**Pivot Tables with dplyr**

*## attach libraries*

library(tidyverse)

library(readxl)

library(here)

library(skimr) *# install.packages('skimr')*

library(kableExtra) *# install.packages('kableExtra')*

*## read in data*

lobsters <- read\_xlsx(here("data/lobsters.xlsx"), skip=4) – to skip the first 4 line in the excel file

To look at summary statistics we’ve used summary, which is good for numeric columns, but it doesn’t give a lot of useful information for non-numeric data. So it means it wouldn’t tell us how many unique sites there are in this dataset. To have a look there I like using the skimr package:

*# explore data*

skimr::skim(lobsters)

This skimr:: notation is a reminder to me that skim is from the skimr package. It is a nice convention: it’s a reminder to others (especially you!).

skim lets us look more at each variable. Here we can look at our character variables and see that there are 5 unique sites (in the n\_unique output). Also, I particularly like looking at missing data. There are 6 missing values in the size\_mm variable.

## **group\_by() %>% summarize()**

In R, we can create the functionality of pivot tables with the same logic: we will tell R to group by something and then summarize by something. Visually, it looks like this:

data %>%

group\_by() %>%

summarize()

### **group\_by one variable**

Let’s use group\_by() %>% summarize() with our lobsters data, just like we did in Excel. We will first group\_by year and then summarize by count, using the function n() (in the dplyr package). n() counts the number of times an observation shows up, and since this is uncounted data, this will count each row.

We can say this out loud while we write it: “take the lobsters data and then group\_by year and then summarize by count in a new column we’ll call count\_by\_year.”

lobsters %>%

group\_by(year) %>%

summarize(count\_by\_year = n())

### group\_by multiple variables

Great. Now let’s summarize by both year and site like we did in the pivot table. We are able to group\_by more than one variable. Let’s do this together:

lobsters %>%

group\_by(site, year) %>%

summarize(count\_by\_siteyear = n())

We put the site first because that is what we want as an end product. But we could easily have put year first. We saw visually what would happen when we did this in the Pivot Table.

### summarize multiple variables

We can summarize multiple variables at a time.

So far we’ve summarized the count of lobster observations. Let’s also calculate the mean and standard deviation. First let’s use the mean() function to calculate the mean. We do this within the same summarize() function, but we can add a new line to make it easier to read. Notice how when you put your curser within the parenthesis and hit return, the indentation will automatically align.

lobsters %>%

group\_by(site, year) %>%

summarize(count\_by\_siteyear = n(),

mean\_size\_mm = mean(size\_mm, na.rm=TRUE),

sd\_size\_mm = sd(size\_mm, na.rm=TRUE))

### Table formatting with kable()

There are several options for formatting tables in RMarkdown; we’ll show one here from the kableExtra package and learn more about it tomorrow.

It works nicely with the pipe operator, so we can build do this from our new object:

*## make a table with our new variable*

siteyear\_summary %>%

kable()

Write this **in Markdown** but replace the # with a backtick (`): “There are #r nrow(lobsters)# total lobsters included in this report.” Let’s knit to see what happens.

I hope you can start to imagine the possibilities. If you wanted to write which year had the most observations, or which site had a decreasing trend, you would be able to.

### **Activity**

1. Build from our analysis and calculate the median lobster size for each site year. Your calculation will use the size\_mm variable and function to calculate the median (Hint: ?median)
2. create and ggsave() a plot.

Then, save, commit, and push your .Rmd, .html, and .png.

Solution (no peeking):

siteyear\_summary <- lobsters %>%

group\_by(site, year) %>%

summarize(count\_by\_siteyear = n(),

mean\_size\_mm = mean(size\_mm, na.rm = TRUE),

sd\_size\_mm = sd(size\_mm, na.rm = TRUE),

median\_size\_mm = median(size\_mm, na.rm = TRUE))

## `summarise()` regrouping output by 'site' (override with `.groups` argument)

*## a ggplot option:*

ggplot(data = siteyear\_summary, aes(x = year, y = median\_size\_mm, color = site)) +

geom\_line()

How to publish Quarto using terminal:

quarto publish quarto-pub

library(gt)

exibble %>%

gt() %>%

fmt\_number(

columns = num,

decimals = 3,

use\_seps = FLASE

)

### dplyr::count()

Now that we’ve spent time with group\_by %>% summarize, there is a shortcut if you only want to summarize by count. This is with a function called count(), and it will group\_by your selected variable, count, and then also ungroup. It looks like this:

lobsters %>%

count(site, year)

*## This is the same as:*

lobsters %>%

group\_by(site, year) %>%

summarize(n = n()) %>%

ungroup()

Hey, we could update our RMarkdown text knowing this: There are #r count(lobsters)# total lobsters included in this summary.

