UCLA EEB Carpentries Workshop (Winter 2025)/Code of Conduct

We are dedicated to providing a welcoming and supportive environment for all people, regardless of background or identity. By participating in this community, participants accept to abide by The Carpentries' Code of Conduct and accept the procedures by which any Code of Conduct incidents are resolved. Any form of behaviour to exclude, intimidate, or cause discomfort is a violation of the Code of Conduct. In order to foster a positive and professional learning environment we encourage the following kinds of behaviours in all platforms and events:

- Use welcoming and inclusive language
- Be respectful of different viewpoints and experiences
- Gracefully accept constructive criticism
- Focus on what is best for the community
- Show courtesy and respect towards other community members

If you believe someone is violating the Code of Conduct, we ask that you report it to The Carpentries Code of Conduct Committee by completing this form (https://docs.google.com/forms/d/e/1FAIpQLSdi0wbplgdydl_6rkVtBIVWbb9YNOHQP_XaANDClmVNu0zs-w/viewform), who will take the appropriate action to address the situation.

For more information: https://docs.carpentries.org/policies/coc/

Day 2 Schedule

Instructors: Kaija Gahm and Peter Laurin
Helpers: Xochitl Ortiz Ross and Heidi Yang

08:30-08:45 Settle in
08:45-08:50 Introducing the Shell
08:50-09:30 Navigating Files and Directories
09:30-10:20 Working With Files and Directories
10:20-10:30 **BREAK**
10:30-11:05 Pipes and Filters
11:05-11:55 Loops
11:55-1:00 **LUNCH**
1:00-1:45 Shell Scripts
1:45-1:55 **BREAK**
1:55-2:40 Finding Things

Sign in:

Name/email/pronouns/(optional) your favorite ice cream flavor or ice cream alternative
Xochitl Ortiz Ross /xortizross@g.ucla.edu/she, they/ chocolate with chocolate bits (brownie or cookie etc)
Heidi Yang / hyangg@g.ucla.edu / they, she / matcha or coffee

Claire Geiman / geiman@ucla.edu / she,her / Ben & Jerry's Half-Baked
Bear Waymire/ BearWaymire@ucla.edu / He, Him / Maple Sea Salt or anything Caramel!
Erika Ono-Kerns/eonokerns@gmail.com/she,her/chocolate chip cookie dough
Kristie Schott/kcschott@ucla.edu/she,her/Maine black bear
Peter Laurin / peterlaurin@g.ucla.edu / he,him / Goat Cheese with Red Cherries
Sasha Gomes / sashagomes@g.ucla.edu / she/her / lavender honey
Richie Ngoy / richiengoy@gmail.com / he/him / green tea
Tai Vuong/vtai2027@ucla.edu/he,him/coffee
Jasmine Herrera/herrera.jasmine159@gmail.com/ she/her / mint chip

(Optional) Feedback from yesterday:

Yesterday I learned:

I am still confused by:

List of commands used so far

Navigating directories:

List --
ls

Print working directory --
pwd

ls -F Desktop
ls -F Desktop/shell-lesson-data
man

Change working directory, going down a hierarchy
cd Desktop
cd shell-lesson-data
cd exercise-data

Move to the parent directory
cd ..

Use -a to show hidden files (like the ..)
ls -F -a

ls -s

ls -S

Working with files and directories:

Make a new directory called thesis--
mkdir thesis

Making directories with subdirectories--
mkdir -p ../project/data ../project/results

List the new diretory hierarchy
ls -FR ../project

Recursively list things
-R

Run a text editor (vim) to create a file called draft.txt
vim draft.txt

To exit the text editor
:q

Create a new empty file witout opening the editor
touch my_file.txt

Remove a file
rm my_file.txt

Rename a file by "moving it" to the same location but with a new name
mv thesis/draft.txt thesis/quotes.txt

You can ask bash to ask for confirmation before overwriting files
mv -i

Copy a file (and change the name)
cp quotes.text thesis/quotations.txt

Copy a directory and all its contents by using the recursive -r
cp -r thesis thesis_backup

Wildcard: matches "everything" including nothing
*

List anything that "ends with" ethane.pdb
ls *ethane.pdb

Wildcard: matches one character
?

List anything that has any 1 character in front of ethane.pdb
ls ?ethane.pdb

Pipes and filters

Word count
wc

Count the number of lines per file
wc -l

Count the numbers of characters
wc -m

Count the number of words
wc -w

Redirect the output to a file with >
wc -l *.pdb > lengths.txt

Concatenate - to show the output of the file
cat lengths.txt

Sort contents
sort

Sort numbers
sort -n

Sort and then save to a file
sort -n lengths.txt > sorted-lengths.txt

Show the first 10 lines
head

Specify the number of lines to show (e.g., 1)
head -n 1

Append to an existing file
>> instead of >

Pipe (use the output from the left as input to the right)
|

Pipe example: give me the file with the fewest line
wc -l *.pdb | sort -n | head -n 1

Loops

for *thing* in *list_of_things*
do

- *command* $*thing*

done

Pint the value of a variable
echo

Interrupt the loop
Ctrl + C

Shell Scripts

vim file.sh
to open the text editor and create a new shell script
i
to start writing in the editor
esc
to stop typing in the editor
:wq
to save and quit the editor

Run the script
bash file.sh

Run an R script using
Rscript file.R

Placeholder for an argument in a script that can be specified when you run it
$1 $2 $3 [for 3 arguments]

Placeholder for "all of the command-line arguments"
$@

selects a column (2nd) from a comma-delimited file (-d,)
cut -d, -f 2 animals.csv

select unique values
uniq

Take Notes!

Use this space to **collaboratively take notes**, ask questions, clarify confusions, or share additional

resources.

Set up page: https://swcarpentry.github.io/shell-novice/instructor/index.html#setup
Download the data, unzip the file and save it on your Desktop

Introducing the Shell

Objectives

- Explain how the shell relates to the keyboard, the screen, the operating system, and users' programs.
- Explain when and why command-line interfaces should be used instead of graphical interfaces.

