command touch
touch my_file.txt

# Look at the size of my_file.txt
ls -l my_file.txt

# Zero bytes in this file, because this is an empty file.

# This might seem like a dumb command, but sometimes you need to have a file existing to write to.

# Let's move on to the next lesson about pipes and filters: http://swcarpentry.github.io/shell-novice/04-pipefilter/

```
# Make sure you're in data-shell
pwd

# Let's list out the contents of the directory molecules
ls molecules

# You can see in this folder there are a whole bunch of files with the extension .pdb (protein
databank)

# Move into the molecules folder
cd molecules

# Let's learn about the very handy wildcard character *
wc *.pdb

# wc stands for wc. It counts the characters, words, and lines in any file
# The wildcard character will match anything. So *.pdb will match anything in the current
folder that ends with *.pdb

# Output anything that starts with the letter p
ls p*

# List anything that has the letter e in the name
ls *e*

# To count the number of lines in each file:
wc -l *.pdb

# Note that when using a wildcard, tabs don't work.

# This just displays the information to the screen. What if we wanted to save this to a file?
Use the > to save the output of a command (what's called the standard output) to a file.
wc -l *.pdb > lengths.txt

# Now list the directory and you'll see lengths.txt
ls

# The cat function will list out the contents of a file to the screen
cat lengths.txt

# cat will list out everything in a file to the screen. If it's a big file, it will list out everything.
If you just want to look at a bit, you can use less
less lengths.txt

# less will bring up one window's worth of information to your screen.

# Let's sort the contents of the file:
```

```
sort -n lengths.txt

# What if we don't include the -n? This treats the contents as a character rather than a
number. So, if you're trying to sort a number use -n
sort lengths.txt

#  Let's sort on the number of lines and save that to another new file called sorted_lengths.txt
sort -n lengths.txt > sorted_lengths.txt

# Another useful command to look at the top bits of files is head. Head will show the top 6
lines. The -n 1 is telling head to only print out the first line.
head -n 1 sorted_lengths.txt

# If we wanted to look at the top two lines:
head -n 2 sorted_lengths.txt

# One thing to keep in mind is to try to not write out to the same file.
# Try this:
sort -n lengths.txt > lengths.txt

# Now look at the contents
cat lengths.txt

# The file is empty, because somewhere along the way something went wrong. Rather than
do this, just create a new file with the results of the sort stored in it.

# The real power of bash comes from using what are called pipes. Pipes will take the output
of a command and pass it to the next command listed after the pipe

wc -l *.pdb | sort -n

# This saves you the intermediate step of having to save the one file and then type another
command to get the output

# Let's add more on, let's only look at the top line of that output
wc -l *.pdb | sort -n | head -n 1

# What happened here?
# 1. You first counted the lines in all of the files that end in .pdb
# 2. The line counts are then passed to sort, which sorts by the number of lines
# 3. That sorted list is passed to head, which then returns only the first line of the sorted list.

# You can see the power here: you can chain commands together in bash

# Let's get to north-pacific-gyre. We'll use a relative path here. Go back one directory (..) and
the to the 2012-07-03, whcih is in north-pacific-gyre
cd ../north-pacific-gyre/2012-07-03/
```

# Type pwd to make sure you're in the right place
```
pwd
```

# Let's count lines of all the text files:
```
wc -l *.txt
```

# You can see there are line counts for each of the 20 or so files in this folder

# Let's look at the number of lines in the 5 smallest files
```
wc -l *.txt | sort -n | head -n 5
```

# All of the files should have 300 lines, but this one doesn't have 300 lines. This is a nice trick to QC your files to double check that they're all the same size that you expect to see.

# You probably don't want to include this file that only has 240 lines.

# Let's look at the bottom few lines rather than the top to make sure there aren't files with more lines than we expect.
```
wc -l *.txt | sort -n | tail -n 5
```

# They're all 300 lines, so that's good.
# However, you'll notice that one of the file names has a Z in the name whereas the rest of the files have A or B in the filename

# Let's list out only the files with a Z in them (must be uppercase):
```
ls *Z*.txt
```

# See there are two samples that list the Z. She forgot to include the depth. So, there are three files that Nelle probably should exclude.

# Let's do a challenge. Near the bottom of this page http://swcarpentry.github.io/shell-novice/04-pipefilter/, do the "What Does >> Mean?" challenge.

# Loops

# Let's move on to the next lesson about loops: http://swcarpentry.github.io/shell-novice/0e5-loop/

# Loops allow us to really automate things.

# Do a pwd to see where you are. Then navigate to the creatures directory in data-shell
```
cd Desktop/data-shell/creatures/
```

# list out contents in the creatures directory
```
ls
```

# See that these are .dat files.

# Let's make some copies of these files, because we're going to manipulate these files and we want a copy that we know hasn't been messed with. This won't work though, because we can't copy multiple files.

```
cp *.dat original_*.dat
```

# You can use your up arrow to scroll through commands you've recently used.

# If you hit control c, it will kill the line you're on

# Control r will do a reverse search. You can search through your recent commands for things you might have done. Start typing something that was in your command (a filename for example). Keep hitting control r to scroll through the options.

