T. Mailund, *Beginning Data Science in R 4*
https://doi.org/10.1007/978-1-4842-8155-0_3

3. Data Manipulation

Thomas Mailund[1]
(1) Aarhus, Denmark

Data science is as much about manipulating data as it is about fitting models to data. Data rarely arrives in a form that we can directly feed into the statistical models or machine learning algorithms we want to analyze them with. The first stages of data analysis are almost always figuring out how to load the data into R and then figuring out how to transform it into a shape you can readily analyze.

Data Already in R

There are some data sets already built into R or available in R packages. Those are useful for learning how to use new methods—if you already know a data set and what it can tell you, it is easier to evaluate how a new method performs—or for benchmarking methods you implement. They are of course less helpful when it comes to analyzing new data.

Distributed together with R is the package `dataset`. We can load the package into R using the `library()` function and get a list of the data sets within it, together with a short description of each, like this:

```
library(datasets)
library(help = "datasets")
```

To load an actual data set into R's memory, use the `data()` function. The data sets are all relatively small, so they are ideal for quickly testing code you are working with. For example, to experiment with plotting x-y plots (Figure 3-1), you could use the `cars` data set that consists of only two columns, a speed and a breaking distance:

```
data(cars)
head(cars)
##   speed  dist
```

```
## 1     4      2
## 2     4     10
## 3     7      4
## 4     7     22
## 5     8     16
## 6     9     10
cars %>% plot(dist ~ speed, data = .)
```

I used the `%>%` pipe operator here, because we need to pass the left-hand side to the `data` argument in `plot`. With `|>` we can only pass the left-hand side to the first argument in the right-hand side function call, but with `%>%` we can use `"."` to move the input to another parameter. Generally, I will use the two pipe operators interchangeably in this chapter, except for cases where one is more convenient than another, like before, and then point out why that is.

Don't worry about the plotting function for now; we will return to plotting in the next chapter.

If you are developing new analysis or plotting code, usually one of these data sets is useful for testing it.

Another package with several useful data sets is `mlbench`. It contains data sets for machine learning benchmarks, so these data sets are aimed at testing how new methods perform on known data sets. This package is not distributed together with R, but you can install it, load it, and get a list of the data sets within it like this:

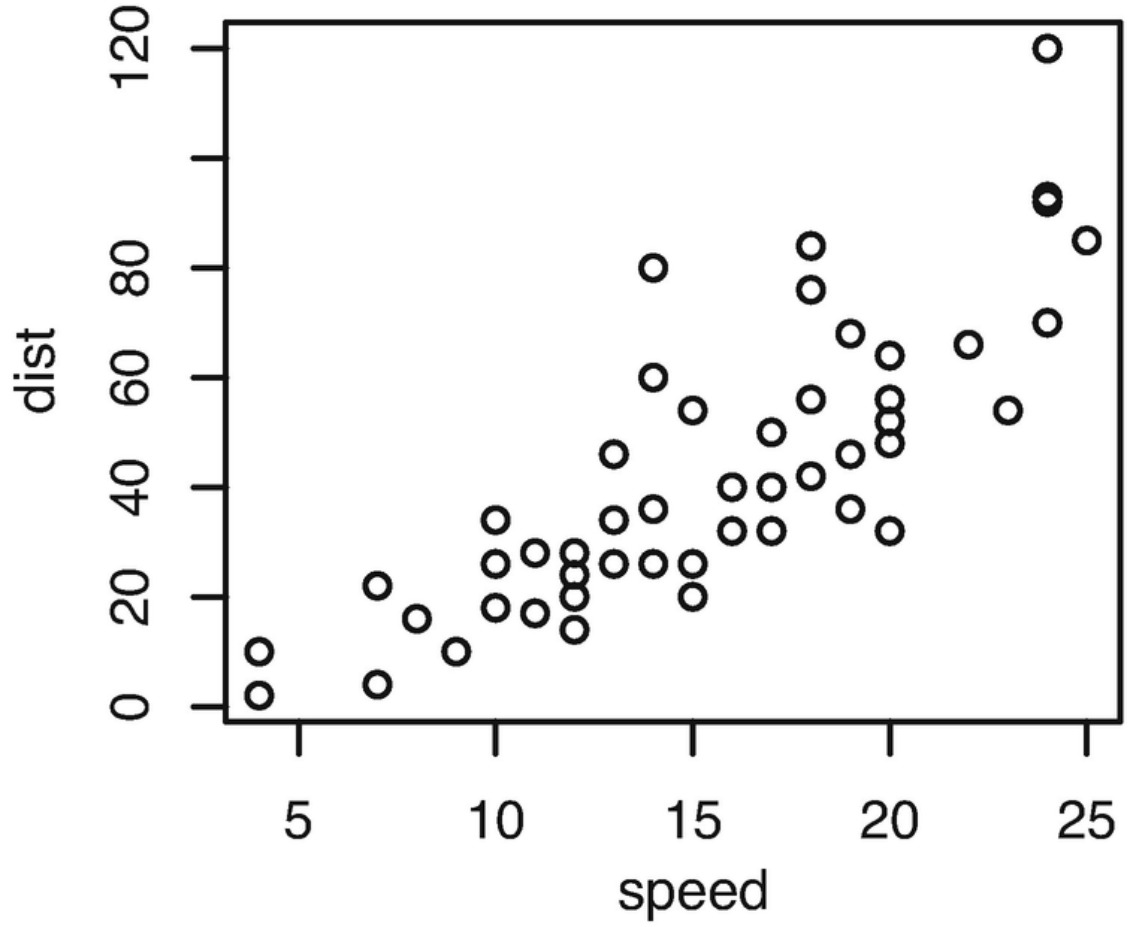

Figure 3-1 Plot of the cars data set

```
install.packages("mlbench")
library(mlbench)
library(help = "mlbench")
```

In this book, I will use data from one of those two packages when giving examples of data analyses.

The packages are convenient for me for giving examples, and if you are developing new functionality for R, they are suitable for testing, but if you are interested in data analysis, presumably you are interested in your own data, and there they are of course useless. You need to know how to get your own data into R. We get to that shortly, but first I want to say a few words about how you can examine a data set and get a quick overview.

Quickly Reviewing Data

Earlier, I have already used the function `head()`. This function shows the first *n* lines of a data frame where *n* is an option with default 6. You can use another *n* to get more or less:

```
cars |> head(3)
##   speed dist
## 1     4    2
## 2     4   10
## 3     7    4
```

The similar function `tail()` gives you the last *n* lines:

```
cars %>% tail(3)
##    speed dist
## 48    24   93
## 49    24  120
## 50    25   85
```

To get summary statistics for all the columns in a data frame, you can use the `summary()` function:

```
cars %>% summary
##     speed           dist
##  Min.   : 4.0  Min.   :  2.00
##  1st Qu.:12.0  1st Qu.: 26.00
##  Median :15.0  Median : 36.00
##  Mean   :15.4  Mean   : 42.98
##  3rd Qu.:19.0  3rd Qu.: 56.00
##  Max.   :25.0  Max.   :120.00
```

It isn't that exciting for the `cars` data set, so let us see it on another built-in data set:

```
data(iris)
iris |> summary()
##   Sepal.Length   Sepal.Width    Petal.Length
##  Min.   :4.300  Min.   :2.000  Min.   :1.000
##  1st Qu.:5.100  1st Qu.:2.800  1st Qu.:1.600
##  Median :5.800  Median :3.000  Median :4.350
##  Mean   :5.843  Mean   :3.057  Mean   :3.758
##  3rd Qu.:6.400  3rd Qu.:3.300  3rd Qu.:5.100
##  Max.   :7.900  Max.   :4.400  Max.   :6.900
##   Petal.Width          Species
##  Min.   :0.100  setosa    :50
##  1st Qu.:0.300  versicolor:50
##  Median :1.300  virginica :50
##  Mean   :1.199
```

```
##  3rd Qu.:1.800
##  Max.   :2.500
```

The summary you get depends on the types the columns have. Numerical data is summarized by their quantiles, while categorical and boolean data are summarized by counts of each category or TRUE/FALSE values. In the `iris` data set, there is one column, `Species`, that is categorical, and its summary is the count of each level.

To see the type of each column, you can use the `str()` function. This gives you the structure of a data type and is much more general than we need here, but it does give you an overview of the types of columns in a data frame and is very useful for that:

```
data(iris)
iris |> str()
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 ..
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 ..
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0...
##  $ Species     : Factor w/ 3 levels "setosa","v"..
```

Reading Data

There are several packages for reading data in different file formats, from Excel to JSON to XML and so on. If you have data in a particular format, try to Google for how to read it into R. If it is a standard data format, the chances are that there is a package that can help you.

Quite often, though, data is available in a text table of some kind. Most tools can import and export those. R has plenty of built-in functions for reading such data. Use

```
?read.table
```

to get a list of them. These functions are all variations of the `read.table()` function, just using different default options. For instance, while `read.table()` assumes that the data is given in whitespace-separated columns, the `read.csv()` function assumes that the data is repre-

sented as comma-separated values, so the difference between the two functions is in what they consider being separating data columns.