Unix shell is both a command-line interface (CLI) and a scripting language, allowing for repetitive tasks to be done automatically and fast.

The shell is a program where users can type commands. Bash is the most popular Unix shell and is the default shell on most modern implementations of Unix.

While a graphical user interface (GUI) allows you to interact with your computer by pointing and clicking and gives you options to click, CLI doesn't give you the options so you must learn the commands.

You can change your current shell to Bash by typing bash and then pressing Return. To check your current shell type echo $0and press Return. To change your default shell to Bash type chsh -s /bin/bash and press the Return key, then reboot for the change to take effect. To change your default back to Zsh, type chsh -s /bin/zsh, press theReturn key and reboot. To check available shells, type cat /etc/shells.

When you open the shell, you will be presented with a prompt, often a $, this means the shell is waiting for input. **Do not type the prompt as part of a command**. We will be using $ to represent the prompt.

First command--type: ls

Short for listing, this will list the contents of your current directory (or folder).

Nelle Nemo is a marine biologist. She has to run 1520 files through a program. Instead of selecting and opening each file by hand, she decides to have her computer do it for her so she can focus on writing up the results.

KEY POINTS

- A shell is a program whose primary purpose is to read commands and run other programs.
- This lesson uses Bash, the default shell in many implementations of Unix.
- Programs can be run in Bash by entering commands at the command-line prompt.
- The shell's main advantages are its high action-to-keystroke ratio, its support for automating repetitive tasks, and its capacity to access networked machines.
- A significant challenge when using the shell can be knowing what commands need to be run and

how to run them.

Navigating Files and Directories

Objectives

- Explain the similarities and differences between a file and a directory.
- Translate an absolute path into a relative path and vice versa.
- Construct absolute and relative paths that identify specific files and directories.
- Use options and arguments to change the behaviour of a shell command.
- Demonstrate the use of tab completion and explain its advantages.

Command: pwd

Stands for "print working directory" and it lets you know which directory you are working in.

On Nelle’s computer, the filesystem looks like this:
"/" - the root directory
"bin" - stores built-in programs
"data" - for data files
"Users" - the users' personal directories
"tmp" - for temporary files

Command: ls -F

tells ls to classify the output by adding a marker to file and directory names to indicate what they are

- a trailing / indicates that this is a directory
- @ indicates a link
- * indicates an executabl

You can clear your terminal using the "clear" command. You can use the up and down arrows to move between lines.

- **Asking for help**
- For Linux and Git Bash type: ls --help
- For Linux and macOS type: man ls
- To search for a character or word in the man pages, use / followed by the character or word you are searching for. To quit the man pages, press q.

We can also use ls to explore other directories, e.g., Desktop

Command: ls -F Desktop

We can also look at the contents of the folders within

Command: ls -F Desktop/shell-lesson-data

To change locations we use cd (change directory) followed by a directory name. It is like double-clicking a folder.

Commands:
$ cd Desktop
$ cd shell-lesson-data
$ cd exercise-data

This only works to go **down** a directory tree. To go up we use cd .. (the directory containing this one, or the parent directory)

Remember: you can use pwd to check where you are.

You can use "ls -F -a" to show the special directory "..". -a stands for "show all"

cd without anything afer it will return you to your home directory.

Remember: each command "element" is separate by a space
ls (command) -F (option) / (argument)

KEY POINTS

- The file system is responsible for managing information on the disk.
- Information is stored in files, which are stored in directories (folders).
- Directories can also store other directories, which then form a directory tree.
- pwd prints the user's current working directory.
- ls [path] prints a listing of a specific file or directory; ls on its own lists the current working directory.
- cd [path] changes the current working directory.
- Most commands take options that begin with a single -.
- Directory names in a path are separated with / on Unix, but \ on Windows.
- / on its own is the root directory of the whole file system.
- An absolute path specifies a location from the root of the file system.
- A relative path specifies a location starting from the current location.
- . on its own means 'the current directory'; .. means 'the directory above the current one'.

Working with Files and Directories

Objectives

- Create a directory hierarchy that matches a given diagram.
- Create files in that hierarchy using an editor or by copying and renaming existing files.
- Delete, copy and move specified files and/or directories.

To change your terminal appearance:

- Preferences > Profiles > choose your fighter!

we can use a tilde "~" to indicate the home directory

- we can access our data using cd ~/Desktop/shell-lesson-data

options for commands:

- they change the behavior of a commond
- short option: use one dash "-"
- long option: two dashes "--", like the --help option
- you can use multiple dashes
- these are also called "flags"
- commands must separated from options by a space
- commands and options are both case-sensitive

$ ls -s

- this shows the file size of files in your working directory

$ ls -S

- this sorts your files by size

a shortcut: the tab is your friend <3 - if you press Tab, it will auto-complete the file name

- pushing Tab twice will show multiple matches for the letter (if there are multiple matches)
- such as if you type $ ls D Desktop/ Downloads/ will both show up

$ cd ~/Desktop/shell-lesson-data/exercise-data/writing/

to make a new directory called "thesis" in the writing directory, use the command
$ mkdir thesis
$ ls -F thesis

- this will show nothing, which means it's empty!