## mutate()

There are a lot of times where you don’t want to summarize your data, but you do want to operate beyond the original data. This is often done by adding a column. We do this with the mutate() function from dplyr. Let’s try this with our original lobsters data. The sizes are in millimeters but let’s say it was important for them to be in meters. We can add a column with this calculation:

lobsters %>%

mutate(size\_m = size\_mm / 1000)

If we want to add a column that has the same value repeated, we can pass it just one value, either a number or a character string (in quotes). And let’s save this as a variable called lobsters\_detailed

lobsters\_detailed <- lobsters %>%

mutate(size\_m = size\_mm / 1000,

millenia = 2000,

observer = "Allison Horst")

## select()

We will end with one final function, select. This is how to choose, retain, and move your data by columns:

Let’s say that we want to present this data finally with only columns for date, site, and size in meters. We would do this:

lobsters\_detailed %>%

select(date, site, size\_m)

# Tidying

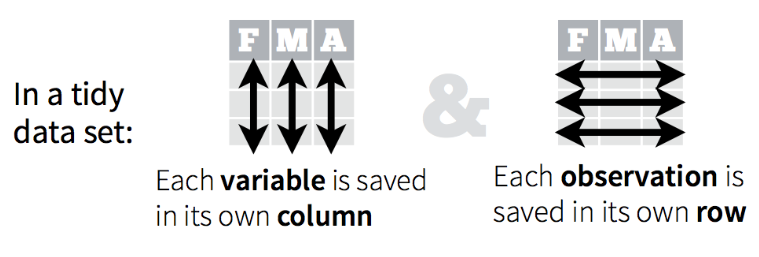
## **Summary**

In previous sessions, we learned to read in data, do some wrangling, and create a graph and table.

Here, we’ll continue by reshaping data frames (converting from long-to-wide, or wide-to-long format), separating and uniting variable (column) contents, and finding and replacing string patterns.

### Tidy data

“Tidy” might sound like a generic way to describe non-messy looking data, but it is actually a specific data structure. When data is tidy, it is rectangular with each variable as a column, each row an observation, and each cell contains a single value



### **Objectives**

In this session we’ll learn some tools to help make our data **tidy** and more coder-friendly. Those include:

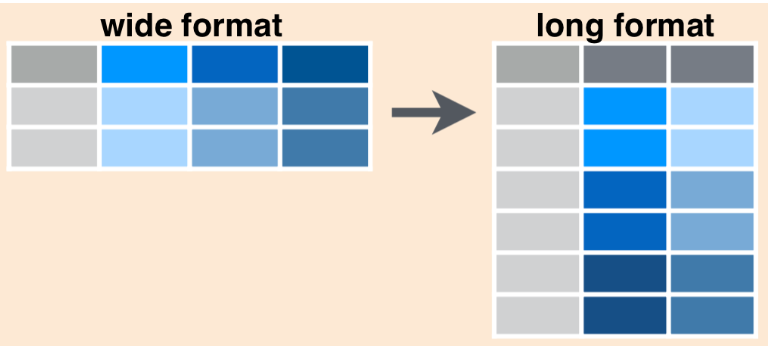
* Use tidyr::pivot\_wider() and tidyr::pivot\_longer() to reshape data frames
* janitor::clean\_names() to make column headers more manageable
* tidyr::unite() and tidyr::separate() to merge or separate information from different columns
* Detect or replace a string with stringr functions

## tidyr::pivot\_longer() to reshape from wider-to-longer format

If we look at inverts, we can see that the year variable is actually split over 3 columns, so we’d say this is currently in **wide format**.

There may be times when you want to have data in wide format, but often with code it is more efficient to convert to **long format** by gathering together observations for a variable that is currently split into multiple columns.