# Back to copying files. Let's learn how to use a for loop to copy the files. Let's use it to just list the head of a file rather than copy, as we're learning

```
for filename in basilisk.dat unicorn.dat
```

# When you hit enter, you see an arrow rather than the normal dollar sign prompt. This is because bash is waiting for you to give it more instructions. Next, type do

```
do
head -n 3 $filename
done
```

# You can see that for each file you listed after the for, it's listed out to the screen the top 3 lines.

# Type echo $filename

# The $ indicates that what you're typing is a variable. After you ran the for loop, the filename unicorn.dat was saved to the variable $filename.
# You can use whatever word you want as the variablename

# Let's try another loop where it prints the filename as it goes through the loop.

```
for filename in basilisk.dat unicorn.dat
do
echo $filename
done
```

# Protip: control a will take you to the beginning of the line and control e will take you to the end of the line

# Change your for loop variable to i, but don't change the body:

```
for i in basilisk.dat unicorn.dat
do
echo $filename
done
```

```
# You can see unicorn.dat is printed twice, because we didn't update the variable in the body.

# You can use the wildcard character to match all of the files in the directory that end in .dat
for filename in *.dat
do
head -n 3 $filename
done

# Let's say we want lines 80-100 for each of the files that end in .dat:
for filename in *.dat
do
echo $filename                              # This line prints the name of the file to the screen.
head -n 100 $filename | tail -n 20          # This line first takes the top 100 lines of the file,
passes that with a pipe to the tail command, which takes the last 20 lines of the top 100 lines.
done

# Back to copying multiple files: Let's copy each of those files, renaming them as original_
$filename
For filename in *.dat
do
cp $filename original_$filename
done

# If you ls, you'll now see the original files and then the original_$filename versions.

# Let's move back to the north-pacific-gyre folder
cd ~/Desktop/data-shell/north-pacific-gyre/2012-07-03/

# Remember we had those two files where there was a Z in the filename rather than an A or
B? We don't want to work with those files. Let's write a loop to only look at files where there
is an A or B in the filename

# We want to match any file that has either and A or B in the filename.
for datafile in *[AB].txt
do
echo $datafile
done

# As a sanity check, you can print out anything matching A or Z
ls *[AZ].txt

# The brackets are saying pick one letter inside these brackets to match. Can do all single
digit numbers
ls *[0-9].txt

#"regex" is short for regular expression.

# Let's add something else after our echo command. This is demonstrating what the for loop
```

```
is going to do, once we actually add a new command.
for datafile in *[AB].txt
do
echo $datafile stats-$datafile
done

# Let's run the stats script called goostats on each of the files. It takes in the first file after the
command ($datafile) and will save the output to the second file listed after the command
(stats-$datafile)
for datafile in *[AB].txt
do
bash goostats $datafile stats-$datafile
done

# You can see this is taking a minute, because it's running some computation.

# If you type ls, now you'll see a whole bunch of stats-*.txt files

# Let's look at the top of one of these files and you'll see some of the stats that were generated
by the goostats program
head stats-NENE02043B.txt

# Let's rerun that loop but make sure it's printing out the name of the file we're on to the
screen.
for datafile in *[AB].txt
do
echo $datafile
bash goostats $datafile stats-$datafile
done

# Control c will cancel an operation in bash.

# Let's remove anything with the stats name
rm stats-*

# Now if you type ls, you only have the original files

# Let's write this loop one more time
for datafile in *[AB].txt
do
echo $datafile                                  # This will print the name of the file we're on to
the screen, so we can track where we are
bash goostats $datafile stats-$datafile       # This tells the computer to run the program
goostats on the file $datafile, and save the output to stats-$datafile
done

# So, you can see that we're iterating through each one of these files and running the program
on each one. This is much faster than if we were running this one by one by typing out
```

everything each time!

# Let's do the exercise "Variables in Loops" in this lesson http://swcarpentry.github.io/shell-novice/05-loop/

# Also try the exercises "Saving to a File in a Loop - Part One" and do Part Two as well.

# A note about when you write your own shell scripts. You need to tell the computer that a file is executable. Nick is going to write a short little script and make it executable.

# In nano, write a hello world script
nano helo_world.sh

# write in that script
echo Hello World

# If you type helo_world.sh, it will not execute

# Let's rename the script
mv helo_world.sh hello_world.sh

# We can use the function chmod to change permissions. -x will make it executable.
chmod +x hello_world.sh

# Let's start on the finding things lesson, but the only thing we're going to focus on is grep http://swcarpentry.github.io/shell-novice/07-find/

# Go to the writing directory:
cd ../../writing/

# Let's look in haiku.txt
cat haiku.txt

# grep is one of the most useful commands in bash. It will pull out any lines containing the word you give it
grep not haiku.txt

# Let's try to grep the world The
grep The haiku.txt

# That matches both The and the beginning of Thesis, because this matches The. If you use the -w option, it will only match words
grep -w The haiku.txt

