INTRO AND RECAP

Welcome to **day 2**! Today, R. But first, recap yesterday.

First, we learned about **managing spreadsheets** efficiently--make a single **rectangle**, **rows=observations**, **columns = variables**. **Header** row at top, **no spaces** in names. **Only 1 type** of data per cell. **Fill all cells** with something, esp. where data are missing. **Code missing** vals consistently. Consider storing **dates** as separate day/m/y. **Keep raw** data files **clean**--don't do calcs in them. Save data as **csv** so it's transferable and always openable. Lastly, **don't** use **color** or highlighting.

Next, **OpenRefine** for **exploring** and **cleaning**. **Don't edit raw**! Use **facets+filters** to **explore**. Use OR to **split columns** easily, **remove extra text**, and find **outliers**. **Reproducible**, so fast to apply again.

Then, **SQL**. Quickly **parse**, **manipulate**, and **summarize** databases. **Keywords**: **SELECT** to choose **columns**, **FROM to specify** data tables, **WHERE** to select specific **rows**, **ORDER BY** to **sort**, **GROUP BY** to organize results by a group, **COUNT** and **SUM** to **summarize by groups**, and **JOIN ON** to **link data** together.

Today, learning **R**. In a bit, we'll see **dplyr**, a set of **tools for** data **manipulation**, very **similar to SQL** (many same or similar keywords).

Before R, **organizing project** files. Data projects are **often chaos**. **Org** is the enemy of chaos!

Here's what we recommend: All files in **common** folder (called "**directory**"). Have separate folders for **raw vs. clean** data. Have separate **folder for** R **code**. Make file **names** meaningful (i.e. **searchable**), **sortable**, & **consistent**! For example, **dates** with year, then 2 digit month, then day. For today, **project folder** on desktop. Name data_carpentry. Make raw_data, clean_data, code, and output folders inside in prep.

Why so **important? You** are your closest **collaborator**! We often must pick analyses back up later--**don't be** a **bad collaborator** with self! Keep things organized.

INTRO R

Ok, now R. Good place to start--**what is R**, anyway?

Programming language. Fairly old. Descended from S. Usefully, **designed with data** processing and **analysis in mind**. Importantly, though, **also program** fluent in R. Good news--**if we learn** to put wishes in **R**, R can put computer to work for us: stats, manipulate data sets, make graphs, and more. So yeah, **learning curve (must learn to speak R)**, but super worth it!

R is worth knowing for many reasons: **Free**, on **all platforms**, **open-source** (anyone make **new features** and upload them for all. 9000+ plus packages and counting!). Also, R knowledge is **in demand** because user base is growing. R user **community** all around you! **Take advantage**: ask questions. I still learn things every day from peers! If not sold yet, just wait. **Today** is **designed to show what R** is good at and **can do** for you!

Today, we'll use **RStudio**, an **add-on** for R. **Integrated Development Environment**--meaning it **integrates all the tools you need** to use R and puts them at your fingertips! Not technically required, but I **can't imagine R without it!**

Turn to R. Show you around. **At top**, **menus** like you're used to. **Bottom-left, console--speak directly** to R here, and where it will **talk back** to us **with products** of code. **Top-left: script editor**. Scripts are **text files for code**. Can type code, run it from here, and **have a record** of it later! **Top-right: Environment**. When we **store stuff** in R, **shows up here** for reference. **Bottom-right**, many useful tabs! **Files--navigate** your project **folder** to look for files. **Plots**--shows **graphics**. **Packages**--turn on add-on packages to **add new features**! **Help**--where help pages appear (more later!). Like I said--all tools in one place.

INTERACTING WITH R

Now, **communicating with R directly thru console**. When R is **ready** to be talked to, **> prompt**. If you have something else, ask us for help! R works well with data, so it can do math very well. Type simple **addition**, then **hit enter to execute code**. R will take it, **operate** on what you gave it, and **return any results**. Here, our sum. Can also do **subtraction**, **multiply**, and **divide**. **Spaces not necessary**, but makes easier reading. Nevermind number in brackets right now!

We can do **complex math** too by using **functions**. Functions are **like SQL's keywords**: they are **shorthand for** one or more **tasks** we might want a computer to do for us. **Using functions** is relatively painless: Put function **name**, put **parentheses** after, then put any necessary **input inside** the **parenthesis**, then run. R will do whatever tasks and return the result.