Schematically, converting from wide to long format using pivot\_longer() looks like this:



We’ll use tidyr::pivot\_longer() to gather data from all years in *inverts* (columns 2016, 2017, and 2018) into two columns:

* one called *year*, which contains the year
* one called *sp\_count* containing the number of each species observed.

The new data frame will be stored as *inverts\_long*:

*# Note: Either single-quotes, double-quotes, OR backticks around years work!*

inverts\_long <- pivot\_longer(data = inverts,

cols = '2016':'2018',

names\_to = "year",

values\_to = "sp\_count")

One thing that isn’t obvious at first (but would become obvious if you continued working with this data) is that since those year numbers were initially column names (characters), when they are stacked into the *year* column, their class wasn’t auto-updated to numeric.

**Explore the class of *year* in *inverts\_long*:**

class(inverts\_long$year)

## [1] "character"

That’s a good thing! We don’t want R to update classes of our data without our instruction. We’ll use dplyr::mutate() in a different way here: to create a new column (that’s how we’ve used mutate() previously) that has the same name of an existing column, in order to update and overwrite the existing column.

In this case, we’ll mutate() to add a column called *year*, which contains an as.numeric() version of the existing *year* variable:

*# Coerce "year" class to numeric:*

inverts\_long <- inverts\_long %>%

mutate(year = as.numeric(year))

Checking the class again, we see that *year* has been updated to a numeric variable:

class(inverts\_long$year)

## [1] "numeric"

## tidyr::pivot\_wider() to convert from longer-to-wider format

In the previous example, we had information spread over multiple columns that we wanted to gather. Sometimes, we’ll have data that we want to spread over multiple columns.

For example, imagine that starting from inverts\_long we want each species in the common\_name column to exist as its **own column**. In that case, we would be converting from a longer to a wider format, and will use tidyr::pivot\_wider().

Specifically for our data, we’ll use pivot\_wider() to spread the common\_name across multiple columns as follows:

inverts\_wide <- inverts\_long %>%

pivot\_wider(names\_from = common\_name,

values\_from = sp\_count)

## janitor::clean\_names() to clean up column names

The janitor package by Sam Firke is a great collection of functions for some quick data cleaning, like:

* janitor::clean\_names(): update column headers to a case of your choosing
* janitor::get\_dupes(): see all rows that are duplicates within variables you choose
* janitor::remove\_empty(): remove empty rows and/or columns
* janitor::adorn\_\*(): jazz up tables

Here, we’ll use janitor::clean\_names() to convert all of our column headers to a more convenient case - the default is **lower\_snake\_case**, which means all spaces and symbols are replaced with an underscore (or a word describing the symbol), all characters are lowercase, and a few other nice adjustments.

For example, janitor::clean\_names() would update these nightmare column names into much nicer forms:

* My...RECENT-income! becomes my\_recent\_income
* SAMPLE2.!test1 becomes sample2\_test1
* ThisIsTheName becomes this\_is\_the\_name
* 2015 becomes x2015

If we wanted to then use these columns (which we probably would, since we created them), we could clean the names to get them into more coder-friendly lower\_snake\_case with janitor::clean\_names():

inverts\_wide <- inverts\_wide %>%

clean\_names()

names(inverts\_wide)

And there are other case options in clean\_names(), like:

* “snake” produces snake\_case (the default)
* “lower\_camel” or “small\_camel” produces lowerCamel
* “upper\_camel” or “big\_camel” produces UpperCamel
* “screaming\_snake” or “all\_caps” produces ALL\_CAPS
* “lower\_upper” produces lowerUPPER
* “upper\_lower” produces UPPERlower

## tidyr::unite() and tidyr::separate() to combine or separate information in column(s)

Sometimes we’ll want to separate contents of a single column into multiple columns, or combine entries from different columns into a single column.

For example, the following data frame has genus and species in separate columns:

We may want to combine the genus and species into a single column, scientific\_name:

Or we may want to do the reverse (separate information from a single column into multiple columns). Here, we’ll learn tidyr::unite() and tidyr::separate() to help us do both.

To demonstrate uniting information from separate columns, we’ll make a single column that has the combined information from *site* abbreviation and *year* in *inverts\_long*.

