Week 2 Lab

Data Wrangling Using TidyVerse

Micaela Wood

01/13/2022

Goals for Today

1. Install and use packages
2. Use `readr` to load data
3. Use `dplyr` to manipulate data
4. Use `tidyr` to clean data
5. Use `tibble` to generate data

Next Week

- Use `ggplot2` to visualize data

Install Packages

- Two ways to install and use packages

```r
install.packages("tidyverse")
library(tidyverse)
```

```r
install.packages("pacman")
p_load(tidyverse)
```

- 8 packages included in `tidyverse`

- dplyr
- readr
- tibble
- tidyr
- ggplot2
- forcats
- purrr
- stringr

Readr

- `readr` allows us to load data to **R**

```
cereal <- read_csv("cereal.csv")
```

- If you want to load excel data you will need to either save it as .csv file or use `readxl` package

Loading/Filtering Data

Usually when using dataframes, we need to get our hands dirty. We will evaluate our options with `base` functions before using functions from `dplyr`.

It may well be the case that there is far more data available than we will need. Three options;

- Cherry pick variables from the source,
- Trim variables from the file,
- Load entire file on R and trim down.

Loading/Filtering Data

```
p_load(gapminder)
head(gapminder)
```

```
#> # A tibble: 6 x 6
#>   country     continent  year lifeExp      pop gdpPercap
#>   <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
#> 1 Afghanistan Asia       1952    28.8  8425333      779.
#> 2 Afghanistan Asia       1957    30.3  9240934      821.
#> 3 Afghanistan Asia       1962    32.0 10267083      853.
#> 4 Afghanistan Asia       1967    34.0 11537966      836.
#> 5 Afghanistan Asia       1972    36.1 13079460      740.
#> 6 Afghanistan Asia       1977    38.4 14880372      786.
```

Let's see some common `dplyr` functions using the gapminder dataframe.

Loading/Filtering Data

To generate a variable in your dataframe use `%>% mutate()`

```
data_lnGDP <- gapminder %>% mutate( GDP = pop*gdpPercap,
                                    lnGDP = log(GDP))
```

To filter out particular rows from your dataframe use `%>% filter()`

```
EurAsia <- data_lnGDP %>% filter(continent %in% c("Asia", "Europe"))
# How many countries did I remove?
length(unique(gapminder$country)) - length(unique(EurAsia$country))
```

To summarize by groups, combine `%>% group_by()` and `%>% summarize`
`desc` places `arrange` variables in descending order

```
sum_EurAsia <- EurAsia %>% group_by(country) %>% summarise(
    avg_pop = mean(pop),avg_gdp = mean(GDP)) %>% arrange(desc(avg_gdp))
sum_EurAsia
```

Binding

Binding vectors

Consider `rbind` and `cbind`: they treat the inputs as either rows or columns, and then binds them together.

```
name ← c("Pam", "George", "Sandy")
favorite ← c("Glazed Yams", "Leeks", "Daffodils")
# What are the dimensions of these?
rbind(name, favorite)
cbind(name, favorite)
```

`rbind` yields a 2x3

`cbind` yields a 3x2

Merging

Binding data frames

You can also use `rbind` and `cbind` to bind data frames.

```r
# Create some data frames for us to work with
name_fav <- cbind(name, favorite)
name_work <- cbind(name, work = c("Bus Driver", NA, "Shopkeeper"))
name_fav
```

```
#>      name     favorite
#> [1,] "Pam"    "Glazed Yams"
#> [2,] "George" "Leeks"
#> [3,] "Sandy"  "Daffodils"
```

```r
name_work
```

```
#>      name     work
#> [1,] "Pam"    "Bus Driver"
#> [2,] "George" NA
#> [3,] "Sandy"  "Shopkeeper"
```

