INTRO AND RECAP

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

First, we learned about **managing spreadsheets**--make a single **rectangle**, **rows = observations**, **columns = variables**. **Header** row at top, **informative names** w/ **no spaces**. **Only 1 type** of data per cell. **Fill all cells** with something, esp. where data are missing. **Code missing** data 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 they're transferable and always openable. Lastly, **don't** use **color** or highlighting--store that information instead.

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

Then, **SQL**. Quickly **explore**, **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.

Remember, y**ou** 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. D**esigned 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 our computer to work for us: do 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! 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. **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.

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--functions are like the verbs of the R language! **Using functions** is relatively painless--3 parts: Put function **name**, put **parentheses** after, then put any necessary **input inside** the **parentheses**, then hit enter to 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 input**, such as data to take the logarithm of. **Others** actually **need to or at least can** to **take multiple** inputs, such as multiple chunks of **data or instructions** on how to work with that data. Even tho it'll work with just data to take the log of, **log()** can actually **take a second input**: 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 base e (**natural log**). However, we **can change** the base by **putting a comma** and then putting a new base--new answer. Meanwhile, the first piece of input is **required**--if we give no data to log, can't work!

This is probably the **single most confusing thing about** using R--understanding functions--so I want you to picture a function in R as a train. Each function is pulling cars, and each car is a **place for** a different **input**. log has two cars--one for the data and one for the base. These cars for input are called "**arguments**." Each train car, or argument, **has a name**: log's first argument is called **x**. The second is called **base**. When we use the log function and we want to give it inputs for both x and base, we use a **comma to separate** the two so R knows what input goes in what car. Thinking about an R function as a train is useful because **R functions assume you're giving them inputs in exactly the same order their cars are in.** If you don't, R won't know any better and you'll get different results! You can see this if you run log(3,5) and log(5,3) in R--you get back very different numbers. However, you can help R ensure that each input always goes in the right place by using the argument names to assign inputs to specific arguments, as shown here. R will put the right inputs in the right cars for you, even if they're out of order!

Now, log is just **1 of thousands of** R **functions**, so R can do thousands of things for you. **Almost all** of your **time** in R will be spent **using functions** because they are so useful. However, as you can see, you use **functions** the most 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**. 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, as well as what order they are in, 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. To do those thing, we can **use script files**. These are text files for R code that can be **shared** with colleagues, **submitted** with journal articles for review, **and reused** over and over, 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! That's because a script file is just a text file--we're not talking directly to R here. However, doing so from a script is easy. There are **two ways**: 1) Put cursor on same line as code and **hit run button**. This flings the code into the console and runs it for us. 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 to run, **highlight** them **to run** them **all** at once.

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 learned two core concepts of R so far that make it very powerful: functions and scripts. Now, we'll learn a third: **objects**. If functions are the "verbs" of R language because they do things, then objects are the "nouns," because they are what things are done to. One way R is so good at working with data is that it allows you **to store data inside of an object** so that you **can work with** that data **repeatedly** with ease. I want you to think of an object as a big storage box with a name label on it. When we create an object in R, we put some stuff in this box and put a name on the box. Then, when we use that name, R knows we really mean all the stuff in the box with that name: the name becomes a nickname for all the stuff! This is so useful--instead of typing out 1000 data points every time we want to work with them, we can store them in an object and just use that object's name instead.

Creating an object, naming it, and putting something in it requires 3 things: the **nme** we want to give our new object goes first, then the **assignment operator <-,** and **then** the **data to store** goes last. We are saying, "Hey R, **store the result** of 15 + 5 inside of a new object called math." Now if we were to type just math, R returns 20, that result. R "knows" means the same thing as 20--20 is in the "math" box. Note--math is now listed **in our Environment**. This is where our created objects are shown for our reference. We **can** now **use math in math problems** in place of 20. We can even use objects we've already made to create new objects and then use those objects in math problems.

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 yet. 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? [follow-up] 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!

Now, just so you know, **the = sign is also** an **assignment operator**. In fact, it's what I **use**! It's shorter and what I was taught. However, **= is used to do other things** in R besides assignment (like associating inputs with arguments in functions), whereas the **arrow only does assignment**. **In that way**, the arrow is **less confusing**. Plus, **many guides use the <-**, **so** you might **see** it **more often**. **Use the one you want** today, but **know both** are out there, and if I mix them up, know that's ok. I should note here that operators, like the arrow, equals sign, the comment operator, math symbols, parentheses, and commas, are the punctuation of R language. Make sure to understand what they do and when to use them!

