Hertie School/SCRIPTS Data Science Workshop Series

Session 2: Modern data management with R

*Therese Anders**
Allison Koh†

January 17, 2020

Modern data management with R

Introduction

Data cleaning and reshaping is one of the tasks that we end up spending the most time on. Today, we will be introducing the `dplyr` library. Together with `stringr` (string operations using regular expressions) and `tidyr`, these packages offer functionality for virtually any data cleaning and reshaping task in `R`.

a selection of commonly used functions from the `tidyverse`. First, we will provide an overview of `haven` for reading and writing data formats used by other statistical packages.

For an overview of the most common functions inside `dplyr`, please refer to the RStudio data wrangling cheat sheet https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf.

Functions in `dplyr`

`dplyr` does not accept tables or vectors—just data frames! `dplyr` uses a strategy called "Split - Apply - Combine". Some of the key functions include:

- `select()`: Subset columns.
- `filter()`: Subset rows.
- `arrange()`: Reorders rows.
- `mutate()`: Add columns to existing data.
- `summarise()` or `summarize()`: Summarizing the data set.

First, lets download the package and call it using the `library()` function.

```
# install.packages("dplyr")
```

```
df <- read.csv("538_presidential_endorsements.csv")
str(df)
```

```
## 'data.frame':    1006 obs. of  10 variables:
##  $ year          : int  2017 2019 2019 2019 2019 2019 2019 2019 2019 2019 ...
##  $ position      : Factor w/ 51 levels "attorney general",..: 35 21 38 38 25 12 12 44 23 25 ...
##  $ state         : Factor w/ 57 levels "AK","AL","AR",..: 24 39 6 11 49 6 10 6 6 6 ...
##  $ endorser      : Factor w/ 1006 levels "A. Donald McEachin",..: 228 40 252 924 822 583 393 317 289
##  $ endorsee      : Factor w/ 17 levels "Amy Klobuchar",..: 8 7 7 7 10 11 2 11 11 11 ...
##  $ endorser.party: Factor w/ 4 levels "D","I","NPP",..: 1 1 1 1 NA 1 1 1 1 1 ...
##  $ source        : Factor w/ 185 levels "http://www.politicspa.com/boyle-cartwright-and-evans-back-b
##  $ order         : int  NA NA NA NA 6 NA NA 22 19 43 ...
##  $ category      : Factor w/ 12 levels "Candidates who have dropped out",..: 9 5 10 10 6 3 3 12 12 6
##  $ points        : int  3 8 6 6 3 1 1 2 2 3 ...
```

*Instructor, Hertie School/SCRIPTS, anders@hertie-school.org.
†Teaching Assistant, Hertie School, kohallison3@gmail.com.

Using `select()` and introducing the Piping Operator %>%

Using the so-called **piping operator** from the `magrittr` package will make the `R` code faster and more legible, because we are not saving every output in a separate data frame, but passing it on to a new function. First, let's use only a subsample of variables in the data frame, specifically the year of the flight, the airline, as well as the origin airport, the destination, and the distance between the airports.

Notice a couple of things in the code below:

- We can assign the output to a new data set.
- We use the piping operator to connect commands and create a single flow of operations.
- We can use the select function to rename variables.
- Instead of typing each variable, we can select sequences of variables.
- Note that the `everything()` command inside `select()` will select all variables.

NOTE: The `::` operator specifies that we want to use the *object* `select()` from the *package* `dplyr`. This explicit programming is useful when functions or objects are contained in multiple packages to avoid confusion.

```
library(magrittr)

df1 <- df %>%
  dplyr::select(year,
                position,
                endorsement.points = points, #Renaming the variable
                state:source) #Selecting a number of columns.
```

Suppose, we didn't really want to select the `source` variable. We can use `select()` to drop variables.

```
df1 <- df1 %>%
  dplyr::select(-source)
```

Introducting `filter()`

There are a number of key operations when manipulating observations (rows).

- `x < y`
- `x <= y`
- `x == y`
- `x != y`
- `x >= y`
- `x > y`
- `x %in% c(a,b,c)` is `TRUE` if `x` is in the vector `c(a, b, c)`.

Suppose, we wanted to filter all the endorsements that come from states that will take part in Super Tuesday. Registered Democrats from these states will vote for the Democratic nominee, who will be on the ballot for the 2020 Presidential Election.

The states that will hold primary elections on this year's Super Tuesday (March 3, 2020) are Alabama (AL), Arkansas (AR), California (CA), Colorado (CO), Maine (ME), Massachussets (MA), Minnesota (MN), North Carolina (NC), Oklahoma (OK), Tennessee (TN), Texas (TX), Utah (UT), Vermont (VT), and Virginia (VA).

```
super_tuesday <- c("AL", "AR", "CA", "CO", "ME", "MA", "MN", "NC", "OK", "TN", "TX", "UT", "VT", "VA")
df2 <- df1 %>%
  dplyr::filter(state %in% super_tuesday)
```

The following also returns all entries from states that will hold primaries on Super Tuesday.

```
head(df1)
```

```
##   year       position endorsement.points state         endorser
## 1 2017 representative                  3    MD      David Trone
## 2 2019       governor                  8    NY     Andrew Cuomo
## 3 2019        senator                  6    CA Dianne Feinstein
## 4 2019        senator                  6    DE Thomas R. Carper
## 5 2019          mayor                  3    TX    Ron Nirenberg
## 6 2019     DNC member                  1    CA  Laphonza Butler
##        endorsee endorser.party
## 1  John Delaney              D
## 2     Joe Biden              D
## 3     Joe Biden              D
## 4     Joe Biden              D
## 5 Julian Castro           <NA>
## 6 Kamala Harris              D
```

```
df1_alt <- df1 %>%
  dplyr::filter(state == c("AL", "AR", "CA", "CO", "ME", "MA", "MN", "NC", "OK", "TN", "TX", "UT", "VT"
```

```
## Warning in `==.default`(state, c("AL", "AR", "CA", "CO", "ME", "MA",
## "MN", : longer object length is not a multiple of shorter object length
```

```
## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple of
## shorter object length
```

```
head(df1_alt)
```

```
##   year        position endorsement.points state         endorser
## 1 2019         senator                  6    CA Dianne Feinstein
## 2 2019       president                  2    MA     Karen Spilka
## 3 2019  representative                  3    TX Veronica Escobar
## 4 2019 state treasurer                  2    VT      Beth Pearce
## 5 2019  representative                  3    MA  Katherine Clark
## 6 2019 speaker pro tem                  2    TX        Joe Moody
##           endorsee endorser.party
## 1        Joe Biden              D
## 2 Elizabeth Warren              D
## 3    Beto O'Rourke              D
## 4   Bernie Sanders              D
## 5 Elizabeth Warren              D
## 6    Julian Castro              D
```

