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:

```r
library(tidyverse)
```

```
## -- Attaching packages --------------------------- tidyverse 1.3.0 --
```

```
## v ggplot2 3.3.2     v purrr   0.3.4
## v tibble  3.0.3     v dplyr   1.0.1
## v tidyr   1.1.2     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.5.0
```

```
## -- 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.

```
##        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:

```
##       country year fertility
## 1     Germany 1960      2.41
## 2 South Korea 1960      6.16
## 3     Germany 1961      2.44
## 4 South Korea 1961      5.99
## 5     Germany 1962      2.47
## 6 South Korea 1962      5.79
```

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:

```
##       country 1960 1961 1962
## 1     Germany 2.41 2.44 2.47
## 2 South Korea 6.16 5.99 5.79
```

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:

```
head(co2, 10)
```

```
##  [1] 315.42 316.31 316.50 317.56 318.13 318.00 316.39 314.65 313.68 313.18
```

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.**

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.

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

```
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.

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. 'UCBAdmissions'

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:

```r
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:

```r
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.

```r
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:

```r
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
```

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.

```r
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.

```
nrow(filter(murders, region != "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               19.37810 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?

```r
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.

```r
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.

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

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:

$$\text{original data} \rightarrow \text{select} \rightarrow \text{filter}$$

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

```r
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:

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

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

We can continue to pipe values along:

```r
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:

```r
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:

```r
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
```

`murders` is the first argument of the `select` function, and the new data frame (formerly `new_table`) is the first argument of the `filter` function.

Note that the pipe works well with functions where the first argument is the input data. Functions in **tidyverse** packages like **dplyr** have this format and can be used easily with the pipe.

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.

```r
murders %>% mutate(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
```

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:

```
data(murders)
my_states <- murders %>%
  mutate(murders, 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
```

4.7 Summarizing data

An important part of exploratory data analysis is summarizing data. The average and standard deviation are two examples of widely used summary statistics. More informative summaries can often be achieved by first splitting data into groups. In this section, we cover two new **dplyr** verbs that make these computations easier: `summarize` and `group_by`. We learn to access resulting values using the `pull` function.

4.7.1 `summarize`

The `summarize` function in **dplyr** provides a way to compute summary statistics with intuitive and readable code. We start with a simple example based on heights. The `heights` dataset includes heights and sex reported by students in an in-class survey.

```
library(dplyr)
library(dslabs)
data(heights)
```

The following code computes the average and standard deviation for females:

```
s <- heights %>%
  filter(sex == "Female") %>%
  summarize(average = mean(height), standard_deviation = sd(height))
s
```

```
##    average standard_deviation
## 1 64.93942           3.760656
```

This takes our original data table as input, filters it to keep only females, and then produces a new summarized table with just the average and the standard deviation of heights. We get to choose the names of the columns of the resulting table. For example, above we decided to use `average` and `standard_deviation`, but we could have used other names just the same.

Because the resulting table stored in `s` is a data frame, we can access the components with the accessor `$`:

```
s$average
```

```
## [1] 64.93942
```

```
s$standard_deviation
```

```
## [1] 3.760656
```

As with most other **dplyr** functions, summarize is aware of the variable names and we can use them directly. So when inside the call to the `summarize` function we write `mean(height)`, the function is accessing the column with the name "height" and then computing the average of the resulting numeric vector. We can compute any other summary that operates on vectors and returns a single value. For example, we can add the median, minimum, and maximum heights like this:

```
heights %>%
  filter(sex == "Female") %>%
  summarize(median = median(height), minimum = min(height),
            maximum = max(height))
```

```
##     median minimum maximum
## 1 64.98031      51      79
```

We can obtain these three values with just one line using the `quantile` function: for example, `quantile(x, c(0,0.5,1))` returns the min (0th percentile), median (50th percentile), and max (100th percentile) of the vector `x`. However, if we attempt to use a function like this that returns two or more values inside `summarize`:

```
heights %>%
  filter(sex == "Female") %>%
  summarize(range = quantile(height, c(0, 0.5, 1)))
```

```
##      range
## 1 51.00000
## 2 64.98031
## 3 79.00000
```

~~we will receive an error: `Error: expecting result of length one, got : 2`. With the function `summarize`, we can only call functions that return a single value. In Section 4.12, we will learn how to deal with functions that return more than one value.~~

For another example of how we can use the `summarize` function, let's compute the average murder rate for the United States. Remember our data table includes total murders and population size for each state and we have already used **dplyr** to add a murder rate column:

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

Remember that the US murder rate is **not** the average of the state murder rates:

```
summarize(murders, mean(rate))
```

```
##   mean(rate)
## 1   2.779125
```

This is because in the computation above the small states are given the same weight as the large ones. The US murder rate is the total number of murders in the US divided by the total US population. So the correct computation is:

```
us_murder_rate <- murders %>%
  summarize(rate = sum(total) / sum(population) * 100000)
us_murder_rate
```

```
##       rate
## 1 3.034555
```

This computation counts larger states proportionally to their size which results in a larger value.

4.7.2 pull

The `us_murder_rate` object defined above represents just one number. Yet we are storing it in a data frame:

```
class(us_murder_rate)
```

```
## [1] "data.frame"
```

since, as most **dplyr** functions, `summarize` always returns a data frame.

This might be problematic if we want to use this result with functions that require a numeric value. Here we show a useful trick for accessing values stored in data when using pipes: when a data object is piped that object and its columns can be accessed using the `pull` function. To understand what we mean take a look at this line of code:

```
us_murder_rate %>% pull(rate)
```

```
## [1] 3.034555
```

This returns the value in the `rate` column of `us_murder_rate` making it equivalent to `us_murder_rate$rate`.

To get a number from the original data table with one line of code we can type:

```
us_murder_rate <- murders %>%
  summarize(rate = sum(total) / sum(population) * 100000) %>%
  pull(rate)

us_murder_rate
```

```
## [1] 3.034555
```

which is now a numeric:

```
class(us_murder_rate)
```

```
## [1] "numeric"
```

4.7.3 Group then summarize with group_by

A common operation in data exploration is to first split data into groups and then compute summaries for each group. For example, we may want to compute the average and standard deviation for men's and women's heights separately. The `group_by` function helps us do this.

If we type this:

```
heights %>% group_by(sex)
```

```
## # A tibble: 1,050 x 2
## # Groups:   sex [2]
##    sex    height
##    <fct>   <dbl>
```

```
##   1 Male       75
##   2 Male       70
##   3 Male       68
##   4 Male       74
##   5 Male       61
##   6 Female     65
##   7 Female     66
##   8 Female     62
##   9 Female     66
## 10 Male       67
## # ... with 1,040 more rows
```

The result does not look very different from `heights`, except we see `Groups: sex [2]` when we print the object. Although not immediately obvious from its appearance, this is now a special data frame called a *grouped data frame*, and **dplyr** functions, in particular `summarize`, will behave differently when acting on this object. Conceptually, you can think of this table as many tables, with the same columns but not necessarily the same number of rows, stacked together in one object. When we summarize the data after grouping, this is what happens:

```r
heights %>%
  group_by(sex) %>%
  summarize(average = mean(height), standard_deviation = sd(height))
```

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

```
## # A tibble: 2 x 3
##   sex    average standard_deviation
##   <fct>    <dbl>              <dbl>
## 1 Female    64.9               3.76
## 2 Male      69.3               3.61
```

The `summarize` function applies the summarization to each group separately.

For another example, let's compute the median murder rate in the four regions of the country:

```r
murders %>%
  group_by(region) %>%
  summarize(median_rate = median(rate))
```

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

```
## # A tibble: 4 x 2
##   region        median_rate
##   <fct>               <dbl>
## 1 Northeast            1.80
## 2 South                3.40
## 3 North Central        1.97
## 4 West                 1.29
```

4.8 Sorting data frames

When examining a dataset, it is often convenient to sort the table by the different columns. We know about the `order` and `sort` function, but for ordering entire tables, the **dplyr** function `arrange` is useful. For example, here we order the states by population size:

```
murders %>%
  arrange(population) %>%
  head()
```

```
##                  state abb        region population total       rate
## 1              Wyoming  WY          West     563626     5  0.8871131
## 2 District of Columbia  DC         South     601723    99 16.4527532
## 3              Vermont  VT     Northeast     625741     2  0.3196211
## 4         North Dakota  ND North Central     672591     4  0.5947151
## 5               Alaska  AK          West     710231    19  2.6751860
## 6         South Dakota  SD North Central     814180     8  0.9825837
```

With `arrange` we get to decide which column to sort by. To see the states by murder rate, from lowest to highest, we arrange by `rate` instead:

```
murders %>%
  arrange(rate) %>%
  head()
```

```
##           state abb        region population total      rate
## 1       Vermont  VT     Northeast     625741     2 0.3196211
## 2 New Hampshire  NH     Northeast    1316470     5 0.3798036
## 3        Hawaii  HI          West    1360301     7 0.5145920
## 4  North Dakota  ND North Central     672591     4 0.5947151
## 5          Iowa  IA North Central    3046355    21 0.6893484
## 6         Idaho  ID          West    1567582    12 0.7655102
```

Note that the default behavior is to order in ascending order. In **dplyr**, the function `desc` transforms a vector so that it is in descending order. To sort the table in descending order, we can type:

```
murders %>%
  arrange(desc(rate))
```

```
##                   state abb        region population total       rate
## 1  District of Columbia  DC         South     601723    99 16.4527532
## 2             Louisiana  LA         South    4533372   351  7.7425810
## 3              Missouri  MO North Central    5988927   321  5.3598917
## 4              Maryland  MD         South    5773552   293  5.0748655
## 5        South Carolina  SC         South    4625364   207  4.4753235
## 6              Delaware  DE         South     897934    38  4.2319369
## 7              Michigan  MI North Central    9883640   413  4.1786225
## 8           Mississippi  MS         South    2967297   120  4.0440846
## 9               Georgia  GA         South    9920000   376  3.7903226
## 10              Arizona  AZ          West    6392017   232  3.6295273
## 11         Pennsylvania  PA     Northeast   12702379   457  3.5977513
## 12            Tennessee  TN         South    6346105   219  3.4509357
```

```
## 13          Florida  FL         South   19687653   669  3.3980688
## 14       California  CA          West   37253956  1257  3.3741383
## 15       New Mexico  NM          West    2059179    67  3.2537239
## 16            Texas  TX         South   25145561   805  3.2013603
## 17         Arkansas  AR         South    2915918    93  3.1893901
## 18         Virginia  VA         South    8001024   250  3.1246001
## 19           Nevada  NV          West    2700551    84  3.1104763
## 20   North Carolina  NC         South    9535483   286  2.9993237
## 21         Oklahoma  OK         South    3751351   111  2.9589340
## 22         Illinois  IL North Central   12830632   364  2.8369608
## 23          Alabama  AL         South    4779736   135  2.8244238
## 24       New Jersey  NJ     Northeast    8791894   246  2.7980319
## 25      Connecticut  CT     Northeast    3574097    97  2.7139722
## 26             Ohio  OH North Central   11536504   310  2.6871225
## 27           Alaska  AK          West     710231    19  2.6751860
## 28         Kentucky  KY         South    4339367   116  2.6732010
## 29         New York  NY     Northeast   19378102   517  2.6679599
## 30           Kansas  KS North Central    2853118    63  2.2081106
## 31          Indiana  IN North Central    6483802   142  2.1900730
## 32    Massachusetts  MA     Northeast    6547629   118  1.8021791
## 33         Nebraska  NE North Central    1826341    32  1.7521372
## 34        Wisconsin  WI North Central    5686986    97  1.7056487
## 35     Rhode Island  RI     Northeast    1052567    16  1.5200933
## 36    West Virginia  WV         South    1852994    27  1.4571013
## 37       Washington  WA          West    6724540    93  1.3829942
## 38         Colorado  CO          West    5029196    65  1.2924531
## 39          Montana  MT          West     989415    12  1.2128379
## 40        Minnesota  MN North Central    5303925    53  0.9992600
## 41     South Dakota  SD North Central     814180     8  0.9825837
## 42           Oregon  OR          West    3831074    36  0.9396843
## 43          Wyoming  WY          West     563626     5  0.8871131
## 44            Maine  ME     Northeast    1328361    11  0.8280881
## 45             Utah  UT          West    2763885    22  0.7959810
## 46            Idaho  ID          West    1567582    12  0.7655102
## 47             Iowa  IA North Central    3046355    21  0.6893484
## 48     North Dakota  ND North Central     672591     4  0.5947151
## 49           Hawaii  HI          West    1360301     7  0.5145920
## 50    New Hampshire  NH     Northeast    1316470     5  0.3798036
## 51          Vermont  VT     Northeast     625741     2  0.3196211
```

4.8.1 Nested sorting

If we are ordering by a column with ties, we can use a second column to break the tie. Similarly, a third column can be used to break ties between first and second and so on. Here we order by `region`, then within region we order by murder rate:

```
murders %>%
  arrange(region, rate) %>%
  head()
```

```
##           state abb    region population total      rate
## 1       Vermont  VT Northeast     625741     2 0.3196211
## 2 New Hampshire  NH Northeast    1316470     5 0.3798036
```