More object tips and tricks. First, o**bjects** can be **named** almost anything. The names **can contain number**s, **but can't start with one**. We recommend **avoiding short names** because many short, genetic terms are **already names for function** in R. Examples include **mean, t, c, and data**. So, that could be really confusing! For the same reason, **don't use periods to separate words** in names because many functions do that as well. Lastly, remember when naming that **R is case-sensitive**, capitals and lowercases are **different**. **Frustrating**, but worth remembering**--be consistent**, especially **with capitals**!

My recommendation--**come up** with a **naming convention** and **stick with it**. For example, I put all my 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 but at least **aim for names that are meaningful** and unique **but not so long** it's a chore to type. If you're interested, there are **formatting guides** out there we can direct you to!

THE SURVEY DATA

We know enough about R now to **start working with our survey data** again. To bring the data into R and save them in an object called surveys, we can use the **read.csv function**. Its **first argument** slot is for the **path** to where the data is on our computer. Our data file is in our data folder, so we type [XXX] here. Now, for its own good, R treats text and numbers as fundamentally different. Because our path is text, we need to mark it as text by using text operators--quotation marks!

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 individual observations in rows and variables as columns. So, just like the data is in Excel.

It's important to explore data once you bring them into R to make sure they look right. We can do this using several functions. **head**() and **tail**() will show us the first and last couple of rows, for example. **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.

Two really powerful exploration 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 column. Note that some of our data are integers (int) and some of them are factors. Factors are special way R stores textual data--we won't cover them today in depth.

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

We've covered four major concepts: functions, objects, scripts, and data importing. We have one more: Indexing. To close this R crash course, we need to teach you how to work with objects--to view and change

their contents. Both of these things require us to understand how to index. To understand how indexing works, I want you to picture our surveys object as a box. Inside of that box is a big shelving unit--it has rows, it has columns, and it has cubbies where each individual data point is stored. Each of these cubbies is individually numbered, just like in this picture. Just as we could in real life, if we wanted to see the contents of a cubby or replace those contents with something else, we can use the cubby's numbers to help us.

Let's say we wanted R to tell us what the value was in the 25th cubby in our data set. To index that value, we need just 3 things: O**ur object's name**, a set of **square brackets**, and our cubby number inside those brackets. That's all there is to it! Square brackets even look like cubbies, which is handy! If we wanted to replace that value with a different one, we simply combine indexing with assignment. [Turn the 25th entry into something else and show that it's changed].

Indexing is a convenient way to view and change specific values in a data set without having to go back to Excel to do it. But what if your data sheet has 30000 entries and you don't know which entry the one you want to work with is? With data frames, you can also index values using their row and column numbers instead. (give reverse example also). Let's say we wanted to know the value in the first row and fifth column. Just put the row and column numbers inside the brackets and separate them with a comma. You can even index whole rows or columns this way by leaving either the row or column slot empty--for example, to get the whole 7th row, [7,]. This means "7th row, all columns." You can also get an entire column using that column's name and then the $ operator to put the column name too. If you wanted a sequence of rows, such as the 5th through the 7th rows, you could use a colon to get that sequence ([5:7,]). Lastly, to get the values in the 5-7th rows for every column except the first, you could use the minus sign: ([5:7, -1]). This excludes the first column from the results.

And, as always, you can save the results of any index into a new object if you'd like. Take a minute to try this out: Use the nrow function and indexing to save just the last row of surveys in a new object called surveys_last.

That concludes the Intro R unit. You now know all the basics needed to do basic work in R! In the next unit, we'll learn some ways to manipulate your data even more powerful than indexing! Any questions?

DPLYR

Welcome back! So far, we've learned how to import data into R and then speak the R language to manipulate that data a bit. By the end of the day, though, our goal is get you to the point where you've fully summarized your data in a useful way, produced a graph of that data, and written a report to give to your supervisor. This unit will address the summarization part, while the two sessions after lunch will handle the graphing and reports parts.

First, though, we need to make sure everyone still has their surveys object from the first unit. Please check your environment tab and confirm you have it (if not, get help to remake it).

In this unit, we're going to learn the powerful R package *dplyr*. In R, a package is a user-created set of functions and data sets that anyone can build, share, download, and use. *Dplyr*, specifically, is a package that provides several functions for quickly and easily manipulating and summarizing a data set. To use a package, we have to turn it on. To do that, we can navigate to the Packages tab, scroll down to where it says *dplyr*, and click the check box. In the console, you see that R used the library function to turn on the package--we can also do that. If you don't see *dplyr*, you may have not installed it properly when preparing for today's workshop. Raise your hand for help!

