Session 4: Tidyverse

Jennifer Lin

2022-04-19

This section on `tidyverse`, and specifically, the `dplyr` package is perhaps one of the most important for developing data wrangling skills. This is necessary when you work with any new data that need to be processed before you can do anything, like generating summary statistics or making plots. Our goals for today are:

1. Learn the basics of cleaning data using `dplyr`
2. Apply data cleaning skills to novel data
3. Generate summary statistics tables using `dplyr`

In the past, you might have manually gone into excel files to clean and recode your data. After this workshop, I hope that you will be able to clean data such that others can take your code and replicate what you did with ease. Also, this will make your workflow more systematic, so you will not risk errors in your data wrangling process. Hopefully, this process will prove to take less time and cause less stress as you continue your data exploration journey.

For this exercise, our data come from the Cooperative Elections Studies (CES) 2020 Study, which surveys respondents from every Congressional District every two years on various issues related to politics. The data for this week are a sample of variables from the dataset but are not cleaned[1].

[1] I should note that I debated whether to use a simplified dataset or have this process reflect downloading data from source – in the interest of time, I decided to use a subset of data but this subset reflects the original coding of the larger data file.

```
CES20 <- read.csv("CES2020_subset.csv")
```

When you are seeing data for the first time, it is always important to know how to go through the corresponding codebook, which includes the variable names and how they are coded. For this workshop, I will give you the codes. However, for your own work, keep the following in mind:

1. Be familiar with the codes and structure of the original data
2. Pay attention to how the original data codes missing values
3. Know the kind of data structure you need for your analyses (Categorical? Numeric?)

With that, let's talk about the `tidyverse` universe.

`Tidyverse`: An Overview

The `tidyverse`[2], as the website defines it is:

[2] `https://www.tidyverse.org/`

> [An] opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures.

You would often load the universe using, though some people, like me, do not like to load the entire universe outright. Sure, it is more convenient if you know you will be using multiple packages from this universe, but the hesitance comes from getting a genuine understanding of what it is you are doing and where your functions originate. Some people also hesitate because there are function conflicts with base R, but this issue is often more minimal compared to the understanding that many value.

```
library(tidyverse)
```

Since the `tidyverse` contains many packages, loading the entire universe makes it hard for one to truly know what function comes from which package. This knowledge may become essential in collaborations as some people need toe `pachage::` in front of any function in order for your code to run on their machine[3]. Under these circumstances, it is not enough to go `tidyverse::function()` as that will not work. Instead, you will need to specify the package, which there are many. Here are just a few and a short description of what each does.

[3] This issue has come up for me multiple times in my own research with collaborators.

- `ggplot2`: Graphs
- `dplyr`: Data manipulation
- `stringr`: String and other Text as Data tools
- `readr`: Read data files
- `forcats`: Handle categorical variables
- `purrr`: Enhance base R tools like for loops
- `tidyr`: Generate tidy data, where each observation is a row and each variable is a column
- `tibble`: Data frame enhancements

Since functions in the `tidyverse` generally follow the same syntax, you can engage with multiple packages in one code chunk so long as you load the packages. A tool that links many of the functions in the `tidyverse` together is the pipe (`%>%`)[4]. Pipes essentially tell the code to move from one step to another without creating a new object.

[4] You can type a pipe using COMMAND+SHIFT+M on a Mac

```
data %>%
  step 1 %>%
  step 2 %>%
  step 3 %>%
  steo 4
```

In each of the steps above, they can be functions from a different, or the same package. Additionally, recall that the way to join elements of a `ggplot` is the `+` sign. The pipe is the `+` equivalent for under `tidyverse` commands.

While there are many great and interesting things that `tidyverse` can do, we will only focus on the `dplyr` package in this section.

`dplyr`: Data Management and Wrangling

Beyond base R, `dplyr` is perhaps the most commonly used package for data manipulation. And this is for a good reason. The functions in this package are easy to use and have rather intuitive names. As you might notice, this is a pattern among packages and functions in `tidyverse`.