Sample function: **log**(), takes the logarithm of the input values. Straightforward enough. **Some** R **functions** are **simple**--they **just** take **one** kind of **input**, such as data to take the logarithm of. **Others** actually **can or need** to **take multiple** forms of input, **whether** it's multiple pieces of **data or instructions** on how to work with that data. **log()** can actually **take two** forms of **input**: values to log, and what base of log to take. In **log's** case, this **second** input **is optional**--log assumes, **by default**, we want to take the log of base e (**natural log**). However, we **can change** base by **putting a comma** and then putting a new base--new answer. Meanwhile, the first piece of input is required--if no values, can't work!

This is probably the **single most confusing thing about** R **functions**: Functions have **slots for** different kinds of **input**. log has two slots, for example. These slots for input are called, strangely, "**arguments**." These argument **slots have names**: log's first argument is called **x**, which we see in this error here. The second one is called **base**. The **comma separates** the two slots so R knows what input goes where. This is important: **R functions are literal**/fussy; **they want** to be given inputs in a very **specific order**. **If** you provide the inputs in **exactly that order**, R is **happy**. If you're all **crazy** about **order**, R will get **cranky**, **or** at least give **different results**! However, **if you refer to** arguments by their **names** and specify, **using the name and =,** that THIS input goes with THIS argument, you **can give inputs in any order** and R will put them in the correct place for you. Once you get familiar with a function, like we are, you can stop naming arguments unless you have to in order to skip all the way to a much later argument. More on that later.

Now, log is just **1 of 1000s of** R **functions**, so R can do thousands of things for you in seconds. **Almost all** of your **time** in R will be spent **using functions**, in fact. However, as you can see, you use **functions** the most **safely** and effectively **when you know** what **inputs** they want, **where** they want them, and **what optional features** they have.

To **learn** this **about** any **function**, type **?function name** and run. Brings up the **help page** for the function! Fair warning, these take practice to read, trust me. But, when you get the hang of it, lots of useful info, including a **description of** what the **function** does, its **arguments** and what they are called and do, and **examples** at the bottom. Use this today as needed!

SCRIPTS

So, we can talk to R directly thru console, but doing so doesn't save our code so we can reference it later, share it, or reuse it. So, instead, we can **use script files**. These can be **shared** with colleagues, **submitted** with journal articles for review, **and reused** over and over to redo analyses, so very useful and empowering!

Before we make our script file, let's tell R where our **project folder** is. To do this, go to **File -> New Project -> existing directory -> browse to data_carpentry, click Create Project**. R sort of restarts. Now it's in "project mode!" This has **changed R's "working directory**" to our project folder, so R knows all data for this project is in this folder and all output produced should go here too. Helps us keep everything in one place.

Now make a **new script** file using this icon. Click, then select new script. Name this script "Intro to R" and **save** it **in** your **code folder**. From here on out, we'll **interact** with R **exclusively via** our **script** file so we have a record of all we've done.

Let's type out some simple math again so we can **see a handy feature**. Our script's name turns **red**. This indicates we have **unsaved changes**. If we click the save icon again, the title reverts to black. I wish Microsoft Word had that!

Code written in our script **won't be run unless** we tell R to run it. Hit enter--nothing happens in console! However, running code from script is easy. There are **two ways**: 1) Put cursor on same line as code and **hit run button**. Alternatively, if you hover over run button, you can see there is a hot-key for running code: **Ctrl + enter** (make sure cursor is on the right line). If you have multiple lines of code, **highlight** them **to run** them **all**.

Another **handy** R **feature**. Just like SQL, R has a **comments** operator, the #. **R ignores** anything written after one, so you can **annotate** your **code** to explain to yourself or any future readers **what your code does**. **USE THIS**! Be a good collaborator, even just with yourself.

OBJECTS

We've seen 2 things that make R especially powerful: functions and scripts. Now, we'll learn about **objects**. R is built to work with data. It allows you **to store data inside an object** so that you **can work with** that data **repeatedly** without having to type it all out every time.

To create an object and store some data in it, we type: the **name** we want to give our new object, the **assignment operator <-,** and **then** the **data to store** on the right. Here we are saying to R, **store the result** of 15 + 5 inside of a new object called math. Now if we were to type just math and run it, R will return 20. R "knows" now that math is equivalent to 20. Also note math is now an object listed **in our Environment** up here.

