Day 2, Lecture One: Data Manipulation with Tidyverse

Andie Creel

January, 2025

How I'd recommend following along for this lecture: have the pdf in front of you so you can follow along some of the points, but mostly write the code and comments in a script or R markdown.

Today, we're going to go through a lot of different concepts. I've tried to write this PDF so you can use it as a reference to refer back to repeatedly.

1 Goal

The goal of this lecture is:

1) Learn about packages, the tidyverse
2) Learn how to manipulate and clean data with tidyverse
3) Learn how to set up a r project

2 Why is data manipulation important

Backing up and remembering we're scientists

- We will have a hypothesis of how the world works
- We want to construct a model that approximates that
- We need data from real world to build that model that approximates the world
- The data we have may not be set up to be plugged into the model we'd like to run
- However, it could be *manipulated* so that we can use the data we have with the model we want

My personal example: American Time Use Survey and the value of outdoor leisure (travel cost model), Credit card data and the value of green space (hedonic model, I think)

3 What are packages and how can I get them?

What are they:

- A package contains a bunch of pre-built functions
- Anyone can load and use them
- Saves you a ton of time because someone already figured out how to do it

Tidyverse

- Collection of R packages
- All are meant for data science
- Have shared syntax
- Makes it easier to import, tidy, transform, visualize, and model data in R
- Shout out to Hadley Wickham and co

How can I install packages (user interface):

1. Go the the files/Plots/Packages quadrant
2. Click on Packages
3. Click Install
4. Search for packages you want ("dplyr", "tidyr", "ggplot2").
    - All of these are tidyverse packages
    - You can install "tidyverse" and have all of them
    - I think it's more valuable to learn one at a time so that you know what function goes with what package

```
# May want to run this as some point, but it takes a long time! so maybe not now
# install.packages("tidyverse")
```

Installing a package vs. loading a package

- You only need to install a package on your local computer once
- You then "load" that package in a script when you want to use it.

Installing a package with r code (one time only)

```
# installing packages using r code
install.packages("dplyr")
install.packages("tidyr")
install.packages("ggplot2")
```

Loading a package

```
library(dplyr)
library(tidyr)
library(ggplot2)
```

4 Manipulating and cleaning with dplyr

- dplyr is my most used package for data cleaning and manipulation
- Going to introduce the primary functions in the dplyr package
    - `mutate()`
    - `if_else()`
    - `filter()`
    - `select()`
    - `group_by()`
    - `summarise()`
    - `left_join()`
- There are a million ways to implement these functions
- Important for your solution strategy to know that these are the functions you can build around
- I would say 90% of my data cleaning is different combinations of these functions
- Why use dplyr instead of base R?
    - It's faster and more memory efficient (good for large data sets)
    - It's easier to read
- Let's go through some examples of what we did yesterday, but redo with dplyr code

5 Redo examples from yesterday with `dplyr`

5.1 `mutate()`

```
# ---------------------------------------------------------------------------
# Let's make a dataframe again
# -------------------------------------------------------------------------
# Build our trusty dataset, but let's call this one myBase
myBase <- data.frame(
  name = c("Andie", "Bridger", "Scott"),
  gender = c("Female", "non-binary", "Male"),
  male = c(FALSE, FALSE, TRUE),
  income_cat = c("middle", "poor", "rich"),
  park_dist = c(1, 0.5, 0.1)
)

myBase
```

```
##      name     gender  male income_cat park_dist
## 1   Andie     Female FALSE     middle       1.0
## 2 Bridger non-binary FALSE       poor       0.5
## 3   Scott       Male  TRUE       rich       0.1
```

```
myDplyr <- myBase

# check to make sure they're the same
identical(myBase, myDplyr)
```

```
## [1] TRUE
```

```
# -------------------------------------------------------------------------
# manipulating a column
# -------------------------------------------------------------------------

# Base R
for (i in seq_along(myBase$park_dist)) {
  myBase$park_dist[i] <- myBase$park_dist[i] + 1 # everyone move one mile
}
myBase
```

```
##      name     gender  male income_cat park_dist
## 1   Andie     Female FALSE     middle       2.0
## 2 Bridger non-binary FALSE       poor       1.5
## 3   Scott       Male  TRUE       rich       1.1
```

```
# dplyr
# add mile to park dist using dplyr
myDplyr <- myDplyr %>%
  mutate(park_dist = park_dist + 1)

# Check to make sure they're identical
(myDplyr == myBase)
```

```
##      name gender male income_cat park_dist
## [1,] TRUE   TRUE TRUE       TRUE      TRUE
## [2,] TRUE   TRUE TRUE       TRUE      TRUE
## [3,] TRUE   TRUE TRUE       TRUE      TRUE
```