From `dplyr` package documentation, here is a brief overview of what this package is and why I find it to be a very useful package:

> `dplyr` is a grammar of data manipulation, providing a consistent set of verbs that help you solve the most common data manipulation challenges:
>
> - `mutate()` adds new variables that are functions of existing variables
> - `select()` picks variables based on their names.
> - `filter()` picks cases based on their values.
> - `summarise()` reduces multiple values down to a single summary.
> - `arrange()` changes the ordering of the rows.

Today, we will cover these functions (and then some)

`select()`: Picking Variables

The `select()` function allows us to pick variables from a given data frame. The function works in one of the following forms:

```
select(.data, ...)
```

```
data %>%
  select(...)
```

Within the parentheses, simply put the names of the variables that you want. Here, you do not need the `c()` to make variable selections.

If we have many variables that have the same name, we can use one of the following within `select()` and use a regular expression to get the variables that we want.

Operator	Description
`everything()`	everything in data
`last_col()`	last column
`starts_with()`	start with a string
`ends_with()`	ends with a string
`contains()`	contains a string

Here is an example on how the `select()` function. Let's look at the existing variable names in the CES dataset

```
names(CES20)
```

```
##  [1] "X"          "starttime"  "endtime"    "caseid"     "region"
##  [6] "countyfips" "countyname" "inputstate" "birthyr"    "gender"
## [11] "votereg"    "urbancity"  "pid3"       "ideo5"      "ownhome"
## [16] "milstat_1"  "milstat_2"  "milstat_3"  "milstat_4"  "milstat_5"
## [21] "vote20"
```

Now, I will move to select a few variables from this data set.

```
CES20_new <- CES20 %>%
  select(
    birthyr,
    gender,
    votereg,
    urbancity,
    pid3,
    ideo5,
    vote20
  )
```

Let's look at the names of the new data set. How does this compare to the original?

```
names(CES20_new)
```

```
## [1] "birthyr"   "gender"    "votereg"   "urbancity" "pid3"      "ideo5"
## [7] "vote20"
```

`filter()`: Picking Cases

The `select()` function above is for picking variables. If you want to pick cases, use `filter()`. This function, for the bakers among us, is like using a sifter to sift sugar or flour in baking. Alternatively, it is like using one of those sand separators to break the components of sand[5]. The goal is to keep the observations that you want and discard the ones that you do not. The syntax for filtering goes as follows:

[5] This is my reference back to High School marine biology, if you have ever taken such course.

```
filter(.data, ...)
```

```
data %>%
  filter(...)
```

When you are filtering observations, you can include one condition or multiple, joined by "and" (&) or "or" (|). Of course, you can also just use multiple pipes, which that works more like the "or" condition. Within the conditions, the format works such that

```
data %>%
  filter(var operator condition)
```

Here,

- `var` is the variable name
- `operator` is one of the operators in the table below
- `condition` is the filtering condition

Operator	Description
==	exactly equal to
<	less than
<=	less than or equal to
`%in%`	is in this list
!=	not equal to
`is.na()`	NA values

Here is an example to keep people who have indicated a party ID response in the CES survey[6].

[6] Notice that the `!` in R means "is not".

```
CES20_new <- CES20_new %>%
  filter(!is.na(pid3))
```

`mutate()`: Creating Variables

Oftentimes, in data wrangling, you need to make your own variables. `mutate()` helps with this. This function allows you to create a new variable by applying a function to an old variable. These functions can denote functions for recoding like `case_when()`, `if_else()` or `recode()`. It can also be math or functions for summary statistics. It can also be a function you wrote yourself!

```
data <- data %>%
  mutate(new = function(old))
```

Here is a general example for how to do simple recoding with a variable using `case_when()`, which is simply a more fancy function for the if else statements in base R or Python programming.

```
data <- data %>%
  mutate(new = case_when(
    old == [old] ~ [new code],
    # For example:
    voterreg == "yes" ~ 1,
    # For all else not
    # previously defined
    TRUE ~ [new code]
  ))
```

Using the CES data, I am going to make some new variables out of existing ones using the `mutate()` function and `case_when()`[7]. In this first chunk, I am creating a three category ideology variable, and a recoded residence variable, and calling both factors, using different mechanisms.