Now that R knows what math means, we **can use math in operations** in place of 20. We can even store the result of those new operations in another object and use that new object in yet more operations. For example, we could use the **sqrt** function on it, or we could **round** it to the **nearest whole number** with the round function. Sidenote: **round** is a function with a **second optional** argument also, called **digits**, which can be changed to affect what decimal is rounded to.

Now, full disclosure, **the = sign also** works as an **assignment operator**. In fact, this is the **one I use**! Mostly because it was the **one I was taught and** it's one **fewer keys** to type. However, **= is used to do a lot of other things** in R besides assignment (like associating function inputs with arguments), whereas the **arrow just does assignments**. **So**, the arrow is **less confusing**. Plus, **many guides** + books **use arrow**, **so** you might **see** it **more often**. **Use the one you want** today, but **know both** are out there, and **I might accidentally switch** back and forth today (sorry!). Incidentally, the hot key to create the arrow is alt + -, if you're computer savvy.

Some more things to know about objects. **Objects** can be **named** almost anything. For example, they **can contain number**s, **but** they **can't start with a number**--R will think you're trying to do math. Names can be short or long. However, you should **aim for the middle**--make the name something **meaningful** and unique **but not so long** it's a chore to type. My recommendation--**come up** with a **naming convention** and **stick with it**. For example, I put all column names in all caps, and I use **underscores** to separate words. Other people use

CamelCase, where each new word starts with a capital. Use what works for you. If you're interested, there are **formatting guides** out there we can direct you to!

Another reason to **avoid short names** is that many short, genetic names are **already function names** in R. Examples include **mean, t, c, and data**. Now you **can technically** name an object mean, **but don't**: it's **confusing** for others to read **and too vague** anyway! Two more cautions: **don't use periods to separate words** because many function names do that as well. Lastly, remember that **R is case-sensitive**, meaning it reads capitals and lowercases as different. **Frustrating**, but worth remembering**--be consistent**, especially **with capitals**!

Alright, time to see if you understand how objects work in R. Let's type these three lines of code into our script, but don't run them. I want you to **take a minute and discuss** with your neighbors--**what will y be equal to** after all three of these lines are run? **Why**? What would we have to do to make y equal to something else?

In short, **R doesn't link objects** together. Just because y was created using x doesn't mean y will update to equal something else just because x changed. **If we want y to change**, we will have to **run this assignment** code **again** to make R aware of the new value. Keep this in mind!

THE SURVEY DATA

We know enough about the basics of R now to **start working with our survey data** set once again. To read the data into R and save the data into an object so we can work with it, we can use the **read.csv function**. Its **first argument** slot is for the **path** to where the data is, so we can put in a URL to get the data from the web. This would be one way, but what if the internet goes down later? **Might be better to download** the **data** first, then read it in. For that, the **download.file()** function. This one has **two arguments** we'll use. The first is the **url**, which we can copy. The second is the **destination file name and path, destfile**. This is where we want to put the file and what we want it called. Now we can **use read.csv to read** in the file **from there**.

In our **Environment**, we see that **surveys is** something called a **data frame** with 34000+ observations of 13 variables. A data frame is an object type in R that **holds data in a rectangle**, with rows as individual observations and columns as variables. Data frames are powerful because **each column** in them **can hold** a **different data type**: one column can be numeric data, another can be for text, another can be for dates, and so on.

We can explore our data frame with several functions. **head**() and **tail**() will show us the first and last couple of rows, for example. This lets us **see if the data look right**. **dim**() shows you the numbers of rows and columns, whereas **nrow**() and **ncol**() give you one or the other. **names**() shows you the names of your columns, while **rownames**() does the same for rows (not too interesting).

Two really powerful functions are **str**() and **summary**(). str() shows you the **structure of** your **data**, combining elements of all these previous functions. You get the **object type** (data.frame), the **dimensions**, the **column names**, their data types, and the **first couple of observations** in each. Note that some of our data are integers (int) and some of them are factors. More on those later. Look at the output here under weight and hindfoot_length. What do you see? **Capital NA** a lot. What do you suppose those values are? **Missing values**! More on that later also.

