Managing data and building pipelines in R

Kate E. Langwig, Alex T. Grimaudo, Macy J. Kailing - IBRC

2022

Field, lab, or software program to R

The first section will focus on best practices for organizing data **from the field, lab, or elsewhere** that will make the transition to wrangling it in R easiest.

Learning objectives

Improve data management through:

- Building pipelines
- Datasheet format
- File management
- Maintaining relational structure between data types
- Meta-data tracking

Why pipelines?

Pipelines are ways of carefully recording and systematizing the steps you take to work with your data

Ideally your project will depend on:

- Some data files: spreadsheets storing data (.csv files from Excel)
- Some scripts (more on this later)
- Something that tells you how these things go together: README file

Getting started | *before beginning data collection*

What will your data **look like** and how will you **analyze it**?

- Determine what your data points/observations/sampling units are (individual bat, acoustic detector, insect trap)
- Use these to link across data types
- Stay consistent by planning ahead
  - spend a little time up front to save lots of time later
- Whenever possible, record in a way that reflects your data entry procedure

For example. . .

To streamline the process between data collection and entry to spreadsheets and reduce transcription errors, we can ***structure datasheets to***:

- minimize extraneous information
- keep single observations to one row
- have distinct pages per sampling event or unit
- organize columns in the same order you will collect the data

Managing data files and folders | *after you collect samples or receive results*

Proper file (electronic and hard copy) management will save you TONS of time, headaches, and **most importantly downstream errors**.

Always. . .

- scan datasheets to save electronically and make copies
- retain unaltered original copies of datasheets or output files
- enter data from scanned sheets
- develop a quality control procedure

Consider. . .

- organizing as if you will continuously build the dataset and anticipate you will need to go back
- "*if someone continued this project, is my data pipeline in a place I could hand it over seamlessly?*"

An example with acoustic data | multiple sites; multiple detectors; thousands of calls

This can quickly become overwhelming with an abundance of folders and files; each containing many dates, similar but different output files, lots of varying information

Tips for organizing bulky data streams

- Organize by detector and give each a unique name
- Add prefix to all call files with unique name (bulk rename utility for PCs/select multiple and rename on Mac)
- Store call files in folders by each **detector**
- Minimize transfer of files by keeping folder structure simple (don't create unnecessary sub-folders)
- Store together the **output files** (from analysis software) you will work with directly
- Store raw files separately and safely

Relational structure | *collating multiple streams of data*

Set yourself up to easily combine different types of data

- Each observation (or row) within any dataframe needs a unique identifier (i.e., sample ID)
- Maintain the same unique identifiers across **ALL YOUR DATA FILES** until the end of time :)
- If observations differ in scale, create unique identifier at the lowest possible level (sampling unit instead of individual observation)
  - Easier to move up than down

Metadata and data dictionaries | *preserving tangential information*

How do you store data that is important but not specific to a single 'observation'?

- Net effort, environmental conditions during sampling event, site characteristics, etc
- Create a separate metadata file that includes the same variables as observational data (e.g., site, date, species)

Metadata and *data dictionaries* | *preserving tangential information*

How would someone stepping into the project understand the data you collect?

- We often collect data that isn't intuitive to non-bat biologists (wing score, RFA, reproductive condition, etc)
- ***Make a data dictionary***:
  - A table of definitions in an Excel spreadsheet (or other program)
  - Each variable from the datasheet is described thoroughly, including precisely what it measures

Record keeping | *long-term pipeline success*

You will inevitably forget your procedure.

- Write a protocol for how your files are organized/linked together
- Track any modifications using your README file or other notebook.
- ***Advantages***:
  - Allows more flexibility and time saving if you need someone to help
  - Provides an easy reference for yourself
  - Reminds you how you set up your pipeline if you need to be away from the project for any time

Now, data is collected, entered to spreadsheets, and filed appropriately!

You have collected your data... now what?

Now you are ready to work with your data in R!

- Important naming conventions
- Types of data R recognizes
  - how to work with dates
- Checking your data for errors
  - unique()
  - hist()
  - head()
  - summary()
  - plot()

Read in your data

Please navigate to the below URL to access the data we will be using today:

https://github.com/VTQuantMethodsEEB/ibrc_workshop

Read in your data

read.csv() is the function to use when reading your spreadsheets into R.