Merging

Binding data frames

`cbind` treats the objects as columns, so they're put side-by-side:

$$[\,A, B\,]$$

```
cbind(name_fav, name_work)
```

```
#>      name     favorite      name     work
#> [1,] "Pam"    "Glazed Yams" "Pam"    "Bus Driver"
#> [2,] "George" "Leeks"       "George" NA
#> [3,] "Sandy"  "Daffodils"   "Sandy"  "Shopkeeper"
```

Merging

Binding data frames

`rbind` treats the objects as rows, so they're stacked:

$$\begin{bmatrix} A \\ B \end{bmatrix}$$

```
rbind(name_fav, name_work) #notice how rbind doesn't care about column names
```

```
#>      name     favorite
#> [1,] "Pam"    "Glazed Yams"
#> [2,] "George" "Leeks"
#> [3,] "Sandy"  "Daffodils"
#> [4,] "Pam"    "Bus Driver"
#> [5,] "George" NA
#> [6,] "Sandy"  "Shopkeeper"
```

Binding data frames

`dplyr` has very similar functions `bind_rows` and `bind_cols`. They work best with tibbles, so we'll go ahead and create tibble versions of our data.

```
name_fav_tib ← as_tibble(name_fav)
name_work_tib ← as_tibble(name_work)
```

```
#> # A tibble: 3 x 2
#>   name   favorite
#>   <chr>  <chr>
#> 1 Pam    Glazed Yams
#> 2 George Leeks
#> 3 Sandy  Daffodils
```

```
#> # A tibble: 3 x 2
#>   name   work
#>   <chr>  <chr>
#> 1 Pam    Bus Driver
#> 2 George <NA>
#> 3 Sandy  Shopkeeper
```

Merging

Binding data frames

```
bind_cols(name_fav_tib, name_work_tib
```

```
#> # A tibble: 3 x 4
#>   name...1 favorite    name...3 work
#>   <chr>    <chr>       <chr>    <chr>
#> 1 Pam      Glazed Yams Pam      Bus Driver
#> 2 George   Leeks       George   <NA>
#> 3 Sandy    Daffodils   Sandy    Shopkeeper
```

```
bind_rows(name_fav_tib, name_work_tib
```

```
#> # A tibble: 6 x 3
#>   name   favorite    work
#>   <chr>  <chr>       <chr>
#> 1 Pam    Glazed Yams <NA>
#> 2 George Leeks       <NA>
#> 3 Sandy  Daffodils   <NA>
#> 4 Pam    <NA>        Bus Driver
#> 5 George <NA>        <NA>
#> 6 Sandy  <NA>        Shopkeeper
```

Set Operations

The `dplyr` set operation functions are `union`, `intersect`, and `setdiff`. These set operations treat observations (rows) as if they were set elements.

```
table_1 <- tribble(
  ~"name", ~"favorites",
  #------|--------
  "Pam", "Glazed Yams",
  "George", "Leeks",
  "Sandy", "Daffodils"
)
table_2 <- tribble(
  ~"name", ~"favorites",
  #------|--------
  "Pam", "Glazed Yams",
  "Gus", "Fish Tacos"
)
```

Set Operations

Create tibbles using an easier to read row-by-row layout. This is useful for small tables of data where readability is important

```
table_1
```

```
#> # A tibble: 3 x 2
#>   name   favorites
#>   <chr>  <chr>
#> 1 Pam    Glazed Yams
#> 2 George Leeks
#> 3 Sandy  Daffodils
```

```
table_2
```

```
#> # A tibble: 2 x 2
#>   name  favorites
#>   <chr> <chr>
#> 1 Pam   Glazed Yams
#> 2 Gus   Fish Tacos
```

Merging

Set Operations

`union` will give you all the observations (rows) that appear in either or both tables. This is similar to `bind_rows`, but `union` will remove duplicates.

```
union(table_1, table_2)
```

```
#> # A tibble: 4 x 2
#>   name   favorites
#>   <chr>  <chr>
#> 1 Pam    Glazed Yams
#> 2 George Leeks
#> 3 Sandy  Daffodils
#> 4 Gus    Fish Tacos
```