Lastly, **summary** will show you interesting **metadata** about your data. **For numeric data**, it presents your **quartiles** and mean and so forth. For factor data, it gives you the **most common values** that variable takes in the data **and** gives you **counts** of those. The most common species was DM, for example.

INDEXING AND SUBSETTING

Now that we know about functions, objects, and data exploration, we need to teach you how to **extract data from objects and subset** objects so you can work with them**. R excels at this** also. To extract data from an object, we use what's called "**indexing**." Let's say we wanted R to report to us the value in the first row, and fifth column of our data sheet. We could index this value as follows: **our object name**, a set of **square brackets**, our **row number**, a **comma**, and then a **column number**. (give reverse example also).

To extract an **entire row or** an entire **column**, just **leave** the appropriate **slot** inside the square brackets **blank**, but **don't forget the comma**! (give examples using column 7). With data frames, you can also **index** using the names of **columns** also, in a couple of different ways. You can even **save your indexed** results **in** a new **object**. (no sex objects! haha).

This new object called sex is different. It doesn't have rows and columns; it's just a **string of values**. Objects like these in R are called **vectors**. Values can be extracted from vectors via indexing also, but now we won't need two different numbers because there aren't two different dimensions.

To **create** your own **vectors**, use the **c** function. Stands for **combine**. **Separate** every new **value** to add with a **comma**. You can use these new vectors to index values that aren't next to each other. You don't even need to store the vector first if you don't want to. If the values you want to extract are near each other, you can use **:** to **make a simple sequence** of values. The **seq** function can make a **more complex sequence**. Another indexing trick is to use a **- sign** to **exclude specific entries**.

All these **tricks work on data frames** too, but now we need two dimension values again, plus our comma.

Take a minute to try this: Use the nrow function and indexing to save just rhe last row of surveys in a new object called surveys last.

MISSING DATA

Ok, just two more topics for this section. First, a couple of words on missing data. **R is good at** handling **missing data because** it has this special value of **NA** for them. Let's make a test vector with a missing value. Now, let's try to take the **mean**() and **range**() of this vector. **We get NAs** back. By default, R assumes we want to know our data contain missing values, which is **good but** also **frustrating**! Thankfully, many functions have an na.rm argument. If we set it to TRUE, R will instead give us the answers we want.

Sidetrack: Try just TRUE. Why doesn't this work? Hint: **check mean's help page**. How many arguments does it have? 3, and na.rm is third. We gave R two inputs here for mean. It's expecting those to be inputs for the first two arguments, and **TRUE doesn't make sense for trim**. **If we want to skip over the second**, trim, and manipulate the third, **we had to tell R that**.

If you work with data with a lot of missing values, two good functions to know. The first is **is**.**na**(). Returns true for each NA value. If we use is.na inside an index, we get just the NA values back because indexing with TRUES and FALSES returns just the TRUES. We can reverse this behavior with the **! operator** to get everything that isn't missing instead. Alternatively, **na.omit**() will produce the same thing.

FACTORS

Lastly, let's talk about factors. **Factors** are **how R stores categorical data**, which it does **really well**. How it does this is easier to see than to hear about, so let's make a simple factor object. You can see this factor has 2 levels, female and male, that are marked as **text** by being **in quotes**. The actual **data**, though, **are integers**, 1 and 2. This is because R stores this data as a **hybrid of text and numbers**: the **categories** themselves are made into **levels** that stay **text**, and then the **data** is **coded** such that **each level** gets its own **integer**. Even though male was the first value in our factor, "**female**" ended up as **level 1**. Any guesses **why**? It's **alphabetically first**! That's how R codes by default.

To change the order of levels, we can use the **levels argument** to the factors function.

Word of warning: **R is not** quite **as user-friendly when** you try to make **factors** with data that **look like numbers**. Let me show you. Converting these data to text is easy using **as.character**(). However, using **as.numeric** doesn't work as well, right? We got the level numbers assigned to the data instead of the data themselves. Interestingly, if we turn the data into text and then into numbers, that works. Something to be aware of.

Another good function to know is **table**, which gives you **counts of** your **factor levels**.

The last tip is just a note that **read.csv** by default **turns any textual data into factors**. If you want to prevent this, just use the **stringsasfactors** argument. **Note autocomplete and camel case here**.