We need to give tidyr::unite() several arguments:

* **data:** the data frame containing columns we want to combine (or pipe into the function from the data frame)
* **col:** the name of the new “united” column
* the **columns you are uniting**
* **sep:** the symbol, value or character to put between the united information from each column

inverts\_unite <- inverts\_long %>%

unite(col = "site\_year", *# What to name the new united column*

c(site, year), *# The columns we'll unite (site, year)*

sep = "\_") *# How to separate the things we're uniting*

#### **Activity:**

**Task:** Create a new object called ‘inverts\_moyr,’ starting from inverts\_long, that unites the month and year columns into a single column named “mo\_yr,” using a slash “/” as the separator. Then try updating the separator to something else! Like “hello!”

**Solution:**

inverts\_moyr <- inverts\_long %>%

unite(col = "mo\_yr", *# What to name the new united column*

c(month, year), *# The columns we'll unite (site, year)*

sep = "/")

**Merging information from > 2 columns (not done in workshop)**

tidyr::unite() can also combine information from more than two columns. For example, to combine the site, common\_name and year columns from inverts\_long, we could use:

*# Uniting more than 2 columns:*

inverts\_triple\_unite <- inverts\_long %>%

tidyr::unite(col = "year\_site\_name",

c(year, site, common\_name),

sep = "-") *# Note: this is a dash*

head(inverts\_triple\_unite)

**Merging information from > 2 columns (not done in workshop)**

tidyr::unite() can also combine information from *more* than two columns. For example, to combine the *site*, *common\_name* and *year* columns from *inverts\_long*, we could use:

*# Uniting more than 2 columns:*

inverts\_triple\_unite <- inverts\_long %>%

tidyr::unite(col = "year\_site\_name",

c(year, site, common\_name),

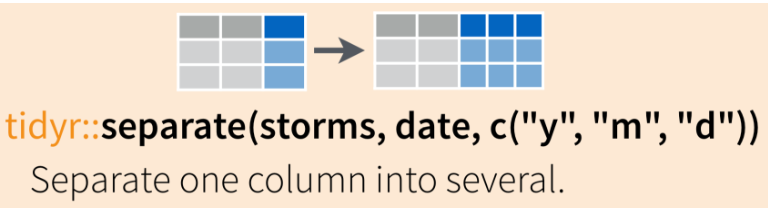
sep = "-") *# Note: this is a dash*

### tidyr::separate() to separate information into multiple columns

While tidyr::unite() allows us to combine information from multiple columns, it’s more likely that you’ll start with a single column that you want to split up into pieces.

For example, I might want to split up a column containing the genus and species (Scorpaena guttata) into two separate columns (Scorpaena | guttata), so that I can count how many Scorpaena organisms exist in my dataset at the genus level.

Use tidyr::separate() to “separate a character column into multiple columns using a regular expression separator.”



Let’s start again with *inverts\_unite*, where we have combined the *site* and *year* into a single column called *site\_year*. If we want to **separate** those, we can use:

inverts\_sep <- inverts\_unite %>%

tidyr::separate(site\_year, into = c("my\_site", "my\_year"))

## stringr::str\_replace() to replace a pattern

Was data entered in a way that’s difficult to code with, or is just plain annoying? Did someone wrongly enter “fish” as “fsh” throughout the spreadsheet, and you want to update it everywhere?

Use stringr::str\_replace() to automatically replace a string pattern.

**Warning**: The pattern will be replaced everywhere - so if you ask to replace “fsh” with “fish,” then “offshore” would be updated to “offishore.” Be careful to ensure that when you think you’re making one replacement, you’re not also replacing something else unexpectedly.

Starting with inverts, let’s any place we find “california” we want to replace it with the abbreviation “CA”:

ca\_abbr <- inverts %>%

mutate(

common\_name =

str\_replace(common\_name,

pattern = "california",

replacement = "CA")

)

# Filters and joins

## Summary

In previous sessions, we’ve learned to do some basic wrangling and find summary information with functions in the dplyr package, which exists within the tidyverse. In this session, we’ll expand our data wrangling toolkit using:

* filter() to conditionally subset our data by **rows**, and
* \*\_join() functions to merge data frames together
* And we’ll make a nicely formatted HTML table with kable() and kableExtra

The combination of filter() and \*\_join() - to return rows satisfying a condition we specify, and merging data frames by like variables - is analogous to the useful VLOOKUP function in Excel.