Using .csv files to store and read-in your data is important!

- Stores data in a text format (human-readable; transferrable)
- Can store large amounts of data in simple format
- Can be read by most programs

Naming conventions

The R environment is very sensitive to how you name objects and slight differences are important.

- “Maple”
- ” Maple”
- “maple”
- “maple”
- “maple.”

Make sure your naming is consistent **in your .csv files!**

Naming conventions

Additional good practices when naming things in R:

- The names of R objects also have to start with a letter, not a number.
- Use mostly numbers, underscores (_), and dots (.) in your names.
- Don't use potentially confusing names like I or O
- Don't use built-in names (like c, list, or data) for variables.
- Make readable variable names using camelCase, snake_case, or kebab.case
- Avoid variableNamesThatAreExcessivelyLong

Data types

There are several types of data that R recognizes, listed below:

- **Logical**: True/False data
- **Numeric**: All real numbers with or without decimal values
- **Integer**: Real numbers without decimal values.
- **Character**: String data. Thing of this as any unique string of values, such as "1dkdl;" or "Apple_Pie". This is the default data type when R does not recognize data as being of another type.
- **Factor**: Used to describe items that can have a known set of values (species, habitat, etc.). Categorical variables, in statistical terms.
- **Date**: calendar date, e.g. "10-22-1994"

What type of data do we have?

A quick way to check what kind of data we have in our dataframe is with the **str() function**

Dates

Dates in R are notorious despite being common forms of data!

- Default format on Mac: mo/day/two digit year –
  e.g. 01/13/18 is January 13, 2018
- Default format on PC: mo/day/four digit year -
  e.g. 01/13/2018

We can tell R to read our data as dates using **as.Date()**.

Dates: lubridate

'**Lubridate**' is a package built to make reading and changing date data easy.

- Can turn numerical data into dates: 20101215 -> 2010-12-15
- Can quickly change format of dates with simple functions
- Helpful when extracting data from dates (for example, if you just want to extract month data from your dates)
- Useful when handling timezone differences.

Checking Data

Checking your data is very important! R can only work with the data you give it – make sure you're giving it good data. Check for common sources of faulty data:

- Naming inconsistencies
  - Follow good naming conventions!
- Duplicated data
  - Are your repeating values real, or a product of a bug in your code?
- Data that doesn't make sense biologically
  - A little brown bat weighing 72 grams whereas the others weighed around 7-8 grams probably isn't real!

Where should you correct your data?

Checking Data

Some useful tools in R to check your data for errors:

- head() – gives you first rows of your dataframe
  - This is a good first pass
- unique() – returns all of the unique values contained in a specified dataset
  - This is particularly useful when looking for naming inconsistencies
- hist() – This will show you the frequency distribution of your specified data.
  - Good for identifying anomalies or outliers in your data. **Interpret carefully!**

Checking Data

Some useful tools in R to check your data for errors:

- summary() – Returns summary statistics of your specified data.
  - Again, good for identifying anomalies or outliers in your data.
- Plot your data!
  - plot() or ggplot()
  - Visualize the shape of your data. Does something look off?

Checking Data

Check your data frequently! You should check your data when you:

- Read in its .csv file.
- Change the shape of your data.
- Merge datasets.
- Do calculations and make new datasets.

Version control

Using version control and management services will vastly reduce the chance a serious error sneaks by.

- **GitHub** (where you downloaded the data today) is a common code management service.
  - Allows team members to collaborate on code.
  - Version control and track-changes.
  - Most journals today ask for your code, and GitHub repositories are a common way for it to be published.

