Day 09: Data Wrangling Part 2

Erin Rossiter

27 March, 2023

Announcements

Announcements

- project update tomorrow
  - see slack
  - any questions
- PS08 posted tomorrow
  - will include some regex practice

Announcements

- project update tomorrow
  - see slack
  - any questions
- PS08 posted tomorrow
  - will include some regex practice

Announcements

- project update tomorrow
  - see slack
  - any questions
- PS08 posted tomorrow
  - will include some regex practice

Announcements

- project update tomorrow
  - see slack
  - any questions
- PS08 posted tomorrow
  - will include some regex practice

Announcements

- project update tomorrow
  - see slack
  - any questions
- PS08 posted tomorrow
  - will include some regex practice

Announcements

- project update tomorrow
  - see slack
  - any questions
- PS08 posted tomorrow
  - will include some regex practice

Last time

Base R apply family

Base R apply family

The basic structure is like this (from help file)

```
apply(X, MARGIN, FUN, ...)
```

- `X` is an "array," so usually **matrices** (a 2-dimensional array)
- `MARGIN` controls how the matrix is analyzed. Should the function be executed on each **row** (margin=1) or each **column** (margin=2)?
- `FUN` is the **function** you want done on each row/column/whatever.
  - *functions are objects, so they can be passed as arguments*
- `...` refers to any argument you want to pass onto `FUN`

Base R apply family

The basic structure is like this (from help file)

```
apply(X, MARGIN, FUN, ...)
```

- `X` is an "array," so usually **matrices** (a 2-dimensional array)
- `MARGIN` controls how the matrix is analyzed. Should the function be executed on each **row** (margin=1) or each **column** (margin=2)?
- `FUN` is the **function** you want done on each row/column/whatever.
  - *functions are objects, so they can be passed as arguments*
- `...` refers to any argument you want to pass onto `FUN`

Base R apply family

The basic structure is like this (from help file)

```
apply(X, MARGIN, FUN, ...)
```

- `X` is an "array," so usually **matrices** (a 2-dimensional array)
- `MARGIN` controls how the matrix is analyzed. Should the function be executed on each **row** (margin=1) or each **column** (margin=2)?
- `FUN` is the **function** you want done on each row/column/whatever.
  - *functions are objects, so they can be passed as arguments*
- `...` refers to any argument you want to pass onto `FUN`

Base R apply family

The basic structure is like this (from help file)

```
apply(X, MARGIN, FUN, ...)
```

- `X` is an "array," so usually **matrices** (a 2-dimensional array)
- `MARGIN` controls how the matrix is analyzed. Should the function be executed on each **row** (margin=1) or each **column** (margin=2)?
- `FUN` is the **function** you want done on each row/column/whatever.
  - *functions are objects, so they can be passed as arguments*
- `...` refers to any argument you want to pass onto `FUN`

Base R apply family

The basic structure is like this (from help file)

```
apply(X, MARGIN, FUN, ...)
```

- `X` is an "array," so usually **matrices** (a 2-dimensional array)
- `MARGIN` controls how the matrix is analyzed. Should the function be executed on each **row** (margin=1) or each **column** (margin=2)?
- `FUN` is the **function** you want done on each row/column/whatever.
  - *functions are objects, so they can be passed as arguments*
- `...` refers to any argument you want to pass onto `FUN`

Base R apply family

The basic structure is like this (from help file)

```
apply(X, MARGIN, FUN, ...)
```

- `X` is an "array," so usually **matrices** (a 2-dimensional array)
- `MARGIN` controls how the matrix is analyzed. Should the function be executed on each **row** (margin=1) or each **column** (margin=2)?
- `FUN` is the **function** you want done on each row/column/whatever.
  - *functions are objects, so they can be passed as arguments*
- `...` refers to any argument you want to pass onto `FUN`

Example

Thing to remember: you are passing a function (ex: `sum`) as an argument to the `apply` function.

```
ex_mat <- matrix(rep(1:3, 3), ncol = 3)
ex_mat
```

```
##      [,1] [,2] [,3]
## [1,]    1    1    1
## [2,]    2    2    2
## [3,]    3    3    3
```

```
# Sum each row
apply(ex_mat, 1, sum)
```

```
## [1] 3 6 9
```

```
# Sum each column
apply(ex_mat, 2, sum)
```

```
## [1] 6 6 6
```

Example

Thing to remember: you are passing a function (ex: `sum`) as an argument to the `apply` function.

```
ex_mat <- matrix(rep(1:3, 3), ncol = 3)
ex_mat
```

```
##      [,1] [,2] [,3]
## [1,]    1    1    1
## [2,]    2    2    2
## [3,]    3    3    3
```

```
# Sum each row
apply(ex_mat, 1, sum)
```

```
## [1] 3 6 9
```

```
# Sum each column
apply(ex_mat, 2, sum)
```

```
## [1] 6 6 6
```

plyr

The `plyr` package

- Consistent naming protocols for the functions to know what you are putting in and taking out.