### Objectives

* Use filter() to subset data frames, returning **rows** that satisfy variable conditions
* Use full\_join(), left\_join(), and inner\_join() to merge data frames, with different endpoints in mind
* Use filter() and \*\_join() as part of a wrangling sequence

## Set-up: Create a new .Rmd, attach packages & get data

Create a new R Markdown document in your r-workshop project and knit to save as **filter\_join.Rmd**. Remove all the example code (everything below the set-up code chunk).

In this session, we’ll attach four packages:

* tidyverse
* readxl
* here
* kableExtra

Attach the packages in the setup code chunk in your .Rmd:

library(tidyverse)

library(readxl)

library(here)

library(kableExtra)

Then create a new code chunk to read in two files from your ‘data’ subfolder:

* fish.csv
* kelp\_fronds.xlsx (read in only the “abur” worksheet by adding argument sheet = "abur" to read\_excel())

*# Read in data:*

fish <- read\_csv(here("data", "fish.csv"))

kelp\_abur <- read\_excel(here("data", "kelp\_fronds.xlsx"), sheet = "abur")

We should always explore the data we’ve read in. Use functions like View(), names(), summary(), head() and tail() to check them out. Now, let’s use filter() to decide which observations (rows) we’ll keep or exclude in new subsets, similar to using Excel’s VLOOKUP function or filter tool.

## dplyr::filter() to conditionally subset by rows

Use filter() to let R know which **rows** you want to keep or exclude, based whether or not their contents match conditions that you set for one or more variables.

Some examples in words that might inspire you to use filter():

* “I only want to keep rows where the temperature is greater than 90°F.”
* “I want to keep all observations **except** those where the tree type is listed as **unknown**.”
* “I want to make a new subset with only data for mountain lions (the species variable) in California (the state variable).”

When we use filter(), we need to let R know a couple of things:

* What data frame we’re filtering from
* What condition(s) we want observations to **match** and/or **not match** in order to keep them in the new subset

### Filter rows by matching a single character string

Let’s say we want to keep all observations from the **fish** data frame where the common name is “garibaldi” (fun fact: that’s California’s official marine state fish, protected in California coastal waters!).

Here, we need to tell R to only keep rows from the **fish** data frame when the common name (**common\_name** variable) exactly matches **garibaldi**.

Use == to ask R to look for exact matches:

fish\_garibaldi <- fish %>%

filter(common\_name == "garibaldi")

Check out the **fish\_garibaldi** object to ensure that only garibaldi observations remain.

Troubleshooting: If you see Error: attempt to use zero-length variable name **it is because you have accidentally highlighted the backticks along with the R code**.

### Filter rows based on numeric conditions

Use expected operators (>, <, >=, <=, ==) to set conditions for a numeric variable when filtering. For this example, we only want to retain observations when the **total\_count** column value is >= 50:

fish\_over50 <- fish %>%

filter(total\_count >= 50)

### Filter to return rows that match this OR that OR that

What if we want to return a subset of the **fish** df that contains garibaldi, blacksmith OR black surfperch?

There are several ways to write an “OR” statement for filtering, which will keep any observations that match Condition A or Condition B or Condition C. In this example, we will create a subset from **fish** that only contains rows where the **common\_name** is garibaldi or blacksmith or black surfperch.

Way 1: You can indicate **OR** using the vertical line operator | to indicate “OR”:

fish\_3sp <- fish %>%

filter(common\_name == "garibaldi" |

common\_name == "blacksmith" |

common\_name == "black surfperch")

Alternatively, if you’re looking for multiple matches in the same variable, you can use the %in% operator instead. Use %in% to ask R to look for any matches within a vector:

fish\_3sp <- fish %>%

filter(common\_name %in% c("garibaldi", "blacksmith", "black surfperch"))

Notice that the two methods above return the same thing.

**Critical thinking:** In what scenario might you NOT want to use %in% for an “or” filter statement? Hint: What if the “or” conditions aren’t different outcomes for the same variable?