The `read.table()` function takes a lot of arguments. These are used to adjust it to the specific details of the text file you are reading. (The other functions take the same arguments, they just have different defaults.) The options I find I use the most are these:

- `header`: This is a boolean value telling the function whether it should consider the first line in the input file a header line. If set to true, it uses the first line to set the column names of the data frame it constructs; if it is set to false, the first line is interpreted as the first row in the data frame.
- `col.names`: If the first line is not used to specify the header, you can use this option to name the columns. You need to give it a vector of strings with a string for each column in the input.
- `dec`: This is the decimal point used in numbers. I get spreadsheets that use both "." and "," for decimal points, so this is an important parameter for me. How important it will be for you probably depends on how many nationalities you collaborate with.
- `comment.char`: By default, the function assumes that "#" is the start of a comment and ignores the rest of a line when it sees it. If "#" is actually used in your data, you need to change this. The same goes if comments are indicated with a different symbol.
- `colClasses`: This lets you specify which type each column should have, so here you can specify that some columns should be factors, and others should be strings. You have to specify all columns, though, which is cumbersome and somewhat annoying since R, in general, is pretty good at determining the right types for a column. The option will only take you so far in any case. You can tell it that a column should be an ordered factor but not what the levels should be and such. I mainly use it for specifying which columns should be factors and which should be strings, but using it will also speed up the function for large data sets since R then doesn't have to figure out the column types itself.

For reading in tables of data, `read.table()` and friends will usually get you there with the right options. If you are having problems reading data, check the documentation carefully to see if you cannot tweak the functions to get the data loaded. It isn't always possible, but it usually is. When it really isn't, I usually give up and write a script in another language to format the data into a form I can load into R. For raw text processing, R isn't really the right tool, and rather than forcing all steps in an analysis into R, I will be pragmatic and choose the best tools for the task, and R isn't always it. But before taking drastic measures, and go programming in another language, you should carefully check if you cannot tweak one of the `read.table()` functions first.

Examples of Reading and Formatting Data Sets

Rather than discussing the import of data in the abstract, let us now see a couple of examples of how data can be read in and formatted.

Breast Cancer Data set

As a first example of reading data from a text file, we consider the `BreastCancer` data set from `mlbench`. Then we have something to compare our results with. The first couple of lines from this data set are

```
library(mlbench)
data(BreastCancer)
BreastCancer %>% head(3)
##        Id Cl.thickness Cell.size Cell.shape
## 1 1000025            5         1          1
## 2 1002945            5         4          4
## 3 1015425            3         1          1
##   Marg.adhesion Epith.c.size Bare.nuclei
## 1             1            2           1
## 2             5            7          10
## 3             1            2           2
##   Bl.cromatin Normal.nucleoli Mitoses  Class
## 1           3               1       1 benign
## 2           3               2       1 benign
## 3           3               1       1 benign
```

The data can be found at **https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Original)** where there is also a description of the data. The URL to the actual data is **https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/breast-cancer-wisconsin.data**, but since this URL is too long to fit on the pages of this book, I have saved it in a variable, `data_url`, that I will use in the following code. To run the code yourself, you simply need to set the variable to the URL:

```
data_url <- "https://..."
```

To get the data downloaded, we could go to the URL and save the file. Explicitly downloading data outside of our R code has pros and cons. It is pretty simple, and we can have a look at the data before we start parsing it, but on the other hand, it gives us a step in the analysis workflow that is not automatically reproducible. Even if the URL is described in our documentation and at a link that doesn't change over time, it is a manual step in the workflow—and a step that people could make mistakes in.

Instead, I am going to read the data directly from the URL. Of course, this is also a risky step in a workflow because I am not in control of the server the data is on, and I cannot guarantee that the data will always be there and that it won't change over time. It is a bit of a risk either way. I will usually add the code to my workflow for downloading the data, but I will also store the data in a file. If I leave the code for downloading the data and saving it to my local disk in a cached Markdown chunk, it will only be run the one time I need it.

I can read the data and get it as a vector of lines using the `readLines()` function. I can always use that to scan the first one or two lines to see what the file looks like:

```
lines <- readLines(data_url) lines[1:5]
## [1] "1000025,5,1,1,1,2,1,3,1,1,2"
## [2] "1002945,5,4,4,5,7,10,3,2,1,2"
## [3] "1015425,3,1,1,1,2,2,3,1,1,2"
## [4] "1016277,6,8,8,1,3,4,3,7,1,2"
## [5] "1017023,4,1,1,3,2,1,3,1,1,2"
```

For this data, it seems to be a comma-separated values file without a header line. So I save the data with the ".csv" suffix. None of the functions for writing or

reading data in R cares about the suffixes, but it is easier for myself to remember what the file contains that way:

```
writeLines(lines, con = "data/raw-breast-cancer.csv")
```

For that function to succeed, I first need to make a `data/` directory. I suggest you have a `data/` directory for all your projects, always, since you want your directories and files structured when you are working on a project.

The file I just wrote to disk can then read in using the `read.csv()` function:

```
raw_breast_cancer <- read.csv("data/raw-breast-cancer.csv")
raw_breast_cancer |> head(3)
##   X1000025 X5 X1 X1.1 X1.2 X2 X1.3 X3 X1.4 X1.5
## 1  1002945  5  4    4    5  7   10  3    2    1
## 2  1015425  3  1    1    1  2    2  3    1    1
## 3  1016277  6  8    8    1  3    4  3    7    1
##   X2.1
## 1    2
## 2    2
## 3    2
```

Of course, I wouldn't write exactly these steps into a workflow. Once I have discovered that the data at the end of the URL is a ".csv" file, I would just read it directly from the URL:

```
raw_breast_cancer <- read.csv(data_url)
raw_breast_cancer |> head(3)
##   X1000025 X5 X1 X1.1 X1.2 X2 X1.3 X3 X1.4 X1.5
## 1  1002945  5  4    4    5  7   10  3    2    1
## 2  1015425  3  1    1    1  2    2  3    1    1
## 3  1016277  6  8    8    1  3    4  3    7    1
##   X2.1
## 1    2
## 2    2
## 3    2
```

The good news is that this data looks similar to the `BreastCancer` data. The bad news is that it appears that the first line in `BreastCancer` seems to have been turned into column names in `raw_breast_cancer`. The `read.csv()` function interpreted the first line as a header. We can fix this using the `header` parameter:

```
raw_breast_cancer <- read.csv(data_url, header = FALSE)
raw_breast_cancer |> head(3)
```

```
##        V1 V2 V3 V4 V5 V6 V7 V8 V9 V10 V11
## 1 1000025  5  1  1  1  2  1  3  1   1   2
## 2 1002945  5  4  4  5  7 10  3  2   1   2
## 3 1015425  3  1  1  1  2  2  3  1   1   2
```

Now the first line is no longer interpreted as header names. That is good, but the names we actually get are not that informative about what the columns contain.

If you read the description of the data from the website, you can see what each column is and choose names that are appropriate. I am going to cheat here and just take the names from the `BreastCancer` data set.

I can set the names explicitly like this:

```
names(raw_breast_cancer) <- names(BreastCancer)
raw_breast_cancer |> head(3)
##        Id Cl.thickness Cell.size Cell.shape
## 1 1000025            5         1          1
## 2 1002945            5         4          4
## 3 1015425            3         1          1
##  Marg.adhesion Epith.c.size Bare.nuclei
## 1             1            2           1
## 2             5            7          10
## 3             1            2           2
##  Bl.cromatin Normal.nucleoli Mitoses Class
## 1           3               1       1     2
## 2           3               2       1     2
## 3           3               1       1     2
```

or I could set them where I load the data:

```
raw_breast_cancer <- read.csv(data_url, header = FALSE,
                              col.names =
names(BreastCancer))
raw_breast_cancer |> head(3)
##        Id Cl.thickness Cell.size Cell.shape
## 1 1000025            5         1          1
## 2 1002945            5         4          4
## 3 1015425            3         1          1
##   Marg.adhesion Epith.c.size Bare.nuclei
## 1             1            2           1
## 2             5            7          10
```

```
## 3             1            2           2
##   Bl.cromatin Normal.nucleoli Mitoses Class
## 1           3               1       1     2
## 2           3               2       1     2
## 3           3               1       1     2
```

Okay, we are getting somewhere. The `Class` column is not right. It encodes the classes as numbers (the web page documentation specifies 2 for benign and 4 for malignant), but in R it would be more appropriate with a factor.

We can translate the numbers into a factor by first translating the numbers into strings and then the strings into factors. I don't like modifying the original data—even if I have it in a file—so I am going to copy it first and then do the modifications:

```
formatted_breast_cancer <- raw_breast_cancer
```

It is easy enough to map the numbers to strings using `ifelse()`:

```
map_class <- function(x) {
    ifelse(x == 2, "bening",
    ifelse(x == 4, "malignant",
           NA))
}
mapped <- formatted_breast_cancer$Class %>% map_class
mapped |> table()
## mapped
##    bening malignant
##       458       241
```

I could have made it simpler with

```
map_class <- function(x) {
ifelse(x == 2, "bening", "malignant")
}
mapped <- formatted_breast_cancer$Class %>% map_class
mapped |> table()
## mapped
##   bening malignant
##      458       241
```

since 2 and 4 are the only numbers in the data

```
formatted_breast_cancer$Class |> unique()
## [1] 2 4
```

but it is always a little risky to assume that there are no unexpected values, so I always prefer to have "weird values" as something I handle explicitly by setting it to NA.

Nested `ifelse()` are easy enough to program, but if there are many different possible values, it also becomes somewhat cumbersome. Another option is to use a table to map between values. To avoid confusion between a table as the one we are going to implement and the function `table()`, which counts how many times a given value appears in a vector, I am going to call the table we create a dictionary. A dictionary is a table where you can look up words, and that is what we are implementing.

For this, we can use named values in a vector. Remember that we can index in a vector both using numbers and using names.