Helper functions

`dplyr` has a number of helper functions–and that is where the magic lies. These can be used with either `select()` or `filter()`. Here are some useful functions:

- `starts_with()`
- `ends_with()`
- `contains()`
- `matches()`: Every name that matches "X", which can be a regular expression (more on that later).
- `one_of()`: Every name that appears in x, which should be a character vector.

For example, let's create a data frame with all variables that contain the word "endorse".

```
library(tidyselect)

test <- df2 %>%
  dplyr::select(contains("endorse"))
```

Introducting `mutate()`

Suppose we want to rescale the `endorsement.points` variable to run from 0 to 1. Since the original variable is scaled from 1 (least valuable endorsement) to 10 (most valuable endorsement; former presidents and vice presidents; current party leaders), we can create a new variable `endorsement.points.rescaled` by dividing the original variable by 10.

```
df2 <- df2 %>%
  dplyr::mutate(endorsement.points.rescaled = endorsement.points/10)
```

Suppose I only wanted to look at endorsements for candidates who are still in the race. I can create a binary variable that codes whether a candidate has dropped out of the race. Here, we introduce the `ifelse()` function that is tremendously helpful in data wrangling exercises. The syntax of `ifelse()` is as follows:

`ifelse(condition, if TRUE this, if FALSE this).`

```
drop <- c("Kamala Harris",
          "Cory Booker",
          "Beto O'Rourke",
          "Julian Castro",
          "Jay Inslee",
          "Steve Bullock",
          "Kirsten Gillibrand",
          "John Hickenlooper",
          "Eric Swalwell")

df3 <- df2 %>%
  dplyr::mutate(dropped = ifelse(endorsee %in% drop, 1, 0)) %>%
  dplyr::filter(dropped == 0)
```

Introducting `summarise()` and `arrange()`

One of the most powerful `dplyr` features is the `summarise()` function, especially in combination with `group_by()`.

First, let's compute the total number of endorsements by endorsee in the Super Tuesday states. Here, we will be using the operator `n()` to tell dplyr to count all the observations for the groups specified in `group_by()`. Also, suppose we want to know the average endorsement points attributed to each candidate. Anothter interesting variable would be the total endorsement points for each candidate to get an idea of the influence of each candidate's endorsements.

Note that, because there are missing values, we need to tell `R` what to do with them.

```
by_endorsee <- df3 %>%
  dplyr::group_by(endorsee) %>%
  dplyr::summarise(n.endorsement=dplyr::n(),
            average.endorsement.points = mean(endorsement.points, na.rm = T),
            total.endorsement.points = sum(endorsement.points))
```

```
## Warning: Factor `endorsee` contains implicit NA, consider using
## `forcats::fct_explicit_na`
```

To get an idea of where each candidate stands among politicians from the Super Tuesday states, lets reorder the output in descending order using `arrange()`.

```
by_endorsee <- df3 %>%
  dplyr::group_by(endorsee) %>%
  dplyr::summarise(n.endorsement=dplyr::n(),
            average.endorsement.points = mean(endorsement.points, na.rm = T),
            total.endorsement.points = sum(endorsement.points)) %>%
  dplyr::arrange(desc(n.endorsement)) #Default is ascending order
```

```
## Warning: Factor `endorsee` contains implicit NA, consider using
## `forcats::fct_explicit_na`
```

```
by_endorsee
```

```
## # A tibble: 9 x 4
##   endorsee       n.endorsement average.endorsement.p~ total.endorsement.po~
##   <fct>                  <int>                  <dbl>                  <int>
## 1 <NA>                     215                   2.19                    470
## 2 Joe Biden                 23                   3                        69
## 3 Bernie Sanders            15                   2.2                      33
## 4 Amy Klobuchar             13                   3.85                     50
## 5 Elizabeth War~            13                   3.23                     42
## 6 Michael Bloom~             5                   2.2                      11
## 7 Pete Buttigieg             4                   3.25                     13
## 8 Michael Bennet             3                   4                        12
## 9 John Delaney               1                   3                         3
```

So, as of January 14, 2020, 215 politicians from the Super Tuesday states have not endorsed a candidate for the Democratic primary elections. Among the endorsees, Joe Biden received the most endorsements and has the highest number of "total endorsement points" per the scale of each endorsement's relative value generated by Five Thirty Eight. However, the candidate with the highest value for "average endorsement points" is Amy Klobuchar. *Perhaps* this indicates that, on average, politicians from Super Tuesday states who have endorsed Amy Klobuchar have more influence.