# There are tons of options in grep. Check them out, because they can be super useful.

# You can use wildcards with grep:
grep T*

```
# You can do the opposite using -v and retain any lines that DON'T have The
grep The -v haiku.txt

# If you want to ignore case you can use -i

# Can save the output of grep to a file using that redirect operator ">". Now you can use this
file in another analysis.
grep The -w haiku.txt > The_haiku.txt
```

# R!!

```
# Go ahead and open up RStudio.

# Let's start a new project by going to File -> New Project -> Empty file . Save as data-
carpentry

# Start a new script by going to File -> New File -> R Script

# In your finder/file explorer, you should see a new folder called data-carpentry saved to your
Desktop.

# Go to the workshop website for the R lessons: http://erdavenport.gith1ub.io/R-ecology-
lesson/index.html

# All the way to the right, go to the code-handout section at the top bar. and save that file.

# Move that file to the new directory we created (data-carpentry) and rename the file data-
carpentry-script.R

# In RStudio, to the bottom right, make sure you're clicked on the tab that says "File". Click
on the button to create a new folder, and create a new folder called data

# When thinking about best practices for organizing your files, useful reference:
http://journals.plos.org/plosbiology/article?id=10.1371/journal.pbio.1001745

# How to work with RStudio:
# The top left pane is the script we're writing. If you hit enter, the command is not run.

# Let's send lines from our script to the comand line:
1 + 1

# You can either click control return (mac or PC) or you can click on run at the top of that
pane.

# You can code directly into the R console, but keep in mind that you can't save those
```

commands for later. If you write commands in the script, you can save that file.

```
# You can do basic math in R
3 + 7

# Division
12 / 2

# Multiplication
5 * 5

# Often we want to save values for use later. We can assign variables in R to store any info
we want. Let's store 55 in the variable weight_kg
weight_kg <- 55

# If we look in the upper right pane, you can see that there's a variable listed.

# If you type weight_kg, you can print the stored value to the screen. Note that tab
completion works in RStudio!

# If you want to assign to a variable and also print to the screen, you can put the assignment
in ()
(weight_kg <- 55)

# Good practice is to use comments to document your code. Anything after the pound sign on
a line will not be evaluated by R
weight_kg <- 55     # Assign value to weight_kg

# Use comments! Future you will be happy you did because you WON'T remember why you
were doing something.

# You can now use variables to do math, because R will use whatever value has been stored
in the variable to do math
weight_lb <- weight_kg * 2.2

# What happens now when we change the value in weight_kg?
weight_kg <- 100

# Let's do another conversion and you can see that the value of weight_kg has been updated
weight_kg * 2.2

# Math is great, but R also comes with a bunch of built in functions for doing more complex
things.
square_root <- sqrt(10)

# Let's try running that function on a variable that we've set:
sqrt(weight_kg)
```

```r
# Some functions have multiple arguments. Let's look at the function to round numbers:
round(3.14159)

# Let's look at the arguments for that function though:
args(round)

# Let's set the number of digits to be more
round(3.14159, digits = 1)

# If you input your argument values in the same order as shown in args(), you don't need to
write the anme of the argument:
round(3.14159, 1)     # good!
round(1, 3.14159)     # bad :(

# If you write the argument names, you can put them in whatever order you want.
round(digis = 1, x = 3.14159)    # Also good

# Almost always, the data you input is what is listed first.

# You can store more than one value in a variable. Let's make what's called a vector:
weight_g <- c(50, 60, 65, 82)
weight_g

# You can also include character vectors in addition to numbers
animals <- c("mouse", "rat", "dog")
animals

# What happens if you don't put your word in quotes? R thinks it's a variable and will throw
an error if that variable isn't already assigned.
animals <- c("mouse", rat)

# Vectors are the work horses of R. Everything in R is vectorized. We'll see that as we learn
more.

# What are some things we might want to know about a vector?
# What's the length of my vector (how many values am I storing)?
length(weight_g)
length(animals)

# Vectors in R get assigned classes (numeric, character, logical, etc). These encode
information differently depending on whether something is a number or letters
class(weight_g)
class(animals)

# We can use the function str() to look at the structure of a vector. Shows the class of the
vector, how many variables, and the first few values.
str(weight_g)
# chr stands for character in R, not chromosome :)

weight_g <- c(weight_g, 90)
weight_g
weight_g <- c(30, weight_g)

# Challenge time! Do the challenge listed under "Vectors and data types".

# Subsetting vectors:

# Sometimes we don't want all of the values stored in a vector. We can pick certain ones.
animals <- c("mouse", "rat", "dog", "cat")

# What if we only want the second value stored in our vector? Use brackets
animals[2]

# For those of you that code in other languages, R starts numbers with 1 rather than 0!

# What if we want the second and third element?
animals[c(2,3)]

# What about the second through fourth element?
animals[2:4]

# Let's create a new vector made from our old vector:
more_animals <- animals[c(1,2,3,2,1,4)]
more_animals

# Let's create a new vector with a bunch of numbers:
weight_g <- c(21, 34, 54, 55, 75)

# We can subset out using logicals (TRUE T and FALSE F)
weight_g[c(TRUE, T, F, T, F)]

# Why is that helpful? Can use it to filter. For example, let's only return the values where the
weight is greater than 50
weight_g > 50

# This returns FALSE FALSE TRUE TRUE TRUE

# Let's combine that with subsetting to return only the values > 50:
big_weight <- weight_g[weight_g > 50]
big_weight

# What about if we want to find something equal to something else? Use two = for
comparison (==).
weight_g[weight_g == 5]
```

```
# You can use more than one condition when doing this type of indexing. What if we want any values where the weight is less than 30 OR less than 50? We can use the pipe |, which in R means OR
weight_g[weight_g < 30 | weight_g > 50]

# You can also use and statements with & (<= is less than or equal to)
weight_g[weight_g <= 30 & weight_g == 21]

animals[animals == "cat"]

# What if you have a bunch of values that you wanted to see if something was in that list of variables? Can use %in%
animals %in% c("rat", "cat", "dog", "duck", "goat")

# Can use this to subset the original vector of animals
animals[animals %in% c("rat", "cat", "dog", "duck", "goat")]

# Often you'll have missing data. In R, missing data is encoded by NA
heights <- c(2,4,4,NA,6)

# Let's try running a function like mean or max on heights
mean(heights)
max(heights)

# That doesn't work. We need to specify in an argument that we should remove NAs
mean(heights, na.rm = T)
max(heights, na.rm = T)

# The exclamation point in R means is not. This returns the vector without anywhere where there was an NA
heights[!is.na(heights)]

# Another way to remove NAs is to use complete.cases()
hegiths[complete.cases(heights)]

# Challenge! At the very bottom of this page http://erdavenport.github.io/R-ecology-lesson/01-intro-to-r.html#challenge8
```