`a_ply`, `aaply`, `adply`, `alply`, `d_ply`, `daply`, `ddply`, `dlply`, `l_ply`, `laply`, `llply`, `m_ply`, `maply`, `mdply`, `mlply`

The `plyr` package

- Consistent naming protocols for the functions to know what you are putting in and taking out.

`a_ply`, `aaply`, `adply`, `alply`, `d_ply`, `daply`, `ddply`, `dlply`, `l_ply`, `laply`, `llply`, `m_ply`, `maply`, `mdply`, `mlply`

The `plyr` package

- Consistent naming protocols for the functions to know what you are putting in and taking out.

`a_ply`, `aaply`, `adply`, `alply`, `d_ply`, `daply`, `ddply`, `dlply`, `l_ply`, `laply`, `llply`, `m_ply`, `maply`, `mdply`, `mlply`

Example

```r
library(plyr)
x <- list(1, 2, 3, 4, 5)
ldply(x, function(x) x +1)
```

```
##   V1
## 1  2
## 2  3
## 3  4
## 4  5
## 5  6
```

dplyr

Using `dplyr`

- Subset data by rows: `filter`
- Reorder by rows: `arrange`
- Subset data by column: `select`
- Create new variables as a function of other variables: `mutate`
- Collapse values down (or extract statistic): `summarise`
- We can use `group_by` to make changes in the scope

Using dplyr

- Subset data by rows: `filter`
- Reorder by rows: `arrange`
- Subset data by column: `select`
- Create new variables as a function of other variables: `mutate`
- Collapse values down (or extract statistic): `summarise`
- We can use `group_by` to make changes in the scope

Using dplyr

- Subset data by rows: `filter`
- Reorder by rows: `arrange`
- Subset data by column: `select`
- Create new variables as a function of other variables: `mutate`
- Collapse values down (or extract statistic): `summarise`
- We can use `group_by` to make changes in the scope

Using `dplyr`

- Subset data by rows: `filter`
- Reorder by rows: `arrange`
- Subset data by column: `select`
- Create new variables as a function of other variables: `mutate`
- Collapse values down (or extract statistic): `summarise`
- We can use `group_by` to make changes in the scope

Using `dplyr`

- Subset data by rows: `filter`
- Reorder by rows: `arrange`
- Subset data by column: `select`
- Create new variables as a function of other variables: `mutate`
- Collapse values down (or extract statistic): `summarise`
- We can use `group_by` to make changes in the scope

Using `dplyr`

- Subset data by rows: `filter`
- Reorder by rows: `arrange`
- Subset data by column: `select`
- Create new variables as a function of other variables: `mutate`
- Collapse values down (or extract statistic): `summarise`
- We can use `group_by` to make changes in the scope

Using `dplyr`

- Subset data by rows: `filter`
- Reorder by rows: `arrange`
- Subset data by column: `select`
- Create new variables as a function of other variables: `mutate`
- Collapse values down (or extract statistic): `summarise`
- We can use `group_by` to make changes in the scope

tidyr

Piping

- The tidyverse includes a nice syntax for combining multiple commands so we don't have to create new objects all of the time.
- The %>% syntax allows us to pass on the results of one line to another

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:plyr':
##
##     arrange, count, desc, failwith, id, mutate, rename, summa
##     summarize

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union

## Rows: 16661 Columns: 33
## -- Column specification ------------------------------------
## Delimiter: ","
```

Another example

```
basicPolls %>%
  group_by(candidate_name, state) %>%
  summarise(average_candidate = mean(pct), count = n()) %>%
  filter(count>10)
```

```
## # A tibble: 206 x 4
## # Groups:   candidate_name [44]
##    candidate_name state          average_candidate count
##    <chr>          <chr>                      <dbl> <int>
##  1 Amy Klobuchar  California                  1.74    40
##  2 Amy Klobuchar  Florida                     1.98    13
##  3 Amy Klobuchar  Iowa                        5.97    61
##  4 Amy Klobuchar  Nevada                      1.71    15
##  5 Amy Klobuchar  New Hampshire               5.42    84
##  6 Amy Klobuchar  Pennsylvania                1.53    11
##  7 Amy Klobuchar  South Carolina              1.27    34
##  8 Amy Klobuchar  Texas                       1.47    23
##  9 Amy Klobuchar  Wisconsin                   2.52    16
## 10 Amy Klobuchar  <NA>                        1.62   503
## # ... with 196 more rows
```

Another example