Introducing Tidyverse

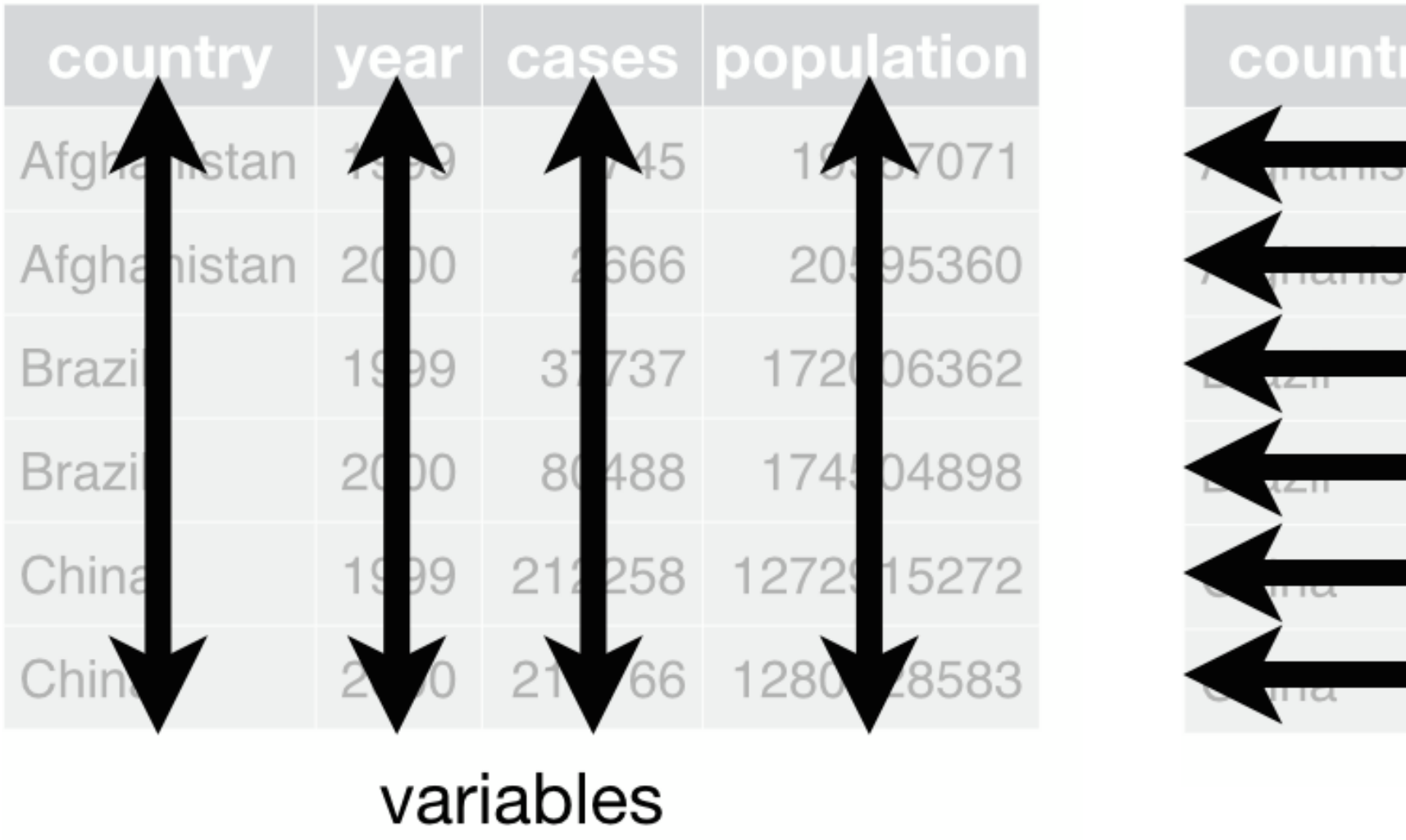
- the tidyverse is a powerful set of separate packages
- https://tidyverse.tidyverse.org/
- powerful packages within the tidyverse include:
  - dplyr (management & manipulation)
  - tidyr (cleaning)
  - stringr (dealing with words inside columns)
  - lubridate (dealing with dates/times)

Tidy(ing) data

Hadley Wickham has defined a concept of tidy data, and has introduced the `tidyverse` package.