In this unit, I'll show you 6 of *dplyr*'s most useful functions--3 easier ones and 3 trickier ones. I'll also show you a new operator introduced in *dplyr* called the pipe that makes doing multiple manipulations faster. I'll also give you chances to practice your *dplyr* skills as we go.

First, the three easier functions. I like to think of these functions in terms of the problem each is designed to solve. The first problem is: "What if I want to see or store only some of a data set's columns?" For that, we use the *select* function. Each column name you give *select* as input will be returned and any you omit won't be. To use *select*, put the data sheet in the first argument slot, and then put each column to save in a new slot.

Challenge: Make an object called *small_surveys* that only has the *species_id*, *sex*, and *weight* columns of the *surveys* data set.

The second problem is "What if I want to sort my data set?" For that, we can use the *arrange* function. To use arrange, you put the data set to sort in the first argument slot and then the column you want to sort by in the second one. You can then sort by another column after that by putting it next, and so on. By default, *arrange* sorts in ascending order. To reverse the order, use the *desc*() function by putting the column to sort by inside that function.

Challenge: Make an object called *sorted_surveys* that sorts *small_surveys* by *species_id* in ascending order and *weight* in descending order.

The third problem we might want to solve is "What if I want to create a new column using a column I already have?" For that, we use the *mutate* function. *mutate* makes a new column using an old column as a reference point. For example, if our weights are in grams and we want to see them in kilograms too, we can use *mutate* to divide the weights by 1000 and make a new column. We can give that column a name too by using assignment.

Challenge: Use the square root function, *sqrt*(), to take the square root of the weights in *sorted_surveys*. Save the result in an object called *mutated_surveys*.

So far so good! We're halfway through *dplyr* already. Now, onto the trickier 3 functions. The first one of these solves the problem "What if I only want to see or store some of the rows of my data set?" To eliminate some rows and keep others, we can use *filter*. *filter* works by using logical rules to determine if it should keep a row or not. For example, if we only want rows in which the *weight* is = 18, we can use *filter* to get those. Now, I used two =s here--can anyone guess why? Yes, = already means something in R, as we've seen, so to do "equals" in a logic sense, R needs a different symbol. You can also do filters like greater than and less than, greater than or equal to, and not equal to. Here, why do you think I needed the quotes? Yes--the *sex* column is text, so I need to remind R it's trying to match text here. Lastly, you can actually have multiple rules as long as you put each new, complete rule in its own argument slot.

Challenge 4: Ok, now use *filter* to make a new object called *filtered_surveys* that filters *mutated_surveys* such that we only have data from females less than or equal to a weight of 50.

Alright, we have one last problem to solve--"What if we want to summarize our data to discover patterns or include the summaries in a report or graph?" This is a more complex problem, so we're going to need two functions to solve it. The first is *group_by* and the second is *summarize*. *group_by* is tricky to understand--it tells R "treat each unique entry in this column as a separate group." Then, when we summarize the data, it will give us summary data on each of those groups as a whole. If we group the data first, we can't see that R has done anything, but it has. To prove it, let's now use *summarize* to find the average weight of each species using the *mean*() function. We now have only a few rows--the data has been collapsed down to a few rows, one for each species, with the mean weight for that species. We don't even have any other columns because those don't even really mean anything with the data summarized like this. In this way, *group_by*, when paired with summarize, kind of acts as a *select* function also. Also, just like with *mutate*, we can name the new column if we want to.

One more trick with *group_by* and *summarize* is that you can group by more than one column. When you then summarize, you will get a summary for each combination of the columns given. Here, we can use the *n*() function to get the number of observations for each species and sex combination.

Now, up until now, we did our summarizations in two steps--we grouped, and then we summarized. *dplyr* actually introduces a way to do that process in one step--the pipe! A pipe is a weird-looking operator--%>%--but it works a lot like a pipe does in the real world. What one does is pump, or pipe, whatever object or product is on its left into the function on its right as the first input. The *group_by* chunk here takes *filtered_surveys* as its first input, then the summarize chunk takes the output of the *group_by* as its first input, making a chain. As you can see, this saves a lot of typing, and means we need to make fewer objects!

Challenge 5: Produce our summarized data set in a single step using pipes, starting from the original *surveys* data set. Select, arrange, mutate, filter, group, and then summarize!

Ok, after lunch, we intend to show you how to graph the results you've found using *dplyr*. As a result, let's make the three data summaries we will need for that lesson now as a way to practice our *dplyr* skills. Your handout has a description of the three data summaries we need. Work with your table to make these and we'll show you how to make them at the end if you'd had trouble.