Putting it all together: The power of piping

In this example, I am starting all the way from the original `df` dataframe to demonstrate the power of the piping operator.

```
df_new <- df %>%
  dplyr::select(year,
                position,
                endorsement.points = points,
                state:source) %>%
  dplyr::filter(state %in% super_tuesday) %>%
  dplyr::mutate(endorsement.points.rescaled = endorsement.points/10,
                dropped = ifelse(endorsee %in% drop, 1, 0)) %>%
  dplyr::filter(dropped == 0) %>%
  dplyr::group_by(endorsee) %>%
  dplyr::summarise(n.endorsement=dplyr::n(),
            average.endorsement.points = mean(endorsement.points, na.rm = T),
```

```
            total.endorsement.points = sum(endorsement.points)) %>%
  dplyr::arrange(desc(n.endorsement))
```

```
## Warning: Factor `endorsee` contains implicit NA, consider using
## `forcats::fct_explicit_na`
```

```
df_new
```

```
## # A tibble: 9 x 4
##   endorsee       n.endorsement average.endorsement.p~ total.endorsement.po~
##   <fct>                  <int>                  <dbl>                  <int>
## 1 <NA>                     215                   2.19                    470
## 2 Joe Biden                 23                   3                        69
## 3 Bernie Sanders            15                   2.2                      33
## 4 Amy Klobuchar             13                   3.85                     50
## 5 Elizabeth War~            13                   3.23                     42
## 6 Michael Bloom~             5                   2.2                      11
## 7 Pete Buttigieg             4                   3.25                     13
## 8 Michael Bennet             3                   4                        12
## 9 John Delaney               1                   3                         3
```

Saving files

Finally, lets save the output of our code. This time, we would like to save it in the .dta format. Remember to load the `foreign()` library before saving.

```
library(foreign)
write.dta(df_new, "super_tuesday_endorsements.dta")
```

Merging data

The data

In R, the package **haven** allows us to access read and write various data formats. Currently, it supports Stata, SPSS and SAS files. Value labels from imported data are translated into a new `labelled()` class, which preserves the original semantics. In addition to some of the functions that can be used to explore the dimensions of any data frame, we can also explore the substantive descriptions of variables.

For this exercise, we will be working with data from the Pew Research Center's *American Trends Panel* (ATP), which is a nationally representative panel of randomly selected U.S. adults. First, we will load the data from the 38th and 39th wave of the panel. The data has been cleaned after downloading from the original sources and subsetted to only include U.S. citizens and respondents who participated in both surveys.

The topics covered in each wave are as follows:

- Wave 38: Pre-election poll, generations research
- Wave 39: Post-election poll

The Wave 38 data contains the following variables:

- `POL1DT_W38`: Do you approve or disapprove of the way Donald Trump is handling his job as president?
- `VTPLAN`: Do you plan to vote in the elections this November?
- `F_PARTY_FINAL`: Party affiliation

The Wave 39 data contains the following variables:

- `POL1DT_W39`: Do you approve or disapprove of the way Donald Trump is handling his job as president?
- `VTHPPYUS_W39`: All in all, are you happy or unhappy with the results of the recent elections that took place across the United States?
- `VOTED_ATP`: Which of the following statements best describes you?
- `F_PARTY_FINAL`: Party affiliation

First, let's load the data from both waves.

```
library(haven)
# load datasets
w38.df <- haven::read_sav("ATP_W38.sav")
w39.df <- haven::read_sav("ATP_W39.sav")
```

Then, let's investigate some of the variables used in each dataset.

```
attributes(w38.df$VTPLAN_W38)
```

```
## $label
## [1] "VTPLAN. Do you plan to vote in the elections this November?"
##
## $format.spss
## [1] "F8.2"
##
## $class
## [1] "haven_labelled"
##
## $labels
##     Yes, I plan to vote ON Election Day
##                                       1
## Yes, I plan to vote BEFORE Election Day
##                                       2
##               No, I do not plan to vote
##                                       3
##                                Not sure
##                                       4
##                                 Refused
##                                      99
```

```
attributes(w39.df$VOTED_ATP_W39)
```

```
## $label
## [1] "VOTED_ATP. Which of the following statements best describes you?"
##
## $format.spss
## [1] "F8.2"
##
## $class
## [1] "haven_labelled"
##
## $labels
##     I did not vote in the 2018 congressional elections
##                                                       1
##                   I planned to vote but wasn't able to
##                                                       2
## I definitely voted in the 2018 congressional elections
##                                                       3
##                                                Refused
```

```
##                                                 99
```

```
attributes(w38.df$F_PARTY_FINAL)
```

```
## $label
## [1] "Party"
##
## $format.spss
## [1] "F8.2"
##
## $class
## [1] "haven_labelled"
##
## $labels
##     Republican       Democrat    Independent Something else        Refused
##              1              2              3              4             99
```

```
attributes(w38.df$F_IDEO)
```

```
## $label
## [1] "Ideology"
##
## $format.spss
## [1] "F8.2"
##
## $class
## [1] "haven_labelled"
##
## $labels
## Very conservative      Conservative          Moderate           Liberal
##                 1                 2                 3                 4
##      Very liberal           Refused
##                 5                99
```

Exercise 1 Suppose we would like to investigate differences between Democrats and Republicans in the Wave 39 data. Add new variables called `dem` and `rep` that code these party affiliations using functions from the `dplyr` package and piping.

```
## # A tibble: 6 x 8
##     QKEY POL1DT_W39 VTHPPYUS_W39 VOTED_ATP_W39 F_PARTY_FINAL  F_IDEO   DEM
##    <dbl>  <dbl+lbl>    <dbl+lbl>     <dbl+lbl>     <dbl+lbl> <dbl+l> <dbl>
## 1 100197 2 [Disapp~  1 [Happy]   3 [I definit~ 2 [Democrat]  4 [Lib~     1
## 2 100260 1 [Approv~  1 [Happy]   3 [I definit~ 1 [Republica~ 1 [Ver~     0
## 3 100314 1 [Approv~  2 [Unhappy] 3 [I definit~ 1 [Republica~ 3 [Mod~     0
## 4 100363 1 [Approv~  1 [Happy]   3 [I definit~ 2 [Democrat]  4 [Lib~     1
## 5 100446 2 [Disapp~  1 [Happy]   3 [I definit~ 2 [Democrat]  4 [Lib~     1
## 6 100588 2 [Disapp~  1 [Happy]   3 [I definit~ 2 [Democrat]  3 [Mod~     1
## # ... with 1 more variable: REP <dbl>
```

