Data Wrangling with R

Claudia A Engel

Last updated: February 23, 2021

Contents

Prerequisites and Preparations

- You should have some **basic knowledge** of R, and be familiar with the topics covered in the Introduction to R.
- Have a recent version of R and RStudio installed.
- Install and load the `tidyverse` package.

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

- Create a new RStudio project `R-data-ws` in a new folder `R-data-ws`. Download both CSV files into a subdirectory called `data` like this:
- Download `MS_trafficstops_bw_age.csv`:

```
download.file("http://bit.ly/MS_trafficstops_bw_age",
              "data/MS_trafficstops_bw_age.csv")
```

- Download `MS_acs2015_bw.csv`:

```
download.file("http://bit.ly/MS_acs_2015_bw",
              "data/MS_acs2015_bw.csv")
```

References

Boehmke, Bradley C. (2016) Data Wrangling with R http://link.springer.com/book/10.1007%2F978-3-319-45599-0

Grolemund, G & Wickham, H (2017): R for Data Science http://r4ds.had.co.nz

Wickham, H. (2014): Tidy Data https://www.jstatsoft.org/article/view/v059i10

Acknowledgements

Part of the materials for this tutorial are adapted from http://datacarpentry.org and http://softwarecarpentry.org.

Chapter 1

Data Manipulation using `dplyr`

> Learning Objectives
>
> - Select columns in a data frame with the **`dplyr`** function `select`.
> - Select rows in a data frame according to filtering conditions with the **`dplyr`** function `filter`.
> - Direct the output of one **`dplyr`** function to the input of another function with the 'pipe' operator `%>%`.
> - Add new columns to a data frame that are functions of existing columns with `mutate`.
> - Understand the split-apply-combine concept for data analysis.
> - Use `summarize`, `group_by`, and `count` to split a data frame into groups of observations, apply a summary statistics for each group, and then combine the results.
> - Join two tables by a common variable.

Manipulation of data frames is a common task when you start exploring your data in R and **`dplyr`** is a package for making tabular data manipulation easier.

> Brief recap: Packages in R are sets of additional functions that let you do more stuff. Functions like `str()` or `data.frame()`, come built into R; packages give you access to more of them. Before you use a package for the first time you need to install it on your machine, and then you should import it in every subsequent R session when you need it.

If you haven't, please install the **`tidyverse`** package.

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

tidyverse is an "umbrella-package" that installs a series of packages useful for data analysis which work together well. Some of them are considered **core** packages (among them **tidyr**, **dplyr**, **ggplot2**), because you are likely to use them in almost every analysis. Other packages, like `lubridate` (to work wiht dates) or **haven** (for SPSS, Stata, and SAS data) that you are likely to use not for every analysis are also installed.

If you type the following command, it will load the **core** `tidyverse` packages.

```
library("tidyverse")    ## load the core tidyverse packages, incl. dplyr
```

If you need to use functions from `tidyverse` packages other than the core packages, you will need to load them separately.

1.1 What is dplyr?

dplyr is one part of a larger **tidyverse** that enables you to work with data in tidy data formats. "Tidy datasets are easy to manipulate, model and visualise, and have a specific structure: each variable is a column, each observation is a row, and each type of observational unit is a table." (From Wickham, H. (2014): Tidy Data https://www.jstatsoft.org/article/view/v059i10)

The package **dplyr** provides convenient tools for the most common data manipulation tasks. It is built to work directly with data frames, with many common tasks optimized by being written in a compiled language (C++). An additional feature is the ability to work directly with data stored in an external database. The benefits of doing this are that the data can be managed natively in a relational database, queries can be conducted on that database, and only the results of the query are returned.

This addresses a common problem with R in that all operations are conducted in-memory and thus the amount of data you can work with is limited by available memory. The database connections essentially remove that limitation in that you can have a database of many 100s GB, conduct queries on it directly, and pull back into R only what you need for analysis.

To learn more about **dplyr** after the workshop, you may want to check out the handy data transformation with **dplyr** cheatsheet.

1.2 Subsetting columns and rows

Let's begin with loading our sample data into a data frame.