Merging

Set Operations

`intersect` will give you only the observations that appear both in `table_1` and in `table_2`: in the intersection of the two tables.

```
intersect(table_1, table_2)
```

```
#> # A tibble: 1 x 2
#>   name  favorites
#>   <chr> <chr>
#> 1 Pam   Glazed Yams
```

Merging

Set Operations

`setdiff(table_1, table_2)` gives you all the observations in table_1 that are not in table_2.

```
setdiff(table_1, table_2)
```

```
#> # A tibble: 2 x 2
#>   name   favorites
#>   <chr>  <chr>
#> 1 George Leeks
#> 2 Sandy  Daffodils
```

Mutating joins

Mutating joins take the first table and add columns from the second table.

There are 3 mutating joins: `left_join`, `inner_join`, and `full_join`.

```
# We'll create 2 new data frames to learn mutating joins:
favorites <- tribble(
  ~"name", ~"fav",
  #------|--------
  "Pam", "Glazed Yams",
  "George", "Leeks",
  "Sandy", "Daffodils"
)
jobs <- tribble(
  ~"name", ~"work",
  #------|--------
  "Pam", "Bus Driver",
  "Gus", "Bartender",
  "Sandy", "Shopkeeper"
)
```

Mutating joins

`left_join(x, y)` takes x and adds the columns of y where the **key** matches. The **key** is a variable that shows up in both tables and you'll specify it with `by = "key_variable"`.

```
left_join(favorites, jobs, by = "name
```

```
#> # A tibble: 3 x 3
#>   name   fav          work
#>   <chr>  <chr>        <chr>
#> 1 Pam    Glazed Yams  Bus Driver
#> 2 George Leeks        <NA>
#> 3 Sandy  Daffodils    Shopkeeper
```

```
# What will be the output?
left_join(jobs, favorites, by = "name
```

Mutating joins

`left_join(x, y)` takes x and adds the columns of y where the **key** matches. The **key** is a variable that shows up in both tables and you'll specify it with `by = "key_variable"`.

```
left_join(favorites, jobs, by = "name
```

```
#> # A tibble: 3 x 3
#>   name   fav          work
#>   <chr>  <chr>        <chr>
#> 1 Pam    Glazed Yams  Bus Driver
#> 2 George Leeks        <NA>
#> 3 Sandy  Daffodils    Shopkeeper
```

```
# What will be the output?
left_join(jobs, favorites, by = "name
```

```
#> # A tibble: 3 x 3
#>   name  work       fav
#>   <chr> <chr>      <chr>
#> 1 Pam   Bus Driver Glazed Yams
#> 2 Gus   Bartender  <NA>
#> 3 Sandy Shopkeeper Daffodils
```

Merging

Mutating joins

`inner_join(x, y)` takes the **intersect** of the key variable and adds columns from both tables.

```
inner_join(favorites, jobs, by = "name")
```

```
#> # A tibble: 2 x 3
#>   name  fav         work
#>   <chr> <chr>       <chr>
#> 1 Pam   Glazed Yams Bus Driver
#> 2 Sandy Daffodils   Shopkeeper
```

Merging

Mutating joins

`full_join(x, y)` takes the **union** of the key variable and adds columns from both tables.

```
full_join(favorites, jobs, by = "name")
```

```
#> # A tibble: 4 x 3
#>   name   fav          work
#>   <chr>  <chr>        <chr>
#> 1 Pam    Glazed Yams Bus Driver
#> 2 George Leeks        <NA>
#> 3 Sandy  Daffodils    Shopkeeper
#> 4 Gus    <NA>         Bartender
```

Filtering joins

Unlike mutating joins, filtering joins will only preserve data from the first table. The observations that are kept depends on the second table. dplyr has 2 types of filtering joins: `semi_join` and `anti_join`.