A primer on data merging using base `R` functions

In principle, we can merge data using the `cbind()` ("column bind") and `rbind()` ("row bind") functions, that we are already familiar with.

```
vec1 <- c(1, 2, 3)
vec2 <- c("a", "a", "b")
```

```
cbind(vec1, vec2)
```

```
##      vec1 vec2
## [1,] "1"  "a"
## [2,] "2"  "a"
## [3,] "3"  "b"
```

```
rbind(vec1, vec2)
```

```
##      [,1] [,2] [,3]
## vec1 "1"  "2"  "3"
## vec2 "a"  "a"  "b"
```

However, this requires the data to align in its dimensions. In addition, for `cbind()` the ordering of the rows has to be the exact same to achieve correct merging; for `rbind()` the ordering of the columns has to be equal to achieve the right output.

```
dat1 <- cbind(vec1, vec2)
vec3 <- c(T, F, T, T)
cbind(dat1, vec3) # The fourth observation in vec3 is omitted
```

```
## Warning in cbind(dat1, vec3): number of rows of result is not a multiple of
## vector length (arg 2)
```

```
##      vec1 vec2 vec3
## [1,] "1"  "a"  "TRUE"
## [2,] "2"  "a"  "FALSE"
## [3,] "3"  "b"  "TRUE"
```

```
obs <- c("c", 4)
rbind(dat1, obs)
```

```
##     vec1 vec2
##     "1"  "a"
##     "2"  "a"
##     "3"  "b"
## obs "c"  "4"
```

As a side note: The `bind_rows()` function in the `dplyr` package is the smart cousin of `rbind()`. Given that two dataframes have an equal number of columns, it can merge them based on the variable name; the ordering of the columns does not matter.

Using dplyr for merging

Luckily, the `dplyr` package in `R` contains a number of functions that allow us to merge data in "smarter" ways. Please refer to the Data Wrangling Cheat Sheet for an overview of the functions `dplyr` has available.

Suppose, we have two data frames: `x` (here WDI) and `y` (here COW). The basic syntax for data merging with `dplyr` is the following:

```
output <- join(A, B, by = "variable")
```

We will focus on the following four join functions:

- `full_join()`: Join data from `x` and `y` upon retaining all rows and values. This is the maximum join possible. Neither `x` nor `y` is the "master."
- `inner_join()`: Join only those rows that appear in both `x` and `y`. Neither `x` nor `y` is the "master."
- `left_join()`: Join only those rows from `y` that appear in `x`, retaining all data in `x`. Here, `x` is the "master."

- `right_join()`: Join only those rows from `x` that appear in `y`, retaining all data in `y`. Here, `y` is the "master."

`full_join()`

Suppose we wanted to keep the maximum amount of data in our new merged data set. We would use the `full_join()` function to merge the Wave 38 and Wave 39 data.

```
full_wrong <- dplyr::full_join(w38.df, w39.df, by = "F_PARTY_FINAL")
nrow(full_wrong)
```

```
## [1] 30847850
```

```
full_wrong
```

```
## # A tibble: 30,847,850 x 12
##    QKEY.x POL1DT_W38 VTPLAN_W38 F_PARTY_FINAL F_IDEO.x QKEY.y POL1DT_W39
##     <dbl>  <dbl+lbl>  <dbl+lbl>     <dbl+lbl> <dbl+lb>  <dbl>  <dbl+lbl>
##  1 100197 2 [Disapp~ 2 [Yes, I~  2 [Democrat] 4 [Libe~ 100197 2 [Disapp~
##  2 100197 2 [Disapp~ 2 [Yes, I~  2 [Democrat] 4 [Libe~ 100363 1 [Approv~
##  3 100197 2 [Disapp~ 2 [Yes, I~  2 [Democrat] 4 [Libe~ 100446 2 [Disapp~
##  4 100197 2 [Disapp~ 2 [Yes, I~  2 [Democrat] 4 [Libe~ 100588 2 [Disapp~
##  5 100197 2 [Disapp~ 2 [Yes, I~  2 [Democrat] 4 [Libe~ 100734 2 [Disapp~
##  6 100197 2 [Disapp~ 2 [Yes, I~  2 [Democrat] 4 [Libe~ 101400 2 [Disapp~
##  7 100197 2 [Disapp~ 2 [Yes, I~  2 [Democrat] 4 [Libe~ 102009 2 [Disapp~
##  8 100197 2 [Disapp~ 2 [Yes, I~  2 [Democrat] 4 [Libe~ 102932 2 [Disapp~
##  9 100197 2 [Disapp~ 2 [Yes, I~  2 [Democrat] 4 [Libe~ 103060 2 [Disapp~
## 10 100197 2 [Disapp~ 2 [Yes, I~  2 [Democrat] 4 [Libe~ 103414 2 [Disapp~
## # ... with 30,847,840 more rows, and 5 more variables:
## #   VTHPPYUS_W39 <dbl+lbl>, VOTED_ATP_W39 <dbl+lbl>, F_IDEO.y <dbl+lbl>,
## #   DEM <dbl>, REP <dbl>
```

A couple of things to notice:

1. There are a couple of duplicate column names in our data. `dplyr` automatically added a suffix to those instances; x for the Wave 38 data (the first data set we passed to the function) and y for the Wave 39 data (the second data set passed to the function). We will rename these variables later on to keep track from which data set they originate.
2. We have over 30 million observations even though our original datasets only have approximately 10,000 observations each. Why do you think this is the case? Do you notice something weird about the `QKEY` variable?

The reason for the high number of observations is that we asked `dplyr` to join the data sets based on the `F_PARTY_FINAL` variable only. This matches each observation in dataset x to each observation in dataset y as long as they have the same party affiliation. However, the rows in our data are not uniquely identified by the `F_PARTY_FINAL` variable. Since we are working with panel data sets, each observation is uniquely identified by the variable `QKEY` to track which respondents participated in multiple waves. We need to pass this information to the `full_join()` function to correctly match up the observations in our data.