$ mkdir -p ../project/data ../project/results

- the "-p" option allows you to make multiple directories simultaneously

^ this can be done even if the project directory doesn't exist, i.e. you can make both a directory and subdirectories at the same time
$ cd .. <- using ".." will go up one level / up one directory
$ ls -FR ../project/

- this will list both what is in the project and all the files in subdirectories

be careful with spaces! you might want to make a directory called "north pacific gyre", but if you say mkdir north pacific gyre, you will be making 3 directories. Instead, you can put underscores or dashes like north_pacific_gyre or north-pacific-gyre

$ cd thesis

- change directory to thesis

to access a text editor (there are multiple - we are going to use Vim):
$ vim draft.txt

- this will open up a text editor that will make a file called draft.txt
- to exit the vim text editor, press :q then Enter
- to exit out of editing mode, press esc
- to save our file, press :wq then Enter
    - w means "save"
    - you can do w and q separately or together

we can also make a file using the following command:
$ touch my_file.txt

- this makes a new empty file

$ ls -l

- shows the size of the files in your directory and time that you made them

to remove a file:
$ rm my_file.txt

Files have filename extenstions, like ".docx" for Microsoft Word documents - this helps computer programs know what kinds of files these are

- .txt = normal text file
- .png = image
    - we can't edit this in the terminal
- just because we have manually made an extentions *does not* mean that it is that type of file

if you want to change the name of a file:
$ mv thesis/draft.txt thesis/quotes.txt

- the first argument is the file we want to "move" or rename
- the second argument is the filename you want to change it to
- mv is a **permanent operation**!! Make sure that there are no other files with the same name, since it will overwrite it
- to be safe, you can use the option "-i" to be prompted whether or not you want to follow through with your command (remember you can't undo commands in bash)

$ mv thesis/quotes.txt .

- what is this dot?
    - .. = the previous directory
    - . = the current directory

$ ls -a

- will show you . .. as well as all files and directories in the current working directory

if we want to copy a file:
$ cp quotes.txt thesis/quotations.txt

- this copies the quotes.txt in the thesis directory with the name "quotations.txt"

to copy a directory:
$ cp -r thesis/ thesis_backup/

- the "-r" option is recursive - you have to use this for a directory

Exercise - we've made a file but we've made a typo in the filename. How do we fix it?
$ touch statstics.txt

options:
$ cp statstics.txt statistics.txt

- will copy the file under a different name "statistics.txt"

mv statstics.txt statistics.txt

- **what we want! will rename the file :)**

mv statstics.txt .

- it won't do anything - it's saying let's move statstics.txt to our current working directory
- bash will return saying that the file is identical

cp statstics.txt .

- this will do the same thing as mv statstics.txt . (nothing will happen and you will get an error message that the file is identical)

we can now remove a file:
$ rm statstics.txt

- remember this cannot be undone! Be careful with this command and back up your files

$ rm -r thesis/

- this will remove the thesis directory and everything inside

wildcards!

- asterisk or * will match anything
    - if we want to find all the files in the alkanes directory ending in .pdb, we can do:
    - $ ls *.pdb
    - let's say we do ls *ethane.pdb:
        - ethane.pdb and methane.pdb
        - * can match zero characters!
- question mark or ? will match exactly one character (which can be anything)
- $ ls ???ane.pdb
    - will result in all files with three characters before ane.pdb

to clear your screen without removing all previous commands: ctrl + L

Exercise - which commands will output ethane.pdb and methane.pdb?

1. ls *t*ane.pdb

- this will result in ethane.pdb methane.pdb but also octane.pdb and pentane.pdb

3. ls *t?ne.*

- this will result in pentane.pdb and octane.pdb, since this specifies just one character between "t" and "ne"

5. ls *t??ne.pdb

- this is the right answer! you have any characters before t, but by specifying two characters between "t" and "ne" we filter just for ethane.pdb and methane.pdb

7. ls ethane.*

- this will only give us ethane.pdb

Key Points

- cp [old] [new] copies a file.
- mkdir [path] creates a new directory.
- mv [old] [new] moves (renames) a file or directory.
- rm [path] removes (deletes) a file.
- * matches zero or more characters in a filename, so *.txt matches all files ending in .txt.
- ? matches any single character in a filename, so ?.txt matches a.txt but not any.txt.
- Use of the Control key may be described in many ways, including Ctrl-X, Control-X, and ^X.
- The shell does not have a trash bin: once something is deleted, it’s really gone.
- Most files’ names are something.extension. The extension isn’t required, and doesn’t guarantee anything, but is normally used to indicate the type of data in the file.
- Depending on the type of work you do, you may need a different text editor than vim.

Pipes and Filters

Objectives

- Explain the advantage of linking commands with pipes and filters.
- Combine sequences of commands to get new output
- Redirect a command’s output to a file.
- Explain what usually happens if a program or pipeline isn’t given any input to process.

$ wc cubane.pdb

- this command stands for "word count" but this will output line count, word count, and character count

$ wc *.pdb

- gives us line, word, character count for all files ending in .pdb

$ wc -l *.pdb

- the "-l" option gives only the line count
- "-w" for words
- "-m" for characters

$ wc -l *.pdb > lengths.txt

- redirects the output of the command into a new file called "lengths.txt" instead of printing it to the screen (default)
- note! if the file "lengths.txt" already exists, **it will overwrite this previous file**

$ cat lengths.txt

- print whatever is in the file to the screen
- stands for concatenate - so it concatenates the entire file and prints it to the screen

let's say we want to sort the file contents:
$ sort numbers.txt

- we get some weird number order that's not in ascending numerical order

$ sort -n numbers.txt

- this is because we are sorting by *numerical* order, not alphanumerical order (what sort without the -n option does)

$ sort -n lengths.txt > sorted_lengths.txt

- this makes a new file with the output of the sort -n command

to see the first X lines of a file:
$ head sorted_lengths.txt

- default is 10 lines

$ head -n 1 sorted_lengths.txt

- use the -n option + a number to specify the number of lines you want to see

you can use >> to append to an existing file:
$ sort -n lengths.txt >> sorted_lengths.txt

it's kind of annoying to make a bunch of intermediate files. Let's use pipes (|):
$ sort -n lengths.txt | head -n 1

- this will "pipe" the output of the sort -n command into the head -n 1 command

we can use multiple pipes too:
$ wc -l *.pdb | sort -n | head -n 1

- this will give us the .pdb file with the fewest line numbers

this can all be done for simple text files!