- Each variable is in a column
- Each observation is in a row
- "Long" rather than "wide" form
- Sometimes duplicates data
- Statistical modeling tools and graphical tools (especially the **ggplot2** package) in R work best with long form

An example of tidy data



Putting your data in tidy format

- Discerning what is a variable can be hard when making data files
- For example, species in my bat dataset is usually a single variable
- I usually also include a "count" column (the number of individuals at a site)
- But what if I wanted to test the effect of the count of one species (e.g.MYSE) on another?
- I'd need to rearrange

Piping

- tidyverse syntax is VERY different from base R
- it relies on using %>%
- this is called a pipe
- it says the word “then”

First load the package

```
#install if you don't have it
library(tidyverse)
```

```
## -- Attaching packages ---------------------------------
## v ggplot2 3.3.5     v purrr   0.3.4
## v tibble  3.1.6     v dplyr   1.0.7
## v tidyr   1.1.4     v stringr 1.4.0
## v readr   2.1.1     v forcats 0.5.1
## -- Conflicts ------------------------------------------
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

Example

```
iris %>%
# This says take the 'built-in' dataset iris
  # THEN filter just the iris species == setosa
  filter(Species=="setosa")
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Spe
## 1           5.1         3.5          1.4         0.2  se
## 2           4.9         3.0          1.4         0.2  se
## 3           4.7         3.2          1.3         0.2  se
## 4           4.6         3.1          1.5         0.2  se
## 5           5.0         3.6          1.4         0.2  se
## 6           5.4         3.9          1.7         0.4  se
## 7           4.6         3.4          1.4         0.3  se
## 8           5.0         3.4          1.5         0.2  se
## 9           4.4         2.9          1.4         0.2  se
## 10          4.9         3.1          1.5         0.1  se
## 11          5.4         3.7          1.5         0.2  se
## 12          4.8         3.4          1.6         0.2  se
```

Critical tools for managing data in tidyverse

- `pivot_longer`, `pivot_wider`
- `mutate`: add a column
- `select`: select columns
- `filter`: select rows
- `group_by`: group then do something (usually mutate or summarise)
- `summarise`: make a summary table
- `arrange`: sort
- `bind_rows`: combines dataframes by stacking them on top of each other - keeps all columns; will make duplicates if the columns don't match (e.g 'species', 'Species')
- `left_join`: merge see other `join` options

Read in two datasets

- you will need to tell R where to find them on your computer
- I am using an .Rproj file, which sets my working directory
- https://github.com/VTQuantMethodsEEB/ibrc_workshop/tree/main/data_management

First read in batdat, each row is a bat

```
# in this dataset, each row is a bat that
# is positive or negative for P. destructans
batdat = read.csv("bat_data.csv")
head(batdat)
```

```
##       swab_id gd         gdL swab_type state site    date
## 1 KL15WI0002  1 0.00007560        BAT    WI HORS 2/27/15
## 2 KL15WI0003  1 0.47879100        BAT    WI HORS 2/27/15
## 3 KL15WI0004  0          NA        BAT    WI HORS 2/27/15
## 4 KL15WI0005  1 0.00000551        BAT    WI HORS 2/27/15
## 5 KL15WI0006  1 0.00003560        BAT    WI HORS 2/27/15
## 6 KL15WI0007  1 0.00003160        BAT    WI HORS 2/27/15
```

Then, read in batcount, each row is a site

```
# in this dataset, each row the count of hibernation site
# for a partcular species on a particular date
# even though sites were counted on the same date
# species are repeated across dates
batcount = read.csv("bat_count.csv")
head(batcount)
```

```
##   site species    date count
## 1 BEAR    EPFU  3/4/15     9
## 2 BEAR    EPFU  3/7/16     5
## 3 BEAR    MYLU 11/9/15   116
## 4 BEAR    MYLU 3/10/17    38
## 5 BEAR    MYLU  3/4/15    97
## 6 BEAR    MYLU  3/7/16   122
```

Group by

- `group_by` is my favorite tidyverse command which has cut my need to write loops in half
- `group_by` allows you to do calculations on groups of things, for example, by species or year
- `group_by` is kind of like using Excel's `filter`
- For example, if I group_by `species, date`, this is the equivalent in Excel of selecting a species (e.g. MYLU) and a specific date (e.g. March 1 2016, 3/1/16) except group_by does this for every species and date combination in your dataset
- This is incredibly useful because in other programs, you might need to write a for loop to have this capability

Group by species