You can create a vector where you use names as the indices. Use the keys you want to map from as the indices and the names you want as results as the values. We want to map from numbers to strings which pose a small problem. If we index into a vector with numbers, R will think we want to get positions in the vector. If we make the vector `v <- c(2 = "benign", 4 = "malignant")`—which we can't, it is a syntax error and for good reasons—then how should `v[2]` be interpreted? Do we want the value at index 2, `"malignant"`, or the value that has key 2, `"benign"`? When we use a vector as a table, we need to have strings as keys. That also means that the numbers in the vector we want to map from should be converted to strings before we look up in the dictionary. The code looks like this:

```
dict <- c("2" = "benign", "4" = "malignant")
map_class <- function(x) dict[as.character(x)]
mapped <- formatted_breast_cancer$Class |> map_class()
mapped |> table()
## mapped
##    benign malignant
##       458       241
```

That worked fine, but if we look at the actual vector instead of summarizing it, we will see that it looks a little strange:

```
mapped[1:5]
##         2         2         2         2         2
```

```
## "benign" "benign" "benign" "benign" "benign"
```

This is because when we create a vector by mapping in this way, we preserve the names of the values. Remember that the dictionary we made to map our keys to values has the keys as names; these names are passed on to the resulting vector. We can get rid of them using the `unname()` function:

```
library(magrittr)
mapped %<>% unname
mapped[1:5]
## [1] "benign" "benign" "benign" "benign" "benign"
```

Here, I used the `magrittr %<>%` operator to both pipe and rename `mapped`; alternatively, we could have used `mapped <- mapped %>% unname` or `mapped <- mapped |> unname()`.

You don't need to remove these names, they are not doing any harm in themselves, but some data manipulation can be slower when your data is dragging names along.

Now we just need to translate this vector of strings into a factor, and we will have our `Class` column.

The entire reading of data and formatting can be done like this:

```
# Download data and put it in a variable
raw_breast_cancer <- read.csv(
  data_url, header = FALSE,
  col.names = names(BreastCancer))
# Get a copy of the raw data that we can transform
formatted_breast_cancer <- raw_breast_cancer
# Reformat the Class variable
formatted_breast_cancer$Class <-
  formatted_breast_cancer$Class %>% {
    c("2" = "benign", "4" = "malignant")[as.character(.)]
  } |> factor(levels = c("benign", "malignant"))
```

In the last statement, we use the `%>%` operator so we can put an expression in curly braces. In there, the incoming class is the "." that we translate to a character and then use to look up the name that we want. Then we pipe the names through `factor()` to get the factor that we went for. We can use either `%>%` or `|>` here. We don't have to remove the names

with `unname()` when we put the result back into `formatted_breast_cancer`, so we don't bother.

It is not strictly necessary to specify the levels in the `factor()` call, but I prefer always to do so explicitly. If there is an unexpected string in the input to `factor()`, it would end up being one of the levels, and I wouldn't know about it until much later. Specifying the levels explicitly alleviates that problem.

Now, you don't want to spend time parsing input data files all the time, so I would recommend putting all the code you write to read in data and transforming it into the form you want in a cached code chunk in an R Markup document. This way, you will only evaluate the code when you change it.

You can also explicitly save data using the `save()` function:

```
formatted_breast_cancer %>%
    save(file = "data/formatted-breast-cancer.rda")
```

Here, I use the suffix `".rda"` for the data. It stands for R data, and your computer will probably recognize it. If you click a file with that suffix, it will be opened in RStudio (or whatever tool you use to work on R). The actual R functions for saving and loading data do not care what suffix you use, but it is easier to recognize the files for what they are if you stick to a fixed suffix.

The data is saved together with the name of the data frame, so when you load it again, using the `load()` function, you don't have to assign the loaded data to a variable. It will be loaded into the name you used when you saved the data:

```
load("data/formatted-breast-cancer.rda")
```

This is both good and bad. I would probably have preferred to control which name the data is assigned to so I have explicit control over the variables in my code, but `save()` and `load()` are designed to save more than one variable, so this is how they work.

I personally do not use these functions that much. I prefer to write my analysis pipelines in Markdown documents, and there it is easier just to

cache the import code.

Boston Housing Data Set

For the second example of loading data, we take another data set from the `mlbench` package, the `BostonHousing` data, which contains information about crime rates and some explanatory variables we can use to predict crime rates:

```
library(mlbench)
data(BostonHousing)
str(BostonHousing)
## 'data.frame':    506 obs. of  14 variables:
##  $ crim   : num  0.00632 0.02731 0.02729 0.03237..
##  $ zn     : num  18 0 0 0 0 0 12.5 12.5 ...
##  $ indus  : num  2.31 7.07 7.07 2.18 2.18 2.18 7..
##  $ chas   : Factor w/ 2 levels "0","1": 1 1 1 1 ..
##  $ nox    : num  0.538 0.469 0.469 0.458 0.458 0..
##  $ rm     : num  6.58 6.42 7.18 7 ...
##  $ age    : num  65.2 78.9 61.1 45.8 54.2 58.7 6..
##  $ dis    : num  4.09 4.97 4.97 6.06 ...
##  $ rad    : num  1 2 2 3 3 3 5 5 ...
##  $ tax    : num  296 242 242 222 222 222 311 311..
##  $ ptratio: num  15.3 17.8 17.8 18.7 18.7 18.7 1..
##  $ b      : num  397 397 393 395 ...
##  $ lstat  : num  4.98 9.14 4.03 2.94 ...
##  $ medv   : num  24 21.6 34.7 33.4 36.2 28.7 22...
```

As before, the link to the actual data is pretty long, so I will give you a tinyURL to it, **http://tinyurl.com/zq2u8vx**, and I have saved the original URL in the variable `data_url`.

I have already looked at the file at the end of the URL and seen that it consists of whitespace-separated columns of data, so the function we need to load it is `read.table()`:

```
boston_housing <- read.table(data_url)
str(boston_housing)
## 'data.frame':   506 obs. of  14 variables:
##  $ V1 : num  0.00632 0.02731 0.02729 0.03237 ...
##  $ V2 : num  18 0 0 0 0 0 12.5 12.5 ...
##  $ V3 : num  2.31 7.07 7.07 2.18 2.18 2.18 7.87 ..
##  $ V4 : int  0 0 0 0 0 0 0 0 ...
```

```
##  $ V5 : num  0.538 0.469 0.469 0.458 0.458 0.458..
##  $ V6 : num  6.58 6.42 7.18 7 ...
##  $ V7 : num  65.2 78.9 61.1 45.8 54.2 58.7 66.6 ..
##  $ V8 : num  4.09 4.97 4.97 6.06 ...
##  $ V9 : int  1 2 2 3 3 3 5 5 ...
##  $ V10: num  296 242 242 222 222 222 311 311 ...
##  $ V11: num  15.3 17.8 17.8 18.7 18.7 18.7 15.2 ..
##  $ V12: num  397 397 393 395 ...
##  $ V13: num  4.98 9.14 4.03 2.94 ...
##  $ V14: num  24 21.6 34.7 33.4 36.2 28.7 22.9 27..
```

If we compare the data that we have loaded with the data from `mlbench`

```
str(BostonHousing)
## 'data.frame':    506 obs. of  14 variables:
##  $ crim   : num  0.00632 0.02731 0.02729 0.03237..
##  $ zn     : num  18 0 0 0 0 0 12.5 12.5 ...
##  $ indus  : num  2.31 7.07 7.07 2.18 2.18 2.18 7..
##  $ chas   : Factor w/ 2 levels "0","1": 1 1 1 1 ..
##  $ nox    : num  0.538 0.469 0.469 0.458 0.458 0..
##  $ rm     : num  6.58 6.42 7.18 7 ...
##  $ age    : num  65.2 78.9 61.1 45.8 54.2 58.7 6..
##  $ dis    : num  4.09 4.97 4.97 6.06 ...
##  $ rad    : num  1 2 2 3 3 3 5 5 ...
##  $ tax    : num  296 242 242 222 222 222 311 311..
##  $ ptratio: num  15.3 17.8 17.8 18.7 18.7 18.7 1..
##  $ b      : num  397 397 393 395 ...
##  $ lstat  : num  4.98 9.14 4.03 2.94 ...
##  $ medv   : num  24 21.6 34.7 33.4 36.2 28.7 22...
```

we see that we have integers and numeric data in our imported data but that it should be a factor for the `chas` variable and numeric for all the rest. We can use the `colClasses` parameter for `read.table()` to fix this. We just need to make a vector of strings for the classes, a vector that is `"numeric"` for all columns except for the `"chas"` column, which should be `"factor"`:

```
col_classes <- rep("numeric", length(BostonHousing))
col_classes[which("chas" == names(BostonHousing))] <-
"factor"
```

We should also name the columns, but again we can cheat and get the names from `BostonHousing`:

```
boston_housing <- read.table(data_url,
                             col.names =
names(BostonHousing),
```

```
                                colClasses = col_classes)
str(boston_housing)
## 'data.frame':    506 obs. of  14 variables:
##  $ crim   : num  0.00632 0.02731 0.02729 0.03237..
##  $ zn     : num  18 0 0 0 0 0 12.5 12.5 ...
##  $ indus  : num  2.31 7.07 7.07 2.18 2.18 2.18 7..
##  $ chas   : Factor w/ 2 levels "0","1": 1 1 1 1 ..
##  $ nox    : num  0.538 0.469 0.469 0.458 0.458 0..
##  $ rm     : num  6.58 6.42 7.18 7 ...
##  $ age    : num  65.2 78.9 61.1 45.8 54.2 58.7 6..
##  $ dis    : num  4.09 4.97 4.97 6.06 ...
##  $ rad    : num  1 2 2 3 3 3 5 5 ...
##  $ tax    : num  296 242 242 222 222 222 311 311..
##  $ ptratio: num  15.3 17.8 17.8 18.7 18.7 18.7 1..
##  $ b      : num  397 397 393 395 ...
##  $ lstat  : num  4.98 9.14 4.03 2.94 ...
##  $ medv   : num  24 21.6 34.7 33.4 36.2 28.7 22...
```

The levels in the `"chas"` factor are “0” and “1.” It is not really good levels as they are very easily confused with numbers—they will print like numbers—but they are not. The numerical values in the factor are actually 1 for “0” and 2 for “1”, so that can be confusing. But it is the same levels as the `mlbench` data frame, so I will just leave it the way it is as well.