```
## 3         Maine  ME Northeast    1328361    11 0.8280881
## 4  Rhode Island  RI Northeast    1052567    16 1.5200933
## 5 Massachusetts  MA Northeast    6547629   118 1.8021791
## 6      New York  NY Northeast   19378102   517 2.6679599
```

4.8.2 The top n

In the code above, we have used the function `head` to avoid having the page fill up with the entire dataset. If we want to see a larger proportion, we can use the `top_n` function. This function takes a data frame as it's first argument, the number of rows to show in the second, and the variable to filter by in the third. Here is an example of how to see the top 5 rows:

```
murders %>% top_n(5, rate)
```

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

Note that rows are not sorted by `rate`, only filtered. If we want to sort, we need to use `arrange`. Note that if the third argument is left blank, `top_n` filters by the last column.

4.9 Exercises

For these exercises, we will be using the data from the survey collected by the United States National Center for Health Statistics (NCHS). This center has conducted a series of health and nutrition surveys since the 1960's. Starting in 1999, about 5,000 individuals of all ages have been interviewed every year and they complete the health examination component of the survey. Part of the data is made available via the **NHANES** package. Once you install the **NHANES** package, you can load the data like this:

```
library(NHANES)
data(NHANES)
```

The NHANES data has many missing values. The `mean` and `sd` functions in R will return `NA` if any of the entries of the input vector is an `NA`. Here is an example:

```
library(dslabs)
data(na_example)
mean(na_example)
```

```
## [1] NA
```

```
sd(na_example)
```

```
## [1] NA
```

To ignore the `NA`s we can use the `na.rm` argument:

```
mean(na_example, na.rm = TRUE)
```

```
## [1] 2.301754
```

```
sd(na_example, na.rm = TRUE)
```

```
## [1] 1.22338
```

Let's now explore the NHANES data. 1. We will provide some basic facts about blood pressure. First let's select a group to set the standard. We will use 20-to-29-year-old females. `AgeDecade` is a categorical variable with these ages. Note that the category is coded like " 20-29", with a space in front! What is the average and standard deviation of systolic blood pressure as saved in the `BPSysAve` variable? Save it to a variable called `ref`.

```
ref <- NHANES %>% filter(AgeDecade == ' 20-29') %>% summarise(mean(BPSysAve, na.rm = TRUE), sd(BPSysAve
ref
```

```
## # A tibble: 1 x 2
##   `mean(BPSysAve, na.rm = TRUE)` `sd(BPSysAve, na.rm = TRUE)`
##                            <dbl>                        <dbl>
## 1                           113.                         11.7
```

2. Using a pipe, assign the average to a numeric variable `ref_avg`. Hint: Use the code similar to above and then `pull`.

```
ref_avg <- NHANES %>% filter(AgeDecade == ' 20-29') %>% summarise(avg = mean(BPSysAve, na.rm = TRUE), s
ref_avg
```

```
## [1] 113.1583
```

3. Now report the min and max values for the same group.

```
NHANES %>% filter(AgeDecade == ' 20-29') %>% summarise(min = min(BPSysAve, na.rm = TRUE), max = max(BPS
```

```
## # A tibble: 1 x 2
##     min   max
##   <int> <int>
## 1    84   179
```

4. Compute the average and standard deviation for females, but for each age group separately rather than a selected decade as in question 1. Note that the age groups are defined by `AgeDecade`. Hint: rather than filtering by age and gender, `filter` by Gender and then use `group_by`.

```
NHANES %>% filter(Gender == 'female') %>% group_by(AgeDecade) %>% summarise(avg = mean(BPSysAve, na.rm =
```

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

```
## # A tibble: 9 x 3
##   AgeDecade   avg    sd
##   <fct>     <dbl> <dbl>
## 1 " 0-9"     100.  9.07
## 2 " 10-19"   104.  9.46
## 3 " 20-29"   108. 10.1
## 4 " 30-39"   111. 12.3
## 5 " 40-49"   115. 14.5
## 6 " 50-59"   122. 16.2
## 7 " 60-69"   127. 17.1
## 8 " 70+"     134. 19.8
## 9  <NA>      142. 22.9
```

5. Repeat exercise 4 for males.