```
primaryPolls %>%
  group_by(candidate_name, state) %>%
  summarise(average_candidate = mean(pct), count = n()) %>%
  filter(count > 10) %>%
  mutate(average_prop = average_candidate/100) %>%
  select(average_prop, candidate_name, state, count)
```

```
## # A tibble: 206 x 4
## # Groups:   candidate_name [44]
##    average_prop candidate_name state          count
##           <dbl> <chr>          <chr>          <int>
##  1       0.0174 Amy Klobuchar  California        40
##  2       0.0198 Amy Klobuchar  Florida           13
##  3       0.0597 Amy Klobuchar  Iowa              61
##  4       0.0171 Amy Klobuchar  Nevada            15
##  5       0.0542 Amy Klobuchar  New Hampshire     84
##  6       0.0153 Amy Klobuchar  Pennsylvania      11
##  7       0.0127 Amy Klobuchar  South Carolina    34
##  8       0.0147 Amy Klobuchar  Texas             23
##  9       0.0252 Amy Klobuchar  Wisconsin         16
## 10       0.0162 Amy Klobuchar  <NA>             503
```

New today: pivots

Basic pivots

- In many cases the data is not quite organized the way we want.
- Right now we have each poll as a separate row. But what if we want each candidate to be a row so we can analyze their trends in polls over time?
- Or what if we get the trend line and want to instead reorganize to look at each poll separately?
- The key commands here are `pivot_wider` and `pivot_longer`

Examples in this section from here

Basic pivots

- In many cases the data is not quite organized the way we want.
- Right now we have each poll as a separate row. But what if we want each candidate to be a row so we can analyze their trends in polls over time?
- Or what if we get the trend line and want to instead reorganize to look at each poll separately?
- The key commands here are `pivot_wider` and `pivot_longer`

Examples in this section from here

Basic pivots

- In many cases the data is not quite organized the way we want.
- Right now we have each poll as a separate row. But what if we want each candidate to be a row so we can analyze their trends in polls over time?
- Or what if we get the trend line and want to instead reorganize to look at each poll separately?
- The key commands here are `pivot_wider` and `pivot_longer`

Examples in this section from here

Basic pivots

- In many cases the data is not quite organized the way we want.
- Right now we have each poll as a separate row. But what if we want each candidate to be a row so we can analyze their trends in polls over time?
- Or what if we get the trend line and want to instead reorganize to look at each poll separately?
- The key commands here are `pivot_wider` and `pivot_longer`

Examples in this section from here

Basic pivots

- In many cases the data is not quite organized the way we want.
- Right now we have each poll as a separate row. But what if we want each candidate to be a row so we can analyze their trends in polls over time?
- Or what if we get the trend line and want to instead reorganize to look at each poll separately?
- The key commands here are `pivot_wider` and `pivot_longer`

Examples in this section from here

Let's speak the same language

- Each **variable** is a column
- Each **observation** is a row
- Each **value** is a cell

Let's draw how this looks for our data

```
## # A tibble: 4 x 3
##   poll_id candidate_name         pct
##     <dbl> <chr>                <dbl>
## 1   63512 Bernard Sanders         28
## 2   63512 Andrew Yang              5
## 3   63512 Pete Buttigieg          26
## 4   63512 Joseph R. Biden Jr.      9
```

```
## # A tibble: 4 x 3
##   poll_id candidate_name         pct
##     <dbl> <chr>                <dbl>
## 1   63511 Bernard Sanders         20
## 2   63511 Joseph R. Biden Jr.     17
## 3   63511 Michael Bloomberg       15
## 4   63511 Elizabeth Warren        11
```

Let's draw

What should our data look like for these RQs:

- For each candidate, how has their polling changed over time in Nevada? (time series)
- Do poll characteristics (like who the polling agency is) correlated with who they find as leader in Nevada?

Example

```
nevadaPrimaries <- primaryPolls %>%
 filter(candidate_name %in% c("Amy Klobuchar", "Bernard Sanders"
                              "Elizabeth Warren", "Joseph R. Bid
                              "Michael Bloomberg", "Pete Buttigi
  filter(state == "Nevada") %>%
  select(poll_id, candidate_name, pct, start_date)
nevadaPrimaries
```