$ cd animal-counts
$ cat animals.csv

$ cat animals.csv | head -n 5 | tail -n 3 | sort -r > final.txt

- step by step:
    - concatentate, or show, the contents of animals.csv
    - show the first 5 lines
    - show the last 3 of the first 5 lines (remember that we've piped only the first 5 from the file)
    - sorts by reverse since we've used the -r option - this sorts by the date since it's the first column, then by the alphabet in the second column
    - redirect the final output to a file called "final.txt"

Let's head into Nelle's North Pacific Gyre dataset:
$ cd ~/Desktop/shell-lesson-data/north-pacific-gyre/

$ wc -l *.txt | sort -n | head -n 5

- this will count all the lines in all .txt files, sort by ascending order, and then choose the first five

Key Points

- wc counts lines, words, and characters in its inputs.
- cat displays the contents of its inputs.
- sort sorts its inputs.
- head displays the first 10 lines of its input by default without additional arguments.
- tail displays the last 10 lines of its input by default without additional arguments.
- command > [file] redirects a command’s output to a file (overwriting any existing content).
- command >> [file] appends a command’s output to a file.
- [first] | [second] is a pipeline: the output of the first command is used as the input to the second.
- The best way to use the shell is to use pipes to combine simple single-purpose programs (filters).

Loops

Objectives

- Write a loop that applies one or more commands separately to each file in a set of files.

- Trace the values taken on by a loop variable during execution of the loop.
- Explain the difference between a variable’s name and its value.
- Explain why spaces and some punctuation characters shouldn’t be used in file names.
- Demonstrate how to see what commands have recently been executed.
- Re-run recently executed commands without retyping them.

what if we want to perform a command on multiple files or multiple times? We use a loop for that

pseudocode:
for thing in list_of_things
do
command $thing
done

- the for loop is running through each thing in list_of_things and doing a commond for each thing
- format:
    - **for __ in __**:
        - **do** {command}
    - **done**

here's an example:
for filename in basilisk.dat minotaur.dat unicorn.dat

- do
    - echo $filename
    - head -n $filename | tail -n 1
- done

- in this case, we refer to what we are looping over using the $ within the for loop, which you define in the blank in for __
    - you can call it anything you want, but it's best to use something relevant to what you're looping over, like "filename" or "animal"
- we can type out the files, or really anything, that we want to loop over after "in"
    - sometimes it's helpful to use a sequence:
        - you can do {0..9} - this makes a sequence of 0 to 9 with a step size of 1
        - this is useful if you want to loop over specific numbers or files

for datafile in *.pdb

- do
    - ls *.pdb
- done

- this will list out all files that end in *.pdb for the number of files that match *.pdb
    - we have 6 files that match, so we would see all files listed 6 times

for alkanes in *.pdb

- do

        - echo $alkanes
        - cat $alkanes > alkanes.pdb
- done

what's happening?

- for every "alkane" that is in the list of the files that end in .pdb
- we are first printing, or "echoing", the file name
- then we are printing the contents of the file name and then redirecting it to a file name called "alkanes.pdb"
    - remember that we use ">" it will overwrite any existing files that have the name alkanes.pdb
    - so we'll only get the last file included in alkanes.pdb
- instead you can use ">>" to append to the same file for each iteration of the for loop

Tip: you can use the command “less” like you would use the command “cat”. You can scroll through the file, and once you’re done you can press q to exit out.

if we wanted to add a prefix to each file, we can't just use cp command with wildcards, since it thinks you're referring to a directory. Instead we need a for loop
for filename in *.dat

- do
    - cp $filename original-$filename
- done
- this copies the files into a new file with the "original-" prefix

Tip: general good practice is to do some sort of printing mechanism like "echo $x" to check where things might go wrong should there be a bug in your command

Let's make a for loop to process through some data files:
for datafile in NENE*A.txt NENE*B.txt

- do
    - echo $datafile
- done
- just doing an echo for loop can check whether it's doing what you want it to do

now let's check if the datafile name would have "stats-" added to the name:
for datafile in NENE*A.txt NENE*B.txt

- do
    - echo $datafile stats-$datafile
- done

lastly, let's run a script on all of our data files:
for datafile in NENE*A.txt NENE*B.txt

- do
    - bash goostats.sh $datafile stats-$datafile

- done

what about a nested loop?
for species in unicorn minotaur frog

- do
    - echo $species
    - for temperature in 25 30
    - do
        - echo $species feels warm in $temperature
    - done
- done

- remember that you need to explicitly define different variables in nested loops
- for any nested for loops, also make sure to be explicit about what you're looping through!

Key Points

- A for loop repeats commands once for every thing in a list.
- Every for loop needs a variable to refer to the thing it is currently operating on.
- Use $name to expand a variable (i.e., get its value). ${name} can also be used.
- Do not use spaces, quotes, or wildcard characters such as '*' or '?' in filenames, as it complicates variable expansion.
- Give files consistent names that are easy to match with wildcard patterns to make it easy to select them for looping.
- Use the up-arrow key to scroll up through previous commands to edit and repeat them.
- Use Ctrl+R to search through the previously entered commands.
- Use history to display recent commands, and ![number] to repeat a command by number.

Shell Scripts

Objectives

- Write a shell script that runs a command or series of commands for a fixed set of files.
- Run a shell script from the command line.
- Write a shell script that operates on a set of files defined by the user on the command line.
- Create pipelines that include shell scripts you, and others, have written.