`semi_join(x, y)` keeps all rows in x where the key matches in y.

```
semi_join(favorites, jobs, by = "name")
```

```
#> # A tibble: 2 x 2
#>   name  fav
#>   <chr> <chr>
#> 1 Pam   Glazed Yams
#> 2 Sandy Daffodils
```

Filtering joins

`anti_join(x, y)` keeps rows in x as long as the key **doesn't** have a match in y.

```
anti_join(favorites, jobs, by = "name")
```

```
#> # A tibble: 1 x 2
#>   name   fav
#>   <chr>  <chr>
#> 1 George Leeks
```

Merging

Pivoting

`pivot_wider()` and `pivot_longer()` aren't two-table topics, but they are useful data manipulation tools in the tidyverse.

```
prefs <- tribble(
  ~"name", ~"preference", ~"item",
  #|-----|--------------|--------|
  "Pam", "loves", "Glazed Yams",
  "Pam", "likes", "Daffodils",
  "Pam", "hates", "Horseradish",
  "George", "loves", "Leeks",
  "George", "likes", "Hazelnuts",
  "George", "hates", "Dandelions"
)
```

Take a look at the data. There are 2 people (Pam and George). Each person has one "love", one "like", and one "hate" item.

Merging

Pivoting

Suppose instead we wanted our data in a different format. What if we had 4 columns instead of 3: `name`, the thing that person `loves`, the thing that person `likes`, and the thing that person `hates`. We'd only need 2 rows (Pam and George).

We want our data to go from having 3 columns to having 4, so we know we can use `tidyr::pivot_wider`.

```
pivot_wider(prefs, names_from = preference, values_from = item)
```

```
#> # A tibble: 2 x 4
#>   name   loves       likes     hates
#>   <chr>  <chr>       <chr>     <chr>
#> 1 Pam    Glazed Yams Daffodils Horseradish
#> 2 George Leeks       Hazelnuts Dandelions
```

Pivoting

Now suppose we want to reverse that operation! We'll start with `prefs_wide` and pivot in to get `prefs` again.

`pivot_longer()` has these arguments:

- **cols**: columns to pivot into the longer format. For us, that will be the columns `loves`, `likes`, and `hates`. We can also say columns 2 through 4: `cols = 2:4`.
- **names_to**: A string. What we should call the new column that holds those old column names: `loves`, `likes`, `hates`: `names_to = "preferences"`
- **values_to**: A string. what we should call the values that are now being pivoted in? `Glazed Yams`, `Daffodils`, etc. So we want `values_to = "items"`