In some cases, `full_join()` and the other join functions from dplyr can often identify which columns to match on based on duplicate column names. However, in our particular case, we run into problems because we have more duplicate column names that unique identifiers for the data; see the message that R outputs when running the command without specifying by which variables to merge.

```
full_alsowrong <- dplyr::full_join(w38.df, w39.df)
```

```
## Joining, by = c("QKEY", "F_PARTY_FINAL", "F_IDEO")
```

```
nrow(full_alsowrong)
```

```
## [1] 11508
```

Before we do the correct `full_join()`, let's change the name of variables that are not unique identifiers in each dataset. To do this, we can use the `select()` function.

```
w38.df <- w38.df %>%
  select(QKEY:VTPLAN_W38,
         F_PARTY_FINAL_W38 = F_PARTY_FINAL,
         F_IDEO_W38 = F_IDEO)
names(w38.df)
```

```
## [1] "QKEY"              "POL1DT_W38"        "VTPLAN_W38"
## [4] "F_PARTY_FINAL_W38" "F_IDEO_W38"
```

```
w39.df <- w39.df %>%
  select(QKEY:VOTED_ATP_W39,
         F_PARTY_FINAL_W39 = F_PARTY_FINAL,
         F_IDEO_W39 = F_IDEO)
names(w39.df)
```

```
## [1] "QKEY"              "POL1DT_W39"        "VTHPPYUS_W39"
## [4] "VOTED_ATP_W39"     "F_PARTY_FINAL_W39" "F_IDEO_W39"
```

Now we are ready to join!

```
merged_full <- full_join(w38.df, w39.df, by = "QKEY")
head(merged_full)
```

```
## # A tibble: 6 x 10
##     QKEY POL1DT_W38 VTPLAN_W38 F_PARTY_FINAL_W~ F_IDEO_W38 POL1DT_W39
##    <dbl>  <dbl+lbl>  <dbl+lbl>        <dbl+lbl>  <dbl+lbl>  <dbl+lbl>
## 1 100197 2 [Disapp~ 2 [Yes, I~  2 [Democrat]    4 [Libera~  2 [Disap~
## 2 100252 2 [Disapp~ 2 [Yes, I~  3 [Independent] 3 [Modera~ NA
## 3 100260 1 [Approv~ 2 [Yes, I~  1 [Republican]  1 [Very c~  1 [Appro~
## 4 100314 1 [Approv~ 1 [Yes, I~  1 [Republican]  3 [Modera~  1 [Appro~
## 5 100363 1 [Approv~ 1 [Yes, I~  2 [Democrat]    4 [Libera~  1 [Appro~
## 6 100446 2 [Disapp~ 1 [Yes, I~  2 [Democrat]    4 [Libera~  2 [Disap~
## # ... with 4 more variables: VTHPPYUS_W39 <dbl+lbl>,
## #   VOTED_ATP_W39 <dbl+lbl>, F_PARTY_FINAL_W39 <dbl+lbl>,
## #   F_IDEO_W39 <dbl+lbl>
```

```
nrow(merged_full)
```

```
## [1] 11487
```

inner_join()

Suppose, instead of retaining as much data as possible, we only wanted to keep data for which we have observations in both data sets. Note that this merging is only done based on the variables that we specify to uniquely identify each observation. `NA` values in all other columns are retained. Please refer to the *Data Wrangling Cheat Sheet* for methods that will select based on matching values beyond the unique identifiers (see for example `intersect()` and `setdiff()`).

```
merged_inner <- inner_join(w38.df, w39.df, by = "QKEY", na.rm = T)
head(merged_inner)
```

```
## # A tibble: 6 x 10
##     QKEY POL1DT_W38 VTPLAN_W38 F_PARTY_FINAL_W~ F_IDEO_W38 POL1DT_W39
##    <dbl>  <dbl+lbl>  <dbl+lbl>        <dbl+lbl>  <dbl+lbl>  <dbl+lbl>
## 1 100197 2 [Disapp~ 2 [Yes, I~   2 [Democrat]   4 [Libera~ 2 [Disapp~
## 2 100260 1 [Approv~ 2 [Yes, I~   1 [Republican] 1 [Very c~ 1 [Approv~
## 3 100314 1 [Approv~ 1 [Yes, I~   1 [Republican] 3 [Modera~ 1 [Approv~
## 4 100363 1 [Approv~ 1 [Yes, I~   2 [Democrat]   4 [Libera~ 1 [Approv~
## 5 100446 2 [Disapp~ 1 [Yes, I~   2 [Democrat]   4 [Libera~ 2 [Disapp~
## 6 100588 2 [Disapp~ 1 [Yes, I~   2 [Democrat]   3 [Modera~ 2 [Disapp~
## # ... with 4 more variables: VTHPPYUS_W39 <dbl+lbl>,
## #   VOTED_ATP_W39 <dbl+lbl>, F_PARTY_FINAL_W39 <dbl+lbl>,
## #   F_IDEO_W39 <dbl+lbl>
```

```
nrow(merged_inner)
```

```
## [1] 9298
```

`left_join()` and `right_join()`

Suppose we had an existing master data set and wanted to add data to this master without adding new rows—just new columns. This could for example be the case if we had a specific time frame that we wanted to study, and only want to merge data which matches this time frame.

Here, suppose we wanted to make the Wave 38 data our master data set and merge data from the Wave 39 data onto this master. The Wave 39 data took place after the 2018 midterm elections, while Wave 38 contains pre-election data. We are interested in keeping all of the observations from pre-election, and add on data for pre-election respondents who also participated in Wave 39. We can show that the Wave 38 data and the merged data frame have the same number of observations (rows).

```
merged_left <- left_join(w38.df, w39.df, by = c("QKEY"))
nrow(w38.df) == nrow(merged_left)
```

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

Exercise 2 How would you achieve the exact same result using the `right_join()` function?

```
## [1] FALSE
```

Data re-shaping with `tidyr`

Another important task in data management is data re-shaping. Often, data does not come in the format that we need for data merging, data visualization, statistical analysis, or vectorized programming. In general, we want data in the following format:

1. Each variable forms a column.
2. Each observation forms a row[1].
3. For panel data, the unit (e.g. country) and time (e.g. year) identifier form columns.