#### Activity

**Task:** Create a subset from **fish** called **fish\_gar\_2016** that keeps all observations if the year is 2016 OR the common name is “garibaldi.”

**Solution:**

fish\_gar\_2016 <- fish %>%

filter(year == 2016 | common\_name == "garibaldi")

### Filter to return observations that match **this** AND **that**

In the examples above, we learned to keep observations that matched any of a number of conditions (**or** statements).

Sometimes we’ll only want to keep observations that satisfy multiple conditions (e.g., to keep this observation it must satisfy this condition **AND** that condition). For example, we may want to create a subset that only returns rows from **fish** where the **year** is 2018 and the **site** is Arroyo Quemado “aque”

In filter(), add a comma (or ampersand ‘&’) between arguments for multiple “and” conditions:

aque\_2018 <- fish %>%

filter(year == 2018, site == "aque")

Like most things in R, there are other ways to do the same thing. For example, you could do the same thing using & (instead of a comma) between “and” conditions:

*# Use the ampersand (&) to add another condition "and this must be true":*

aque\_2018 <- fish %>%

filter(year == 2018 & site == "aque")

Or you could just do two filter steps in sequence:

*# Written as sequential filter steps:*

aque\_2018 <- fish %>%

filter(year == 2018) %>%

filter(site == "aque")

### Activity: combined filter conditions

**Challenge task:** Create a subset from the **fish** data frame, called **low\_gb\_wr** that only contains:

* Observations for garibaldi or rock wrasse
* AND the total\_count is less than or equal to 10

**Solution:**

low\_gb\_wr <- fish %>%

filter(common\_name %in% c("garibaldi", "rock wrasse"),

total\_count <= 10)

### stringr::str\_detect() to filter by a partial pattern

Sometimes we’ll want to keep observations that contain a specific string pattern within a variable of interest.

For example, consider the fantasy data below:

|  |  |
| --- | --- |
| **id** | **species** |
| 1 | rainbow rockfish |
| 2 | blue rockfish |
| 3 | sparkle urchin |
| 4 | royal blue fish |

There might be a time when we would want to use observations that:

* Contain the string “fish,” in isolation or within a larger string (like “rockfish”)
* Contain the string “blue”

In those cases, it would be useful to **detect** a string pattern, and potentially keep any rows that contain it. Here, we’ll use stringr::str\_detect() to find and keep observations that contain our specified string pattern.

Let’s detect and keep observations from **fish** where the **common\_name** variable contains string pattern “black.” Note that there are two fish, blacksmith and black surfperch, that would satisfy this condition.

Using filter() + str\_detect() in combination to find and keep observations where the **site** variable contains pattern “sc”:

fish\_bl <- fish %>%

filter(str\_detect(common\_name, pattern = "black"))

So str\_detect() returns is a series of TRUE/FALSE responses for each row, based on whether or not they contain the specified pattern. In that example, any row that does contain “black” returns TRUE, and any row that does not contain “black” returns FALSE.

### **Activity**

**Task:** Create a new object called **fish\_it**, starting from **fish**, that only contains observations if the **common\_name** variable contains the string pattern “it.” What species remain?

**Solution:**

fish\_it <- fish %>%

filter(str\_detect(common\_name, pattern = "it"))

*# blacksmITh and senorITa remain!*

We can also exclude observations that contain a set string pattern by adding the negate = TRUE argument within str\_detect().

## dplyr::\*\_join() to merge data frames

There are a number of ways to merge data frames in R. We’ll use full\_join(), left\_join(), and inner\_join() in this session.

From R Documentation (?join):

* full\_join(): “returns all rows and all columns from both x and y. Where there are not matching values, returns NA for the one missing.” Basically, nothing gets thrown out, even if a match doesn’t exist - making full\_join() the safest option for merging data frames. When in doubt, full\_join().
* left\_join(): “return all rows from x, and all columns from x and y. Rows in x with no match in y will have NA values in the new columns. If there are multiple matches between x and y, all combinations of the matches are returned.”
* inner\_join(): “returns all rows from x where there are matching values in y, and all columns from x and y. If there are multiple matches between x and y, all combination of the matches are returned.” This will drop observations that don’t have a match between the merged data frames, which makes it a riskier merging option if you’re not sure what you’re trying to do.