GGPLOT

Alright everyone, time to **introduce** you to the **ggplot2** package, another powerful add-on for R. Before we can start, though we need to **download a reduced version** of our survey data to work with. For this, we'll use the **download.file** function again, putting the data in the raw data folder. **Then**, use **read.csv** to read the file in as an object called reduced.

In R, to make graphics, we **could use** the **basic plotting** functions that R comes with, like **plot**(). These are **simple**-ish to use, but also simple in structure. To make really eye-catching graphics, **ggplot**, a package built by **Hadley Wickham** based on the concepts of the **grammar of graphics**, which is what gg stands for. Ggplot code is **more complex** than base graphing code, but once you get it, you can make **publishable quality** graphs in minutes or less. All my recent pubs feature ggplots!

To make sure our dplyr and ggplot2 packages are turned on, let's turn them on using the **library function**. (Or can turn on on the packages tab).

OUR FIRST GGPLOTS

To start, let's make a **scatterplot of weight vs. hindfoot length**. The **base structure** of a ggplot has **three parts**. Once you get these three parts down, you are off and running! The first part is the **actual ggplot function** call. Its **first argument is the data** sheet, which **must be a data frame**, which ours is. Its second argument is **mapping**--this is where we tell it **what data goes where**, which we do with the **aesthetics function**, **aes**(). Put in our x and y.

Note, if we **just run this**, we see that it **sets up a graphing area** for us, but nothing's on it. That's cuz we're missing our **third basic piece**--a **geom**. A geom is a function that tells ggplot how, specifically, we want our data graphed. Here, we'll use **geom_point**, the **scatterplot** layer. **To add a** feature or **layer** to a ggplot, you literally **add** it **with a +.**

This is basically how ggplot works--you tell it what data to use, how to use it, and what features to add. ggplot then **layers all** of this **information** on top of one another and **panini-presses it** into a single graph. Because ggplots are built in additive pieces, we **can** actually **save** these **pieces into objects** and then work with those pieces to try out alternative graphs. Let's save our base ggplot piece in an object, then we can save the whole thing in another object. Because ggplot code can be lengthy, this can save a lot of time while experimenting!

Let's experiment with the scales of the axes just to show you a bit more of what ggplot is capable of. If we want to **log base-10 transform y**, we can add on the **scale_y_log10()** function. If we want to **sqrt-transform x**, we can add on the **scale_x_sqrt()** function. There are a bunch of these autoscaling functions available, so scaling axes in ggplot is easy!

Ok, time to **challenge** you all to use your **combined dplyr and ggplot2 knowledge**! Take a minute and, with your neighbors, make this same scatterplot here (plot2) but only with the animals from species_id "DM." First, filter to get just DM in new object, then call ggplot2.

MORE EXPERIMENTING

Let's do more experimenting with our basic scatterplot (plot1). **Each** individual **geom has** a number of **arguments** that allows us **to customize** our graphic. For example, to make the points **less opaque**, we could use a lower **alpha**. We can also add color with the **color** argument. There are hundreds of colors ggplot recognizes; you can look up color palettes online. To change the **size** of points, that's the size argument.

Changing aspects of the graphs this way changes all of the points being plotted. However, if we wanted to, say, plot each species's points as a different color, we could do that by **mapping color to species**. We can do that because **each geom** also **has an aesthetics argument**, just like ggplot does.

Ok, dplyr + ggplot challenge #2! I want you to get the **mean weights and hindfoot_lengths for each species**, **plus** a **count** of observations for each species. Then, I want you to **plot mean weight vs. hindfoot_length** like this but with the **point size varying by the count**! Answer: First, group by species ID, then use mean to get two means, then use tally or n(). Then, ggplot, with geom_point + size = count. Is that freaking cool or what??

OTHER GEOMS

Let's explore some other geoms now. **Geom_line** allows us to make a **line graph**. Let's make a line graph of **observations of each species by year**. Group by year, then tally. Then make plot3 object.

If we add in a geom_point, we **can add** the **points** to the graph. We can even make these two different elements different colors. When we do that, we notice that the **points** are on top of the lines, right? Why do you think? Because they are **plotted last because** they were **added last**. When I said we are adding layers to a graph, we are literally adding layers on top of one another! So, order matters (reverse order). Now, lines on top.