New things introduced

- Pipes %>%: pipes take input (our dataframe) and passes it onto the next function. You can chain pipes together.
- `mutate()` mutate is a function from the dplyr package

- A data frame is piped to the function `mutate()` and then the function executes some small function you gave it (park_dist + 1) and returns the new value

```
# ------------------------------------------------------------------------
# Making a new column
# ------------------------------------------------------------------------

# base R
for (i in 1:length(myBase$park_dist)) {
  myBase$park_dist_new[i] <- myBase$park_dist[i] + 1 # everyone moved one mile
}

# dplyr
myDplyr <- myDplyr %>%
  mutate(park_dist_new = park_dist + 1)

myDplyr
```

```
##      name     gender  male income_cat park_dist park_dist_new
## 1   Andie     Female FALSE     middle       2.0           3.0
## 2 Bridger non-binary FALSE       poor       1.5           2.5
## 3   Scott       Male  TRUE       rich       1.1           2.1
```

```
# check if identical
(myBase == myDplyr)
```

```
##      name gender male income_cat park_dist park_dist_new
## [1,] TRUE   TRUE TRUE       TRUE      TRUE          TRUE
## [2,] TRUE   TRUE TRUE       TRUE      TRUE          TRUE
## [3,] TRUE   TRUE TRUE       TRUE      TRUE          TRUE
```

New things introduced

- `mutate()` can also construct a new a new column

5.2 if_else()

Last lecture, we said that women and non-binary people had their distances from parks recorded wrong. Non-male people were a quarter mile closer to the park than originally recorded.

```
# ------------------------------------------------------------------------
# goes through each row and changes age if someone is male
# ------------------------------------------------------------------------

# base r
for (i in seq_along(myBase$male)) {
  if (myBase$male[i] == FALSE) { # check if someone is "not male"
    myBase$park_dist_correct[i] <- myBase$park_dist[i] - 0.25 # adjust
  }else{
    myBase$park_dist_correct[i] <- myBase$park_dist[i]
  }
}
myBase
```

```
##      name     gender  male income_cat park_dist park_dist_new park_dist_correct
## 1   Andie     Female FALSE     middle       2.0           3.0              1.75
## 2 Bridger non-binary FALSE       poor       1.5           2.5              1.25
## 3   Scott       Male  TRUE       rich       1.1           2.1              1.10
```

```r
# dplyr: if_else
myDplyr <- myDplyr %>%
  mutate(park_dist_correct = if_else(male == FALSE, park_dist - 0.25, park_dist))
myDplyr
```

```
##      name     gender  male income_cat park_dist park_dist_new park_dist_correct
## 1   Andie     Female FALSE     middle       2.0           3.0              1.75
## 2 Bridger non-binary FALSE       poor       1.5           2.5              1.25
## 3   Scott       Male  TRUE       rich       1.1           2.1              1.10
```

```r
# Check if identical
(myBase == myDplyr)
```

```
##      name gender male income_cat park_dist park_dist_new park_dist_correct
## [1,] TRUE   TRUE TRUE       TRUE      TRUE          TRUE              TRUE
## [2,] TRUE   TRUE TRUE       TRUE      TRUE          TRUE              TRUE
## [3,] TRUE   TRUE TRUE       TRUE      TRUE          TRUE              TRUE
```

New things introduced

- `if_else()` combined with `mutate()`
  - the first part is what you're evaluating to check if it's true (`male == FALSE`)
  - Next entry is what to return if true (`park_dist - 0.25`)
  - Final part is what to return if false (`park_dist`)

6 New functions in the dplyr package

6.1 filter()

You have a data set, and want subset to only some observations conditioned on a characteristic.