The `readr` Package

The `read.table()` class of functions will usually get you to where you want to go with importing data. I use these in almost all my work. But there is a package aimed at importing data that tries to both speed up the importing and being more consistent in how data is imported, so I think I should mention it.

That package is the `readr` package:

```
library(readr)
```

It implements the same class of import functions as the built-in functions. It just uses underscores except for dots in the function names. So where you would use `read.table()`, the `readr` package gives you

`read_table()`. Similarly, it gives you `read_csv()` as a substitute for `read.csv()`.

The `readr` package has different defaults for how to read data. Other than that, its main call to fame is being faster than the built-in R functions. This shouldn't concern you much if you put your data import code in a cached code chunk, and in any case if loading data is an issue, you need to read Chapter 5. The functions from `readr` do not return data frames but the tibble data structure we briefly discussed before. For most purposes, this makes no difference, so we can still treat the loaded data the same way.

Let us look at how to import data using the functions in the package. We return to the breast cancer data we imported earlier. We downloaded the breast cancer data and put it in a file called `"data/raw-breast-cancer.csv"`, so we can try to read it from that file. Obviously, since it is a CSV file, we will use the `read_csv()` function:

```
raw_breast_cancer <- read_csv("data/raw-breast-cancer.csv",
                              show_col_types = FALSE)
## New names:
## • `1` -> `1...3`
## • `1` -> `1...4`
## • `1` -> `1...5`
## • `2` -> `2...6`
## • `1` -> `1...7`
## • `1` -> `1...9`
## • `1` -> `1...10`
## • `2` -> `2...11`
raw_breast_cancer %>% head(3)
## # A tibble: 3 × 11
##   `1000025`   `5` `1...3` `1...4` `1...5` `2...6`
##       <dbl> <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
## 1   1002945     5       4       4       5       7
## 2   1015425     3       1       1       1       2
## 3   1016277     6       8       8       1       3
## # ... with 5 more variables: `1...7` <chr>,
## #   `3` <dbl>, `1...9` <dbl>, `1...10` <dbl>,
## #   `2...11` <dbl>
```

(The `show_col_types = FALSE` option just says that we don't want to see inferred column types; it removes a bunch of output lines that we aren't interested in here. If you want to see what the function prints, you can try it out for yourself.)

The function works similar to the `read.csv()` function and interprets the first line as the column names. The warning we get, and the weird column names, is because of that. We don't want this, but this function doesn't have the option to tell it that the first line is not the names of the columns. Instead, we can inform it what the names of the columns are, and then it will read the first line as actual data:

```
raw_breast_cancer <- read_csv("data/raw-breast-cancer.csv",
                              col_names =
names(BreastCancer),
                              show_col_types = FALSE)
raw_breast_cancer %>% head(3)
## # A tibble: 3 × 11
##        Id Cl.thickness Cell.size Cell.shape
##     <dbl>        <dbl>     <dbl>      <dbl>
## 1 1000025            5         1          1
## 2 1002945            5         4          4
## 3 1015425            3         1          1
## # ... with 7 more variables: Marg.adhesion <dbl>,
## #  Epith.c.size <dbl>, Bare.nuclei <chr>,
## #  Bl.cromatin <dbl>, Normal.nucleoli <dbl>,
## #  Mitoses <dbl>, Class <dbl>
```

Which functions you use to import data doesn't much matter. You can use the built-in functions or the `readr` functions. It is all up to you.

Manipulating Data with `dplyr`

Data frames are ideal for representing data where each row is an observation of different parameters you want to fit in models. Nearly all packages that implement statistical models or machine learning algorithms in R work on data frames. But to actually manipulate a data frame, you often have to write a lot of code to filter data, rearrange data, summarize it in various ways, and such. A few years ago, manipulating data frames re-

quired a lot more programming than actually analyzing data. That has improved dramatically with the `dplyr` package (pronounced "d plier" where "plier" is pronounced as "pliers").

This package provides a number of convenient functions that let you modify data frames in various ways and string them together in pipes using the `%>%` or `|>` operator. If you import `dplyr`, you get a large selection of functions that let you build pipelines for data frame manipulation using pipelines.

Earlier, we have formatted data by manipulating data columns by directly assigning to them, like we did with `formatted_breast_cancer$Class <- formatted_breast_cancer$Class %>% ...` statements. There is nothing inherently wrong with this approach, but it breaks the idea of using pipelines to send data through a series of transformations, if we have to assign the results to old or new columns from time to time. With `dplyr`, you get functions that manipulate data frames as a whole, but where you can still modify, add, or remove columns as you please.

The transformations we did on the breast cancer set you can also do with `dplyr`, going directly from the raw data set to the formatted data, without first creating the data frame for the formatted data and then updating it. The way you would do it could look like this:

```
library(dplyr)
# Download data and put it in a variable
raw_breast_cancer <- read.csv("data/raw-breast-cancer.csv",
                              header = FALSE,
                              col.names =
names(BreastCancer))
# Reformat the Class column as a benign/malignant factor
formatted_breast_cancer <- raw_breast_cancer |>
  mutate(
    Class =
        case_when(Class == 2 ~ "benign", Class == 4 ~
"malignant") |>
        factor(levels = c("benign", "malignant"))
```

```
)
```

We still get the raw data by downloading it, we don't have any other choice, but then we take this raw data and pipe it through a function called `mutate`: `raw_breast_cancer |> mutate(...)` to change the data frame. The `mutate()` function (see later) can modify the columns in the data frame, and in the preceding expression, that is what we do. With `raw_breast_cancer |> mutate(Class = ...)`, we say that we want to change the `Class` column to what is to the right of the `=` inside `mutate`. You can modify more than one column with the same `mutate`, but we only change one. The right-hand side of `=` is a bit complex because we are doing all the work we did earlier, translating 2 and 4 into the strings `"benign"` and `"malignant"` and then making a factor out of that. It is not that bad if we split the operation into two pieces, though. First, we use a `dplyr` function called `case_when`. It works a bit like `ifelse` but is more general. Its input is a sequence of so-called formulas—expressions on the form `lhs ~ rhs`—where the left-hand side (lhs) should be a boolean expression and the right-hand side (rhs) the value you want if the left-hand side is `TRUE`. Here, the expressions are simple: we test if `Class` is two or four and return `"benign"` or `"malignant"` accordingly. This maps the 2/4 encoding to strings, and we then pipe that through `factor` to get a factor out of that. The result is written to the `Class` column. Notice that we can refer directly to the values in a column, here `Class`, using just the column name. We couldn't do that in the preceding section where we worked with the data frame's `Class` column, where we had to write `raw_breast_cancer$Class` to get at it. The functions in `dplyr` do some magic that lets you treat data frame columns as if they were variables, and this makes the expressions you have to write a bit more manageable.

Some Useful `dplyr` Functions

I will not be able to go through all of the `dplyr` functionality in this chapter. In any case, it is updated frequently enough that, by the time you read this, there is probably more functionality than at the time I write. So you will need to check the documentation for the package.

The following are just the functions I use on a regular basis. They all take a data frame or equivalent as the first argument, so they work perfectly with pipelines. When I say "data frame equivalent," I mean that they take as an argument anything that works like a data frame. Quite often, there are better representations of data frames than the built-in data structure. For large data sets, it is often better to use a different representation than the built-in data frame, something we will return to in Chapter 5. Some alternative data structures are better because they can work with data on disk—R's data frames have to be loaded into memory, and others are just faster to do some operations on. Or maybe they just print better. If you write the name of a data frame into the R terminal, it will print the entire data. Other representations will automatically give you just the head of the data.

The `dplyr` package has several representations. The tibbles, mentioned a few times in the previous chapters, are a favorite of mine that I use just because I prefer the output when I print such tables. You can translate a base data frame into a tibble using the `as_tibble` function:

```
iris %>% as_tibble()
## # A tibble: 150 × 5
##    Sepal.Length Sepal.Width Petal.Length
##           <dbl>       <dbl>        <dbl>
##  1          5.1         3.5          1.4
##  2          4.9         3            1.4
##  3          4.7         3.2          1.3
##  4          4.6         3.1          1.5
##  5          5           3.6          1.4
##  6          5.4         3.9          1.7
##  7          4.6         3.4          1.4
##  8          5           3.4          1.5
##  9          4.4         2.9          1.4
## 10          4.9         3.1          1.5
## # ... with 140 more rows, and 2 more variables:
## #   Petal.Width <dbl>, Species <fct>
```

It only prints the first ten rows, and it doesn't print all columns. The output is a little easier to read than if you get the entire data frame.

Anyway, let us get to the `dplyr` functions.

select—Pick selected columns and get rid of the rest.

The `select()` function selects columns of the data frame. It is equivalent to indexing columns in the data.