The `tidyr` package offers two main functions for data reshaping:

- `gather()`: Shaping data from wide to long.
- `spread()`: Shaping data from long to wide.

[1] Hadley Wickham (2014, "Tidy Data" in *Journal of Statistical Analysis*) adds another condition, namely that "Each type of observational unit forms a table." We will not go into this here, but I can highly recommend you read Wickham's piece if you want to dive deeper into tidying data.

Wide versus long data

For **wide** data formats, each unit's responses are in a single row. For example:

Country	Area	Pop1990	Pop1991
A	300	56	58
B	150	40	45

For **long** data formats, each row denotes the observation of a unit at a given point in time. For example:

Country	Year	Area	Pop
A	1990	300	56
A	1991	300	58
B	1990	150	40
B	1991	150	45

`gather()`

We use the `gather()` function to reshape data from wide to long. In general, the syntax of the data is as follows:

```
new_df <- gather(old_df, key, value, columns to gather)
```

We will be working with the `merged_inner` data set. Suppose, we wanted to have a single column for military expenditures, and another column identifying the data set that the data comes from.

Gathering the military expenditure variables in this way will result in duplicate rows for each country-year. Whether or not this data format is desirable depends on the intended use for the data. Most statistical analysis methods require the data to have a single row per observation (for example country-year). However, some data visualization methods fare better when each concept (for example the variable `milex`) is captured in a single column with additional columns specifying supplementary attributes of the observation (see below).

Below, note that since `tidyr` and `dplyr` are sibling packages from the "tidyverse," we can use them seamlessly in the same pipe (here using `group_by()`, `summarize()`, `mutate()`, and `filter()` from `dplyr`).

```
library(tidyr)
merged_long <- merged_inner %>%
  dplyr::group_by(F_IDEO_W39) %>%
  dplyr::summarize(n=n(),
                   POL1DT_W38=sum(POL1DT_W38),
                   POL1DT_W39=sum(POL1DT_W39)) %>%
  dplyr::mutate(drop = ifelse(F_IDEO_W39==99, 1, 0)) %>% # remove non-responses for political ideology
  dplyr::filter(drop == 0) %>%
  tidyr::gather(variable, # name for categorical name indicator
                key, # values
                c(3:4), # variables to be reshaped
                na.rm = T)
merged_long
```

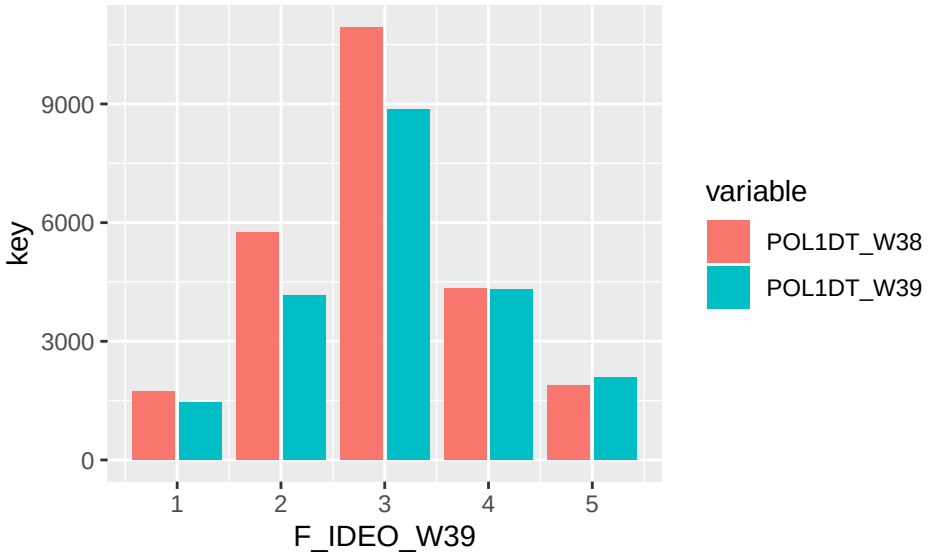
```
## # A tibble: 10 x 5
##               F_IDEO_W39     n  drop variable     key
##                <dbl+lbl> <int> <dbl> <chr>      <dbl>
##  1 1 [Very conservative]   757     0 POL1DT_W38  1743
```

```
##  2 2 [Conservative]       2090     0 POL1DT_W38  5749
##  3 3 [Moderate]           3344     0 POL1DT_W38 10947
##  4 4 [Liberal]            2011     0 POL1DT_W38  4332
##  5 5 [Very liberal]        959     0 POL1DT_W38  1890
##  6 1 [Very conservative]   757     0 POL1DT_W39  1443
##  7 2 [Conservative]       2090     0 POL1DT_W39  4159
##  8 3 [Moderate]           3344     0 POL1DT_W39  8854
##  9 4 [Liberal]            2011     0 POL1DT_W39  4323
## 10 5 [Very liberal]        959     0 POL1DT_W39  2084
```

We can use this format to compare pre- and post-election attitudes about Donald Trump by political ideology among the panel respondents (Note: without applying survey weights, this is not a representative estimate). Here, I use the `ggplot2` package (also from the "tidyverse") to show how to create bar graphs that use the properties of "tidy" data, namely the fact that a specific feature is represented with a single column (here `F_IDEO_W39`), with additional columns specifying supplementary properties of this feature (here `variable`). Note that `F_IDEO_W39==1` indicates "very conservative", while `F_IDEO_W39==5` indicates "very liberal".

```
library(ggplot2)
ggplot(merged_long,
       aes(x = F_IDEO_W39, y = key, fill = variable)) +
  geom_bar(stat = "identity", position = "dodge2")
```

```
## Don't know how to automatically pick scale for object of type haven_labelled. Defaulting to continuou
```



spread()

Suppose we wanted to revert our operation (or generall shape data from a long to a wide format), we can use `tidyr`'s `spread()` function. The syntax is similar to `gather()`.

