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
- Let's go through some examples of what we did yesterday, but redo with dplyr code
- Why use dplyr instead of base R?
    - It's faster and more memory efficient (good for large datasets)
    - It's easier to read

4.1 Examples from yesterday

```r
# --------------------------------------------------------------------------
# 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 (from above: version one of referencing a cell)
# --------------------------------------------------------------------------

# 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
```

```r
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

Now, let's introduce some more functions that are helpful for data cleaning that we haven't seen yet.

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

myEmps
```

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

4.2.1 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

```r
# -----------------------------------------------------------------------
# filter
# -----------------------------------------------------------------------
myEmp_marketing <- myEmps %>%
  filter(department == "Marketing")

myEmp_marketing
```

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

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

```r
# -----------------------------------------------------------------------
# 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")
)

myNAs
```

```
##   id     name  age
## 1  1    Andie   25
## 2  2 Bridger   30
## 3  3    Nancy <NA>
## 4  4    Scott   28
## 5  5     Alex   35
## 6  6     Tash <NA>
## 7  7     Kenz   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
```

4.2.2 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.

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  Marketing  55000
## 3  3         HR  40000
## 4  4  Marketing  60000
## 5  5         IT  50000
```

4.2.3 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
# ------------------------------------------------------------------------
```

```r
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     Marketing   55000            115000
## 3     3 Charlie HR          40000             40000
## 4     4 David   Marketing   60000            115000
## 5     5 Eva     IT          50000             95000
```

4.2.4 summarize()

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

```r
# ---------------------------------------------------------------------------
# summarize
# ---------------------------------------------------------------------------
myEmp_summary <- myEmps %>%
  group_by(department) %>%
  summarize(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 Marketing    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 `summarize()` function and return how many observations are in a group.

4.2.5 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 first. This is what I almost exclusively use.

```r
# --------------------------------------------------------------------------
# create second dataset
# --------------------------------------------------------------------------
myDepts <- data.frame(
  department = c("IT", "Marketing", "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  Marketing  55000 Building B Jane Smith
## 3  3 Charlie         HR  40000 Building C Mike Brown
## 4  4   David  Marketing  60000 Building B Jane Smith
## 5  5     Eva         IT  50000 Building A   John Doe
```

5 Manipulating and cleaning with tidyr

pivot long and pivot wide