```
batdat$lgdL = log10(batdat$gdL)
batdat %>%
  group_by(species) %>%
    summarise(mean.fungal.loads=mean(lgdL,na.rm=TRUE))
```

```
## # A tibble: 5 x 2
##   species   mean.fungal.loads
##   <chr>                 <dbl>
## 1 EPFU                  -3.64
## 2 MYLU                  -3.03
## 3 MYSE                  -3.69
## 4 PESU                  -2.04
## 5 SUBSTRATE             -4.11
```

Summarise versus Mutate

- `summarise` creates a new dataframe
- `mutate` does a calculation where it add a new column to your existing dataframe

Importance of assigning

```
fungal.load.table = batdat %>%
  group_by(species) %>%
  summarise(mean.fungal.loads=mean(lgdL,na.rm=TRUE))

fungal.load.table
```

```
## # A tibble: 5 x 2
##   species   mean.fungal.loads
##   <chr>                 <dbl>
## 1 EPFU                  -3.64
## 2 MYLU                  -3.03
## 3 MYSE                  -3.69
## 4 PESU                  -2.04
## 5 SUBSTRATE             -4.11
```

Assigning

- When using summarise, it's best to call your summarised object a new name (e.g. fungal.load.table)
- This is typically not necessary when using mutate which just adds a column to an existing dataset

```
batdat = batdat %>%
  #take batdat, then group by somethings
  #we re-assign batdat to batdat because we want to add th
    group_by(site,species,date) %>%
  #you can group_by multiple things
    mutate(sample.size=n())
#this adds a column to the dataframe
#using the function n(), which counts things (e.g n rows i
```

What does our dataframe look like now?

```
head(batdat %>%
       select(c("swab_id", "site","species","date","gd","sa
```

```
## # A tibble: 6 x 6
## # Groups:   site, species, date [2]
##   swab_id     site  species date       gd sample.size
##   <chr>       <chr> <chr>   <chr>   <int>       <int>
## 1 KL15WI0002 HORS  MYSE    2/27/15     1           4
## 2 KL15WI0003 HORS  MYLU    2/27/15     1          20
## 3 KL15WI0004 HORS  MYLU    2/27/15     0          20
## 4 KL15WI0005 HORS  MYLU    2/27/15     1          20
## 5 KL15WI0006 HORS  MYLU    2/27/15     1          20
## 6 KL15WI0007 HORS  MYLU    2/27/15     1          20
```

```
#this is just showing a few columns for effect
```

Joining

- Joining datasets together is a useful skill, especially if we have two datasets we need to match on a specific column or set of columns
- https://dplyr.tidyverse.org/reference/mutate-joins.html
- Always call your new dataframe something new; don't write over and old dataframe in case you make a mistake joining

Join functions

- `inner_join()`: includes all rows in x and y.
- when inner joining, non-matching rows will be dropped!
- `left_join()`: includes all rows in x.
- when left joining, every row in x is kept, but only those matching x are kept in y
- `right_join()`: includes all rows in y.
- when right joining, every row in y is kept, but only those matching y are kept in x
- `full_join()`: includes all rows in x or y.
- every row is kept in both x and y

Joining actual datasets

- We frequently have data coming from different sources (lab, field, etc.)
- Let's join our infection data with our count data
- We might want to do this to ask 'How do bat infections at a site influence bat abundance at the same site?'

Join batdat and batcount

```
batdat_count = left_join(
  x = batdat,
  y = batcount,
  by = c("site","species","date")
  #columns to join on
)
head(batdat_count %>%
       select(c("swab_id","site","date","species","count"))
```

```
## # A tibble: 6 x 5
## # Groups:   site, species, date [2]
##   swab_id    site  date    species count
##   <chr>      <chr> <chr>   <chr>   <int>
## 1 KL15WI0002 HORS  2/27/15 MYSE        3
## 2 KL15WI0003 HORS  2/27/15 MYLU     1110
## 3 KL15WI0004 HORS  2/27/15 MYLU     1110
## 4 KL15WI0005 HORS  2/27/15 MYLU     1110
## 5 KL15WI0006 HORS  2/27/15 MYLU     1110
```

Pivoting

- https://tidyr.tidyverse.org/articles/pivot.html
- sometimes our datasets are not in the format we want for an analysis

Why pivot?