`new_df <- spread(old_df, key, value)`,

where `key` refers to the colum which contains the values that are to be converted to column names and `value` specifies the column that contains the value which is to be stored in the newly created columns.

```
head(merged_long, 3)
```

```
## # A tibble: 3 x 5
##              F_IDEO_W39     n  drop variable     key
```

```
##                F_IDEO_W39 <dbl+lbl> <int> <dbl> <chr>      <dbl>
## 1 1 [Very conservative]   757     0 POL1DT_W38  1743
## 2 2 [Conservative]       2090     0 POL1DT_W38  5749
## 3 3 [Moderate]           3344     0 POL1DT_W38 10947
```

```
merged_wide <- spread(merged_long, variable, key)
merged_wide
```

```
## # A tibble: 5 x 5
##               F_IDEO_W39     n  drop POL1DT_W38 POL1DT_W39
##                <dbl+lbl> <int> <dbl>      <dbl>      <dbl>
## 1 1 [Very conservative]   757     0       1743       1443
## 2 2 [Conservative]       2090     0       5749       4159
## 3 3 [Moderate]           3344     0      10947       8854
## 4 4 [Liberal]            2011     0       4332       4323
## 5 5 [Very liberal]        959     0       1890       2084
```

String operations

String operations play a large role in the data cleaning component of data management, as well as in data gathering tasks such as web scraping. All string operations work on objects of the class **character**. Base `R` has a number of string operations, but they are often not user-friendly. The `stringr` package provides a comprehensive library of string operations. As part of the "tidyverse," we can use `stringr` functions together with other funtions from the `dplyr` and `tidyr` packages within the same pipe.

Basic string operations

The simplest string operation is base `R`'s `paste()` command. It simply concatenates two strings, by default with a space. We can also specify a custom expression to concatenate the characters with the `sep =` parameter.

```
paste("My", "string")
```

```
## [1] "My string"
```

```
paste("My", "string", sep = "_")
```

```
## [1] "My_string"
```

```
paste("My", "string", sep = "pretty")
```

```
## [1] "Myprettystring"
```

In data management, we most often use the `paste()` function when combining the values of two variables.

```
countries <- c("a", "b", "c")
years <- c(1990, 1990, 2000)
paste(countries, years, sep = "_")
```

```
## [1] "a_1990" "b_1990" "c_2000"
```

Pattern matching using `stringr`

Pattern matching is the work horse of string operations. Here, we focus on pattern matching functions that are most useful for the data management of data that is already organized in some kind of structure, such as vectors or data frames. There are many other string operation functions, such as `str_locate()` and

`str_locate_all()` that are most useful for data cleaning and reorganization of unstructured data. Here, we will cover the following `stringr` functions:

- `str_sub()`: Extract substrings from a character vector by indices.
- `str_extract()`: Extract substrings from a character vector by matched pattern.
- `str_detect()`: Returns `TRUE` or `FALSE` if the pattern in matched in the input string.
- `str_replace()`: Replaces a matched pattern with a new pattern.

str_sub()

Since the `str_sub()` function operates with indices, is most useful when all elements of a vector (or some other data object of interest) have the same pattern. It extracts the information that matches the indices specified by the user. The general syntax is as follows.

```
str_sub(pattern, start = , end = )
```

```
library(stringr)
locations <- c("CA Los Angeles", "NV Las Vegas", "CA San Diego")
locations_state <- str_sub(locations, start = 1, end = 2)
locations_state
```

```
## [1] "CA" "NV" "CA"
```

We can also specify open intervals. For example, if we wanted to extract only the cities, we could extract all characters starting at the fourth position.

```
locations_city <- str_sub(locations, 4)
locations_city
```

```
## [1] "Los Angeles" "Las Vegas"   "San Diego"
```

str_extract()

Think of `str_extract()` as the smarter sibling of `str_sub()`. Rather than just extracting characters based on indices, it extracts characters based on matched patterns. Together with the regular expression operations we introduce below, this is very powerful in extracting information. Here is the syntax for `str_extract()`:

```
str_extract(string, pattern)
```

```
locations_state2 <- str_extract(locations, "CA")
locations_state2
```

```
## [1] "CA" NA   "CA"
```

str_detect()

Think of `str_detect()` as a logical operator. It returns `TRUE` if the pattern is matched and `FALSE` otherwise. The function has the following syntax:

```
str_detect(string, pattern)
```

```
str_detect(locations, "L")
```

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

```
str_detect(locations, "Los")
```

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

`str_replace()`

`str_replace()` is very helpful in data management tasks, in particular when re-coding variables. As an example, suppose we wanted to replace the state abbreviations with the full state names. The syntax of the function is as follows:

```
str_replace(string, pattern, replacement)
```

```
locations_fullname <- str_replace(locations, "CA", "California") %>%
  str_replace("NV", "Nevada")
locations_fullname
```

```
## [1] "California Los Angeles" "Nevada Las Vegas"
## [3] "California San Diego"
```

Regular Expressions

String operations are useful, but their true potential is realized when used in combination with regular expressions. Regular expressions help us to define search patterns. This obliviates the need to type out entire character strings in pattern matching. Regular expressions are like a language that is understood by more than just one program. With small changes, you would be able to use the regular expressions discussed here in other programming languages, such as Python.

Suppose, our strings weren't as well organized as before. How would you extract the state from the elements of the following character vector?

```
locs <- c("CA Los Angeles", "Las NV Vegas", "San Diego CA")
```

Even though the elements of the character vector aren't as ordered any more, there is still a pattern! State abbreviations always have two letters and they are always capitalized. Below, we will learn a number of regular expression that will help us state this pattern ("two letters, capitalized") in a language that `R` understands.

Note that unless otherwise specified, regular expressions are case sensitive. The following patterns are examples for specifying the substantive content of the pattern.

- `^`: String **starts with** following pattern..
- `$`: String **ends with** preceding pattern.
- `[ ]`: or.
- `[a-z]`: Any letter.
- `[0-9]`: Any digit.

There is also a number of regular expressions that are used to specify the quantity of matches.