Ex. 1: Let's say we have observations of different ecosystems and want to filter to low pollution levels

```r
# -------------------------------------------------------------------------
# make an ecosystem dataset
# -------------------------------------------------------------------------
# ecosystem dataset
df_env_data <- data.frame(
  ecosystem = c("Forest", "Desert", "Wetland", "Grassland","Urban"),
  species_richness = c(120, 45, 80, 60, 30),
  pollution_level = c("Low", "High", "Medium", "Low", "High")
)

# display
df_env_data
```

```
##   ecosystem species_richness pollution_level
## 1    Forest              120             Low
## 2    Desert               45            High
## 3   Wetland               80          Medium
## 4 Grassland               60             Low
## 5     Urban               30            High
```

```r
# -------------------------------------------------------------------------
# filter
# -------------------------------------------------------------------------
# filter to include only locations with low pollution levels
low_pollution_data <- df_env_data %>%
  filter(pollution_level == "Low")

# display the filtered dataset
low_pollution_data
```

```
##   ecosystem species_richness pollution_level
## 1    Forest              120             Low
## 2 Grassland               60             Low
```

Ex. 2: drop NAs

You will frequently have datasets that have missing data (i.e., the cell has a NA value). Many functions and models won't work with NAs, so they have to be cleaned out of the data set. Let's imagine we're missing a location for one of our ecosystem observations

```
# -------------------------------------------------------------------------
# filter NAs
# -------------------------------------------------------------------------

# example df
df_env_data_na <- data.frame(
  ecosystem = c("Forest", "Desert", "Wetland", NA,"Urban"),
  species_richness = c(120, 45, 80, 60, 30),
  pollution_level = c("Low", "High", "Medium", "Low", "High")
)

df_env_data_na
```

```
##   ecosystem species_richness pollution_level
## 1    Forest              120             Low
## 2    Desert               45            High
## 3   Wetland               80          Medium
## 4      <NA>               60             Low
## 5     Urban               30            High
```

```
# drop rows with NAs in the 'location' column using filter
df_env_data_clean <- df_env_data_na %>%
  filter(!is.na(ecosystem))

df_env_data_clean
```

```
##   ecosystem species_richness pollution_level
## 1    Forest              120             Low
## 2    Desert               45            High
## 3   Wetland               80          Medium
## 4     Urban               30            High
```

6.2 select()

You have a data set with more columns than you need. Sometimes you want to only work with some of the variables, and it is space efficient to get rid of the rest.

For example, let's say you're only interested in the polution at diffent locations, not the species richness.

```
# -------------------------------------------------------------------------
# select
# -------------------------------------------------------------------------
pollution_data <- df_env_data %>%
  select(ecosystem, pollution_level)

pollution_data
```

```
##   ecosystem pollution_level
## 1    Forest             Low
## 2    Desert            High
## 3   Wetland          Medium
## 4 Grassland             Low
## 5     Urban            High
```

6.3 group_by()

Group by is useful for when you want to aggregate information up to a higher level. So, if you need a variable that is the average species richness by ecosystem rather than the species richness at individual sites.