You can use it to pick out a single column:

```
iris %>% as_tibble() %>% select(Petal.Width) %>% head(3)
## # A tibble: 3 × 1
##   Petal.Width
##         <dbl>
## 1         0.2
## 2         0.2
## 3         0.2
```

Or several columns:

```
iris %>% as_tibble() %>%
  select(Sepal.Width, Petal.Length) %>% head(3)
## # A tibble: 3 × 2
##   Sepal.Width Petal.Length
##         <dbl>        <dbl>
## 1         3.5          1.4
## 2         3            1.4
## 3         3.2          1.3
```

You can even give it ranges of columns:

```
iris %>% as_tibble() %>%
  select(Sepal.Length:Petal.Length) %>% head(3)
## # A tibble: 3 × 3
##   Sepal.Length Sepal.Width Petal.Length
##          <dbl>       <dbl>        <dbl>
## 1          5.1         3.5          1.4
## 2          4.9         3            1.4
## 3          4.7         3.2          1.3
```

but how that works depends on the order the columns are in for the data frame, and it is not something I find all that useful.

I pipe `iris` through `as_tibble()` in these pipelines, because I like the formatting more. You don't have to, to use the `dplyr` functions, but if you don't, you are working on a base data frame instead of a tibble, and R will print your data slightly differently.

The real usefulness comes with pattern matching on column names. There are different ways to pick columns based on the column names:

```
iris |> as_tibble() |> select(starts_with("Petal")) |>
head(3)
## # A tibble: 3 × 2
##   Petal.Length Petal.Width
##          <dbl>       <dbl>
## 1          1.4         0.2
## 2          1.4         0.2
## 3          1.3         0.2
iris |> as_tibble() |> select(ends_with("Width")) |> head(3)
## # A tibble: 3 × 2
##   Sepal.Width Petal.Width
##         <dbl>       <dbl>
## 1         3.5         0.2
## 2         3           0.2
## 3         3.2         0.2
iris |> as_tibble() |> select(contains("etal")) |> head(3)
## # A tibble: 3 × 2
##   Petal.Length Petal.Width
##          <dbl>       <dbl>
## 1          1.4         0.2
## 2          1.4         0.2
## 3          1.3         0.2
iris |> as_tibble() |> select(matches(".t.")) |> head(3)
## # A tibble: 3 × 4
##   Sepal.Length Sepal.Width Petal.Length
##          <dbl>       <dbl>        <dbl>
## 1          5.1         3.5          1.4
## 2          4.9         3            1.4
## 3          4.7         3.2          1.3
## # ... with 1 more variable: Petal.Width <dbl>
```

The `matches` function searches for a regular expression and in this example will select any name that contains a `t` except if it is the first or last letter.

Check out the documentation for `dplyr` to see which options you have for selecting columns.

You can also use `select()` to remove columns. The preceding examples select the columns you want to include, but if you use "–" before the selection criteria, you will remove, instead of include, the columns you specify:

```
iris %>% as_tibble() %>%
  select(-starts_with("Petal")) %>% head(3)
## # A tibble: 3 × 3
##   Sepal.Length Sepal.Width Species
##          <dbl>       <dbl> <fct>
## 1          5.1         3.5 setosa
## 2          4.9         3   setosa
## 3          4.7         3.2 setosa
```

mutate—Add computed values to your data frame.

The `mutate()` function lets you add a column to your data frame by specifying an expression for how to compute it:

```
iris %>% as_tibble() %>%
  mutate(Petal.Width.plus.Length = Petal.Width +
Petal.Length) %>%
  select(Species, Petal.Width.plus.Length) %>%
  head(3)
## # A tibble: 3 × 2
##   Species Petal.Width.plus.Length
##   <fct>                     <dbl>
## 1 setosa                      1.6
## 2 setosa                      1.6
## 3 setosa                      1.5
```

You can add more columns than one by specifying them in the `mutate()` function:

```
iris %>% as_tibble() %>%
   mutate(Petal.Width.plus.Length = Petal.Width +
Petal.Length,
          Sepal.Width.plus.Length = Sepal.Width +
Sepal.Length) %>%
   select(Petal.Width.plus.Length, Sepal.Width.plus.Length)
%>%
   head(3)
## # A tibble: 3 × 2
##   Petal.Width.plus.Length Sepal.Width.plus.Length
##                     <dbl>                   <dbl>
## 1                     1.6                     8.6
```

```
## 2                    1.6                    7.9
## 3                    1.5                    7.9
```

but you could of course also just call `mutate()` several times in your pipeline.

transmute—Add computed values to your data frame and get rid of all other columns.

The `transmute()` function works just like the `mutate()` function, except it combines it with a `select()` so the result is a data frame that only contains the new columns you make:

```
iris %>% as_tibble() %>%
  transmute(Petal.Width.plus.Length = Petal.Width +
Petal.Length) %>%
  head(3)
## # A tibble: 3 × 1
##   Petal.Width.plus.Length
##                     <dbl>
## 1                     1.6
## 2                     1.6
## 3                     1.5
```

arrange—Reorder your data frame by sorting columns.

The `arrange()` function just reorders the data frame by sorting columns according to what you specify:

```
iris %>% as_tibble() %>%
  arrange(Sepal.Length) %>%
  head(3)
## # A tibble: 3 × 5
##   Sepal.Length Sepal.Width Petal.Length
##          <dbl>       <dbl>        <dbl>
## 1          4.3         3            1.1
## 2          4.4         2.9          1.4
## 3          4.4         3            1.3
## # ... with 2 more variables: Petal.Width <dbl>,
## #   Species <fct>
```

By default, it orders numerical values in increasing order, but you can ask for decreasing order using the `desc()` function:

```
iris %>% as_tibble() %>%
```

```
  arrange(desc(Sepal.Length)) %>%
  head(3)
## # A tibble: 3 × 5
##   Sepal.Length Sepal.Width Petal.Length
##          <dbl>       <dbl>        <dbl>
## 1          7.9         3.8          6.4
## 2          7.7         3.8          6.7
## 3          7.7         2.6          6.9
## # ... with 2 more variables: Petal.Width <dbl>,
## #   Species <fct>
```

filter—Pick selected rows and get rid of the rest.

The `filter()` function lets you pick out rows based on logical expressions. You give the function a predicate specifying what a row should satisfy to be included:

```
iris %>% as_tibble() %>%
  filter(Sepal.Length > 5) %>%
  head(3)
## # A tibble: 3 × 5
##   Sepal.Length Sepal.Width Petal.Length
##          <dbl>       <dbl>        <dbl>
## 1          5.1         3.5          1.4
## 2          5.4         3.9          1.7
## 3          5.4         3.7          1.5
## # ... with 2 more variables: Petal.Width <dbl>,
## #   Species <fct>
```

You can get as inventive as you want here with the logical expressions:

```
iris %>% as_tibble() %>%
  filter(Sepal.Length > 5 & Species == "virginica") %>%
  select(Species, Sepal.Length) %>%
  head(3)
## # A tibble: 3 × 2
##   Species      Sepal.Length
##   <fct>               <dbl>
##      1      virginica   6.3
##      2      virginica   5.8
##      3      virginica   7.1
```

group_by—Split your data into subtables based on values of some of the columns.

The `group_by()` function tells `dplyr` that you want to work on data separated into different subsets.

By itself, it isn't that useful. It just tells `dplyr` that, in future computations, it should consider different subsets of the data as separate data sets. It is used with the `summarise()` function where you want to compute summary statistics.

You can group by one or more variables; you just specify the columns you want to group by as separate arguments to the function. It works best when grouping by factors or discrete numbers; there isn't much fun in grouping by real numbers:

```
iris %>% as_tibble() %>% group_by(Species) %>% head(3)
## # A tibble: 3 × 5
## # Groups:   Species [1]
##   Sepal.Length Sepal.Width Petal.Length
##          <dbl>       <dbl>        <dbl>
## 1          5.1         3.5          1.4
## 2          4.9         3            1.4
## 3          4.7         3.2          1.3
## # ... with 2 more variables: Petal.Width <dbl>,
## #   Species <fct>
```

Not much is happening here. You have restructured the data frame such that there are groupings, but until you do something with the new data, there isn't much to see. The power of `group_by()` is the combination with the `summarise()` function.

summarise/summarize—Calculate summary statistics.

The spelling of this function depends on which side of the pond you are on. It is the same function regardless of how you spell it.

The `summarise()` function is used to compute summary statistics from your data frame. It lets you compute different statistics by expressing what you want to summarize. For example, you can ask for the mean of values:

```
iris %>%
  summarise(Mean.Petal.Length = mean(Petal.Length),
            Mean.Sepal.Length = mean(Sepal.Length))
```

```
##   Mean.Petal.Length Mean.Sepal.Length
## 1             3.758          5.843333
```

Where it is really powerful is in the combination with `group_by()`. There, you can split the data into different groups and compute the summaries for each group:

```
iris %>%
  group_by(Species) %>%
  summarise(Mean.Petal.Length = mean(Petal.Length))
## # A tibble: 3 × 2
##   Species    Mean.Petal.Length
##   <fct>                  <dbl>
## 1 setosa                  1.46
## 2 versicolor              4.26
## 3 virginica               5.55
```

Depending on your version of `dplyr`, you might get a warning here that `summarise` is "ungrouping" the output. This has to do with which groupings a data frame has when we give it to `summarise` and which groupings the output has. Until `dplyr` 1.0.0, there was only one option, which of course you couldn't change, but now there are four and a default behavior, which may or may not be what you want (but is probably there to at least make the function backward compatible with older code). If you get a warning, it is because you have a version of `dplyr` that has the new behavior, but that is still old enough to warn you that you are using the default behavior. It will then tell you that you can change this using the `.groups` argument.

The four behaviors we can get are

- `.groups = "drop_last"` removes the last grouping we introduced. If we are summarizing over some grouping, we end up with a table with one row per group, and there is no need to keep the group information for that any longer. This is the old default behavior.
- `.groups = "drop"` removes all groupings, so we have called `group_by` multiple times we lose all the groups; with `drop_last`, we would only lose the last grouping.
- `.groups = "keep"` keeps all the groups exactly as they are.