- A script can be made to save a series of operations - these can be considered like small programs
- everything under the hood on your computer is basically a series of shell scripts!
- this makes computing (and your life) much easier and more efficient
- also good for reproducibility

we need to use a text editor to create our script (vim is one option):
$ vim middle.sh

- then press i to start writing
- when you're done writing, press esc to stop writing
- then press :wq to save and quit out of vim

to run our new script we can do the following:
$ bash middle.sh

- the command bash is looking for a plain text file

we can specify arguments in our script:
$ vim middle.sh

- add a $1 to indicate you are using the first argument you put following the script name when you call it.
- you can use any number of arguments, $1, $2, etc.

in our script, we can put that we expect 3 arguments:
head -n $2 $1 | tail -n $3

- $1 = the file name
- $2 = the number of lines we want printed at the header
- $3 = the number of lines from the tail of the subset lines from the first command

then we can use the script!
$ bash middle.sh propane.pdb 15 5

we can also add comments in our script using # as follows:
Select lines from the middle of a file.
Usage: bash middle.sh filename end_line num_lines
head -n $2 $1 | tail -n $3

- these are very helpful when sharing your scripts with others and with your future self

we can use $@ to take in all arguments

we can subset a file and identify a column in our file:
$cut -d , -f 2 animals.csv | sort | uniq

- -d = delimiter option. In this case we specify our separator is a comma
- -f = column number. In this case we specify the second column
- sort them alphabetically
- get the unique values in the second column

lastly, you can refer to shell scripts in a shell script!

```
for datafile in $@
do
  echo $datafile
  bash goostats.sh $datafile stats-$datafile
done
```

- now you can call the stats command on the files you specify in your larger script command

---

Day 1 Schedule

Instructors: Kaija Gahm and Xochitl Ortiz Ross
Helpers: Peter Laurin

08:30-09:00 - Introductions, installation help, and settle in
09:00-09:45 - Introduction to R and RStudio
09:45-11:15 - Data visualization with ggplot
11:15-11:25 - **BREAK**
11:25-12:25 - Exploring and understanding data
12:25-1:25 - **LUNCH**
1:25-2:25 - Working with data (part 1)
2:25-2:35 - **BREAK**
2:35-3:35 - Working with data (part 2)

Sign in:

Name/email/pronouns/(optional) fun fact about yourself
Kaija /kgahm@ucla.edu/she,her
Xochitl (só-chill)/ xortizross@g.ucla.edu / she,they / this is my first workshop as a certified Carpentries instructor!
Claire / geiman@ucla.edu / she,her / I enjoy wood/metalworking
Maya Sagarin/msagarin@ucla.edu/she;her
Erika/eonokerns@ucla.edu/she,her/
Alex/ alcisneroscarey@g.ucla.edu/ she/they/ big fan of tinned fish
Jeremy/ jeremy.m.gingras@gmail.com/ he/him
George/ougeorge21@gmail.com/he, him

Kristie/kcschott@ucla.edu/sheher

Ibrahim/bruinsurfer@g.ucla.edu
Bear Waymire/bearwaymire@ucla.edu /He/Him /
Bella/awatson1@g.ucla.edu/she her
Sasha Gomes/ sashagomes@ucla.edu/she/her
Aliya Alidaee/aliyakalidaee@g.ucla.edu/she,her
Richie Ngoy/ richiengoy@gmail.com/ he/him
Auxenia Grace / aprivettmendoza@ucla.edu / she/her
Peter Laurin / peterlaurin@g.ucla.edu / he,him / fun fact: my dog is in the room with me, so if I seem distracted that's why!
Jasmine Herrera / herrera.jasmine159@gmail.com / she,her
Nico Andrade / nicoandrade@ucla.edu / he/him/his / I love playing saxophone!
Tai Vuong/vtai2027@ucla.edu/he,him

Covered functions:

Help keep track of which functions have been covered in the lesson and what package they come from.

Take Notes!

Use this space to collaboratively take notes, ask questions, clarify confusions, or share additional resources.
I have some stuff already on R studio from an assignment from a stats class like 2 years ago. How do I clear environment on R?
Peter: If you go to the menu on top and go to "Session --> Restart R" that will clear your environment, remove any plots, clear your memory, etc. However, if you have scripts open, you'll have to close those manually as well.
You can click the little broom icon!

Introduction to R and R Studio

Objectives

- Understand the difference between R and RStudio
- Describe the purpose of the different RStudio panes
- Organize files and directories into R Projects
- Use the RStudio help interface to get help with R functions
- Be able to format questions to get help in the broader R community

It is good practice to keep your projects self-contained, the best way to do this in R is to create an R Project. This will automatically set your working directory as the top folder of the project whenever you open the R project.

Sub-folders within R project - best practice to keep raw and cleaned data separate to ensure raw data is never modified and cleaned data is reproducible from scripts

Scripts are where you keep track of what you did and it allows you to repeat all the same steps again in the future.

Use a # to write comments and notes in your script.

Data visualization with ggplot

Objectives

- Produce scatter plots and boxplots using ggplot2.
- Represent data variables with plot components.
- Modify the scales of plot components.
- Iteratively build and modify ggplot2 plots by adding layers.
- Change the appearance of existing ggplot2plots using premade and customized themes.

- Describe what faceting is and apply faceting inggplot2.
- Save plots as image files.

You can install the ggplot2 package using the code `install.packages("ggplot2")
Load the package using library(ggplot2)
Installing happens just once, like downloading a program from the internet. Loading happens every time (so, library() calls should go in your script) to allow you to access the package.

The ratdat package contains data from the Portal Project

- Read more about the Portal Project here: https://portal.weecology.org/

See documentation for a function by typing its name with a question mark in front of it. For example, ? complete_old shows the documentation for the complete_old dataset in ratdat.

Exploring your dataset

View() shows a dataset in R in tabular form
str() shows the structure of the dataset
The $ in the dataset output indicates a column

Plotting basics

We'll use `ggplot` as a function to tell ggplot2 package to create a plot.
We can specify the "mapping" of variables to the plot using `aes()` (for example, x axis, y axis, color)
To specify a specific graphic to be associated to the mapping we denoted, we can add a "geom". For example, `geom_point` plots a scattergraph of points.
Note: R doesn't care much about indentations/newlines, so you can indent your code to make it fit better on your screen.

Alpha changes the transparency: values vary between 0 and 1
Resource for changing colors: https://derekogle.com/NCGraphing/resources/colors
other commonly-used aes for geom_point: shape, size, alpha (transparency)

Exercise 1:

Challenge: try modifying the plot so that the shape of the point varies by sex
Hint: you will set the shape the same way you set the color
Bonus: do you think this is a good way to represent sex with these data?

```
ggplot(data = complete_old, mapping = aes(x = weight, y = hindfoot_length, shape = sex))
+ geom_point()
```

Not a very clear way to represent these data!