Let's work with a longer version of our ecosystem dataset

```
# -------------------------------------------------------------------------
# create longer dataset
# -------------------------------------------------------------------------

df_env_long <- data.frame(
    ecosystem = c("Forest", "Desert", "Wetland", "Grassland", "Urban", "Forest", "Desert", "Wetland", "
    species_richness = c(120, 45, 80, 60, 30, 110, 50, 85, 65, 35),
    pollution_level = c("Low", "High", "Medium", "Low", "High", "Low", "High", "Medium", "Low", "High")
)

# display the dataset
df_env_long
```

```
##    ecosystem species_richness pollution_level
## 1     Forest              120             Low
## 2     Desert               45            High
## 3    Wetland               80          Medium
## 4  Grassland               60             Low
## 5      Urban               30            High
## 6     Forest              110             Low
## 7     Desert               50            High
## 8    Wetland               85          Medium
## 9  Grassland               65             Low
## 10     Urban               35            High
```

```
# group by ecosystem and calculate the mean species richness
df_env_grouped <- df_env_long %>%
    group_by(ecosystem) %>%
    mutate(mean_species_richness = mean(species_richness))

# display the updated dataset with mean species richness
df_env_grouped
```

```
## # A tibble: 10 x 4
## # Groups:   ecosystem [5]
##    ecosystem species_richness pollution_level mean_species_richness
##    <chr>                <dbl> <chr>                           <dbl>
##  1 Forest                 120 Low                             115  
##  2 Desert                  45 High                             47.5
##  3 Wetland                 80 Medium                           82.5
##  4 Grassland               60 Low                              62.5
##  5 Urban                   30 High                             32.5
##  6 Forest                 110 Low                             115  
##  7 Desert                  50 High                             47.5
##  8 Wetland                 85 Medium                           82.5
##  9 Grassland               65 Low                              62.5
## 10 Urban                   35 High                             32.5
```

6.4 summarise()

Finally, the summarise() function can help you create summary tables. These are useful for getting a quick snapshot of data. Using them is particularly important when you have a large enough datset that you're unable to look at the dataset and glean insight (which, for me, happens if the dataset is longer than 5 observations).

```r
# -------------------------------------------------------------------------
# summarise
# -------------------------------------------------------------------------
# group by ecosystem and calculate the mean and total species richness
summary_table <- df_env_long %>%
    group_by(ecosystem) %>%
    summarise(
        mean_species_richness = mean(species_richness),
        total_species_richness = sum(species_richness)
    )

# display the summary table
summary_table
```

```
## # A tibble: 5 x 3
##   ecosystem mean_species_richness total_species_richness
##   <chr>                     <dbl>                  <dbl>
## 1 Desert                     47.5                     95
## 2 Forest                    115                      230
## 3 Grassland                  62.5                    125
## 4 Urban                      32.5                     65
## 5 Wetland                    82.5                    165
```

Quick note on the `n()` function: it looks like is didn't get an input, but remember we used all the pipes. The `n()` function is designed to be used with the `summarise()` function and return how many observations are in a group.

Disclaimer: There is also a `summarize()` function that should work the same, but sometimes may not. The recommendation is to use the `summarise()`!

6.5 join() functions

You may need data from multiple data sets to be in one data set. We do this with `join()` functions.

The most common is `left_join()`, but you may also see `right_join()`, `inner_join()`, and `full_join()`.

A `left_join()` adds variables from a second dataset to your original dataset. This is what I almost exclusively use.

```
# -------------------------------------------------------------------------
# recall our env dataset
# -------------------------------------------------------------------------
df_env_data
```

```
##   ecosystem species_richness pollution_level
## 1    Forest              120             Low
## 2    Desert               45            High
## 3   Wetland               80          Medium
## 4 Grassland               60             Low
## 5     Urban               30            High
```

```
# -------------------------------------------------------------------------
# create second dataset
# -------------------------------------------------------------------------
df_pollution <- data.frame(
  pollution_level = c("Low", "High", "Medium"),
  water_quality = c("drink and swim", "no swim", "swim only"),
  super_fund_site = c(F, T, F)
)

# -------------------------------------------------------------------------
# left_join
# -------------------------------------------------------------------------
df_join <- left_join(x = df_env_data, y = df_pollution, by = "pollution_level")
df_join
```

```
##   ecosystem species_richness pollution_level  water_quality super_fund_site
## 1    Forest              120             Low drink and swim           FALSE
## 2    Desert               45            High        no swim            TRUE
## 3   Wetland               80          Medium      swim only           FALSE
## 4 Grassland               60             Low drink and swim           FALSE
## 5     Urban               30            High        no swim            TRUE
```

7 Manipulating and cleaning with tidyr

Just like `dplyr`, `tidyr` has loads of very useful functions. There are two that I think are the most important to be aware of

- `pivot_longer()`
- `pivot_wider()`

7.1 `pivot_longer()`

Sometimes you'll want to pivot your data longer. Most models I work will require data to be formatted so that they are "long" by variables because that's usually the format best for regression models.