```
NHANES %>% filter(Gender == 'male') %>% group_by(AgeDecade) %>% summarise(avg = mean(BPSysAve, na.rm = T
```

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

```
## # A tibble: 9 x 3
##   AgeDecade   avg    sd
##   <fct>     <dbl> <dbl>
## 1 " 0-9"     97.4  8.32
## 2 " 10-19"  110.   11.2
## 3 " 20-29"  118.   11.3
## 4 " 30-39"  119.   12.3
## 5 " 40-49"  121.   14.0
## 6 " 50-59"  126.   17.8
## 7 " 60-69"  127.   17.5
## 8 " 70+"    130.   18.7
## 9  <NA>     136.   23.5
```

6. We can actually combine both summaries for exercises 4 and 5 into one line of code. This is because `group_by` permits us to group by more than one variable. Obtain one big summary table using `group_by(AgeDecade, Gender)`.

```
NHANES %>% group_by(AgeDecade, Gender) %>% summarise(avg = mean(BPSysAve, na.rm = TRUE), sd = sd(BPSysA
```

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

```
## # A tibble: 18 x 4
## # Groups:   AgeDecade [9]
##    AgeDecade Gender   avg    sd
##    <fct>     <fct>  <dbl> <dbl>
##  1 " 0-9"    female 100.   9.07
##  2 " 0-9"    male    97.4  8.32
##  3 " 10-19"  female 104.   9.46
##  4 " 10-19"  male   110.  11.2
##  5 " 20-29"  female 108.  10.1
##  6 " 20-29"  male   118.  11.3
##  7 " 30-39"  female 111.  12.3
##  8 " 30-39"  male   119.  12.3
```

```
##  9 " 40-49"  female 115.  14.5
## 10 " 40-49"  male   121.  14.0
## 11 " 50-59"  female 122.  16.2
## 12 " 50-59"  male   126.  17.8
## 13 " 60-69"  female 127.  17.1
## 14 " 60-69"  male   127.  17.5
## 15 " 70+"    female 134.  19.8
## 16 " 70+"    male   130.  18.7
## 17  <NA>     female 142.  22.9
## 18  <NA>     male   136.  23.5
```

7. For males between the ages of 40-49, compare systolic blood pressure across race as reported in the `Race1` variable. Order the resulting table from lowest to highest average systolic blood pressure.

```
Race1 <- NHANES %>% filter(AgeDecade == " 40-49" & Gender == "male") %>% group_by(Race1) %>% summarise(a
```

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

```
Race1
```

```
## # A tibble: 5 x 3
##   Race1      avg    sd
##   <fct>    <dbl> <dbl>
## 1 Black     126.  17.1
## 2 Mexican   122.  13.9
## 3 Hispanic  122.  11.1
## 4 Other     120.  16.2
## 5 White     120.  13.4
```

4.10 Tibbles

Tidy data must be stored in data frames. We introduced the data frame in Section 2.4.1 and have been using the `murders` data frame throughout the book. In Section 4.7.3 we introduced the `group_by` function, which permits stratifying data before computing summary statistics. But where is the group information stored in the data frame?

```
murders %>% group_by(region)
```

```
## # A tibble: 51 x 6
## # Groups:   region [4]
##    state                abb   region    population total  rate
##    <chr>                <chr> <fct>          <dbl> <dbl> <dbl>
##  1 Alabama              AL    South        4779736   135  2.82
##  2 Alaska               AK    West          710231    19  2.68
##  3 Arizona              AZ    West         6392017   232  3.63
##  4 Arkansas             AR    South        2915918    93  3.19
##  5 California           CA    West        37253956  1257  3.37
##  6 Colorado             CO    West         5029196    65  1.29
##  7 Connecticut          CT    Northeast    3574097    97  2.71
##  8 Delaware             DE    South         897934    38  4.23
##  9 District of Columbia DC    South         601723    99 16.5
## 10 Florida              FL    South       19687653   669  3.40
## # ... with 41 more rows
```

Notice that there are no columns with this information. But, if you look closely at the output above, you see the line `A tibble` followd by dimensions. We can learn the class of the returned object using:

```
murders %>% group_by(region) %>% class()
```

```
## [1] "grouped_df" "tbl_df"     "tbl"        "data.frame"
```

The `tbl`, pronounced tibble, is a special kind of data frame. The functions `group_by` and `summarize` always return this type of data frame. The `group_by` function returns a special kind of `tbl`, the `grouped_df`. We will say more about these later. For consistency, the **dplyr** manipulation verbs (`select`, `filter`, `mutate`, and `arrange`) preserve the class of the input: if they receive a regular data frame they return a regular data frame, while if they receive a tibble they return a tibble. But tibbles are the preferred format in the tidyverse and as a result tidyverse functions that produce a data frame from scratch return a tibble. For example, in Chapter 5 we will see that tidyverse functions used to import data create tibbles.

Tibbles are very similar to data frames. In fact, you can think of them as a modern version of data frames. Nonetheless there are three important differences which we describe next.

4.10.1 Tibbles display better

The print method for tibbles is more readable than that of a data frame. To see this, compare the outputs of typing `murders` and the output of murders if we convert it to a tibble. We can do this using `as_tibble(murders)`. If using RStudio, output for a tibble adjusts to your window size. To see this, change the width of your R console and notice how more/less columns are shown.

4.10.2 Subsets of tibbles are tibbles

If you subset the columns of a data frame, you may get back an object that is not a data frame, such as a vector or scalar. For example:

```
class(murders[,4])
```

```
## [1] "numeric"
```

is not a data frame. With tibbles this does not happen:

```
class(as_tibble(murders)[,4])
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

This is useful in the tidyverse since functions require data frames as input.

With tibbles, if you want to access the vector that defines a column, and not get back a data frame, you need to use the accessor `$`:

```
class(as_tibble(murders)$population)
```

```
## [1] "numeric"
```

A related feature is that tibbles will give you a warning if you try to access a column that does not exist. If we accidentally write `Population` instead of `population` this:

```r
murders$Population
```

```
## NULL
```

returns a `NULL` with no warning, which can make it harder to debug. In contrast, if we try this with a tibble we get an informative warning:

```r
as_tibble(murders)$Population
```

```
## Warning: Unknown or uninitialised column: `Population`.
```

```
## NULL
```

4.10.3 Tibbles can have complex entries

While data frame columns need to be vectors of numbers, strings, or logical values, tibbles can have more complex objects, such as lists or functions. Also, we can create tibbles with functions:

```r
tibble(id = c(1, 2, 3), func = c(mean, median, sd))
```

```
## # A tibble: 3 x 2
##      id func
##   <dbl> <list>
## 1     1 <fn>
## 2     2 <fn>
## 3     3 <fn>
```

4.10.4 Tibbles can be grouped

The function `group_by` returns a special kind of tibble: a grouped tibble. This class stores information that lets you know which rows are in which groups. The tidyverse functions, in particular the `summarize` function, are aware of the group information.

4.10.5 Create a tibble using tibble instead of data.frame

It is sometimes useful for us to create our own data frames. To create a data frame in the tibble format, you can do this by using the `tibble` function.

```r
grades <- tibble(names = c("John", "Juan", "Jean", "Yao"),
                     exam_1 = c(95, 80, 90, 85),
                     exam_2 = c(90, 85, 85, 90))
class(grades$names)
```

```
## [1] "character"
```

Note that base R (without packages loaded) has a function with a very similar name, `data.frame`, that can be used to create a regular data frame rather than a tibble. One other important difference is that by default `data.frame` coerces characters into factors without providing a warning or message:

```r
grades <- data.frame(names = c("John", "Juan", "Jean", "Yao"),
                     exam_1 = c(95, 80, 90, 85),
                     exam_2 = c(90, 85, 85, 90))
class(grades$names)
```

```
## [1] "character"
```

To avoid this, we use the rather cumbersome argument `stringsAsFactors`:

```r
grades <- data.frame(names = c("John", "Juan", "Jean", "Yao"),
                     exam_1 = c(95, 80, 90, 85),
                     exam_2 = c(90, 85, 85, 90),
                     stringsAsFactors = FALSE)
class(grades$names)
```

```
## [1] "character"
```

To convert a regular data frame to a tibble, you can use the `as_tibble` function.

```r
as_tibble(grades) %>% class()
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

4.11 The dot operator

One of the advantages of using the pipe `%>%` is that we do not have to keep naming new objects as we manipulate the data frame. As a quick reminder, if we want to compute the median murder rate for states in the southern states, instead of typing:

```r
tab_1 <- filter(murders, region == "South")
tab_2 <- mutate(tab_1, rate = total / population * 10^5)
rates <- tab_2$rate
median(rates)
```

```
## [1] 3.398069
```

We can avoid defining any new intermediate objects by instead typing:

```r
filter(murders, region == "South") %>%
  mutate(rate = total / population * 10^5) %>%
  summarize(median = median(rate)) %>%
  pull(median)
```