[7] As you can tell, `case_when()` is one of my favorite recoding variables. `recode()` is useful too and does similar things. See the function documentation for more details.

```
CES20_new <- CES20_new %>%
  mutate(
    ideo3  = case_when(
      ideo5 %in% c(1:2) ~ "Liberal",
      ideo5  ==  3      ~ "Moderate",
      ideo5 %in% c(4:5) ~ "Conservative"),
    ideo3  = as.factor(ideo3),
    residence      = case_when(
      urbancity == 1 ~ "City",
      urbancity == 2 ~ "Suburb",
      urbancity == 3 ~ "Town",
      urbancity == 4 ~ "Rural"),
    residence = factor(
      residence,
      levels = c("City", "Suburb", "Town", "Rural"))
  )
```

In this next chunk, I create a variable to reflect age by doing some simple math, subtracting `birthyear` from the survey year. In addition, I generate age brackets and create some variables to reflect vote choice. Notice that using `TRUE ~ something` means that if there is a value for the old variable, it will be assigned to whatever the `something` is, regardless of what the original value is.

```
CES20_new <- CES20_new %>%
  mutate(
```

```
    age      = 2020 - birthyr,
    AGE      = case_when(
      age %in% c(18:24) ~ "18 - 24",
      age %in% c(25:44) ~ "25 - 44",
      age %in% c(45:64) ~ "45 - 64",
      age %in% c(65:80) ~ "65+"),
    TRUMP    = case_when(
      vote20 == 1 ~ TRUE,
      TRUE ~ FALSE),
    pres_vote = case_when(
      vote20 == 1 ~ "Donald J. Trump",
      vote20 == 2 ~ "Joseph R. Biden"),
  )
```

Collectively, notice that for single "is exactly equal to" cases, you use the `==` double equal sign. However, when you want to match something to a list of items, you need to use the `%in%` operator. Also, notice how logical variables work under `case_when()`. You should simply type out `TRUE` or `FALSE` in all caps, and ideally spell out the entire word.

`summarise()`: Get to the Point

We now get the summary statistics. In base R, you can use `summary()` and you will get a bunch of pre-determined summary statistics for your variable. However, what if you want to dictate what summary statistics to generate? `summarize()` in dplyr[8] helps us do that.

[8] The British `summarise()` also works.

Before going into `summarize()`, I should introduce `group_by()`, which allows you to generate summary statistics by a categorical variable. This is useful if you want to calculate something by a respondent's partisanship or vote status. Often, we use `group_by()` before `summarize()` to note which grouping mechanism we want to use. In the newer `dplyr` versions, we need to notate, within `summarize()` whether to keep or drop the groups using `.groups = 'keep'`. Here is the general syntax:

```
data %>%
  group_by(grouping_variable) %>%
  summarise(
    newvar_name1 = function(variable),
    newvar_name2 = function(variable),
    .groups = 'keep)
```

Below, I have two examples. First, I want to calculate the number of people in the survey who self-report living in any given place of

residence and then I want to calculate average age for people by place of residence. I can do that as follows

```
n_by_residence <- CES20_new %>%
  group_by(residence) %>%
  summarise(
    n = n(),
    avg_age = mean(age, na.rm = TRUE),
    .groups = 'keep'
  ) %>%
  filter(!is.na(residence))

n_by_residence
```

```
## # A tibble: 4 x 3
## # Groups:   residence [4]
##   residence     n avg_age
##   <fct>     <int>   <dbl>
## 1 City      16668    44.9
## 2 Suburb    23482    49.3
## 3 Town       8809    49.0
## 4 Rural     11693    51.1
```