We can even **map an aesthetic to just one** of these two **layers** by only giving only one of them additional mapping directions. Only here the points change color with year--the line doesn't. If we instead wanted to affect all the geoms at once at make color vary by year for all of them, we **could put** that **mapping in the ggplot** function call **instead**. This is one thing that makes **ggplot** so **flexible**--you can either make aesthetics **global**, affecting all layers, or **local** to just one geom layer.

(Can't skip.) dplyr + ggplot2 challenge #3! I want you to make a line plot of the counts of just two species by year: DM and DS. Plot the points also.

GROUPING

Now, notice that this graph is connect points from both species together. Imagine we want to **keep** the **lines black, but split the two species points into different colors**. We might try mapping color to species inside of geom_point. This gets the color, but the lines are still connected. We could make the color mapping global, but now the lines are different colors too. We could just go in and make the lines black, but there's a better way: the **group aesthetic**. It tells ggplot that **unless we say otherwise**, we want **groups treated the same**. Then, we can change only the point color inside of geom_point. Neat huh?

OTHER GEOMS

Ggplot can of course make plots like histograms and boxplots too. Using the whole reduced data set, use **geom_histogram** to make a histogram of weights. **Map weights to x**. Then, play around with the **bins argument** inside of histogram.

Boxplots are similar; they use **geom_boxplot**. **We can add point**s on top of this boxplot to visualize the distribution of the data, but to avoid having all the points be plotted on top of each other, we can use **geom_jitter**, which plots points randomly spaced out a bit. Of course, now we've covered up our boxplots! How could we fix this? Yes, just reverse order.

FACETS

We've been grouping different groups so far by color or size, but we **can** actually **make ggplot split** the whole graph **into separates graph for each group** too **using facets**. Let's make line graphs yearly counts by species by altering our code from earlier slightly. Then, let's make our fancy ggplot. Here are all our lines on one graph. To split them, we can add on **facet.wrap**, telling ggplot to split them up by species_id. Super cool!

Now, what if we wanted to **split by species and sex**? What would we have to change? Add in sex, then change group + color. Even cooler, right??

Facet_wrap arranges the new graphs in whatever shape will fit best in your plotting window, but **facet_grid** will allow you to **arrange** the graphs **in rows or columns by a factor or factors**. This makes ggplot split the graphs out by sex, with each sex in a row. Reversing the order will make it a column instead. We can even split by species_id and sex. Is ggplot powerful or what?

CUSTOMIZING

And we're just scratching the surface of what ggplot can do. I want to show you just a few more ways you can customize your graphs. Earlier, we saw the scale_x and **scale_y function families**. We can use these functions to **change** aspects of **how** our **axes are plotted**. Since our data are continuous, we'll use the continuous versions of these functions. First, if we wanted to change the range of data plotted, we could use the **limits argument**. It takes a vector of 2 values, a **min and a max**. You can also adjust what **values** are used as **labels** by using the **breaks argument**. This one takes a vector of whatever values you want to mark, even if they are spaced out.

You can **also** use these functions to **change** what **colors** are being used to mark data by groups. Say you had plotted these data by sex but didn't like these default colors. You could use **scale_fill_manual and** its **argument values** to change the colors to whatever you liked. Is it a bad time to mention I'm actually a Michigan grad? Makes me wanna start singing hail to the victors!

The last customization tool to show you today is the **theme function**. Let's you **change** a bunch of **"little things"** about a graph, from the size of legend fonts to the colors of tick marks to the margin widths. Let's say we hated that damn gray default **background** ggplots have. We could use theme, and its **panel.background** to change that to white instead, using the **element_rectangle** function. If we hated those pesky **grid-lines**, we could make those go away using theme, the **argument panel.grid**, and the function **element_blank**. Theme has at least 100 arguments, so it offers you a ton of ways to change your graphs. There are even some **pre-baked themes**, like theme_bw. (show other options).

SAVING GRAPHS

The last thing for the ggplot unit is **saving graphs**. For this, ggplot has the function **ggsave**. Let's save our plot2. The first argument is the **path** to where we want the file saved, then the **name** and **file type** we want. The **second** is the graph to save, stored in an **object**. **Then, height + width** control the image size in inches. If we **change** the **file extension** in the first argument, we can even change the graph type!