There was a question over the break of websites you can use if you want to learn more. There are a bunch of websites where you can learn interactively. There will be material you can read and then there will be a prompt on the website where actually test those skills. The one I've heard about the most is datacamp for learning R or Python:

https://www.datacamp.com

And there are options for learning command line as well:
https://www.codecademy.com/en/learn/learn-the-command-line
http://www.learnshell.org/

```
# Let's start working with data
download.file("https://ndownloader.figshare.com/files/2292169", "data/
portal_data_joined.csv")

# In the bottom right pane where it says files, you should now see this portal_data_joined.csv
file in your file system.

# We've downloaded the data, but now let's read it into R and save it into a variable named
surveys
surveys <- read.csv("data/portal_data_joined.csv")

# We can look at the top few lines of this data.frame using head (just like in bash!)
head(surveys)

# A data.frame is a bunch of vectors all smooshed together into columns. We can use str() to
get the info for each column
str(surveys)

# When you read in data, R looks at your data and guesses what kind of data it is.

# We can look to see how many rows and columns our data.frame is. Row is always listed
first, and columns are listed second
dim(surveys)

# If you just want the number of rows, you can use the function nrow() (ncol() for number of
columns)
nrow(surveys)
ncol(surveys)

# Can look at the first few lines and can specify the number of rows to look at (let's look at 5)
head(surveys, 5)

# Can look at the last few lines as well using tail
tail(surveys, 5)

# You can see the column names by typing names
names(surveys)

# Look at the row names by using rownames(surveys)
rownames(surveys)

# We can index and subset data.frames just like we did with vectors. What if we want the
whole first row?
surveys[1, ]

# What if we want the whole first column?
surveys[,1]
```

# What if we want the value in the first row and the first column
surveys[1,1]

# What if we want the first three rows, and the third through 5th columns?
surveys[1:3, 3:5]

# Can use indexing to exclude rows as well. Let's print out the first row and everything but the first column
surveys[1, -1]

# What about the first row with everything but the second column
surveys[1, -2]

# Note, don't put commas in numbers in R!

# What about if wanted to print just the first 6 lines, but using the -?
surveys[-c(7:34786),]

# You can also use column names to subset columns. Let's pull out all rows, but just the column species_id
surveys[, "species_id"]

# Can also pick out columns by using the dollar sign. Let's do the same as above, but using the dollar sign
surveys$species_id

# Challenge! Let's do the challenge here listed under "Indexing and subsetting data frames" http://erdavenport.github.io/R-ecology-lesson/02-starting-with-data.html#challenge7

surveys_200 <- surveys[200,]

```
nrow(surveys)                      # Get the number of rows in survey
surveys[nrow(surveys), ]            # Pull just the last row of the dataset
surveys[nrow(surveys)/2, ]          # Pull the row in the middle
surveys[ , nrow(surveys)]           # Will return an error because there aren't that many
columns
```

# Factors - special data.class in R that deals with categorical variables.
str(surveys)

# If you look at the sex column, you can see there are three levels for sex: male, female, and ""

# Let's create a new variable that contains female and male as factors
sex <- factor(c("male", "female", "female", "male"))