Within the `group_by()` function, you can include multiple groups to help you get summary statistics for a combination of group identities. Here, I am demonstrating this by calculating the number of respondents per ideology and presidential vote choice along with their average age. I take out people who answered NA to both.

```
vote_by_ideo <- CES20_new %>%
  group_by(ideo3, pres_vote) %>%
  summarise(
    n = n(),
    avg_age = mean(age, na.rm = TRUE),
    .groups = 'keep') %>%
  filter(!is.na(ideo3) & !is.na(pres_vote))

vote_by_ideo
```

```
## # A tibble: 6 x 4
## # Groups:   ideo3, pres_vote [6]
##   ideo3        pres_vote           n avg_age
##   <fct>        <chr>           <int>   <dbl>
## 1 Conservative Donald J. Trump  1772    59.3
## 2 Conservative Joseph R. Biden   364    52.9
## 3 Liberal      Donald J. Trump    66    44.1
```

```
## 4 Liberal      Joseph R. Biden  5430    48.2
## 5 Moderate     Donald J. Trump   603    51.8
## 6 Moderate     Joseph R. Biden  2923    54.3
```

arrange(): Put in Order

We now come to the last function for today, but this is by far, not the last function that you will ever use in the `dplyr` package. There are plenty more to explore but the selection I present here reflects the most common functions.

`arrange()` helps us put things in order. This might be particularly useful when organizing your data set or summary statistics tables. To put the observations in order by characteristics on a given variable, use one of the two below, which represents the general syntax for `arrange()`.

```
arrange(
  data, variable,
  grouping_variable)

data %>%
  arrange(
    variable,
    grouping_variable)
```

If you want to arrange observations in descending order, simply use `desc()` with the variable name in the arrange syntax.

```
arrange(
  data,
  desc(variable),
  grouping_variable)

data %>%
  arrange(
    desc(variable),
    grouping_variable)
```

I should note that in the above syntax examples, the `grouping_variable` component is often optional. It is included if you want to arrange observations in order within a certain group. For example, if you want to organize your data such that all liberals, moderates and conservatives are put together and then, within that, you want to organize the observations by voters for Donald Trump before voters for Joe Biden (based on alphabetical order by first name), you can use the grouping variable to tell `arrange()` to group by political ideology and the sorting variable would be presidential vote.

Now, let me show you some examples. I use the summary statistics tables from the previous section in these demonstrations. Here, I am arranging by the average age, from greatest to least

```
n_by_residence <- n_by_residence %>%
  arrange(avg_age)
```

residence	n	avg_age
City	16668	44.90659
Town	8809	49.04949
Suburb	23482	49.26667
Rural	11693	51.14479

Now, let's do another example, sorting by descending average age.

```
vote_by_ideo <- vote_by_ideo %>%
  arrange(desc(avg_age))
```

ideo3	pres_vote	n	avg_age
Conservative	Donald J. Trump	1772	59.31151
Moderate	Joseph R. Biden	2923	54.25830
Conservative	Joseph R. Biden	364	52.93681
Moderate	Donald J. Trump	603	51.81758
Liberal	Joseph R. Biden	5430	48.16096
Liberal	Donald J. Trump	66	44.10606

Exercises

1. Using `mutate()` and `case_when()`, recode the `votereg` and `gender` variables such that:
    1. `gender` is coded as 1 is Male and 2 is Female and is a factor variable
    2. `votereg` is coded as 1 is yes and 2 is no
2. Pick any combination of 2 variables, either from the demonstration or your recodes in Part 1. Create a data frame that satisfies the following:
    1. The dataframe ONLY includes both of these variables
    2. There are no NA values on both of these variables
    3. The dataset is arranged in alphabetical order on one of these variables
3. From your outcome in Part 2, generate some sort of summary statistics table. Use reasonable grouping mechanisms.