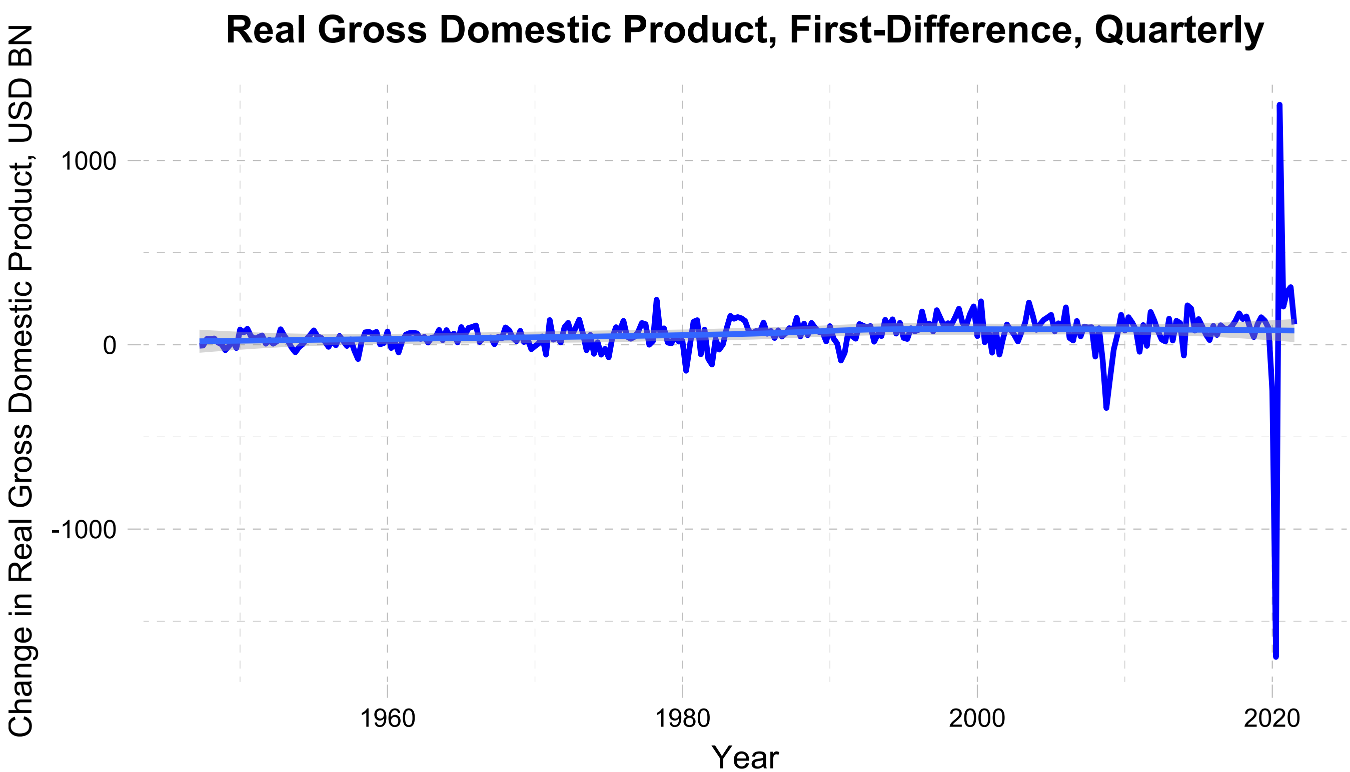
Pivoting

```
prefs_wide %>% pivot_longer(cols = 2:4, names_to = "preferences", values_to = "i
```