# levels() shows what the categories are (know as levels in R)

```r
levels(sex)

# How many levels are there? Can use nlevels()
nlevels(sex)

# The default in R is that levels are stored alphabetically. It doesn't sort levels in order that they are in the dataframe, but by the alphabet.
# Sometimes you may want to switch the order of levels (for plotting, for example).
# Can use the levels argument of factor to reset the order
sex <- factor(sex, levels = c("male", "female"))

# We can convert factors back to characters using the as.character() function
as.character(sex)

# What happens with numbers though when we try to convert from factors to numbers?
f <- factor(c(1990, 1983, 1998, 1990))

# What if we just use as.numeric()
as.numeric(f)

# AH, bad! This is an R quirk. R stores factors as numbers. When you try to convert, convert first to a character.
as.numeric(as.character(f))

# This is really annoying of R and we all hate it! But be very careful when converting from factors.

# Another way you can do this
as.numeric(levels(f))[f]

# Let's make our first plot
plot(surveys$sex)

# You can see there are a whole bunch of values that are neither male nor female
sex <- surveys$sex
levels(sex)

# You can see there's this random slot of missing values.
# Let's index on this first level and rename it to "missing"
levels(sex)[1] <- "missing"

# If you look at levels(sex) you can see "" is now "missing"
levels(sex)

# Let's replot and you can see these things have been relabeled
plot(surveys$sex)

# Let's read in surveys again and let's use the argument stringsAsFactors = T
```

```
surveys <- read.csv("data/portal_data_joined.csv", stringsAsFactors = T)

# This looks the same, but let's set stringsAsFactors = F. This will force R to set the class as character rather than factors.
surveys <- read.csv("data/portal_data_joined.csv", stringsAsFactors = F)

# Nick (and I do too) loads in data with stringsAsFactors = F by default.

# Challenge! Do this challenge: http://erdavenport.github.io/R-ecology-lesson/02-starting-with-data.html#challenge13

# To download the tidyverse packages, in R
install.packages("tidyverse")

# Here is the history of my commands:
https://www.dropbox.com/s/jo4afzs7ig4tsbb/gHEADit_commands.txt?dl=0
```

Hi guys Nick here taking notes!

Lets get started with git!

All the problems you have with writing papers - eg writing re-writing changes - also happen when writing code. We should have a better way to track these changes.

```
# Sets username to your name
git config --global user.name "Your Name Here"

# Sets email to your email
git config --global user.email "imsocoll@coomail.com"

# Makes colors consistent
git config --global color.ui "auto"

# Sets the text editor you want to use
git config --global core.editor "nano/notepad"

# Look at all the settings
git config --list

# Okay git will only track changes in a directory we specific so lets create a directory - we obviously don't want to track changes of every file and folder on our computer
mkdir planets

# I'm horrible at spelling so feel free to edit any mistakes!!!!!! Remeber levles ;)

# lets move into the directory planets and check we are there
```

cd planets
pwd

# Now we need to tell git that we want to track all of the changes within this directory. Git will tack things on a directory by directory level. Lets tell git we want to initialize a repository in the directory planets
git init

# If we type ls we don't see anything different
ls

# But if we type ls -a we can see there is a hidden folder .git this folder is what makes git work.
ls -a

#Some people have a new prompt set. See this: https://stackoverflow.com/questions/32356595/why-is-mingw64-appearing-on-my-git-bash

# You can always change your prompt back to a simple $ using
PS1='$ '

# We can look at the status of our git repo using git status we will use this over and over and over and over ... you'll see how it's useful later today

git status

# Lets do a challenge http://erdavenport.github.io/git-lessons/03-create.html

# Okay lets make a new file
nano mars.txt

# Lets write "Mars is cold and dry, but everything is my fav color" then save

# Lets see whats going on with git

git status # we see that there is an untracked files "mars.txt"

# We need to tell git to track this file lets do that now
git add mars.txt

# Now we need to actually commit these changes. Emily gave the example of git add is telling people to line up for a picture and git commit is actually taking the picture. We always have to add a message when we commit. These should be short and descriptive.
git commit -m "Started notes on Mars as a base" # The -m tells what your message is going to be