- `.groups = "rowwise"` makes each row in the output its own group.

You can check which columns in the data frame are part of a grouping using the `group_vars`, so we can see which groups the output of a `summarise` has for each of the four options. First, we make a grouping for the `iris` data and check the group variables:

```
grouped_iris <- iris %>% as_tibble() %>%
  group_by(Species, Petal.Length)
grouped_iris %>% group_vars()
## [1] "Species"      "Petal.Length"
```

With `drop_last`, the second grouping variable, `Petal.Length`, is removed, but the first is not; we are dropping the last variable and only the last:

```
grouped_iris %>%
  summarise(Mean.Petal.Length = mean(Petal.Length),
            .groups = "drop_last") %>%
  group_vars()
## [1] "Species"
```

If we use `drop` instead, all the grouping variables are removed, and `group_vars` gives us `character(0)` which is the empty string vector:

```
grouped_iris %>%
  summarise(Mean.Petal.Length = mean(Petal.Length),
            .groups = "drop") %>%
  group_vars()
## character(0)
```

If we use `keep`, none of the grouping variables are removed:

```
grouped_iris %>%
  summarise(Mean.Petal.Length = mean(Petal.Length),
            .groups = "keep") %>%
  group_vars()
## [1] "Species"      "Petal.Length"
```

At first glance, the `rowwise` option looks a lot like `keep`:

```
grouped_iris %>%
  summarise(Mean.Petal.Length = mean(Petal.Length),
            .groups = "rowwise") %>%
  group_vars()
## [1] "Species"      "Petal.Length"
```

It isn't the same thing, though. If we summarize with `rowwise`, we get a different kind of data structure, a so-called `rowwise_df` (a row-wise data frame), and we can see this if we ask for the "class" of the result:

```
grouped_iris %>%
  summarise(Mean.Petal.Length = mean(Petal.Length),
```

```
            .groups = "rowwise") %>%
  class()
## [1] "rowwise_df" "tbl_df"     "tbl"
## [4] "data.frame"
```

The class is a list of types, and we will see how this works in the second half of the book. The purpose of this rowwise data type is a different kind of manipulation, where we manipulate rows of data frames. They are beyond the scope of this book, and I will leave it at that.

For the `summarise()` function, the `.groups` option is only important if you do more than one summary. In most usage, you don't. You manipulate your data, then you compute some summary statistics, and then that is your result. In cases such as those, it doesn't matter if you keep or drop any grouping variables. If you don't care about the grouping in the result of a summary, you can turn the warning off using the option `dplyr.summarise.inform = FALSE`:

```
options(dplyr.summarise.inform = FALSE)
summary <- grouped_iris %>%
  summarise(Mean.Petal.Length = mean(Petal.Length))
```

Here, even though you didn't use `.groups`, you shouldn't get a warning.

If you turn the option on again, you should get the warning:

```
options(dplyr.summarise.inform = TRUE)
```

Although you usually don't care, it is safer to specify `.groups`. At least you have to think about whether you want the result to have any groups, and if you do, you have to consider if you get the right ones.

A summary function worth mentioning here is `n()` which just counts how many observations you have in a subset of your data:

```
iris %>%
  summarise(Observations = n())
##  Observations
## 1          150
```

Again, this is more interesting when combined with `group_by()`:

```
iris %>%
  group_by(Species) %>%
  summarise(Number.Of.Species = n(), .groups = "drop")
## # A tibble: 3 × 2
##   Species    Number.Of.Species
```

```
##   <fct>                <int>
## 1 setosa                  50
## 2 versicolor              50
## 3 virginica               50
```

You can combine summary statistics simply by specifying more than one in the `summary()` function:

```
iris %>%
  group_by(Species) %>%
  summarise(Number.Of.Samples = n(),
            Mean.Petal.Length = mean(Petal.Length),
            .groups = "drop")
## # A tibble: 3 × 3
##   Species  Number.Of.Samples Mean.Petal.Length
##   <fct>                <int>             <dbl>
## 1 setosa                  50              1.46
## 2 versicolor              50              4.26
## 3 virginica               50              5.55
```

Breast Cancer Data Manipulation

To get a little more feeling for how the `dplyr` package can help us explore data, let us see it in action.

Let us return to the breast cancer data. We start with the modifications we used to transform the raw data we imported from the CSV file (stored in the variable `raw_breast_cancer`). Using `dplyr` functions, we could create the `formatted_breast_cancer` data, as we saw earlier, like this:

```
formatted_breast_cancer <- raw_breast_cancer |>
  as_tibble() |>
  mutate(
    Class =
      case_when(Class == 2 ~ "benign", Class == 4 ~
"malignant") |>
      factor(levels = c("benign", "malignant"))
  )
```

I piped the raw data through `as_tibble` first, this time, because I prefer to work with tibbles. Otherwise, it is the same code as earlier.

We can check if things look the way they should using a `select` and a `head`:

```
formatted_breast_cancer |> select(Normal.nucleoli:Class) |>
head(5)
## # A tibble: 5 × 3
##   Normal.nucleoli Mitoses Class
##             <int>   <int> <fct>
## 1               1       1 benign
## 2               2       1 benign
## 3               1       1 benign
## 4               7       1 benign
## 5               1       1 benign
```

Now let us look a little at the actual data. This is a very crude analysis of the data we can do for exploratory purposes. It is not a proper analysis, but we will return to that in Chapter 6 later.

We could be interested in how the different parameters affect the response variable, the `Class` variable. For instance, is cell thickness different for benign and malignant tumors? To check that, we can group the data by the `Cell` parameter and look at the mean cell thickness:

```
formatted_breast_cancer %>%
  group_by(Class) %>%
  summarise(mean.thickness = mean(Cl.thickness), .groups =
"drop")
## # A tibble: 2 × 2
##   Class     mean.thickness
##   <fct>              <dbl>
## 1 benign              2.96
## 2 malignant           7.20
```

It looks like there is a difference. Now whether this difference is significant requires a proper test—after all, we are just comparing means here, and the variance could be huge. But just exploring the data, it gives us a hint that there might be something to work with here.

We could ask the same question for other variables, like cell size:

```
formatted_breast_cancer %>%
  group_by(Class) %>%
  summarise(mean.size = mean(Cell.size), .groups = "drop")
## # A tibble: 2 × 2
##   Class     mean.size
##   <fct>         <dbl>
```

```
## 1 benign         1.33
## 2 malignant      6.57
```

Another way of looking at this could be to count, for each cell size, how many benign tumors and how many malignant tumors we see. Here, we would need to group by both cell size and class and then count, and we would probably want to arrange the data so we get the information in order of increasing or decreasing cell size:

```
formatted_breast_cancer %>%
  arrange(Cell.size) %>%
  group_by(Cell.size, Class) %>%
  summarise(ClassCount = n(), .groups = "drop")
## # A tibble: 18 × 3
##    Cell.size Class   ClassCount
##        <int> <fct>        <int>
## 1          1 benign         380
## 2          1 malignant        4
## 3          2 benign          37
## 4          2 malignant        8
## 5          3 benign          27
## 6          3 malignant       25
## 7          4 benign           9
## 8          4 malignant       31
## 9          5 malignant       30
## 10         6 benign           2
## 11         6 malignant       25
## 12         7 benign           1
## 13         7 malignant       18
## 14         8 benign           1
## 15         8 malignant       28
## 16         9 benign           1
## 17         9 malignant        5
## 18        10 malignant       67
```

Here again, we get some useful information. It looks like there are more benign tumors compared to malignant tumors when the cell size is small and more malignant tumors when the cell size is large. Again something we can start to work from when we later want to build statistical models.

This kind of grouping only works because the cell size is measured as discrete numbers. It wouldn't be helpful to group by a floating-point number. There, plotting is more useful. But for this data, we have the cell size as integers, so we can explore the data just by building tables in this way.

We can also try to look at combined parameters. We have already seen that both cell size and cell thickness seem to be associated with how benign or malignant a tumor is, so let us try to see how the cell thickness behaves as a function of both class and cell size:

```
formatted_breast_cancer %>%
  group_by(Class, as.factor(Cell.size)) %>%
  summarise(mean.thickness = mean(Cl.thickness),
            .groups = "drop")
## # A tibble: 18 × 3
##    Class     `as.factor(Cell.size)` mean.thickness
##    <fct>     <fct>                           <dbl>
##  1 benign    1                                2.76
##  2 benign    2                                3.49
##  3 benign    3                                3.81
##  4 benign    4                                5.11
##  5 benign    6                                5
##  6 benign    7                                5
##  7 benign    8                                6
##  8 benign    9                                6
##  9 malignant 1                                7.25
## 10 malignant 2                                6.75
## 11 malignant 3                                6.44
## 12 malignant 4                                7.71
## 13 malignant 5                                6.87
## 14 malignant 6                                6.88
## 15 malignant 7                                6.89
## 16 malignant 8                                7.18
## 17 malignant 9                                8.8
## 18 malignant 10                               7.52
```

I am not sure how much I learn from this. It seems that for the benign tumors, the thickness increases with the cell size, but for the malignant, there isn't that pattern.