We will be working a small subset of the data from the Stanford Open Policing Project. It contains information about traffic stops for blacks and whites in the state of Mississippi during January 2013 to mid-July of 2016.

```
stops <- read_csv("data/MS_trafficstops_bw_age.csv")
```

```
#> Parsed with column specification:
#> cols(
#>   id = col_character(),
#>   stop_date = col_date(format = ""),
#>   county_name = col_character(),
#>   county_fips = col_double(),
#>   police_department = col_character(),
#>   driver_gender = col_character(),
#>   driver_birthdate = col_date(format = ""),
#>   driver_race = col_character(),
#>   officer_id = col_character(),
#>   driver_age = col_double(),
#>   violation = col_character()
#> )
```

```
stops
```

```
#> # A tibble: 211,211 x 11
#>    id    stop_date  county_name county_fips police_departme~ driver_gender
#>    <chr> <date>     <chr>             <dbl> <chr>            <chr>
#>  1 MS-2~ 2013-01-01 Jones             28067 Mississippi Hig~ male
#>  2 MS-2~ 2013-01-01 Lauderdale        28075 Mississippi Hig~ male
#>  3 MS-2~ 2013-01-01 Pike              28113 Mississippi Hig~ male
#>  4 MS-2~ 2013-01-01 Hancock           28045 Mississippi Hig~ male
#>  5 MS-2~ 2013-01-01 Holmes            28051 Mississippi Hig~ male
#>  6 MS-2~ 2013-01-01 Jackson           28059 Mississippi Hig~ female
#>  7 MS-2~ 2013-01-01 Jackson           28059 Mississippi Hig~ female
#>  8 MS-2~ 2013-01-01 Grenada           28043 Mississippi Hig~ female
#>  9 MS-2~ 2013-01-01 Holmes            28051 Mississippi Hig~ male
#> 10 MS-2~ 2013-01-01 Holmes            28051 Mississippi Hig~ male
#> # ... with 211,201 more rows, and 5 more variables: driver_birthdate <date>,
#> #   driver_race <chr>, officer_id <chr>, driver_age <dbl>, violation <chr>
```

You may have noticed that by using `read_csv` we have generated an object of class `tbl_df`, also known as a "tibble". Tibble's data structure is very similar to a data frame. For our purposes the only differences are that

- (1) columns of class `character` are never converted into factors[1],
- (2) it tries to recognize and `date` types
- (3) the output displays the data type of each column under its name, and

[1] This is now also true for the base `read.csv` starting with R version 4.

- (4) it only prints the first few rows of data and only as many columns as fit on one screen. If we wanted to print all columns we can use the print command, and set the `width` parameter to `Inf`. To print the first 6 rows for example we would do this: `print(my_tibble, n=6, width=Inf)`.

To select columns of a data frame with `dplyr`, use `select()`. The first argument to this function is the data frame (`stops`), and the subsequent arguments are the columns to keep.

```
select(stops, police_department, officer_id, driver_race)
```

```
#> # A tibble: 211,211 x 3
#>    police_department          officer_id driver_race
#>    <chr>                      <chr>      <chr>
#>  1 Mississippi Highway Patrol J042       Black
#>  2 Mississippi Highway Patrol B026       Black
#>  3 Mississippi Highway Patrol M009       Black
#>  4 Mississippi Highway Patrol K035       White
#>  5 Mississippi Highway Patrol D028       White
#>  6 Mississippi Highway Patrol K023       White
#>  7 Mississippi Highway Patrol K032       White
#>  8 Mississippi Highway Patrol D021       White
#>  9 Mississippi Highway Patrol R021       White
#> 10 Mississippi Highway Patrol R021       White
#> # ... with 211,201 more rows
```

It is worth knowing that `dplyr` is backed by another package with a number of helper functions, which provide convenient functions to select columns based on their names. For example:

```
select(stops, starts_with("driver"))
```

```
#> # A tibble: 211,211 x 4
#>    driver_gender driver_birthdate driver_race driver_age
#>    <chr>         <date>           <chr>            <dbl>
#>  1 male          1950-06-14       Black               63
#>  2 male          1967-04-06       Black               46
#>  3 male          1974-04-15       Black               39
#>  4 male          1981-03-23       White               32
#>  5 male          1992-08-03       White               20
#>  6 female        1960-05-02       White               53
#>  7 female        1953-03-16       White               60
#>  8 female        1993-06-14       White               20
#>  9 male          1947-12-11       White               65
#> 10 male          1984-07-14       White               28
#> # ... with 211,201 more rows
```

Check out the tidyselect reference for more.

To subset rows based on specific criteria, we use `filter()`:

```
filter(stops, county_name == "Yazoo")
```

```
#> # A tibble: 3,528 x 11
#>    id    stop_date  county_name county_fips police_departme~ driver_gender
#>    <chr> <date>     <chr>             <dbl> <chr>            <chr>
#>  1 MS-2~ 2013-01-02 Yazoo             28163 Mississippi Hig~ male
#>  2 MS-2~ 2013-01-02 Yazoo             28163 Mississippi Hig~ female
#>  3 MS-2~ 2013-01-02 Yazoo             28163 Mississippi Hig~ male
#>  4 MS-2~ 2013-01-02 Yazoo             28163 Mississippi Hig~ female
#>  5 MS-2~ 2013-01-02 Yazoo             28163 Mississippi Hig~ male
#>  6 MS-2~ 2013-01-03 Yazoo             28163 Mississippi Hig~ male
#>  7 MS-2~ 2013-01-03 Yazoo             28163 Mississippi Hig~ male
#>  8 MS-2~ 2013-01-04 Yazoo             28163 Mississippi Hig~ male
#>  9 MS-2~ 2013-01-04 Yazoo             28163 Mississippi Hig~ male
#> 10 MS-2~ 2013-01-04 Yazoo             28163 Mississippi Hig~ female
#> # ... with 3,518 more rows, and 5 more variables: driver_birthdate <date>,
#> #   driver_race <chr>, officer_id <chr>, driver_age <dbl>, violation <chr>
```

Here are some other ways to subset rows:

- by row number: `slice(stops, 1:3) # rows 1-3`
- rows with highest or lowest values of a variable:
  - `slice_min(stops, driver_age) # likewise slice_max()`
- random rows:
  - `slice_sample(stops, n = 5) # number of rows to select`
  - `slice_sample(stops, prop = .0001) # fraction of rows to select`

To sort rows by variables use the `arrange` function:

```
arrange(stops, county_name, stop_date)
```

```
#> # A tibble: 211,211 x 11
#>    id    stop_date  county_name county_fips police_departme~ driver_gender
#>    <chr> <date>     <chr>             <dbl> <chr>            <chr>
#>  1 MS-2~ 2013-02-09 Adams             28001 Mississippi Hig~ male
#>  2 MS-2~ 2013-03-02 Adams             28001 Mississippi Hig~ female
#>  3 MS-2~ 2013-03-16 Adams             28001 Mississippi Hig~ female
#>  4 MS-2~ 2013-03-20 Adams             28001 Mississippi Hig~ female
#>  5 MS-2~ 2013-04-06 Adams             28001 Mississippi Hig~ female
#>  6 MS-2~ 2013-04-13 Adams             28001 Mississippi Hig~ female
#>  7 MS-2~ 2013-04-19 Adams             28001 Mississippi Hig~ female
#>  8 MS-2~ 2013-04-21 Adams             28001 Mississippi Hig~ female
#>  9 MS-2~ 2013-04-24 Adams             28001 Mississippi Hig~ male
#> 10 MS-2~ 2013-04-24 Adams             28001 Mississippi Hig~ male
```

```
#> # ... with 211,201 more rows, and 5 more variables: driver_birthdate <date>,
#> #   driver_race <chr>, officer_id <chr>, driver_age <dbl>, violation <chr>
```

1.3 Pipes

What if you wanted to filter **and** select on the same data? For example, lets find drivers over 85 years and only keep the violation and gender columns. There are three ways to do this: use intermediate steps, nested functions, or pipes.

- Intermediate steps:

With intermediate steps, you essentially create a temporary data frame and use that as input to the next function. This can clutter up your workspace with lots of objects.

```
tmp_df <- filter(stops, driver_age > 85)
select(tmp_df, violation, driver_gender)
```

- Nested functions

You can also nest functions (i.e. placce one function inside of another). This is handy, but can be difficult to read if too many functions are nested as things are evaluated from the inside out.

```
select(filter(stops, driver_age > 85), violation, driver_gender)
```

- Pipes!

The last option, called "pipes". Pipes let you take the output of one function and send it directly to the next, which is useful when you need to do many things to the same dataset. Pipes in R look like `%>%` and are made available via the `magrittr` package, which is installed automatically with **`dplyr`**. If you use RStudio, you can type the pipe with Ctrl + Shift + M if you have a PC or Cmd + Shift + M if you have a Mac.

```
stops %>%
  filter(driver_age > 85) %>%
  select(violation, driver_gender)
```

In the above, we use the pipe to send the `stops` data first through `filter()` to keep rows where `driver_race` is Black, then through `select()` to keep only the `officer_id` and `stop_date` columns. Since `%>%` takes the object on its left and passes it as the first argument to the function on its right, we don't need to explicitly include it as an argument to the `filter()` and `select()` functions anymore.

If we wanted to create a new object with this smaller version of the data, we could do so by assigning it a new name:

```
senior_drivers <- stops %>%
  filter(driver_age > 85) %>%
  select(violation, driver_gender, driver_race)

senior_drivers
```

```
#> # A tibble: 3 x 3
#>   violation driver_gender driver_race
#>   <chr>     <chr>         <chr>
#> 1 Seat belt male          White
#> 2 Speeding  male          White
#> 3 Seat belt male          Black
```

Note that the final data frame is the leftmost part of this expression.

> Challenge
>
> Using pipes, subset the `stops` data to include stops in Tunica county only and retain the columns `stop_date`, `driver_age`, and `violation`. Bonus: sort the table by driver age.

1.4 Add new columns

Frequently you'll want to create new columns based on the values in existing columns or. For this we'll use `mutate()`. We can also reassign values to an existing column with that function.

Be aware that new and edited columns will not permanently be added to the existing data frame – unless we explicitly save the output.

So here is an example using the `year()` function from the lubridate package to extract the year of the drivers' birthdate:

```
library(lubridate)

stops %>%
  mutate(birth_year = year(driver_birthdate))
```

We can keep adding columns like this:

```
stops %>%
  mutate(birth_year = year(driver_birthdate),
         birth_cohort = floor(birth_year/10)*10)
```

We are beginning to see the power of piping. Here is a slightly expanded example, where we select the column `birth_cohort` that we have created and send it to plot:

```
stops %>%
  mutate(birth_year = year(driver_birthdate),
         birth_cohort = floor(birth_year/10)*10,
         birth_cohort = factor(birth_cohort)) %>%
    select(birth_cohort) %>%
    plot()
```

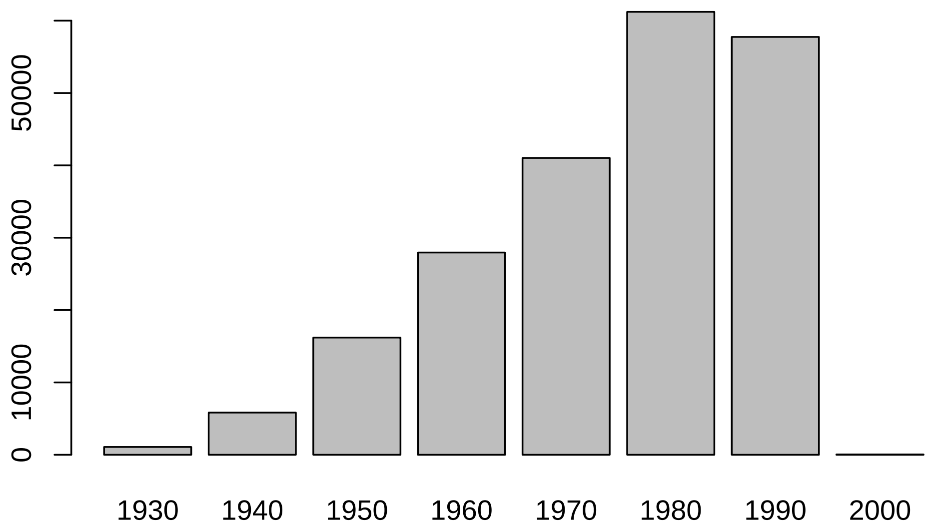

Figure 1.1: Driver Birth Cohorts

Mutate can also be used in conjunction with logical conditions. For example, we could create a new column, where we assign everyone born after the year 2000 to a group “millenial” and overyone before to “pre-millenial”.

In order to do this we take advantage of the `ifelse` function:

`ifelse(a_logical_condition, if_true_return_this, if_false_return_this)`

In conjunction with mutate, this works like this:

```
stops %>%
  mutate(cohort = ifelse(year(driver_birthdate) < 2000, "pre-millenial", "millenial"))
  select(driver_birthdate, cohort)
```

```
#> # A tibble: 211,211 x 2
#>    driver_birthdate cohort
#>    <date>           <chr>
#>  1 1950-06-14       pre-millenial
#>  2 1967-04-06       pre-millenial
#>  3 1974-04-15       pre-millenial
#>  4 1981-03-23       pre-millenial
```

```
#>  5 1992-08-03       pre-millenial
#>  6 1960-05-02       pre-millenial
#>  7 1953-03-16       pre-millenial
#>  8 1993-06-14       pre-millenial
#>  9 1947-12-11       pre-millenial
#> 10 1984-07-14       pre-millenial
#> # ... with 211,201 more rows
```

More advanced conditional recoding can be done with `case_when()`.

> Challenge
>
> Create a new data frame from the `stops` data that meets the following criteria: contains only the `violation` column for female drivers of age 50 that were stopped on a Sunday. For this add a new column to your data frame called `weekday_of_stop` containing the number of the weekday when the stop occurred. Use the `wday()` function from `lubridate` (Sunday = 1).
>
> Think about how the commands should be ordered to produce this data frame!

1.5 What is split-apply-combine?

Many data analysis tasks can be approached using the *split-apply-combine* paradigm: split the data into groups, apply some analysis to each group, and then combine the results.

`dplyr` makes this possible through the use of the `group_by()` function.

`group_by()` is often used together with `summarize()`, which collapses each group into a single-row summary of that group. `group_by()` takes as arguments the column names that contain the **categorical** variables for which you want to calculate the summary statistics. So to view the mean age for black and white drivers:

```
stops %>%
  group_by(driver_race) %>%
  summarize(mean_age = mean(driver_age, na.rm=TRUE))
```

```
#> `summarise()` ungrouping output (override with `.groups` argument)
```

```
#> # A tibble: 3 x 2
#>   driver_race mean_age
#>   <chr>          <dbl>
#> 1 Black           34.2
#> 2 White           36.2
#> 3 <NA>            34.5
```

If we wanted to remove the line with `NA` we could insert a `filter()` in the chain:

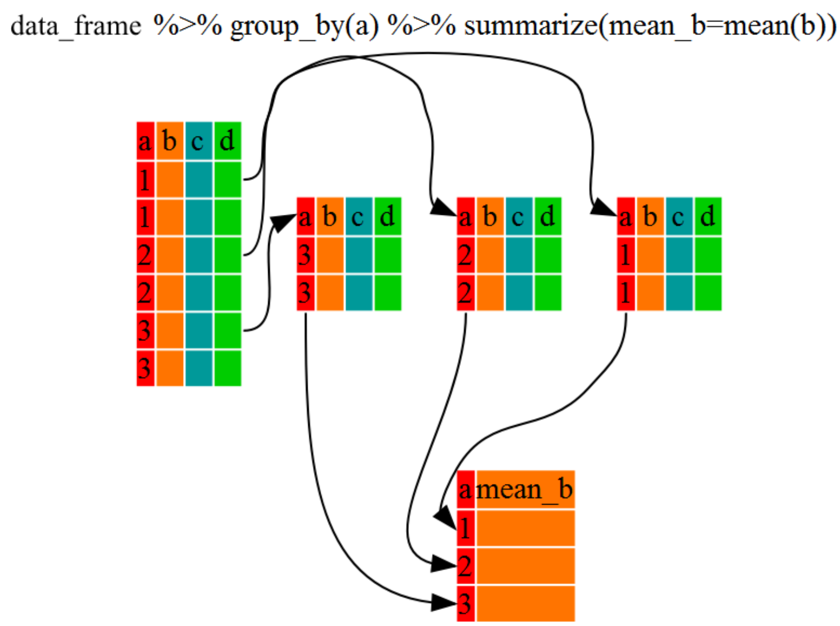

Figure 1.2: Split - Apply - Combine

```
stops %>%
  filter(!is.na(driver_race)) %>%
  group_by(driver_race) %>%
  summarize(mean_age = mean(driver_age, na.rm=TRUE))
```

```
#> `summarise()` ungrouping output (override with `.groups` argument)
```

```
#> # A tibble: 2 x 2
#>   driver_race mean_age
#>   <chr>          <dbl>
#> 1 Black           34.2
#> 2 White           36.2
```

Recall that `is.na()` is a function that determines whether something is an `NA`. The `!` symbol negates the result, so we're asking for everything that is *not* an `NA`.

You can also group by multiple columns:

```
stops %>%
  filter(!is.na(driver_race)) %>%
  group_by(county_name, driver_race) %>%
  summarize(mean_age = mean(driver_age, na.rm=TRUE))
```

```
#> `summarise()` regrouping output by 'county_name' (override with `.groups` argument)
```

```
#> # A tibble: 163 x 3
#> # Groups:   county_name [82]
#>    county_name driver_race mean_age
#>    <chr>       <chr>          <dbl>
#>  1 Adams       Black           36.2
#>  2 Adams       White           40.0
#>  3 Alcorn      Black           34.6
#>  4 Alcorn      White           33.6
#>  5 Amite       Black           37.5
#>  6 Amite       White           42.1
#>  7 Attala      Black           36.4
#>  8 Attala      White           38.6
#>  9 Benton      Black           34.7
#> 10 Benton      White           32.0
#> # ... with 153 more rows
```

Once the data are grouped, you can also summarize multiple variables at the same time (and not necessarily on the same variable). For instance, we could add a column indicating the minimum age in each group (i.e. county):

```
stops %>%
  filter(!is.na(driver_race)) %>%
  group_by(county_name, driver_race) %>%
```

```
  summarize(mean_age = mean(driver_age, na.rm=TRUE),
            min_age = min(driver_age, na.rm=TRUE))
```

```
#> `summarise()` regrouping output by 'county_name' (override with `.groups` argument)
```

```
#> # A tibble: 163 x 4
#> # Groups:   county_name [82]
#>    county_name driver_race mean_age min_age
#>    <chr>       <chr>          <dbl>   <dbl>
#>  1 Adams       Black           36.2      16
#>  2 Adams       White           40.0      16
#>  3 Alcorn      Black           34.6      17
#>  4 Alcorn      White           33.6      15
#>  5 Amite       Black           37.5      17
#>  6 Amite       White           42.1      15
#>  7 Attala      Black           36.4       8
#>  8 Attala      White           38.6      15
#>  9 Benton      Black           34.7      18
#> 10 Benton      White           32.0      18
#> # ... with 153 more rows
```

1.6 Tallying

When working with data, it is also common to want to know the number of observations found for categorical variables. For this, `dplyr` provides `count()`. For example, if we wanted to see how many traffic stops each officer recorded:

```
stops %>%
  count(officer_id)
```

Bu default, count will name the column with the counts `n`. We can change this by explicitly providing a value for the `name` argument:

```
stops %>%
  count(officer_id, name = "n_stops")
```

We can optionally sort the results in descending order by adding `sort=TRUE`:

```
stops %>%
  count(officer_id, name = "n_stops", sort = TRUE)
```

`count()` calls `group_by()` transparently before counting the total number of records for each category. Similarly, we can count subgroups within groups:

```
stops %>%
  count(officer_id, violation, name = "n_stops")
```

Alternatives:

```
stops %>%
  group_by(officer_id) %>%
  tally(sort = TRUE) # tally() requires group_by before counting

stops %>%
  group_by(officer_id) %>%
  summarize(n = n()) %>% # n() is useful when the count is needed within a calculation
  arrange(desc(n))
```

Challenge

Which 5 counties were the ones with the most stops in 2013? Hint: use the year() function from lubridate.

1.7 Joining two tables

It is not uncommon that we have our data spread out in different tables and need to bring those together for analysis. In this example we will combine the numbers of stops for black and white drivers per county together with the numbers of the black and white total population for these counties. The population data are the estimated values of the 5 year average from the 2011-2015 American Community Survey (ACS):

```
acs <- read_csv("data/MS_acs2015_bw.csv")
```

```
#> Parsed with column specification:
#> cols(
#>   County = col_character(),
#>   FIPS = col_double(),
#>   black_pop = col_double(),
#>   white_pop = col_double(),
#>   bw_pop = col_double()
#> )
```

```
acs
```

```
#> # A tibble: 82 x 5
#>    County      FIPS black_pop white_pop bw_pop
#>    <chr>      <dbl>     <dbl>     <dbl>  <dbl>
#>  1 Jones      28067     19711     47154  66865
#>  2 Lauderdale 28075     33893     43482  77375
#>  3 Pike       28113     21028     18282  39310
#>  4 Hancock    28045      4172     39686  43858
#>  5 Holmes     28051     15498      3105  18603
#>  6 Jackson    28059     30704    101686 132390
#>  7 Grenada    28043      9417     11991  21408
#>  8 Scott      28123     10562     16920  27482
```

```
#>  9 Wayne      28153      8015     12154  20169
#> 10 Bolivar    28011     21648     11197  32845
#> # ... with 72 more rows
```

In a first step we count all the stops per county.

```
stops %>%
  count(county_name, name = "n_stops")
```

```
#> # A tibble: 82 x 2
#>    county_name n_stops
#>    <chr>         <int>
#>  1 Adams           942
#>  2 Alcorn         3345
#>  3 Amite          2921
#>  4 Attala         4203
#>  5 Benton          214
#>  6 Bolivar        4526
#>  7 Calhoun        1658
#>  8 Carroll        1788
#>  9 Chickasaw      3869
#> 10 Choctaw         613
#> # ... with 72 more rows
```

We will then pipe this into our next operation where we bring the two tables together. We will use `left_join`, which returns all rows from the left table, and all columns from the left and the right table. As ID, which uniquely identifies the corresponding records in each table we use the County names.

```
stops %>%
  count(county_name, name = "n_stops") %>%
  left_join(acs, by = c("county_name" = "County"))
```

```
#> # A tibble: 82 x 6
#>    county_name n_stops  FIPS black_pop white_pop bw_pop
#>    <chr>         <int> <dbl>     <dbl>     <dbl>  <dbl>
#>  1 Adams           942 28001     17757     12856  30613
#>  2 Alcorn         3345 28003      4281     31563  35844
#>  3 Amite          2921 28005      5416      7395  12811
#>  4 Attala         4203 28007      8194     10649  18843
#>  5 Benton          214 28009      3078      5166   8244
#>  6 Bolivar        4526 28011     21648     11197  32845
#>  7 Calhoun        1658 28013      3991     10103  14094
#>  8 Carroll        1788 28015      3470      6702  10172
#>  9 Chickasaw      3869 28017      7549      9522  17071
#> 10 Choctaw         613 28019      2596      5661   8257
#> # ... with 72 more rows
```

Now we can, for example calculate the stop rate, i.e. the number of stops per population in each county.

> Challenge
>
> Which county has the highest and which one the lowest stop rate? Use the snippet from above and pipe into the additional operations to do this.

`dplyr` join functions are generally equivalent to `merge` from the base command, but there are a few advantages:

- rows are kept in existing order
- it runs faster
- tells you what keys you're merging by (if you don't supply them)
- also works with database tables.

https://groups.google.com/d/msg/manipulatr/OuAPC4VyfIc/Qnt8mDfq0WwJ

See `?dplyr::join` for all the possible joins.

Chapter 2

Data Manipulation using `tidyr`

Learning Objectives

- Understand the concept of a wide and a long table format and for which purpose those formats are useful.
- Understand what key-value pairs are.
- Reshape a data frame from long to wide format and back with the `pivot_wider` and `pivot_longer` commands from the **`tidyr`** package.
- Export a data frame to a .csv file.

`dplyr` pairs nicely with **`tidyr`** which enables you to swiftly convert between different data formats for plotting and analysis.

The package **`tidyr`** addresses the common problem of wanting to reshape your data for plotting and use by different R functions. Sometimes we want data sets where we have one row per observation. Sometimes we want a data frame where each observation type has its own column, and rows are instead more aggregated groups - like surveys, where each column represents an answer. Moving back and forth between these formats is nontrivial, and **`tidyr`** gives you tools for this and more sophisticated data manipulation.

To learn more about **`tidyr`** after the workshop, you may want to check out this cheatsheet about **`tidyr`**.

2.1 About long and wide table format

The 'long' format is where:

- each column is a variable
- each row is an observation

In the 'long' format, you usually have 1 column for the observed variable and the other columns are ID variables.

For the 'wide' format a row, for example could be a reserarch subject for which you have multiple observation variables containing the same type of data, for example responses to a set of survey questions, or repeated observations over time, or a mix of both. Here is an example:

	subject_ID	question_1	question_2	question_3
1	A	4.00	3.00	4.00
2	B	4.00	1.00	5.00
3	C	2.00	5.00	2.00

You may find data input may be simpler or some other applications may prefer the 'wide' format. However, many of `R`'s functions have been designed assuming you have 'long' format data. This tutorial will help you efficiently transform your data regardless of original format.

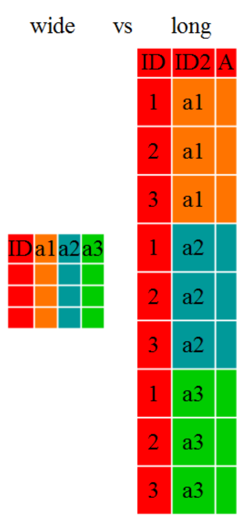

Figure 2.1: Wide vs. Long Table Format

The choice of data format affects readability. For humans, the wide format is often more intuitive, since we can often see more of the data on the screen due to its shape. However, the long format is more machine readable and is closer to the formatting of databases. The `ID` variables in our dataframes are similar to the fields in a database and observed variables are like the database values.

Challenge 1

Is stops in a long or wide format?

2.2 Long to Wide with pivot_wider

Now let's see this in action. First, using **dplyr**, let's create a data frame with the counts of different violations for each county:

```
violations <- stops %>%
  count(county_name, violation)

violations
```

```
#>   county_name                violation   n
#> 1       Adams        Breaks-Lights-etc   7
#> 2       Adams         Careless driving  48
#> 3       Adams License-Permit-Insurance 118
#> 4       Adams         Other or unknown  35
#> 5       Adams                Seat belt 229
#> 6       Adams                 Speeding 505
```

Now, to make this long data wide, we use `pivot_wider` from `tidyr` to turn the driver gender into columns. In addition to our data table we provide `pivot_wider` with two arguments: `names_from` describes which column to use for name of the output column, and `values_from` tells it from column to get the cell values. We'll use a pipe so we can ignore the data argument.

```
violations_wide <- violations %>%
  pivot_wider(names_from = violation,
              values_from = n)

violations_wide
```

```
#> # A tibble: 82 x 7
#>    county_name `Breaks-Lights-~ `Careless drivi~ `License-Permit~
#>    <chr>                  <int>            <int>            <int>
#>  1 Adams                      7               48              118
#>  2 Alcorn                    62              100              737
#>  3 Amite                     47               86              370
#>  4 Attala                    99              113              526
#>  5 Benton                     3                9               73
#>  6 Bolivar                   57              139             1034
#>  7 Calhoun                   26               38              383
#>  8 Carroll                   26               40              323
#>  9 Chickasaw                 42               53             1378
#> 10 Choctaw                    8                6               73
#> # ... with 72 more rows, and 3 more variables: `Other or unknown` <int>, `Seat
```

```
#> #   belt` <int>, Speeding <int>
```

2.3 Wide to long with pivot_longer

What if we had the opposite problem, and wanted to go from a wide to long format? For that, we use `pivot_longer`, which will increase the number of rows and decrease the number of columns. We provide the functino with thee arguments: `cols` which are the columns we want to pivot into the long format, `names_to`, which is a string specifying the name of the column to create from the data stored in the column names, and `values_to`, which is also a string, specifying the name of the column to create from the data stored in cell values. So, to go backwards from `violations_wide`, and exclude `county_name` from the long format, we would do the following:

```
violations_long <- violations_wide %>%
  pivot_longer(cols = -county_name,        # exclude column with county name
               names_to = "violation",      # name is a string!
               values_to = "n")             # also a string

violations_long
```

```
#> # A tibble: 492 x 3
#>    county_name violation                 n
#>    <chr>       <chr>                 <int>
#>  1 Adams       Breaks-Lights-etc         7
#>  2 Adams       Careless driving         48
#>  3 Adams       License-Permit-Insurance   118
#>  4 Adams       Other or unknown         35
#>  5 Adams       Seat belt               229
#>  6 Adams       Speeding                505
#>  7 Alcorn      Breaks-Lights-etc        62
#>  8 Alcorn      Careless driving        100
#>  9 Alcorn      License-Permit-Insurance   737
#> 10 Alcorn      Other or unknown        418
#> # ... with 482 more rows
```

We could also have used a specification for what columns to include. This can be useful if you have a large number of identifying columns, and it's easier to specify what to gather than what to leave alone. And if the columns are adjacent to each other, we don't even need to list them all out – we can use the `:` operator!

```
violations_wide %>%
  pivot_longer(cols = `Breaks-Lights-etc`:Speeding,      # this also works
               names_to = "violation",
               values_to = "n")
```

```
#> # A tibble: 492 x 3
#>    county_name violation                    n
#>    <chr>       <chr>                    <int>
#>  1 Adams       Breaks-Lights-etc            7
#>  2 Adams       Careless driving            48
#>  3 Adams       License-Permit-Insurance   118
#>  4 Adams       Other or unknown            35
#>  5 Adams       Seat belt                  229
#>  6 Adams       Speeding                   505
#>  7 Alcorn      Breaks-Lights-etc           62
#>  8 Alcorn      Careless driving           100
#>  9 Alcorn      License-Permit-Insurance   737
#> 10 Alcorn      Other or unknown           418
#> # ... with 482 more rows
```

There are many powerful operations you can do with the `pivot_*` functions. To learn more review the vignette:

```
vignette("pivot")
```

> Challenge
>
> 1.From the stops dataframe create a wide data frame `tr_wide` with "year" as columns, each row is a different violation, and the values are the number of traffic stops per each violation, roughly like this:
>
> ```
> violation      | 2013 | 2014 | 2015 ... Break-Lights   |
> 65 |     54|    67 ... Speeding        |  713 |    948|  978
> ... ...
> ```
>
> Use `year()` from the lubridate package. Hint: You will need to summarize and count the traffic stops before reshaping the table.
>
> 2. Now take the data frame, and make it long again, so each row is a unique violation - year combination, like this:
>
> ```
> violation   | year | n of stops    Speeding   | 2013 | 65
> Speeding    | 2014 | 54 ... etc
> ```

2.4 Exporting data

Similar to the `read_csv()` function used for reading CSV files into R, there is a `write_csv()` function that generates CSV files from data frames.

Before using `write_csv()`, we are going to create a new folder, `data_output`, in our working directory that will store this generated dataset. We don't want to write generated datasets in the same directory as our raw data. It's good practice to keep them separate. The `data` folder should only contain the raw, unaltered data, and should be left alone to make sure we don't delete or modify it.

In contrast, our script will generate the contents of the `data_output` directory, so even if the files it contains are deleted, we can always re-generate them.

We can now save the table generated above in our `data_output` folder:

```
write_csv(violation_wide, "data_output/county_violations.csv")
```