# Now lets do git status and see how it looks
git status #we see there is nothing to commit as the most recent version of all files have been

```
commited

# Okay lets see how we examine the commits and messages we have generated thus far
git log #you should see the author, data, time, and notes about the commit

# Within that hidden .git directory is all of your tracked changes IF YOU DELETE THESE
WILL BE GONE

# Okay lets make some more edits and show how this works

nano mars.txt

# Write something like "The two moons may be a problem for wolfman" # save and exit

# Okay lets do another git status
git status # we should see that mars is now not being tacked

# We can use another command to see the differences between the last commit and our new
file
git diff

# Lets try and commit this file
git commit -m "Add concerns about effects of Mars' moons on Wolfman"

# Let's look at what happend
git status

# This didn't work because we always have to do the three step process. So let's try again
with an add command first
git add mars.txt
git commit -m "Add concerns about effects of Mars' moons on Wolfman"

#Okay, let's try and figure out more about staging. Start by making a new change
nano mars.txt
# Write "But the Mummy will appreciate the lack of humidity" and save

# Now let's try git diff
git diff

# Now lets see what happens after we add mars.txt
git add mars.txt
git diff # we see there are no changes to show! Once a file is staged it is being tracked by git
so git diff doesn't work.

# We can look at changes in the staged files with another argument
git diff --staged

# Now let's commit this final change
```

```
git commit -m "Discuss concerns about Mars' climate for Mummy"

# Okay let's look at the git status to make sure everything is working and the git log which should show three commits and three messages
git status
git log

# Lets do a challange while I catch up on notes! Very bottom of this page: http://erdavenport.github.io/git-lessons/04-changes.html

# Okay let's move back to planets and get ready to explore our history! http://erdavenport.github.io/git-lessons/05-history.html
cd ~/planets

# Okay lets look at the current version of mars.txt with the previous version. We'll do this using a shortcut

git diff HEAD~1

# We can also do this by looking at the commit identifiers. Remeber if we type git log we can see the identifiers
git log # Then copy and past the identifier you want to look at - you can sometimes do the first 10-15 identifiers

git diff IDENTIFIER_YOU_COPIED mars.txt

# Okay but what if we want to go back to an older version? Let start by messing up mars.txt
nano mars.txt # delete everything and write a bunch of junk

# Oh no we just deleted all of our hard work. We can use checkout to get the last version
git checkout HEAD mars.txt #This checks out the last committed version of mars.txt
# You can also use the unique commit identifier in place of head

# Okay we usually don't want to track changes to data files - espcially things like raw sequecing data. Let's see how we'd do this. Let's make a directory and a few fake files
mkdir results
touch a.dat b.dat c.dat results/a.out results/b.out

# Okay if we do a git status we'll see these files all show up as untracked
git status

# Now let's create a special file recognized by git called .gitignore
nano .gitignore
# In here let's write "*.dat and on a new line results/" This is telling git to ignore everything that ends with .dat - remember our wildcard and everything in our results folder.

# So now let's do git status
git status # We can see the gitignore file shows up but none of the other data or results files
```

# Let's add the gitignore file for good practice
git add .gitignore
git commit -m "Add the ignore file"
git status

# We can see we're all caught up even though we aren't tacking our data files. Let's see what happens if we try to add one
git add a.dat # You should get a message saying this doesn't work. This is good because we don't want to do this on accent - eg acciently start tracking a 100G bam file

# If you want to see what files you're ignoring
git status --ignored

# Everyone needs to go and create a new account on github.com

# Okay guys time to get in deep. On the internet go to github.com and create a new repository let's use the same name as we did on our computer - planets

# Now we need to connect the repository on the internet and the command line using this code in the command line
git remote add orgin https://github.com/USERNAME/planets.git # The https link comes from the github.com repository Use your username not mine

# We can use git remote -v to look at where this repository is connected to the web
git remote -v

# Okay so we have connected these repositotries but we haven't synced the files on our computer to the website. Let's do that now using git push
git push origin master

# We can also pull information from the internet using pull
git pull origin master

# Okay let's simulate working on two computers. Lets first move to the tmp folder on you computer from the terminalls
cd /tmp

# Now we are going to clone the directory from the internet. Go to the github website and navigate to the repository. You should see a gree button that says "Clone or Download". Click that and copy the link. Go back to the terminal and do the following
git clone LINK_FROM_GITHUB

# If you do an ls you should see the planets directory
ls

# Okay so within your /tmp/planets directory let's make a new file
nano pluto.txt # Write something awesome

```
# Let's add and commit
git add pluto.txt
git commit -m "Some notes about Pluto"
cl# Okay if we do git status we see that everything is up to date, but we get a message that we
are ahead of the origin/master
git status

# Let's push the changes to the world wide web
git push origin master

# You should now see that this file is on your repository online. This is still not synced on
your other "computer" - i.e. ~/planets/ . Lets navigate there.

cd ~/planets/

# Let's pull down the most recent commits from the webs

git pull origin master

# Okay let's make a change to mars.txt in ~/planets
nano mars.txt

# Let's add and commit
git add mars.txt
git commit -m "added a line to our home copy"

# Let's push this to the internet
git push origin master

# Okay let's navigate to the /tmp/planets and make some differnt chagnes to mars.txt in this
direcotry
nano mars.txt

# Let's add, commit, and push
git add mars.txt
git commit -m "added diff line to tmp copy"
git push origin master

# This should have given you an error when you tried to push
# Let's go ahead and git pull
git pull origin master

# Now if you cat mars.txt
cat mars.txt

# You'll see both changes are displayed here. We have to clean these up by hand. Let's do
nano and delete all of the junk and add a new fourth line saying we fixed merge.
nano mars.txt

# Let's git add and commit
git add mars.txt
git commit -m "fixed merge issues"

# Let's do git status to see what's happening here
git status # everything looks good

# Let's do git push
git push origin master # now this works because we have fixed the errors

# Okay let's navigate back to the ~/planets directory and do a git pull
cd ~/planets
git pull origin master

# Let's look at mars.txt now
cat mars.txt

# Now we can see that mars.txt is up to date everywhere. We didn't get another merge error
since we told git exactly what we wanted when we did the manual edits. git understands we
did this and so no merge error1

# Get geared up for a challenge!!!!

# We are going to do the challange at the bottom of this page: http://erdavenport.github.io/
git-lessons/09-conflict.html

# We are going to clone a repository from emily https://github.com/erdavenport/bio.git.
Clone this anywhere you want as long as it's not inside planets. /tmp is a good option


# Okay guys hope you had a good lunch. Get ready for R.

# Alright let's re-read in our data from yesterday
surveys <- read.csv("data/portal_data_joined.csv")

# Let's look at the first few lines of the data frame to see how everythging looks
head(surveys)

# Let's do a quick plot of our data

hist(surveys$weights) # hist is a function in R which allows you to make histograms

# Remember to code in the source (top left) and send to console (bottom left) using command
or control + enter

# We can change the color of the plot using the col argument within hist and the title using
```

main
hist(sruveys$weight, col = "red, main = "Distribution of weights") # the color and title need to be within double quotes