```
head(batdat_count)
```

```
## # A tibble: 6 x 13
## # Groups:   site, species, date [2]
##   swab_id       gd          gdL swab_type state site  dat
##   <chr>      <int>        <dbl> <chr>     <chr> <chr> <ch
## 1 KL15WI0002     1  0.0000756  BAT       WI    HORS  2/2
## 2 KL15WI0003     1  0.479      BAT       WI    HORS  2/2
## 3 KL15WI0004     0 NA          BAT       WI    HORS  2/2
## 4 KL15WI0005     1  0.00000551 BAT       WI    HORS  2/2
## 5 KL15WI0006     1  0.0000356  BAT       WI    HORS  2/2
## 6 KL15WI0007     1  0.0000316  BAT       WI    HORS  2/2
## # ... with 3 more variables: lgdL <dbl>, sample.size <in
```

```
# right now, species is in long format
# but we could imagine wanting to test the abundance of one
# and how that influences another
# for example, does the number of MYLU influence the numbe
# for that, we would need to make columns of each species
```

pivot_wider is how we take long data, and make it wide

```
batcounts.wide<- batcount %>%
  #this says - make a new df called batcounts.wide using ba
  pivot_wider(names_from = species,
              values_from = count
              )
##make columns for each of the values in the species colum
head(batcounts.wide)
```

```
## # A tibble: 6 x 6
##   site  date     EPFU  MYLU  MYSE  PESU
##   <chr> <chr>   <int> <int> <int> <int>
## 1 BEAR  3/4/15      9    97     0    55
## 2 BEAR  3/7/16       5   122    16    50
## 3 BEAR  11/9/15     NA   116     7    50
## 4 BEAR  3/10/17     NA    38    NA    22
## 5 HORS  3/3/16       4   188    NA    NA
## 6 HORS  11/7/15     NA   646     1    NA
```

adding 0s into a pivot

- when we perform this, it automatically fills missing with NAs but we know that missing actually mean 0

```
batcounts.wide = batcount %>%
  pivot_wider(
  names_from = species,
  values_from = count,
  values_fill = 0
)
head(batcounts.wide)
```

```
## # A tibble: 6 x 6
##   site  date     EPFU  MYLU  MYSE  PESU
##   <chr> <chr>   <int> <int> <int> <int>
## 1 BEAR  3/4/15      9    97     0    55
## 2 BEAR  3/7/16       5   122    16    50
## 3 BEAR  11/9/15      0   116     7    50
## 4 BEAR  3/10/17      0    38     0    22
## 5 HORS  3/3/16       4   188     0     0
```

Going from wide to long format

- more often, we might be working in the opposite direction
- A more common issue is that programs output information in columns that we would rather have in rows

```
batcounts.long = batcounts.wide %>%
  pivot_longer(
    cols = c("EPFU","MYLU","MYSE","PESU"),
    #what are the existing columns I want to make into rows
    names_to =   "species",
    #put the names of the columns in a column called 'spec
    values_to = "count"
    #the values that were in each of the columns get moved
  )
```

Check out new batcounts.long

```
head(batcounts.long)
```

```
## # A tibble: 6 x 4
##   site  date   species count
##   <chr> <chr>  <chr>   <int>
## 1 BEAR  3/4/15 EPFU        9
## 2 BEAR  3/4/15 MYLU       97
## 3 BEAR  3/4/15 MYSE        0
## 4 BEAR  3/4/15 PESU       55
## 5 BEAR  3/7/16 EPFU        5
## 6 BEAR  3/7/16 MYLU      122
```

Often, multiple ways to do the same thing

```
#option #2
batcounts.long = batcounts.wide %>%
  pivot_longer(
    cols = c(starts_with("MY"), "EPFU", "PESU"),
    #if all the columns start with the same thing (or many
    #we can use 'starts_with
    names_to =   "species",
    values_to = "count"
  )
```

Option 3 - pivoting longer

```
batcounts.long = batcounts.wide %>%
  pivot_longer(
    cols = -c(site, date),
    #this says use every column but site and date
    names_to =   "species",
    values_to = "count"
  )
```