```
## # A tibble: 76 x 4
##    poll_id candidate_name       pct start_date
##      <dbl> <chr>              <dbl> <date>
##  1   63269 Joseph R. Biden Jr.  19.4 2020-01-08
##  2   63269 Bernard Sanders      17.6 2020-01-08
##  3   63269 Elizabeth Warren     10.6 2020-01-08
##  4   63269 Pete Buttigieg        8.2 2020-01-08
##  5   63269 Amy Klobuchar         3.6 2020-01-08
##  6   63254 Elizabeth Warren     14   2020-01-06
##  7   63254 Bernard Sanders      29   2020-01-06
##  8   63254 Joseph R. Biden Jr.  28   2020-01-06
##  9   63254 Pete Buttigieg        6   2020-01-06
## 10   63254 Amy Klobuchar         4   2020-01-06
```

Pivot wider

- Like our time series example

country	year	cases
Angola	1999	800
Angola	2000	750
Angola	2001	925
Angola	2002	1020
India	1999	20100
India	2000	25650
India	2001	26800
India	2002	27255
Mongolia	1999	450
Mongolia	2000	512
Mongolia	2001	510
Mongolia	2002	586

country	1999	2000	2001	2002
Angola	800	750	925	1020
India	20100	25650	26800	27255
Mongolia	450	512	510	586

Pivot data wider

```
data %>%
  pivot_wider(
    names_from = "year",
    values_from = "cases"
  )
```

Our example

```
wide_nv <- nevadaPrimaries %>%
  pivot_wider(
    id_cols = candidate_name,
    names_from = poll_id,
    values_from = pct)
# dropped unused by default
wide_nv
```

```
## # A tibble: 6 x 16
##   candidate_name `63269` `63254` `63245` `62629` `62643` `625
##   <chr>            <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <d
## 1 Joseph R. Bid~    19.4      28      23      24      33      2
## 2 Bernard Sande~    17.6      29      17      18      23      1
## 3 Elizabeth War~    10.6      14      12      18      21      2
## 4 Pete Buttigieg     8.2       6       6       8       9
## 5 Amy Klobuchar      3.6       4       2       2       2
## 6 Michael Bloom~    NA        NA       2      NA      NA      N
## # ... with 7 more variables: `59640` <dbl>, `59508` <dbl>, `5
## #   `58876` <dbl>, `58427` <dbl>, `58069` <dbl>, `57799` <dbl
```

Our example, but with dates as column names

Time series

```
wide_nv2 <- nevadaPrimaries %>%
  pivot_wider(
    id_cols = candidate_name,
    names_from = start_date,
    values_from = pct)
wide_nv2
```

```
## # A tibble: 6 x 16
##   candidate_name 2020-~1 2020-~2 2020-~3 2019-~4 2019-~5 2019
##   <chr>            <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <d
## 1 Joseph R. Bid~    19.4      28      23      24      33     2
## 2 Bernard Sande~    17.6      29      17      18      23     1
## 3 Elizabeth War~    10.6      14      12      18      21     2
## 4 Pete Buttigieg     8.2       6       6       8       9
## 5 Amy Klobuchar      3.6       4       2       2       2
## 6 Michael Bloom~    NA        NA       2      NA      NA     N
## # ... with 7 more variables: `2019-09-19` <dbl>, `2019-08-28`
## #   `2019-08-14` <dbl>, `2019-08-02` <dbl>, `2019-06-06` <dbl
## #   `2019-05-09` <dbl>, `2019-03-28` <dbl>, and abbreviated
```

Pivote longer

Pivote longer

country	1999	2000	2001	2002
Angola	800	750	925	1020
India	20100	25650	26800	27255
Mongolia	450	512	510	586

Pivot data longer

```
data %>%
  pivot_longer(
    cols = 1999:2002,
    names_to = "year",
    values_to = "cases"
  )
```

country	year	cases
Angola	1999	800
Angola	2000	750
Angola	2001	925
Angola	2002	1020
India	1999	20100
India	2000	25650
India	2001	26800
India	2002	27255
Mongolia	1999	450
Mongolia	2000	512
Mongolia	2001	510
Mongolia	2002	586

Our example

– Of course sometimes we want to do the reverse using `pivot_longer`

```r
long_nv <- wide_nv2 %>%
  # select two dates to demonstrate
  select(candidate_name, "2020-01-08", "2020-01-06") %>%
  pivot_longer(
    cols = c("2020-01-08", "2020-01-06"),
    names_to = "start_date",
    values_to = "pct")
long_nv
```

```
## # A tibble: 12 x 3
##    candidate_name      start_date   pct
##    <chr>               <chr>      <dbl>
##  1 Joseph R. Biden Jr. 2020-01-08  19.4
##  2 Joseph R. Biden Jr. 2020-01-06  28
##  3 Bernard Sanders     2020-01-08  17.6
##  4 Bernard Sanders     2020-01-06  29
##  5 Elizabeth Warren    2020-01-08  10.6
##  6 Elizabeth Warren    2020-01-06  14
##  7 Pete Buttigieg      2020-01-08   8.2
```