- Could also add color, add facets (separating the points into different graphs by sex)

Exercise 2 (bonus, optional):

Now try changing the plot so that the color of the points varies by year. Do you notice a difference in the color scale compared to changing color by plot type? Why do you think this happened?

```
ggplot(data = complete_old, mapping = aes(x = weight, y = hindfoot_length, color = year))
+ geom_point()
```

We notice that the color scale is now continuous instead of discrete! Ggplot2 understands whether the variable you're using is discrete or continuous and it picks the colors accordingly.

A new layer: scales

By adding `scale_(...)` (fill in the ... for yourself) we can change many default features of how a geom (graphic) plots the data.
For example, scale_color_viridis_d() will change geoms to use the "viridis" palette for the color aesthetic instead of the default.

Scales are useful for other things too. For example, let's change the x axis scale to logarithmic

```
ggplot(data = complete_old, mapping = aes(x = weight, y = hindfoot_length, color = plot_type))
+ geom_point()+ scale_color_viridis_d()+scale_x_log10()
```

A different type of plot: boxplots!

```
ggplot(data = complete_old, mapping = aes(x = plot_type, y = hindfoot_length))
+ geom_boxplot()
```

Quick detour: how to make the labels look prettier!
We add scale_x_discrete() because we're going to modify the

```
ggplot(data = complete_old, mapping = aes(x = plot_type, y = hindfoot_length))
+ geom_boxplot() + scale_x_discrete(labels = label_wrap_gen(width = 10))
```

Changing the color:

```
ggplot(data = complete_old, mapping = aes(x = plot_type, y = hindfoot_length, color = plot_type))
+ geom_boxplot() + scale_x_discrete(labels = label_wrap_gen(width = 10))
```

- color only controls the 1-dimensional aspects of the plot, such as points and lines
- to change the fill color, we use "fill = " instead of "color = "

```
ggplot(data = complete_old, mapping = aes(x = plot_type, y = hindfoot_length, fill = plot_type))
+ geom_boxplot() + scale_x_discrete(labels = label_wrap_gen(width = 10))
```

Adding points on top of the boxplot

ggplot(data = complete_old, mapping = aes(x = plot_type, y = hindfoot_length)) + geom_boxplot() + scale_x_discrete(labels = label_wrap_gen(width = 10)) + geom_point()

The problem with this is that all the points are in a single line!
Let's jitter them instead, using geom_jitter

ggplot(data = complete_old, mapping = aes(x = plot_type, y = hindfoot_length)) + geom_boxplot() + scale_x_discrete(labels = label_wrap_gen(width = 10)) + geom_jitter()

This kind of obscures the plot. Instead, let's add geom_jitter first and then the boxplot.
ggplot(data = complete_old, mapping = aes(x = plot_type, y = hindfoot_length)) + geom_jitter() + geom_boxplot() + scale_x_discrete(labels = label_wrap_gen(width = 10))

Other layers below are affected by changes to layers at the top. For example, let's change the color of everything:
ggplot(data = complete_old, mapping = aes(x = plot_type, y = hindfoot_length, color = plot_type)) + geom_jitter() + geom_boxplot() + scale_x_discrete(labels = label_wrap_gen(width = 10))

Additional arguments for each layer can be found by looking up the documentation for that function (e.g. ?geom_boxplot).
To get rid of outliers in the boxplot, we can use:
ggplot(data = complete_old, mapping = aes(x = plot_type, y = hindfoot_length, color = plot_type)) + geom_jitter() + geom_boxplot(outliers = FALSE) + scale_x_discrete(labels = label_wrap_gen(width = 10))

Challenge 2: Change geoms

Violin plots are similar to boxplots- try making one using plot_type and hindfoot_length as the x and y variables. Remember that all geom functions start with geom_, followed by the type of geom.

This might also be a place to test your search engine skills. It is often useful to search for R package_name stuff you want to search. So for this example we might search for R ggplot2 violin plot.

Answer:
ggplot(data = complete_old, mapping = aes(x = plot_type, y = hindfoot_length)) +
geom_jitter(aes(color = plot_type)) +
geom_violin(fill=NA) +
scale_x_discrete(labels = label_wrap_gen(width=10))

Some key takeaways

- The base layer includes the data and the mapping (which variable goes to which axis)
- We can change scales to change how our data is represented
- We can add layers using the + sign. There are many different layers, such as boxplots, scatterplots, jitter plots, violins, etc.

- We can also change the theme!

Changing the theme (background, etc)

First, let's save the plot that we made as an "object" using the assignment arrow

```
myplot <- ggplot(data = complete_old, mapping = aes(x = plot_type, y = hindfoot_length)) +
    geom_jitter(aes(color = plot_type)) +
    geom_violin(fill=NA) +
    scale_x_discrete(labels = label_wrap_gen(width=10))
```

And now we can change the theme
myplot + theme_bw() # black and white theme
myplot + theme_classic() # classic theme

We can also modify specific elements of the theme

myplot + theme_bw() +

- theme(axis.title. = element_text(size = 14), # make the axis titles larger
- panel.grid.major.x = element_blank(), # Remove some of the gridlines

  legend.position = "none") # remove the legend

Note: there are various ways of removing things in ggplot2; it's not always consistent. Google is your friend.

Changing the labels

myplot + theme_bw() + labs(title = "Rodent size by plot type",

- x = "Plot type",
- y = "Hindfoot length (mm)")

Making several plots, separated by the values of a variable

myplot + facet_wrap(vars(sex), ncol = 1) # make small "facets" of the plot by sex and arrange them in a single column

Bonus exercise:

Challenge 4: Make your own plot!

