# Day 2, Lecture One: Data Manipulation with Tidyverse

Andie Creel

January, 2023

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 Travel Cost Model, Cell Photo Data and green space

## 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 pacakge**

- 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 datasets)
  - It's easier to read
- Let's go through some examples of what we did yesterday, but redo with dplyr code

## 4.1 Examples from yesterday

```
# ------------------------------------------------------------------------
# Let's make a dataframe again
# ------------------------------------------------------------------------
myBase <-   data.frame(
   gender = c("Male", "non-binary","Female"),
   male = c(T, F, F),
   height = c(152, 171.5, 165),
   weight = c(81, 93, 78),
   age = c(42,38,26)
)

myDplyr <- myBase

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

# Base R
for (i in 1:length(myBase$age)) {
  myBase$age[i] <- myBase$age[i] + 1 # everyone aged one year
}

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

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

```
##      gender male height weight  age
## [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 (age +1) and returns the new value

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

# base R
for (i in 1:length(myBase$age)) {
  myBase$age_new[i] <- myBase$age[i] + 1 # everyone aged one year
}

# dplyr
myDplyr <- myDplyr %>%
  mutate(age_new = age + 1)

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

```
##      gender male height weight  age age_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

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

# base r
for (i in 1:length(myBase$male)) {
   if (myBase$male[i] == TRUE) {
    myBase$age_new_m[i] <- myBase$age[i] - 3
  }else{
    myBase$age_new_m[i] <- myBase$age[i]
  }
}

# dplyr
myDplyr <- myDplyr %>%
  mutate(age_new_m = if_else(male == TRUE, age - 3, age))

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

```
##      gender male height weight  age age_new age_new_m
## [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 == TRUE`)
  - Next entry is what to return if true (`age - 3`)
  - Final part is what to return if false (`age`)

## 4.2 New functions in the dplyr package: `filter()`

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

Ex 1: You just want people in the marketing department

```
# ------------------------------------------------------------------------
# make a data frame of employees
# ------------------------------------------------------------------------
myEmps <- data.frame(
  id = 1:5,
  name = c("Alice", "Bob", "Charlie", "David", "Eva"),
  department = c("IT", "Market", "HR", "Market", "IT"),
  salary = c(45000, 55000, 40000, 60000, 50000)
)

myEmps
```

```
##   id    name department salary
## 1  1   Alice         IT  45000
## 2  2     Bob     Market  55000
## 3  3 Charlie         HR  40000
## 4  4   David     Market  60000
## 5  5     Eva         IT  50000
```

```
# ------------------------------------------------------------------------
# filter
# ------------------------------------------------------------------------
myEmp_m <- myEmps %>%
  filter(department == "Market")

myEmp_m
```

```
##   id  name department salary
## 1  2   Bob     Market  55000
## 2  4 David     Market  60000
```

Ex 2: you have missing variables and need to drop those observations (a lot of models will require you to do this)

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

# example df
myNAs <- data.frame(
```

```
  id = 1:7,
  name = c("Andie", "Bridger", "Nancy", "Scott", "Alex", "Tash", "Kenz"),
  age = c(25, 30, NA, 28, 35, NA, "40")
)

# filter NAs
myNAs_filter <- myNAs %>%
  filter(!is.na(age))

myNAs_filter # id 3 and 6 are dropped now
```

```
##   id    name age
## 1  1   Andie  25
## 2  2 Bridger  30
## 3  4   Scott  28
## 4  5    Alex  35
## 5  7    Kenz  40
```

### 4.3 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.

Ex: You want to "anonymize" the data by dropping people's name

```
# ------------------------------------------------------------------------
# select
# ------------------------------------------------------------------------
myEmp_thin <- myEmps %>%
  select(id, department, salary)

myEmp_thin
```

```
##   id department salary
## 1  1         IT  45000
## 2  2     Market  55000
## 3  3         HR  40000
## 4  4     Market  60000
## 5  5         IT  50000
```

### 4.4 group_by()

You need to group by a variable, and then run a function on those groups. For example, you want to know the total amount of salary for each department.

```
# ------------------------------------------------------------------------
# group_by
# ------------------------------------------------------------------------
myEmps_grouped <- myEmps %>%
  group_by(department) %>%
  mutate(dept_total_salary = sum(salary)) %>%
```

```
  ungroup(department)

myEmps_grouped
```

```
## # A tibble: 5 x 5
##      id name    department salary dept_total_salary
##   <int> <chr>   <chr>       <dbl>              <dbl>
## 1     1 Alice   IT          45000              95000
## 2     2 Bob     Market      55000             115000
## 3     3 Charlie HR          40000              40000
## 4     4 David   Market      60000             115000
## 5     5 Eva     IT          50000              95000
```

### 4.5 summarise()

When used with group by, can give you fast and simple summaries of the data table.

Ex. you're interested in some stats about the department

```
# ------------------------------------------------------------------------
# summarise
# ------------------------------------------------------------------------
myEmp_summary <- myEmps %>%
  group_by(department) %>%
  summarise(avg_sal = mean(salary), # avg salary by department
            num_employees = n()) # number of employess

myEmp_summary
```

```
## # A tibble: 3 x 3
##   department avg_sal num_employees
##   <chr>        <dbl>         <int>
## 1 HR           40000             1
## 2 IT           47500             2
## 3 Market       57500             2
```

**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. Eliana highlighted this, and the recommendation is to use the `summarise()`!

### 4.6 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.

```r
# ------------------------------------------------------------------------
# create second dataset
# ------------------------------------------------------------------------
myDepts <- data.frame(
  department = c("IT", "Market", "HR"),
  location = c("Building A", "Building B", "Building C"),
  boss = c("John Doe", "Jane Smith", "Mike Brown")
)

# ------------------------------------------------------------------------
# left_join
# ------------------------------------------------------------------------
myEmps_join<- left_join(x = myEmps, y = myDepts, by = "department")
myEmps_join
```

```
##   id    name department salary   location       boss
## 1  1   Alice         IT  45000 Building A   John Doe
## 2  2     Bob     Market  55000 Building B Jane Smith
## 3  3 Charlie         HR  40000 Building C Mike Brown
## 4  4   David     Market  60000 Building B Jane Smith
## 5  5     Eva         IT  50000 Building A   John Doe
```

# 5 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()`

## 5.1 `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.

```r
# ------------------------------------------------------------------------
# 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
```

## 5.2 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)
)

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

myRain_wide <- myRain %>%
```

```
  pivot_wider(names_from = day, values_from = rainfall_mm)

# ------------------------------------------------------------------------
# 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