# Alright get ready for some loops!

#First we are going to do use the command if
if(surveys$year[1] == 1984){ # This is saying is the first element of the survey$year vector is euqal to 1984 then advance

- print("Great Scott, it's 1984") # You'll notice this is positioned to the right by one tab - this is common synatx for loops. Anything after the { is usually indented to say this is within the loop

}

# This didn't print anything since the first element is 1977. Let's update the code with a new statement called else

if(surveys$year[1] == 1984){

- print("Great Scott, it's 1984")
- } else { # This else statement tells R to do something in the case where the if statment isn't true
- print ("It's not 1984)
- }
-

# Challenge is here about halfway down: http://erdavenport.github.io/R-ecology-lesson/03-loops-and-functions.html

# Let's solve the challange together!
if(surveys$weight[40] >= 28.3){
   print("It's a biggun")
   } else {
      print ("It's not very big")

- }
-

# Okay now let's do some for loops. These are similar in principle to the for loops we learned about in bash yesterday
for (i in 1:10){
   print(i)
   }
#This should print 1-10. What happens if we print out a string?

for (i in 1:10{
   print("cookie")

```
    }
#This prints cookie 10 times because nothing is adaptive.

# Let's loop over something non-numeric
for(i in c("cat", "dog", "mouse")){
    print(i)
    }
# You can see this prints cat, dog, and mouse

# We can use this function to loop over the rows in surveys. Let's try that now.
for( i in 1:dim(surveys)[1]){ #This indexing is saying to loop from 1 to the number of rows in surveys - ie ~37K
  if (surveys$year[i] == 1984){ # This is called a nested loop as it is within the other for loop
    print("Great Scott, it's 1984")
  } else {
    print("It's not 1984")
  }
}
```

-

```
# Okay let's do something more useful. First let's create a new data frame
surveys_adjusted <- surveys # This just made of copy of surveys called surveys_adjusted

# Now let's make a new loop where we cahgne the weight variable by 10% if the year is 1984
for( i in 1:dim(surveys_adjusted)[1]){
  if (surveys_adjusted$year[i] == 1984){
    surveys_adjusted$weight[i] <- surveys_adjusted$weight[i] * 1.1
  }


#Okay now that we succesfully built this awesome for loop let's see if it worked!

#Let's look at the mean weight of samples from 1984 in the adjusted and non-adjusted data frames
o_weights <- mean(surveys$weight[surveys$year == 1984], na.rm = T)

a_weights <- mean(surveys_adjusted$weight[surveys_adjusted$year == 1984], na.rm = T)

#Challange acepted: http://erdavenport.github.io/R-ecology-lesson/03-loops-and-functions.html#challenge5

surveys_adj_no_na <- surveys_adjusted[!is.na(surveys_adjusted$weight), ]
big_animals <- 0
for( i in 1:nrow(surveys_adj_no_na)){
    if (surveys_adj_no_na$weight[i] > 28.3 {
        big_animals <- big_animals + 1
        }
        }
big animals

# Time for dplyr and ggplot2: http://erdavenport.github.io/R-ecology-lesson/04-dplyr.html

# Let's load up the tidyverse (code name for dplyr plust some other fun packages to make the
data tidy)
library(tidyverse)

# Let's subset on some columns
select(surveys, plot_id, species_id, weight) # similar to some of our other indexing but it
doesn't need quote

# Let's subset using a slice
select(surveys, month:year)

# Okay now let's try to filter by a row
filter(surveys, year == 1995) # This selects all rows where surveys$year is equal to 1995

# Let's explore the utility of pipes. Let's first do an operation with pipes. First lets filter on
surveys with wieght less than 5
weight5 <- filter(surveys, weight < 5)

# Then select three columns
select(weight5, species_id, sex, weight)

# Now let's try this with a pipe. This remove the intermediate variable and makes things go a
bit faster
surveys_sml <- surveys %>%
  filter(weight < 5) %>%
  select(species_id, sex, weight)


#Challenge - Using pipes, subset the survey data to include individuals collected before 1995
and retain only the columns year, sex, and weight.

surveys %>%
  filter(year < 1995) %>%
  select(year, sex, weight)

# MUTATE - not in the cancer/X-men sense of the word
surveys %>%
  mutate(weight_kg = weight/1000) # This returns a new data frame with an additional
column called wieght_kg. This data frame could be saved

# Mutate looking at only the first few lines
surveys %>%
  mutate(weight_kg = weight/1000) %>%
  head

# Now let's filter out those pesky NAs
surveys %>%
  filter(!is.na(weight)) %>%
  mutate(weight_kg = weight/1000) %>%
  head

  #Challange: http://erdavenport.github.io/R-ecology-lesson/04-dplyr.html#mutate