```
## [1] 3.398069
```

We can do this because each of these functions takes a data frame as the first argument. But what if we want to access a component of the data frame. For example, what if the `pull` function was not available and we wanted to access `tab_2$rate`? What data frame name would we use? The answer is the dot operator.

For example to access the rate vector without the `pull` function we could use

```
rates <-   filter(murders, region == "South") %>%
  mutate(rate = total / population * 10^5) %>%
  .$rate
median(rates)
```

```
## [1] 3.398069
```

In the next section, we will see other instances in which using the `.` is useful.

4.12 do

The tidyverse functions know how to interpret grouped tibbles. Furthermore, to facilitate stringing commands through the pipe `%>%`, tidyverse functions consistently return data frames, since this assures that the output of a function is accepted as the input of another. But most R functions do not recognize grouped tibbles nor do they return data frames. The `quantile` function is an example we described in Section 4.7.1. The `do` function serves as a bridge between R functions such as `quantile` and the tidyverse. The `do` function understands grouped tibbles and always returns a data frame.

~~In Section 4.7.1, we noted that if we attempt to use `quantile` to obtain the min, median and max in one call, we will receive an error: `Error: expecting result of length one, got : 2`.~~

```
data(heights)
heights %>%
  group_by(sex) %>%
  summarize(range = quantile(height, c(0, 0.5, 1)))
```

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

```
## # A tibble: 6 x 2
## # Groups:   sex [2]
##   sex    range
##   <fct>  <dbl>
## 1 Female  51
## 2 Female  65.0
## 3 Female  79
## 4 Male    50
## 5 Male    69
## 6 Male    82.7
```

We can use the do function to fix this.

First we have to write a function that fits into the tidyverse approach: that is, it receives a data frame and returns a data frame.

```
my_summary <- function(dat){
  x <- quantile(dat$height, c(0, 0.5, 1))
  tibble(min = x[1], median = x[2], max = x[3])
}
```

We can now apply the function to the heights dataset to obtain the summaries:

```r
heights %>%
  group_by(sex) %>%
  my_summary
```

```
## # A tibble: 1 x 3
##     min median   max
##   <dbl>  <dbl> <dbl>
## 1    50   68.5  82.7
```

But this is not what we want. We want a summary for each sex and the code returned just one summary. This is because `my_summary` is not part of the tidyverse and does not know how to handled grouped tibbles. `do` makes this connection:

```r
heights %>%
  group_by(sex) %>%
  do(my_summary(.))
```

```
## # A tibble: 2 x 4
## # Groups:   sex [2]
##   sex      min median   max
##   <fct>  <dbl>  <dbl> <dbl>
## 1 Female    51   65.0  79
## 2 Male      50   69    82.7
```

If you do not use the parenthesis, then the function is not executed and instead `do` tries to return the function. This gives an error because `do` must always return a data frame. You can see the error by typing:

```r
heights %>%
  group_by(sex) %>%
  do(my_summary)
```

```
## Error: Results 1, 2 must be data frames, not function
```

4.13 The purrr package

In Section 3.5 we learned about the `sapply` function, which permitted us to apply the same function to each element of a vector. We constructed a function and used `sapply` to compute the sum of the first `n` integers for several values of `n` like this:

```r
compute_s_n <- function(n){
  x <- 1:n
  sum(x)
}
n <- 1:25
s_n <- sapply(n, compute_s_n)
```

This type of operation, applying the same function or procedure to elements of an object, is quite common in data analysis. The **purrr** package includes functions similar to `sapply` but that better interact with other tidyverse functions. The main advantage is that we can better control the output type of functions. In contrast, `sapply` can return several different object types; for example, we might expect a numeric result from a line of code, but `sapply` might convert our result to character under some circumstances. **purrr** functions will never do this: they will return objects of a specified type or return an error if this is not possible.

The first **purrr** function we will learn is `map`, which works very similar to `sapply` but always, without exception, returns a list:

```
library(purrr)
s_n <- map(n, compute_s_n)
class(s_n)
```

```
## [1] "list"
```

If we want a numeric vector, we can instead use `map_dbl` which always returns a vector of numeric values.