Ex. You want to have month be a variable in a regression (so you control for the month), but instead there are 12 columns for the month. You can pivot those columns into one column called `month`. You'd say you're data is "long by month" after you've done this formatting.

```
# -------------------------------------------------------------------------
# Create data set
# -------------------------------------------------------------------------
```

```
myTemps <- data.frame(
  location = c("Forest", "Desert"),
  Jan = c(5, 20),
  Feb = c(6, 22),
  Mar = c(10, 25)
)
myTemps
```

```
##   location Jan Feb Mar
## 1   Forest   5   6  10
## 2   Desert  20  22  25
```

```
# -------------------------------------------------------------------------
# pivot longer
# -------------------------------------------------------------------------
myTemps_long <- myTemps %>%
  pivot_longer(
  cols = Jan:Mar, # the columns we want to pivot
  names_to = "Month", # the new variable where the names of columns will be assigned
  values_to = "Temperature" # the new variable where the values will be assigned
)
myTemps_long
```

```
## # A tibble: 6 x 3
##   location Month Temperature
##   <chr>    <chr>       <dbl>
## 1 Forest   Jan             5
## 2 Forest   Feb             6
## 3 Forest   Mar            10
## 4 Desert   Jan            20
## 5 Desert   Feb            22
## 6 Desert   Mar            25
```

7.2 pivot_wider()

I usually pivot data frames from a long format to a wide format when the model I want to aggregate the data i.e., I want the data to be *less* granular. Usually you want data to me more grandular, but there are cases where one data set is less granular than another, and so you need them to match in order to merge them together.

Ex. You have one data set that is long by day. You want to combine it with another data set that isn't long by day and instead is only long by station and week. Therefore you want to get the total weekly rainfall, instead of having it as daily rainfall.

```
# -------------------------------------------------------------------------
# Create long data
# -------------------------------------------------------------------------
myRain <- data.frame(
  station = c("StationA", "StationA", "StationA",
              "StationB", "StationB", "StationB"),
  day = c("Monday", "Tuesday", "Wednesday",
          "Monday", "Tuesday", "Wednesday"),
  rainfall_mm = c(5, 10, 3,
                  0, 0, 12)
)
```

```
myRain
```

```
##    station       day rainfall_mm
## 1 StationA    Monday           5
## 2 StationA   Tuesday          10
## 3 StationA Wednesday           3
## 4 StationB    Monday           0
## 5 StationB   Tuesday           0
## 6 StationB Wednesday          12
```

```
# -------------------------------------------------------------------------
# pivot so it's wide by station
# -------------------------------------------------------------------------

myRain_wide <- myRain %>%
  pivot_wider(names_from = day, values_from = rainfall_mm)

myRain_wide
```

```
## # A tibble: 2 x 4
##   station  Monday Tuesday Wednesday
##   <chr>     <dbl>   <dbl>     <dbl>
## 1 StationA      5      10         3
## 2 StationB      0       0        12
```

```
# -------------------------------------------------------------------------
# get weekly total rainfall
# -------------------------------------------------------------------------

myRain_weekly <- myRain_wide %>%
  mutate(weekly_rain = Monday + Tuesday + Wednesday) %>%
  select(-Monday, -Tuesday, -Wednesday)

myRain_weekly
```

```
## # A tibble: 2 x 2
##   station  weekly_rain
##   <chr>          <dbl>
## 1 StationA          18
## 2 StationB          12
```

New things introduced

- we used a “-” sign with the `select()` command to drop columns rather than selecting them