Maybe we can learn more by ordering the data in a different way. What if we look at the numbers of benign and malignant tumors for each cell size and see what the thickness is?

```
formatted_breast_cancer %>%
  group_by(as.factor(Cell.size), Class) %>%
  summarise(mean.thickness = mean(Cl.thickness),
            .groups = "drop")
```

```
## # A tibble: 18 × 3
##    `as.factor(Cell.size)` Class     mean.thickness
##    <fct>                  <fct>              <dbl>
##  1 1                      benign              2.76
##  2 1                      malignant           7.25
##  3 2                      benign              3.49
##  4 2                      malignant           6.75
##  5 3                      benign              3.81
##  6 3                      malignant           6.44
##  7 4                      benign              5.11
##  8 4                      malignant           7.71
##  9 5                      malignant           6.87
## 10 6                      benign              5
## 11 6                      malignant           6.88
## 12 7                      benign              5
## 13 7                      malignant           6.89
## 14 8                      benign              6
## 15 8                      malignant           7.18
## 16 9                      benign              6
## 17 9                      malignant           8.8
## 18 10                     malignant           7.52
```

I am not sure how much we learned from that either, but at least it looks like for each cell size where we have both benign and malignant tumors the thickness is higher for the malignant than the benign. That is something at least. A place to start the analysis. But we can learn more when we start plotting data and when we do a proper statistical analysis of them. We will return to that in later chapters. For now, we leave it at that.

Tidying Data with `tidyr`

I am not really sure where the concept of "tidy data" comes from. Hadley Wickham, the author of many of the essential packages you will use in your R data analysis, describes tidy data as such:

- Tidy data is a standard way of mapping the meaning of a data set to its structure. A data set is messy or tidy depending on how rows, columns and tables are matched up with observations, variables and types.

In my experience, tidy data means that I can plot or summarize the data efficiently. It mostly comes down to what data is represented as columns in a data frame and what is not.

In practice, this means that I have columns in my data frame that I can work with for the analysis I want to do. For example, if I want to look at the `iris` data set and see how the `Petal.Length` varies among species, then I can look at the `Species` column against the `Petal.Length` column:

```
iris |>
  as_tibble() |>
  select(Species, Petal.Length) |>
  head(3)
## # A tibble: 3 × 2
##    Species Petal.Length
##    <fct>          <dbl>
## 1  setosa           1.4
## 2  setosa           1.4
## 3  setosa           1.3
```

I have a column specifying the `Species` and another specifying the `Petal.Length`, and it is easy enough to look at their correlation. I can plot one against the other (we will cover visualization in the next chapter). I can let the x-axis be species and the y-axis be `Petal.Length` (see Figure 3-2).

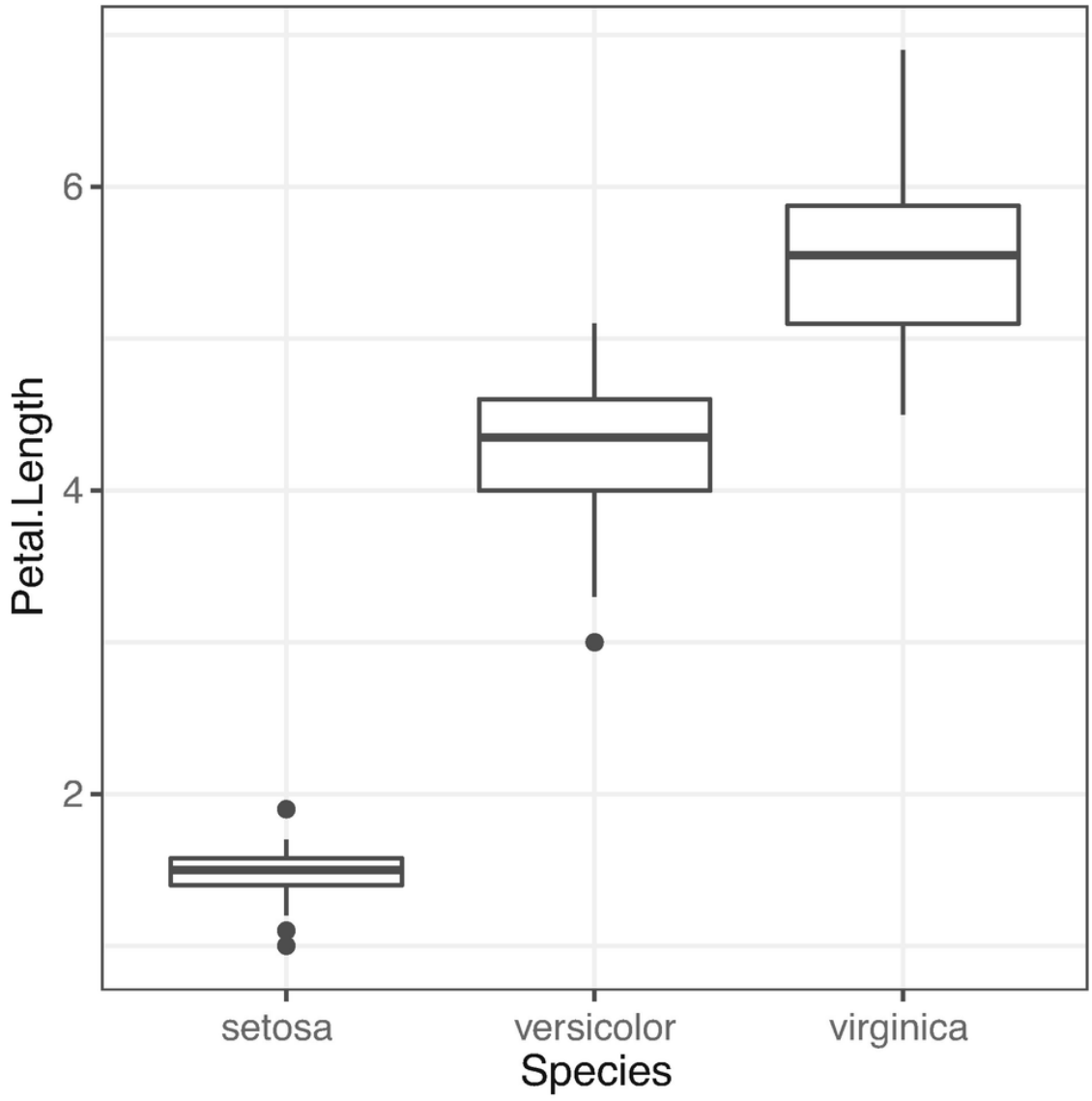

Figure 3-2 Plot species vs. petal length

```
library(ggplot2)
iris %>%
  select(Species, Petal.Length) %>%
  qplot(Species, Petal.Length, geom = "boxplot", data = .)
```

(In this pipeline, we need to use the `%>%` operator rather than `|>` because we need the input of `qplot` to go into the `data` argument, where the "." is, and `|>` cannot do this.)

This works because I have a column for the x-axis and another for the y-axis. But what happens if I want to plot the different measurements of the irises to see how those are? Each measurement is a separate column. They are `Petal.Length`, `Petal.Width`, and so on.

Now I have a bit of a problem because the different measurements are in different columns in my data frame. I cannot easily map them to an x-axis and a y-axis.

The `tidyr` package lets me fix that:

```
library(tidyr)
```

It has a function, `pivot_longer()`, that modifies the data frame, so columns become names in a factor and other columns become values.

What it does is essentially transforming the data frame such that you get one column containing the name of your original columns and another column containing the values in those columns.

In the `iris` data set, we have observations for sepal length and sepal width. If we want to examine `Species` vs. `Sepal.Length` or `Sepal.Width`, we can readily do this. We have more of a problem if we want to examine for each species both measurements at the same time. The data frame just doesn't have the structure we need for that.

If we want to see `Sepal.Length` and `Sepal.Width` as two measurements, we can plot against their values, and we would need to make a column in our data frame that tells us if a measurement is a length or a width and another column that shows us what the measurement actually is. The `pivot_longer()` function from `tidyr` lets us do that:

```
iris |>
  pivot_longer(
    c(Sepal.Length, Sepal.Width),
    names_to = "Attribute",
    values_to = "Measurement"
  ) |>
  head()
## # A tibble: 6 × 5
##   Petal.Length Petal.Width Species Attribute
##          <dbl>       <dbl> <fct>   <chr>
## 1          1.4         0.2 setosa  Sepal.Length
## 2          1.4         0.2 setosa  Sepal.Width
## 3          1.4         0.2 setosa  Sepal.Length
## 4          1.4         0.2 setosa  Sepal.Width
```

```
## 5          1.3         0.2 setosa  Sepal.Length
## 6          1.3         0.2 setosa  Sepal.Width
## # ... with 1 more variable: Measurement <dbl>
```

The preceding code tells `pivot_longer()` to take the columns `Sepal.Length` and `Sepal.Width` and make them names and values in two new columns. Here, you should read "names" as the name of the input columns a value comes from, and you should read "values" as the actual value in that column. We are saying that we want two new columns that hold all the values from the two columns `Sepal.Length` and `Sepal.Width`. The first of these columns, we get to name it with the `names_to` parameter and we give it the name `Attribute`, will contain the original column name we got the value from, and the second column, `values_to` that we name `Measurement` gets the values from the original columns. The original columns that we didn't specify are still there, but values are duplicated to match that the `Sepal.Length` and `Sepal.Width` values now are merged into a single column.