  surveys %>%
  filter(!is.na(hindfoot_length)) %>%
  mutate(hindfoot_half = hindfoot_length/2) %>%
  filter(hindfoot_half < 30) %>%
  select(species_id, hindfoot_half)

  # Okay, let;s see how we can use a varable to group and summarize. Here we will group by
sex and calculate the mean weight for each grouping
  surveys %>%
  group_by(sex) %>%
  summarise(mean_weight <- mean(weight, na.rm =T))

  # We can also group by multiple categories - here we will group out by sex and species
surveys %>%
  filter(!is.na(weight)) %>%
  group_by(sex, species_id) %>%
  summarise(mean_weight <- mean(weight))

# Let's get our data ready to work in ggplot2 by removing missing values and NAs
surveys_complete <- surveys %>%
  filter(species_id != "", !is.na(weight), !is.na(hindfoot_length), sex != "")

#If you are following along here is the link for the next lesson. Let's get our plot on! http://
erdavenport.github.io/R-ecology-lesson/05-visualization-ggplot2.html

#Let's initialize a ggplot
ggplot(data = surveys_complete) # This should just give a grey background

#Let's add some axis
ggplot(data = surveys_complete,
       aes(x = weight, y = hindfoot_length)) # This should give you the specified x and y axis

# Let's actually make a scatter plot
ggplot(data = surveys_complete,
       aes(x = weight, y = hindfoot_length)) +
  geom_point() # geom_point is the geom for scatterplot - a geom is a shape or graph type in
ggplot

# We can save this base ggplot as an object
```

```
surveys_plot = ggplot(data = surveys_complete,
       aes(x = weight, y = hindfoot_length))

# Now to get the scatter plot we can do this
surveys_plot + geom_point()

# Some of these points are hard to see so let's add an argument to geom_point which will
shade on density
ggplot(data = surveys_complete,
                 aes(x = weight, y = hindfoot_length)) +
  geom_point(alpha = 0.1)

# Now let's add some awesome color
ggplot(data = surveys_complete,
                 aes(x = weight, y = hindfoot_length)) +
  geom_point(alpha = 0.1, color = "blue")

# We can also add another aesthetic (aes) to color by species
ggplot(data = surveys_complete,
       aes(x = weight, y = hindfoot_length)) +
  geom_point(alpha = 0.1, aes(color = species_id))

# Okay, let's try a slighlty differnt graph. Let's look at species_id and hindfoot_length in a
boxplot rather than a scatter plot
ggplot(data = surveys_complete,
       aes(x = species_id, y = hindfoot_length)) +
  geom_boxplot()

# Now lets add some ponts to the boxplot where we show points with a "jitter"
ggplot(data = surveys_complete,
       aes(x = species_id, y = hindfoot_length)) +
  geom_boxplot() + geom_jitter(alpha = 0.3, color = "tomato")

# The boxplot here gets a bit lost, so let's flip jitter and boxplot
ggplot(data = surveys_complete,
       aes(x = species_id, y = hindfoot_length)) +
  geom_jitter(alpha = 0.3, color = "tomato") + geom_boxplot()

# Okay let's combine some of what we already learned and ggplot
yearly_counts <- surveys_complete %>%
  group_by(year, species_id) %>%
  tally # This counts for each group

# Now let's use these data to make a new plot
ggplot(data = yearly_counts,
       aes(x = year, y = n)) +
  geom_line()
  # This does not look like what we want let's do a better job splitting by group and coloring
each line
  # Now let's use these data to make a new plot
ggplot(data = yearly_counts,
       aes(x = year, y = n, group = species_id, color = species_id)) +
  geom_line()


# Let's talk about faceting
ggplot(data = yearly_counts,
       aes(x = year, y = n, group = species_id, color = species_id)) +
  geom_line() +
  facet_wrap(~ species_id) # this breaks each species_id into an individual plot


# Let's prepare our data so we can also group by sex
yearly_sex_counts <- surveys_complete %>%
  group_by(year, species_id, sex) %>%
  tally

# Now let's change our grouping to be by sex instead of species_id
ggplot(data = yearly_sex_counts,
       aes(x = year, y = n, group = sex, color = sex)) +
  geom_line() +
  facet_wrap(~ species_id)

# Now let's tweak some more of our plot parameters. We are going to get rid of the grey
backgound
ggplot(data = yearly_sex_counts,
       aes(x = year, y = n, group = sex, color = sex)) +
  geom_line() +
  facet_wrap(~ species_id) +
  theme_bw() # get's rid of the grey background but keeps the grid lines

  # Let's save this plot
  myplot <- ggplot(data = yearly_sex_counts,
       aes(x = year, y = n, group = sex, color = sex)) +
  geom_line() +
  facet_wrap(~ species_id) +
  theme_bw()
  ggsave(myplot, file = "mysweetplot.pdf")
```