Try making your own plot! You can run str(complete_old) or ?complete_old to explore variables you

might use in your new plot. Feel free to use variables we have already seen, or some we haven't explored yet.
Here are a couple ideas to get you started:

- make a histogram of one of the numeric variables
- try using a different color scale_
- try changing the size of points or thickness of lines in a geom

Exploring and Understanding Data

Objectives

- Explore the structure and content of data.frames
- Understand vector types and missing data
- Use vectors as function arguments
- Create and convert factors
- Understand how R assigns values to objects

Make a new script

As a reminder, scripts are separate files that store and execute code independently. There are several ways to open a new script, including in the top left corner, or with command + shift + N on a mac.

Many people open scripts with comments in a small header, including their name, the date and the purpose of the script

The data.frame

The "class" function allows you to check what **type** an object is

The "head" function prints the first 6 rows (by default) of a data.frame or other object
"tail" prints the last 6 rows of a data.frame

> head(complete_old)

A tibble: 6 × 13

	record_id	month	day	year	plot_id	species_id	sex	hindfoot_length	weight	genus	species	taxa
	<int>	<int>	<int>	<int>	<int>	<chr>	<chr>	<int>	<int>	<chr>	<chr>	<chr>
1	1	7	16	1977	2	NL	M	32	NA	Neotoma	albigu…	Rode…
2	2	7	16	1977	3	NL	M	33	NA	Neotoma	albigu…	Rode…
3	3	7	16	1977	2	DM	F	37	NA	Dipodom…	merria…	Rode…
4	4	7	16	1977	7	DM	M	36	NA	Dipodom…	merria…	Rode…
5	5	7	16	1977	3	DM	M	35	NA	Dipodom…	merria…	Rode…
6	6	7	16	1977	1	PF	M	14	NA	Perogna…	flavus	Rode…

Functions look for particular arguments. For example, head expects an object, x. However, because we put `complete_old` in first, head knows that `complete_old` is x. An additional argument, n, looks for the number of lines to print.

> head(complete_old, n = 2)

> head(complete_old, n = 2)
A tibble: 2 × 13

	record_id	month	day	year	plot_id	species_id	sex	hindfoot_length	weight	genus	species	taxa
	<int>	<int>	<int>	<int>	<int>	<chr>	<chr>	<int>	<int>	<chr>	<chr>	<chr>
1	1	7	16	1977	2	NL	M	32	NA	Neotoma	albigula	Rode…
2	2	7	16	1977	3	NL	M	33	NA	Neotoma	albigula	Rode…

If we name arguments, we can put them in any order.

> head(n = 2, x = complete_old) # this works

Some functions to learn more about your data.frame:
summary() # gives a summary of each variable (e.g. mean, min, max)
str() #structure, gives the dimensions of the df, as well as the type and first few entries for each variable

The $ operator allows us to extract individual columns from a data.frame
complete_old$year

Vectors: the building block of data

Each column in a dataframe is made up of a vector. Many things in R are vectors (or vectors stitched together).

Atomic vector types: integer c(-2L, -1L, 0L, 1L, 2L, ...) , character c("hi there", "no way"), numeric c(-0.2, -0.1, 0.0, 0.1, 0.2), logical c(TRUE, FALSE, FALSE, TRUE)

Vectors can only be 1 **type.** Multiple types inserted into one vector will be "coerced" into the most compatible type for those data (see below).

The assignment operator ("<-") will assign an object to a variable, so we can use it later.
> num <- c(1, 2, 5, 12, 4)
> class(num)
[1] "numeric"

Challenge 1: Coercion

Since vectors can only hold one type of data, something has to be done when we try to combine different types of data into one vector.

1. What type will each of these vectors be? Try to guess without running any code at first, then run the code and use class() to verify your answers.

```
num_logi <- c(1, 4, 6, TRUE)
num_char <- c(1, 3, "10", 6)
char_logi <- c("a", "b", TRUE)

tricky <- c("a", "b", "1", FALSE)
```

Answers:
> class(num_logi) #all can become numeric, TRUE -- > 1
[1] "numeric"
> class(num_char) # all can become char
[1] "character"
> class(char_logi) # all can become char
[1] "character"
> class(tricky) # all can become char
[1] "character"

Missing data

Missing data in R is represented explicitly by NA, and does not count as a data type for the purposes of coercion.

weights <- c(25, 34, 12, NA, 42) # this is valid!
R is careful not to assume what to do with missing data. For example,
> min(weights)
[1] NA
In effect, we do not know the minimum because there is missing data! To rectify this, and explicitly acknowledge our missing data, we can use:
> min(weights, na.rm=TRUE)
[1] 12

Vectors as arguments

Vectors are often used as arguments in functions themselves.
`> quantile(complete_old$weight, probs = c(0.25, 0.5, 0.75), na.rm = TRUE) #` here, both weight and probs are vectors

Assignment, objects, and values

Challenge 3: Assignments and objects

What is the value of y after running the following code?
```
x <- 5
y <- x
x <- 10
```

Answer: y is 5!
Y does not become linked to y in any way, y takes on the *current* value of x as its assignment. We evauluate the right hand side before we assign it to y.

When we're naming objects, they should be clear and informative, without being too long.
For a variable representing weight in kilograms,
weight_kg works better than the shorter
wgt or the full
weight_in_kilograms

reserved names in R or dots are also bad practice.

Often, we use snake_case or camelCase.