- `?`: Preceding pattern is optional (matched 0 or 1 times).
- `*`: Preceding pattern is matched 0 or more times.
- `+`: Preceding pattern is matched at least once.
- `{n}`: Preceding pattern is matched exactly`n` times.
- `{n, m}`: Preceding pattern is matched at least `n` times and up to `m` times.
- `{n,}`: Preceding pattern is matched at least `n` times.

So in our example above, we could use the following regular expression to extract the state names:

```
str_extract(locs, "[A-Z]{2}")
```

```
## [1] "CA" "NV" "CA"
```

Examples for regular expressions

```
fruits <- c("apple", "banana", "pineapple", "cherries4")
```

Exercise 1 What is the output of the following command?

```
str_detect(fruits, "a")
```

Exercise 2 What is the output of the following command?

```
str_detect(fruits, "^a")
```

Exercise 3 How would you ask `R` to print out `TRUE` for all elements of the `fruits` vector that end with the letter `e`?

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

Exercise 4 What is the output of the following command?

```
str_detect(fruits, "[ie]")
```

Exercise 5 How would you extract the number from the last element of the `fruits` vector?

```
## [1] "4"
```

We can use regular expressions to create more complex queries. Suppose, we wanted to know which strings start with an `a` and end with an `e`.

```
str_detect(fruits, "^a[a-z]*e$")
```

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

Exercise 6 What do you think is the output of the following command?

```
str_detect(fruits, "a[a-z]*e$")
```

Escape character

Regular expressions can also be used to match special characters. Note, however, that since some special characters have a special meaning in `R`, we sometimes have to use a so-called **escape character**, i.e. a backslash, to specify these patterns within regular expressions. For example, since the open square bracket `[` is part of the regular expression for `OR`, in order to match a `[` in the text, you need to type `\\[` to get the desired output.

```
test <- "a []"
#str_detect(test, "[") # This throws and error.
str_detect(test, "\\[") # This correctly recognized the pattern.
```

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

Application: Phone Numbers

Suppose, we have data on contact information and wanted to extract only the values that match phone numbers. Unfortunately, phone numbers come in different formats, but there are a number of common patterns.

What are common patterns for phone numbers you guys can think of?

```
phone <- c("123 456 7890",
           "(123) 456 7890",
           "123-456-7890")
```

We cannot use simple indices any longer to extract the phone numbers from this vector. The lengths of the elements differ!

```
nchar(phone[1])
```

```
## [1] 12
```

```
nchar(phone[2])
```

```
## [1] 14
```

Lets write a regular expression that outputs `TRUE` for all of these phone numbers.

```
str_detect(phone, "[(]?[0-9]{3}[)]?[ -]{1}[0-9]{3}[ -]{1}[0-9]{3}")
```

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

Exercise 7 How would you alter this if I gave you the following number format and asked you to match it as well? `123.456.7890`

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

Application: Creating a data frame

Why do we need this? Suppose I gave you a long string of data and asked you to clean the data and give be a data set of phone numbers and ZIP codes. You could make your life **a lot** easier if you know regular expressions.

Load the file `data.RData` and implicitly print out at its content, a vector called `vec`.

```
# load("data.RData")
# vec
# # First, lets get the phone numbers (since we already have a parser)
# phone <- str_extract(vec, "[(]?[0-9]{3}[)]?[ -.]{1}[0-9]{3}[ -.]{1}[0-9]{3}")
# phone
# # Second, get ZIP codes
# zip <- str_extract(vec, "[0-9]{5}")
# zip
# # Lastly, lets get the IDs
# id <- str_extract(vec, "[0-9]+:")
# id
# id <- str_replace(id, ":", "")
# id
# # Putting it all into one data frame
# df <- data.frame(id, phone, zip)
# print(df)
# # Now cleaning it
# # Assumptions:
# # a) no one has two zip codes or phone numbers,
# # b) No id is missing.
# library(dplyr)
# library(tidyr)
# df_cleaned <- df %>%
#   gather(indicator, value, phone:zip) %>%
```

```
#   na.omit() %>%
#   spread(indicator, value)
# df_cleaned
```

`for` Loops in `R`

Loops are a staple of programming. Loops allow us to automate our code, and are particularily useful when you find yourself doing a task over and over again. While in practice, we try to vectorize our operations as much as possible (see the solution above where we convert factors to characters), being comfortable with loops is crucial for many programming tasks.

Basic loop structure

General syntax of a for loop:

`for(iterator in iterations){function/output}`

Lets write a loop that prints out the numbers 1 through 10.

```
for(i in 1:10){
  print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

Suppose, the numbers we wanted to print out are part of a vector. We can use loops to iterate through this vector.

```
vec <- seq(11, 20, 1)
for(k in vec){
  print(k)
}
```

```
## [1] 11
## [1] 12
## [1] 13
## [1] 14
## [1] 15
## [1] 16
## [1] 17
## [1] 18
## [1] 19
## [1] 20
```

Exercise 1 What do you think does the following output do?

```
for(l in 5:length(vec)){
  print(l)
}
```

Of course, we can use loops to automate more complex tasks. For example, we could use it to change a batch of variables to `character()`. To try this, re-load the data from session 5.

```
# data_new <- read.csv("hw5_data.csv")
# str(data_new)
# test1 <- data_new
# for(l in 1:ncol(test1)){
#   test1[,l] <- as.character(test1[,l])
# }
# str(test1)
```

As a more sophisticated version, we could convert only those variables that are factors (not numerical variables like `Density.Pop....km2..`) to characters, using an `if` statement.

```
# test2 <- data_new
# for(k in 1:ncol(test2)){
#   if(is.factor(test2[,k])){
#     test2[,k] <- as.character(test2[,k])
#   }
# }
# str(test2)
```

Sources

Bacoffe, A. and Silver, N., 2019. *The 2020 Endorsement Primary*. FiveThirtyEight. Retrieved January 13, 2020.

Pew Research Center. 2015. Building Pew Research Center's American Trends Panel, Technical Report, Washington D.C. Available at http://pewrsr.ch/1Jo4nKE.