We don't necessarily want to keep all columns after a transformation like this. If we just want to plot `Sepal.Length` against `Sepal.Width`, maybe colored by `Species`, we can use `select` to pick the columns we want to keep. That would be `Species` for the colors, `Attributes` so we can tell which values are `Sepal.Length` and which are `Sepal.Width`, and then `Measurement` for the actual values:

```
iris |>
  pivot_longer(
     c(Sepal.Length, Sepal.Width),
     names_to = "Attribute",
     values_to = "Measurement"
  ) |>
  select(Species, Attribute, Measurement) |>
  head(3)
## # A tibble: 3 × 3
##   Species Attribute  Measurement
##   <fct>   <chr>            <dbl>
## 1 setosa  Sepal.Length       5.1
## 2 setosa  Sepal.Width        3.5
## 3 setosa  Sepal.Length       4.9
```

This transforms the data into a form where we can plot the attributes against measurements (see Figure 3-3 for the result):

```
iris |>
  pivot_longer(
    c(Sepal.Length, Sepal.Width),
    names_to = "Attribute",
    values_to = "Measurement"
  ) |>
  select(Species, Attribute, Measurement) %>%
  qplot(Attribute, Measurement,
          geom = "boxplot",
          facets = . ~ Species, data = .)
```

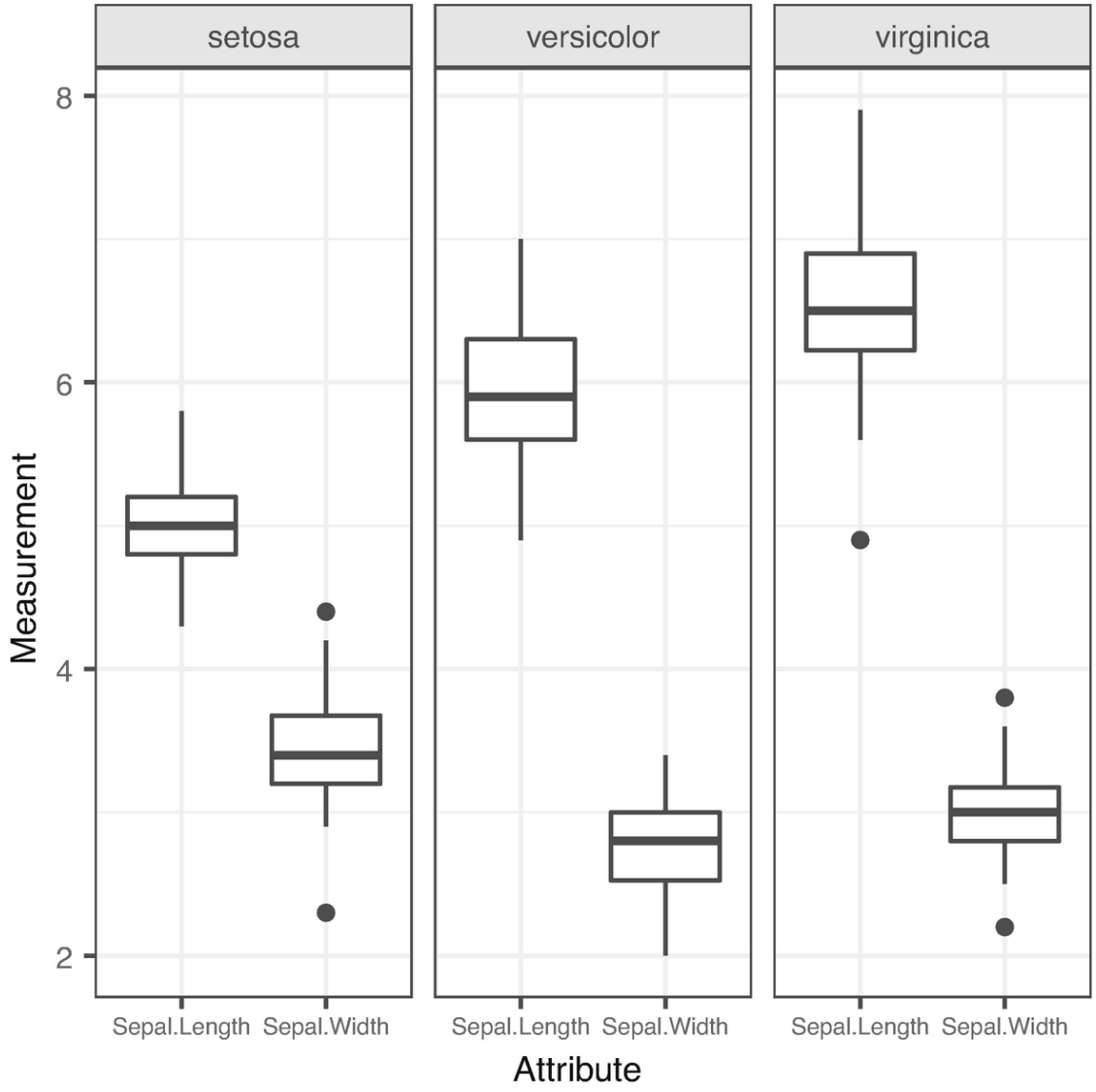

Figure 3-3 Plot measurements vs. values

The `tidyr` package also has a function, `pivot_wider`, for transforming data frames in the other direction. It is not a reverse of `pivot_longer`, because `pivot_longer` removes information about correlations in the data; we cannot, after pivoting, see which lengths originally sat in the same rows as which width, unless the remaining columns uniquely identified this, which they are not guaranteed to do (and do not for the `iris` data set). With `pivot_wider`, you get columns back that match the names you used with `pivot_wider`, and you get the values you specified there as well, but if the other variables do not uniquely identify how they should match up, you can get multiple values in the same column:

```
iris |> as_tibble() |>
  pivot_longer(
  c(Sepal.Length, Sepal.Width),
  names_to = "Attribute",
  values_to = "Measurement"
) |>
pivot_wider(
  names_from = Attribute,
  values_from = Measurement
)
## Warning: Values from `Measurement` are not uniquely
identified; output will
## * Use `values_fn = list` to suppress this warning.
## * Use `values_fn = {summary_fun}` to summarise duplicates.
## * Use the following dplyr code to identify duplicates.
##   {data} %>%
##     dplyr::group_by(Petal.Length, Petal.Width, Species,
Attribute) %>%
##     dplyr::summarise(n = dplyr::n(), .groups = "drop") %>%
##     dplyr::filter(n > 1L)
## # A tibble: 103 × 5
##    Petal.Length Petal.Width Species Sepal.Length
##           <dbl>       <dbl> <fct>   <list>
## 1           1.4         0.2 setosa  <dbl [8]>
## 2           1.3         0.2 setosa  <dbl [4]>
## 3           1.5         0.2 setosa  <dbl [7]>
## 4           1.7         0.4 setosa  <dbl [1]>
## 5           1.4         0.3 setosa  <dbl [3]>
## 6           1.5         0.1 setosa  <dbl [2]>
## 7           1.6         0.2 setosa  <dbl [5]>
```

```
## 8           1.4         0.1 setosa  <dbl [2]>
## 9           1.1         0.1 setosa  <dbl [1]>
## 10          1.2         0.2 setosa  <dbl [2]>
## # ... with 93 more rows, and 1 more variable:
## #   Sepal.Width <list>
```

You should get some warnings here; after the first transformation, we do not have enough information to transform back. Instead, R has to map some keys to multiple values. A column with type `<list>` is where you have multiple values, and the values are printed as their type and the number, for example, `<dbl [8]>`. Working with this kind of data is beyond the scope of this book, since it is atypical data for most data science applications. We are only ending up in this situation now, because we are trying to reverse an operation we did, when that operation isn't reversible. You would never try to reverse a `pivot_longer()` with a `pivot_wider()` in this way, and I have never had to attempt it myself.

Still, there are cases where your data is in a tidy format and you want to transform it back to a variable per column, and then `pivot_wider()` is your function of choice. You just have to deal with cases that might arise, where remaining columns do not provide enough information to do it. One option is to summarize all the values that would have to go into the same row, that is, those that we got as lists in the `iris` example earlier. You can provide a function for that using the `values_fn` parameter. I don't know if it makes sense to summarize the values here by their mean, but we can if we want to:

```
iris |> as_tibble() |>
  pivot_longer(
    c(Sepal.Length, Sepal.Width),
    names_to = "Attribute",
    values_to = "Measurement"
  ) |>
  pivot_wider(
    names_from = Attribute,
    values_from = Measurement,
    values_fn = mean
  ) |>
  # Let's just look at the columns we summarised...
  select(Sepal.Length, Sepal.Width)
## # A tibble: 103 × 2
```

```
##    Sepal.Length Sepal.Width
##           <dbl>       <dbl>
## 1          4.96        3.39
## 2          4.75        3.22
## 3          5.07        3.41
## 4          5.4         3.9
## 5          4.83        3.3
## 6          5.05        3.6
## 7          4.88        3.3
## 8          4.85        3.3
## 9          4.3         3
## 10         5.4         3.6
## # ... with 93 more rows
```

Exercises

It is time to put what we have learned into practice. There are only a few exercises, but I hope you will do them. You can't learn without doing exercises after all.

Importing Data

To get a feeling of the steps in importing and transforming data, you need to try it yourself. So try finding a data set you want to import. You can do that from one of the repositories I listed in the first chapter:

- RDataMining.com
- UCI Machine Learning Repository
- KDNuggets
- Reddit r/data sets
- GitHub Awesome Public Data sets

Or maybe you already have a data set you would like to analyze.

Have a look at it and figure out which import function you need. You might have to set a few parameters in the function to get the data loaded correctly, but with a bit of effort, you should be able to. For column

names, you should choose some appropriate ones from reading the data description, or if you are loading something in that is already in `mlbench`, you can cheat as I did in the preceding examples.

Using `dplyr`

Now take the data you just imported and examine various summaries. It is not so important what you look at in the data as it is that you try summarizing different aspects of it. We will look at proper analyses later. For now, just use `dplyr` to explore your data.

Using `tidyr`

Look at the preceding `dplyr` example. There, I plotted `Sepal.Length` and `Sepal.Width` for each species. Do the same thing for `Petal.Length` and `Petal.Width`.

If there is something similar to do with the data set you imported in the first exercise, try doing it with that.