We will use **kelp\_abur** as our “left” data frame, and **fish** as our “right” data frame, to explore different join outcomes.

### full\_join() to merge data frames, keeping everything

When we join data frames in R, we need to tell R a couple of things (and it does the hard joining work for us):

* Which data frames we want to merge together
* Which variables to merge by

Use full\_join() to safely combine two data frames, keeping everything from both and populating with NA as necessary.

Example: use full\_join() to combine **kelp\_abur** and **fish**:

abur\_kelp\_fish <- kelp\_abur %>%

full\_join(fish, by = c("year", "site"))

Let’s look at the merged data frame with View(abur\_kelp\_fish). A few things to notice about how full\_join() has worked:

1. All columns that existed in **both data frames** still exist
2. All observations are retained, even if they don’t have a match. In this case, notice that for other sites (not ‘abur’) the observation for fish still exists, even though there was no corresponding kelp data to merge with it.
3. The kelp frond data is joined to all observations where the joining variables (year, site) are a match, which is why it is repeated 5 times for each year (once for each fish species).

Because all data (observations & columns) are retained, full\_join() is the safest option if you’re unclear about how to merge data frames.

### left\_join(x,y) to merge data frames, keeping everything in the ‘x’ data frame and only matches from the ‘y’ data frame

Now, we want to keep all observations in kelp\_abur, and merge them with fish while only keeping observations from fish that match an observation within abur. When we use left\_join(), any information from fish that don’t have a match (by year and site) in kelp\_abur won’t be retained, because those wouldn’t have a match in the left data frame.

kelp\_fish\_left <- kelp\_abur %>%

left\_join(fish, by = c("year","site"))

Notice when you look at **kelp\_fish\_left**, data for other sites that exist in fish do **not** get joined, because left\_join(df\_a, df\_b) will only keep observations from df\_b if they have a match in df\_a!

### inner\_join() to merge data frames, only keeping observations with a match in **both**

Use inner\_join() if you **only** want to retain observations that have matches across **both data** frames. Caution: this is built to exclude any observations that don’t match across data frames by joined variables - double check to make sure this is actually what you want to do!

For example, if we use inner\_join() to merge fish and kelp\_abur, then we are asking R to **only return observations where the joining variables (year and site) have matches in both data frames.** Let’s see what the outcome is:

kelp\_fish\_injoin <- kelp\_abur %>%

inner\_join(fish, by = c("year", "site"))

*# kelp\_fish\_injoin*

Here, we see that only observations (rows) where there is a match for year and site in both data frames are returned.

Here, we see that only observations (rows) where there is a match for year and site in both data frames are returned.

### filter() and join() in a sequence

Now let’s combine what we’ve learned about piping, filtering and joining!

Let’s complete the following as part of a single sequence (remember, check to see what you’ve produced after each step) to create a new data frame called **my\_fish\_join**:

* Start with **fish** data frame
* Filter **fish** to only including observations for 2017 at Arroyo Burro
* Join the **kelp\_abur** data frame to the resulting subset using left\_join()
* Add a new column that contains the ‘fish per kelp fronds’ density (total\_count / total\_fronds)

That sequence might look like this:

my\_fish\_join <- fish %>%

filter(year == 2017, site == "abur") %>%

left\_join(kelp\_abur, by = c("year", "site")) %>%

mutate(fish\_per\_frond = total\_count / total\_fronds)

## An HTML table with kable() and kableExtra

With any data frame, you can a nicer looking table in your knitted HTML using knitr::kable() and functions in the kableExtra package.

Start by using kable() with my\_fish\_join, and see what the default HTML table looks like in your knitted document:

kable(my\_fish\_join)

Simple, but quick to get a clear & useful table! Now let’s spruce it up a bit with kableExtra::kable\_styling() to modify HTML table styles:

my\_fish\_join %>%

kable() %>%

kable\_styling(bootstrap\_options = "striped",

full\_width = FALSE)

…with many other options for customizing HTML tables! Make sure to check out [“Create awesome HTML tables with knitr::kable() and kableExtra” by Hao Zhu](https://cran.r-project.org/web/packages/kableExtra/vignettes/awesome_table_in_html.html) for more examples and options.