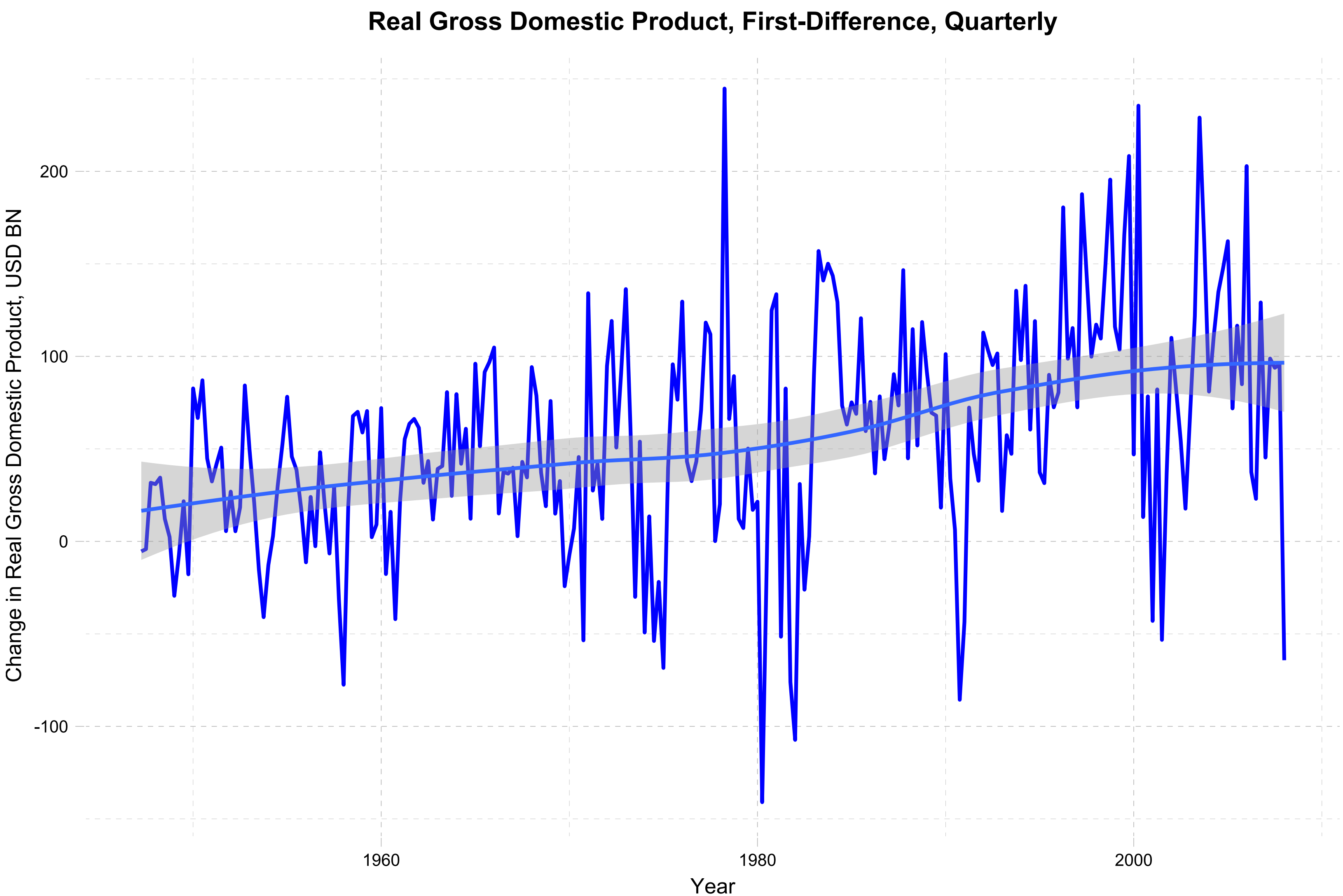
```
#> # A tibble: 6 x 3
#>   name   preferences items
#>   <chr>  <chr>       <chr>
#> 1 Pam    loves       Glazed Yams
#> 2 Pam    likes       Daffodils
#> 3 Pam    hates       Horseradish
#> 4 George loves       Leeks
#> 5 George likes       Hazelnuts
#> 6 George hates       Dandelions
```

Cleaning

Outliers may also be present in data. In macroeconomics, one may be trying to assess mean and variance of real gross domestic product in the United States.



Cleaning


Real Gross Domestic Product, First-Difference, Quarterly
Change in Real Gross Domestic Product, USD BN
200
100
0
-100
1960
1980
2000
Year

Cleaning

Consider the 1.5 IQR rule of thumb. This is used to identify mild outliers. For extreme outliers only, shift to a 3 IQR.

- IQR = 75th Percentile Value - 25th Percentile Value
- Lower Outlier Boundary = 25th - 1.5*IQR
- Upper Outlier Boundary = 75th + 1.5*IQR

```
#Extreme Outliers identified and cleaned
Phase1 ← summary(data$columnX)
OutLower ← Phase1[2]-3*(Phase1[5]-Phase1[2])
OutHigher ← Phase1[5]+3*(Phase1[5]-Phase1[2])
house_w ← filter(house, columnX > OutLower)
house_w ← filter(house, columnX < OutHigher)
```

See example of extreme outlier cleaning in Davies, R., & T., Jeppesen, 2015 *"Export mode, firm heterogeneity, and source country characteristics,* Review of World Economics, Vol. 151(2), pp 169-195.

Cleaning

Resources

- Datacamp: Joining data with dplyr
- RStudio dplyr Cheat Sheet
- Tidyverse