Key Points

- functions like head(), str(), and summary() are useful for exploring data.frames
- most things in R are vectors, vectors stitched together, or functions
- make sure to use class() to check vector types, especially when using new functions
- factors can be useful, but behave differently from character vectors

Working with Data

Often, we have to read data in from an external file. To do so, we can use a different package of the tidyverse: readr.
For a comma-separated-values (csv) file, we use read_csv:

```
surveys <- read_csv("data/cleaned/surveys_complete_77_89.csv")
```

To extract certain solumns in our data, we use the "select" function of dplyr:

select(surveys, plot_id, species_id, hindfoot_length)

This can be generalized to column number, range, type, containing a certain character match, etc.

```
select(surveys, where(is.numeric))
```

We can extract rows by using the **dplyr** command "filter":

```
filter(surveys, year == 1985) # where is year equal to 1985
```

Other logical operators: >, >=, <, <=, !
We can alos use %in% to quickly include a vector of values:

```
filter(surveys, year %in% c(1988, 1987))
```

which is equivalent to

```
filter(surveys, species_id == 1988 | species_id == 1987)
```

CHALLENGE 1

Use the surveys data to make a data.frame that has only data with years from 1980 to 1985.
Answer:

```
surveys_filtered <- filter(surveys, year >= 1980 & year <= 1985)
```

The pipe: %>%

The **pipe** allows us to 'nest' a series of functions into one another to quickly deal with multiple functions without having to make intermediate variables. For example,

```
surveys_sub <- surveys %>%
```

- select(-day) %>%
- filter(month >= 7)

selects for all columns (besides day) and then filters to all entries with the month column greater than or equal to 7. When using pipes, each step is evaluated left to right, like if functions were nested inside one another:

```
surveys_sub <- filter(select(surveys, -day), month >= 7)
```

CHALLENGE 2

Use the surveys data to make a data.frame that has the columns record_id, month, and species_id, with data from the year 1988. Use a pipe between the function calls.

Answer:

```
surveys_1988 <- surveys %>%
```

- filter(year == 1988) %>%
- select(record_id, month, species_id)

Making new columns with mutate()

s

```
surveys %>%
```

- mutate(weight_kg = weight/1000)

You can also create multiple columns at the same time

surveys %>%

- mutate(weight_kg = weight/1000,
    - weights_lbs = weight_kg*2.2)

You can also create a new column based on multiple existing columns. For example, we can make a new date column using the year, month, and day columns. We can then use relocate() to move that column after the year column instead of at the end

surveys %>%

- mutate(date = paste(year, month, day, sep = '-')) %>%
- relocate(date, .after = year)

But out date column is now a character type, not a date type. Dates are a special type because they have a specific order. We can use the package lubridate and the function ymd() to change this.

surveys %>%

- mutate(date = paste(year, month, day, sep = '-'),
    - date = ymd(date))

The pipe takes the thing on the left and passes it into the thing on the right putting it in as the first argument. This means we can use the pipeline for ggplot too:

surveys %>%

- mutate(weight_kg = weight/1000) %>%
- ggplot(mapping = aes(x= weight_kg, y= hindfoot_length)) +
- geom_point()

The split-apply-combine approach

We use group_by() to group rows by a particular variable, and then summarize() to apply a certain function to each group and then recombine the results in a single output.

surveys %>%

- group_by(sex) %>%
- summarize(mean_weight = mean(weight, na.rm = T))

n() is a function that counts number of rows

surveys %>%

- group_by(sex) %>%
- summarize(mean_weight = mean(weight, na.rm = T),
    - n = n())

You can also group by multiple columns (e.g., species and sex)

surveys %>% group_by(species_id, sex) %>%

- summarize(mean_weight = mean(weight, na.rm=T),
    - n = n()))

NaN = Not a Number. Occurs when you try to perform an operation with an empty vector.

We can start off by filtering out our NAs so we don't have to worry about it every time.

surveys %>%

- filter(!is.na(weight)) %>%
- group_by(species...

We can also organize the order of our mean weights using arrange()--for which the default is ascending order--or arrange(desc()), if we want descending order.

At the end of your pipeline, you will want to add ungroup(), otherwise your data will stay grouped and will continue to use those groups in future analyses.

What if you don't want your data to be collapsed into single rows per group? You can pair group_by() with mutate() instead of summarize()

surveys %>%

- filter(!is.na(weight)) %>%
- group_by(species_id, sex) %>%
- mutate(mean_weight = mean(weight),
    - weight_diff = weight - mean_weight)

We can use contains() inside of select() to select all columns that contain a certain character string

surveys %>%

- select(species_id, sex, contains('weight'))

Reshaping data with tidyr

We can also use the package tidyr, part of the tidyverse, to "reshape" our data.

Let’s say we are interested in comparing the mean weights of each species across our different plots. We can begin this process using the group_by() + summarize() approach:

sp_by_plot <- surveys %>%

- filter(!is.na(weight)) %>%
- group_by(species_id, plot_id) %>%
- summarise(mean_weight = mean(weight)) %>%
- arrange(species_id, plot_id)

It is a bit difficult to compare values across plots. It would be nice if we could reshape this data.frame to make those comparisons easier. Well, the tidyr package from thetidyverse has a pair of functions that

allow you to reshape data by pivoting it: pivot_wider() and pivot_longer().

pivot_wider() will make the data wider, which means increasing the number of columns and reducing the number of rows.
pivot_longer() will do the opposite, reducing the number of columns and increasing the number of rows.

```
sp_by_plot_wide <- sp_by_plot %>%
```

- pivot_wider(names_from = plot_id,
    - values_from = mean_weight)

Now each column is a different plot_id!

We can go back to the long format by using pivot_longer()

```
sp_by_plot_wide %>%
```

- pivot_longer(cols = -species_id, names_to = "PLOT", values_to = "MEAN_WT")

Exporting data

Let's start by renaming those columns in pivor_wider() to be more informative

```
sp_by_plot %>%
```

- pivot_wider(names_from = plot_id, values_from = mean_weight,
    - names_prefix = "plot_")

This adds the prefix "plot_" to each of the new column names.

To save the file we will use write_csv()

```
write_csv(surveys_sp, "data/cleaned/surveys_meanweight_species_plot.csv")
```

KEY POINTS

- use filter() to subset rows and select() to subset columns
- build up pipelines one step at a time before assigning the result
- it is often best to keep components of dates separate until needed, then use mutate() to make a date column
- group_by() can be used with summarize() to collapse rows or mutate() to keep the same number of rows
- pivot_wider() and pivot_longer() are powerful for reshaping data, but you should plan out how to use them thoughtfully

The END!

Feedback:

Today I learned:

I am still confused by: