

## Chapter 4 The tidyverse

Up to now we have been manipulating vectors by reordering and subsetting them through indexing. However, once we start more advanced analyses, the preferred unit for data storage is not the vector but the data frame. In this chapter we learn to work directly with data frames, which greatly facilitate the organization of information. We will be using data frames for the majority of this book. We will focus on a specific data format referred to as tidy and on specific collection of packages that are particularly helpful for working with tidy data referred to as the tidyverse.

We can load all the tidyverse packages at once by installing and loading the tidyverse package:

```
library(tidyverse)

## -- Attaching packages ----- tidyverse 1.3.1 --

## v ggplot2 3.3.5      v purrr   0.3.4
## v tibble  3.1.4      v dplyr  1.0.7
## v tidyr   1.1.3      v stringr 1.4.0
## v readr   2.0.1      v forcats 0.5.1

## -- Conflicts ----- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

We will learn how to implement the tidyverse approach throughout the book, but before delving into the details, in this chapter we introduce some of the most widely used tidyverse functionality, starting with the dplyr package for manipulating data frames and the purrr package for working with functions. Note that the tidyverse also includes a graphing package, ggplot2, which we introduce later in Chapter 7 in the Data Visualization part of the book; the readr package discussed in Chapter 5; and many others. In this chapter, we first introduce the concept of tidy data and then demonstrate how we use the tidyverse to work with data frames in this format.

### 4.1 Tidy data

We say that a data table is in tidy format if each row represents one observation and columns represent the different variables available for each of these observations. The murders dataset is an example of a tidy data frame.

```
library(dslabs)
head(murders)

##      state abb region population total
## 1  Alabama  AL  South   4779736   135
## 2   Alaska  AK   West    710231    19
## 3  Arizona  AZ   West   6392017   232
## 4 Arkansas  AR  South   2915918    93
## 5 California CA   West  37253956  1257
## 6  Colorado CO   West   5029196    65
```

Each row represent a state with each of the five columns providing a different variable related to these states: name, abbreviation, region, population, and total murders.

To see how the same information can be provided in different formats, consider the following example:

This tidy dataset provides fertility rates for two countries across the years. This is a tidy dataset because each row presents one observation with the three variables being country, year, and fertility rate. However, this dataset originally came in another format and was reshaped for the dslabs package. Originally, the data was in the following format:

The same information is provided, but there are two important differences in the format: 1) each row includes several observations and 2) one of the variables, year, is stored in the header. For the tidyverse packages to be optimally used, data need to be reshaped into tidy format, which you will learn to do in the Data Wrangling part of the book. Until then, we will use example datasets that are already in tidy format.

Although not immediately obvious, as you go through the book you will start to appreciate the advantages of working in a framework in which functions use tidy formats for both inputs and outputs. You will see how this permits the data analyst to focus on more important aspects of the analysis rather than the format of the data.

## 4.2 Exercises

1. Examine the built-in dataset `co2`. Which of the following is true:

- a. `co2` is tidy data: it has one year for each row.
- b. `co2` is not tidy: we need at least one column with a character vector.
- c. `co2` is not tidy: it is a matrix instead of a data frame.
- d. `co2` is not tidy: to be tidy we would have to wrangle it to have three columns (year, month and value), then each `co2` observation would have a row.

-> answer: d

2. Examine the built-in dataset `ChickWeight`. Which of the following is true:

```
head(ChickWeight)
```

```
##   weight Time Chick Diet
## 1     42    0     1    1
## 2     51    2     1    1
## 3     59    4     1    1
## 4     64    6     1    1
## 5     76    8     1    1
## 6     93   10     1    1
```

- a. `ChickWeight` is not tidy: each chick has more than one row.
- b. `ChickWeight` is tidy: each observation (a weight) is represented by one row. The chick from which this measurement came is one of the variables.
- c. `ChickWeight` is not tidy: we are missing the year column.
- d. `ChickWeight` is tidy: it is stored in a data frame.

-> answer: b

3. Examine the built-in dataset `BOD`. Which of the following is true:

```
head(BOD)
```

```
##   Time demand
## 1     1     8.3
## 2     2    10.3
## 3     3    19.0
## 4     4    16.0
## 5     5    15.6
## 6     7    19.8
```

- a. BOD is not tidy: it only has six rows.
- b. BOD is not tidy: the first column is just an index.
- c. BOD is tidy: each row is an observation with two values (time and demand)
- d. BOD is tidy: all small datasets are tidy by definition.

-> answer: c

4. Which of the following built-in datasets is tidy (you can pick more than one):

- a. BJsales
- b. EuStockMarkets
- c. DNase
- d. Formaldehyde
- e. Orange
- f. UCBA admissions

-> answer: b, c, d, e

## 4.3 Manipulating data frames

The dplyr package from the tidyverse introduces functions that perform some of the most common operations when working with data frames and uses names for these functions that are relatively easy to remember. For instance, to change the data table by adding a new column, we use mutate. To filter the data table to a subset of rows, we use filter. Finally, to subset the data by selecting specific columns, we use select.

### 4.3.1 Adding a column with mutate

We want all the necessary information for our analysis to be included in the data table. So the first task is to add the murder rates to our murders data frame. The function mutate takes the data frame as a first argument and the name and values of the variable as a second argument using the convention name = values. So, to add murder rates, we use:

```
library(dslabs)
data("murders")
murders <- mutate(murders, rate= total / population*100000)
```

Notice that here we used total and population inside the function, which are objects that are not defined in our workspace. But why don't we get an error?

This is one of dplyr's main features. Functions in this package, such as mutate, know to look for variables in the data frame provided in the first argument. In the call to mutate above, total will have the values in murders\$total. This approach makes the code much more readable.

We can see that the new column is added:

```
head(murders)
```

```
##      state abb region population total    rate
## 1  Alabama AL  South   4779736   135 2.824424
## 2  Alaska  AK   West    710231    19 2.675186
## 3  Arizona AZ   West   6392017   232 3.629527
## 4  Arkansas AR  South   2915918    93 3.189390
## 5 California CA  West  37253956  1257 3.374138
## 6  Colorado CO   West   5029196    65 1.292453
```

Although we have overwritten the original murders object, this does not change the object that loaded with data(murders). If we load the murders data again, the original will overwrite our mutated version.

#### 4.3.2 Subsetting with filter

Now suppose that we want to filter the data table to only show the entries for which the murder rate is lower than 0.71. To do this we use the filter function, which takes the data table as the first argument and then the conditional statement as the second. Like mutate, we can use the unquoted variable names from murders inside the function and it will know we mean the columns and not objects in the workspace.

```
filter(murders, rate <= 0.71)
```

```
##      state abb      region population total    rate
## 1  Hawaii  HI        West   1360301     7 0.5145920
## 2   Iowa  IA North Central  3046355    21 0.6893484
## 3 New Hampshire NH      Northeast  1316470     5 0.3798036
## 4 North Dakota ND North Central   672591     4 0.5947151
## 5  Vermont VT      Northeast   625741     2 0.3196211
```

#### 4.3.3 Selecting columns with select

Although our data table only has six columns, some data tables include hundreds. If we want to view just a few, we can use the dplyr select function. In the code below we select three columns, assign this to a new object and then filter the new object:

```
new_table <- select(murders, state, region, rate)
filter(new_table, rate <= 0.71)
```

```
##      state      region    rate
## 1  Hawaii        West 0.5145920
## 2   Iowa North Central 0.6893484
## 3 New Hampshire Northeast 0.3798036
## 4 North Dakota North Central 0.5947151
## 5  Vermont      Northeast 0.3196211
```

In the call to select, the first argument murders is an object, but state, region, and rate are variable names.

## 4.4 Exercises

1. Load the dplyr package and the murders dataset.

```
library(dplyr)
library(dslabs)
data(murders)
```

You can add columns using the dplyr function mutate. This function is aware of the column names and inside the function you can call them unquoted:

```
murders <- mutate(murders, population_in_millions = population / 10^6)
```

We can write population rather than murders\$population. The function mutate knows we are grabbing columns from murders.

Use the function mutate to add a murders column named rate with the per 100,000 murder rate as in the example code above. Make sure you redefine murders as done in the example code above ( murders <- [your code]) so we can keep using this variable.

```
murders <- mutate(murders, rate= total / population*100000)
```

2. If rank(x) gives you the ranks of x from lowest to highest, rank(-x) gives you the ranks from highest to lowest. Use the function mutate to add a column rank containing the rank, from highest to lowest murder rate. Make sure you redefine murders so we can keep using this variable.

```
murders <- mutate(murders, rank= rank(-rate))
```

3. With dplyr, we can use select to show only certain columns. For example, with this code we would only show the states and population sizes:

```
select(murders, state, population) %>% head()
```

```
##      state population
## 1  Alabama    4779736
## 2   Alaska     710231
## 3  Arizona    6392017
## 4 Arkansas    2915918
## 5 California 37253956
## 6  Colorado    5029196
```

Use select to show the state names and abbreviations in murders. Do not redefine murders, just show the results.

```
select(murders, state, abb)
```

```
##      state abb
## 1  Alabama  AL
## 2   Alaska  AK
## 3  Arizona  AZ
```

```

## 4      Arkansas AR
## 5      California CA
## 6      Colorado CO
## 7      Connecticut CT
## 8      Delaware DE
## 9 District of Columbia DC
## 10     Florida FL
## 11     Georgia GA
## 12     Hawaii HI
## 13     Idaho ID
## 14     Illinois IL
## 15     Indiana IN
## 16     Iowa IA
## 17     Kansas KS
## 18     Kentucky KY
## 19     Louisiana LA
## 20     Maine ME
## 21     Maryland MD
## 22     Massachusetts MA
## 23     Michigan MI
## 24     Minnesota MN
## 25     Mississippi MS
## 26     Missouri MO
## 27     Montana MT
## 28     Nebraska NE
## 29     Nevada NV
## 30     New Hampshire NH
## 31     New Jersey NJ
## 32     New Mexico NM
## 33     New York NY
## 34     North Carolina NC
## 35     North Dakota ND
## 36     Ohio OH
## 37     Oklahoma OK
## 38     Oregon OR
## 39     Pennsylvania PA
## 40     Rhode Island RI
## 41     South Carolina SC
## 42     South Dakota SD
## 43     Tennessee TN
## 44     Texas TX
## 45     Utah UT
## 46     Vermont VT
## 47     Virginia VA
## 48     Washington WA
## 49     West Virginia WV
## 50     Wisconsin WI
## 51     Wyoming WY

```

4. The `dplyr` function `filter` is used to choose specific rows of the data frame to keep. Unlike `select` which is for columns, `filter` is for rows. For example, you can show just the New York row like this:

```
filter(murders, state == "New York")
```

```
##      state abb      region population total population_in_millions    rate rank
## 1 New York  NY Northeast   19378102    517                19.3781 2.66796    29
```

You can use other logical vectors to filter rows.

Use filter to show the top 5 states with the highest murder rates. After we add murder rate and rank, do not change the murders dataset, just show the result. Remember that you can filter based on the rank column.

```
filter(murders, rank <= 5)
```

```
##      state abb      region population total
## 1 District of Columbia DC      South    601723    99
## 2      Louisiana LA      South   4533372   351
## 3      Maryland MD      South   5773552   293
## 4      Missouri MO North Central   5988927   321
## 5      South Carolina SC      South   4625364   207
##      population_in_millions    rate rank
## 1          0.601723 16.452753    1
## 2          4.533372  7.742581    2
## 3          5.773552  5.074866    4
## 4          5.988927  5.359892    3
## 5          4.625364  4.475323    5
```

5. We can remove rows using the `!=` operator. For example, to remove Florida, we would do this:

```
no_florida <- filter(murders, state != "Florida")
```

Create a new data frame called `no_south` that removes states from the South region. How many states are in this category? You can use the function `nrow` for this.

```
no_south <- filter(murders, region != "South")
nrow(no_south)
```

```
## [1] 34
```

6. We can also use `%in%` to filter with `dplyr`. You can therefore see the data from New York and Texas like this:

```
filter(murders, state %in% c("New York", "Texas"))
```

```
##      state abb      region population total population_in_millions    rate rank
## 1 New York  NY Northeast   19378102    517                19.3781 2.66796    29
## 2      Texas TX      South   25145561    805                25.14556 3.20136    16
```

Create a new data frame called `murders_nw` with only the states from the Northeast and the West. How many states are in this category?

```
murders_nw <- filter(murders, region %in% c("Northeast", "West"))
nrow(murders_nw)
```

```
## [1] 22
```

7. Suppose you want to live in the Northeast or West and want the murder rate to be less than 1. We want to see the data for the states satisfying these options. Note that you can use logical operators with filter. Here is an example in which we filter to keep only small states in the Northeast region.

```
filter(murders, population < 5000000 & region == "Northeast")
```

```
##           state abb    region population total population_in_millions    rate
## 1  Connecticut  CT Northeast   3574097    97             3.574097 2.7139722
## 2         Maine  ME Northeast   1328361    11             1.328361 0.8280881
## 3 New Hampshire NH Northeast   1316470     5             1.316470 0.3798036
## 4  Rhode Island RI Northeast   1052567    16             1.052567 1.5200933
## 5     Vermont  VT Northeast    625741     2             0.625741 0.3196211
##   rank
## 1    25
## 2    44
## 3    50
## 4    35
## 5    51
```

Make sure murders has been defined with rate and rank and still has all states. Create a table called my\_states that contains rows for states satisfying both the conditions: it is in the Northeast or West and the murder rate is less than 1. Use select to show only the state name, the rate, and the rank.

```
my_states <- filter(murders, region %in% c("Northeast", "West") & rate < 1)
select(my_states, state, rate, rank)
```

```
##           state    rate rank
## 1      Hawaii 0.5145920   49
## 2      Idaho 0.7655102   46
## 3      Maine 0.8280881   44
## 4 New Hampshire 0.3798036   50
## 5      Oregon 0.9396843   42
## 6      Utah 0.7959810   45
## 7      Vermont 0.3196211   51
## 8      Wyoming 0.8871131   43
```

## 4.5 The pipe: %>%

With dplyr we can perform a series of operations, for example select and then filter, by sending the results of one function to another using what is called the pipe operator: %>%. Some details are included below.

We wrote code above to show three variables (state, region, rate) for states that have murder rates below 0.71. To do this, we defined the intermediate object new\_table. In dplyr we can write code that looks more like a description of what we want to do without intermediate objects:

```
original data -> select -> filter
```



For such an operation, we can use the pipe `%>%`. The code looks like this:

```
murders %>% select(state, region, rate) %>% filter(rate <= 0.71)
```

```
##           state      region    rate
## 1      Hawaii        West 0.5145920
## 2      Iowa North Central 0.6893484
## 3 New Hampshire    Northeast 0.3798036
## 4 North Dakota North Central 0.5947151
## 5      Vermont    Northeast 0.3196211
```

This line of code is equivalent to the two lines of code above. What is going on here?

In general, the pipe sends the result of the left side of the pipe to be the first argument of the function on the right side of the pipe. Here is a very simple example:

```
16 %>% sqrt()
```

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

We can continue to pipe values along:

```
16 %>% sqrt() %>% log2()
```

```
## [1] 2
```

The above statement is equivalent to `log2(sqrt(16))`.

Remember that the pipe sends values to the first argument, so we can define other arguments as if the first argument is already defined:

```
16 %>% sqrt() %>% log(base = 2)
```

```
## [1] 2
```

Therefore, when using the pipe with data frames and dplyr, we no longer need to specify the required first argument since the dplyr functions we have described all take the data as the first argument. In the code we wrote:

```
murders %>% select(state, region, rate) %>% filter(rate <= 0.71)
```

```
##           state      region    rate
## 1      Hawaii        West 0.5145920
## 2      Iowa North Central 0.6893484
## 3 New Hampshire    Northeast 0.3798036
## 4 North Dakota North Central 0.5947151
## 5      Vermont    Northeast 0.3196211
```

## 4.6 Exercises

1. The pipe `%>%` can be used to perform operations sequentially without having to define intermediate objects. Start by redefining `murder` to include `rate` and `rank`.

```
murders <- mutate(murders, rate = total / population * 100000,
                  rank = rank(-rate))
```

In the solution to the previous exercise, we did the following:

```
my_states <- filter(murders, region %in% c("Northeast", "West") &
                  rate < 1)

select(my_states, state, rate, rank)
```

##	state	rate	rank
## 1	Hawaii	0.5145920	49
## 2	Idaho	0.7655102	46
## 3	Maine	0.8280881	44
## 4	New Hampshire	0.3798036	50
## 5	Oregon	0.9396843	42
## 6	Utah	0.7959810	45
## 7	Vermont	0.3196211	51
## 8	Wyoming	0.8871131	43

The pipe `%>%` permits us to perform both operations sequentially without having to define an intermediate variable `my_states`. We therefore could have mutated and selected in the same line like this:

```
mutate(murders, rate = total / population * 100000, rank = rank(-rate)) %>% select(state, rate, rank)
```

##	state	rate	rank
## 1	Alabama	2.8244238	23
## 2	Alaska	2.6751860	27
## 3	Arizona	3.6295273	10
## 4	Arkansas	3.1893901	17
## 5	California	3.3741383	14
## 6	Colorado	1.2924531	38
## 7	Connecticut	2.7139722	25
## 8	Delaware	4.2319369	6
## 9	District of Columbia	16.4527532	1
## 10	Florida	3.3980688	13
## 11	Georgia	3.7903226	9
## 12	Hawaii	0.5145920	49
## 13	Idaho	0.7655102	46
## 14	Illinois	2.8369608	22
## 15	Indiana	2.1900730	31
## 16	Iowa	0.6893484	47
## 17	Kansas	2.2081106	30
## 18	Kentucky	2.6732010	28
## 19	Louisiana	7.7425810	2
## 20	Maine	0.8280881	44
## 21	Maryland	5.0748655	4
## 22	Massachusetts	1.8021791	32
## 23	Michigan	4.1786225	7
## 24	Minnesota	0.9992600	40
## 25	Mississippi	4.0440846	8
## 26	Missouri	5.3598917	3

## 27	Montana	1.2128379	39
## 28	Nebraska	1.7521372	33
## 29	Nevada	3.1104763	19
## 30	New Hampshire	0.3798036	50
## 31	New Jersey	2.7980319	24
## 32	New Mexico	3.2537239	15
## 33	New York	2.6679599	29
## 34	North Carolina	2.9993237	20
## 35	North Dakota	0.5947151	48
## 36	Ohio	2.6871225	26
## 37	Oklahoma	2.9589340	21
## 38	Oregon	0.9396843	42
## 39	Pennsylvania	3.5977513	11
## 40	Rhode Island	1.5200933	35
## 41	South Carolina	4.4753235	5
## 42	South Dakota	0.9825837	41
## 43	Tennessee	3.4509357	12
## 44	Texas	3.2013603	16
## 45	Utah	0.7959810	45
## 46	Vermont	0.3196211	51
## 47	Virginia	3.1246001	18
## 48	Washington	1.3829942	37
## 49	West Virginia	1.4571013	36
## 50	Wisconsin	1.7056487	34
## 51	Wyoming	0.8871131	43

Notice that `select` no longer has a data frame as the first argument. The first argument is assumed to be the result of the operation conducted right before the `%>%`.

Repeat the previous exercise, but now instead of creating a new object, show the result and only include the state, rate, and rank columns. Use a pipe `%>%` to do this in just one line.

```
murders %>% filter(region %in% c("Northeast", "West") & rate < 1) %>% select(state, rate, rank)
```

##	state	rate	rank
## 1	Hawaii	0.5145920	49
## 2	Idaho	0.7655102	46
## 3	Maine	0.8280881	44
## 4	New Hampshire	0.3798036	50
## 5	Oregon	0.9396843	42
## 6	Utah	0.7959810	45
## 7	Vermont	0.3196211	51
## 8	Wyoming	0.8871131	43

2. Reset `murders` to the original table by using `data(murders)`. Use a pipe to create a new data frame called `my_states` that considers only states in the Northeast or West which have a murder rate lower than 1, and contains only the state, rate and rank columns. The pipe should also have four components separated by three `%>%`. The code should look something like this: `my_states <- murders %>% mutate SOMETHING %>% filter SOMETHING %>% select SOMETHING`

```
data(murders)

my_states <- murders %>%
```

```
mutate(rate = total / population * 100000, rank = rank(-rate)) %>%
  filter(region %in% c("Northeast", "West") & rate<1) %>%
  select(state, rate, rank)
my_states
```

```
##           state      rate rank
## 1      Hawaii 0.5145920   49
## 2       Idaho 0.7655102   46
## 3       Maine 0.8280881   44
## 4 New Hampshire 0.3798036   50
## 5       Oregon 0.9396843   42
## 6       Utah 0.7959810   45
## 7    Vermont 0.3196211   51
## 8    Wyoming 0.8871131   43
```