```
s_n <- map_dbl(n, compute_s_n)
class(s_n)
```

```
## [1] "numeric"
```

This produces the same results as the `sapply` call shown above.

A particularly useful **purrr** function for interacting with the rest of the tidyverse is `map_df`, which always returns a tibble data frame. However, the function being called needs to return a vector or a list with names. For this reason, the following code would result in a `Argument 1 must have names` error:

```
s_n <- map_df(n, compute_s_n)
```

```
## Error: Argument 1 must have names.
```

We need to change the function to make this work:

```
compute_s_n <- function(n){
  x <- 1:n
  tibble(sum = sum(x))
}
s_n <- map_df(n, compute_s_n)
```

The **purrr** package provides much more functionality not covered here. For more details you can consult this online resource.

4.14 Tidyverse conditionals

A typical data analysis will often involve one or more conditional operations. In Section 3.1 we described the `ifelse` function, which we will use extensively in this book. In this section we present two **dplyr** functions that provide further functionality for performing conditional operations.

4.14.1 `case_when`

The `case_when` function is useful for vectorizing conditional statements. It is similar to `ifelse` but can output any number of values, as opposed to just `TRUE` or `FALSE`. Here is an example splitting numbers into negative, positive, and 0:

```r
x <- c(-2, -1, 0, 1, 2)
case_when(x < 0 ~ "Negative",
          x > 0 ~ "Positive",
          TRUE  ~ "Zero")
```

```
## [1] "Negative" "Negative" "Zero"     "Positive" "Positive"
```

A common use for this function is to define categorical variables based on existing variables. For example, suppose we want to compare the murder rates in four groups of states: *New England*, *West Coast*, *South*, and *other*. For each state, we need to ask if it is in *New England*, if it is not we ask if it is in the *West Coast*, if not we ask if it is in the *South*, and if not we assign *other*. Here is how we use `case_when` to do this:

```r
murders %>%
  mutate(group = case_when(
    abb %in% c("ME", "NH", "VT", "MA", "RI", "CT") ~ "New England",
    abb %in% c("WA", "OR", "CA") ~ "West Coast",
    region == "South" ~ "South",
    TRUE ~ "Other")) %>%
  group_by(group) %>%
  summarize(rate = sum(total) / sum(population) * 10^5)
```

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

```
## # A tibble: 4 x 2
##   group        rate
##   <chr>       <dbl>
## 1 New England  1.72
## 2 Other        2.71
## 3 South        3.63
## 4 West Coast   2.90
```

4.14.2 `between` A common operation in data analysis is to determine if a value falls inside an interval. We can check this using conditionals. For example, to check if the elements of a vector `x` are between `a` and `b` we can type

```r
x >= a & x <= b
```

```
## Error in eval(expr, envir, enclos): object 'a' not found
```

However, this can become cumbersome, especially within the tidyverse approach. The `between` function performs the same operation.

```r
between(x, a, b)
```

```
## Error in between(x, a, b): object 'a' not found
```

4.15 Exercises

1. Load the `murders` dataset. Which of the following is true?

   a. `murders` is in tidy format and is stored in a tibble.
   b. **`murders` is in tidy format and is stored in a data frame.**
   c. `murders` is not in tidy format and is stored in a tibble.
   d. `murders` is not in tidy format and is stored in a data frame.

2. Use `as_tibble` to convert the `murders` data table into a tibble and save it in an object called `murders_tibble`.

```
murders_tibble <- as_tibble(murders)
```

3. Use the `group_by` function to convert `murders` into a tibble that is grouped by region.

```
murders_group <- murders %>% group_by(region)
class(murders_group)
```

```
## [1] "grouped_df" "tbl_df"     "tbl"        "data.frame"
```

4. Write tidyverse code that is equivalent to this code:

```
exp(mean(log(murders$population)))
```

```
## [1] 3675209
```

Write it using the pipe so that each function is called without arguments. Use the dot operator to access the population. Hint: The code should start with `murders %>%`.

```
murders %>% .$population %>% log %>% mean %>% exp
```

```
## [1] 3675209
```

5. Use the `map_df` to create a data frame with three columns named `n`, `s_n`, and `s_n_2`. The first column should contain the numbers 1 through 100. The second and third columns should each contain the sum of 1 through n with n the row number.

```
compute_s_n <- function(n){
  tibble(n = n, s_n = n*(n+1)/2, s_n_2 = sum(1:n))
}
map_df(1:100, compute_s_n)
```

```
## # A tibble: 100 x 3
##         n   s_n s_n_2
##     <int> <dbl> <int>
##   1     1     1     1
##   2     2     3     3
##   3     3     6     6
##   4     4    10    10
```

```
##  5      5     15     15
##  6      6     21     21
##  7      7     28     28
##  8      8     36     36
##  9      9     45     45
## 10     10     55     55
## # ... with 90 more rows
```