Excerpt from "Modern Data Science with R" (2017)
https://mdsr-book.github.io/
copyright CRC Press

Chapter 4

Data wrangling

This chapter introduces basics of how to wrangle data in R. Wrangling skills will provide an intellectual and practical foundation for working with modern data.

4.1 A grammar for data wrangling

In much the same way that `ggplot2` presents a grammar for data graphics, the `dplyr` package presents a grammar for data wrangling [234]. Hadley Wickham, one of the authors of `dplyr`, has identified five *verbs* for working with data in a data frame:

`select()` take a subset of the columns (i.e., features, variables)

`filter()` take a subset of the rows (i.e., observations)

`mutate()` add or modify existing columns

`arrange()` sort the rows

`summarize()` aggregate the data across rows (e.g., group it according to some criteria)

Each of these functions takes a data frame as its first argument, and returns a data frame. Thus, these five verbs can be used in conjunction with each other to provide a powerful means to slice-and-dice a single table of data. As with any grammar, what these verbs mean on their own is one thing, but being able to combine these verbs with nouns (i.e., data frames) creates an infinite space for data wrangling. Mastery of these five verbs can make the computation of most any descriptive statistic a breeze and facilitate further analysis. Wickham's approach is inspired by his desire to blur the boundaries between R and the ubiquitous relational database querying syntax SQL. When we revisit SQL in Chapter 12, we will see the close relationship between these two computing paradigms. A related concept more popular in business settings is the OLAP (online analytical processing) hypercube, which refers to the process by which multidimensional data is "sliced-and-diced."

4.1.1 `select()` and `filter()`

The two simplest of the five verbs are `filter()` and `select()`, which allow you to return only a subset of the rows or columns of a data frame, respectively. Generally, if we have a data frame that consists of n rows and p columns, Figures 4.1 and 4.2 illustrate the effect of filtering this data frame based on a condition on one of the columns, and selecting a subset of the columns, respectively.

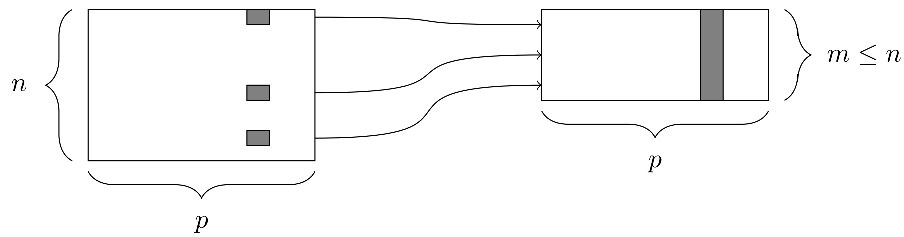

Figure 4.1: The `filter()` function. At left, a data frame that contains matching entries in a certain column for only a subset of the rows. At right, the resulting data frame after filtering.

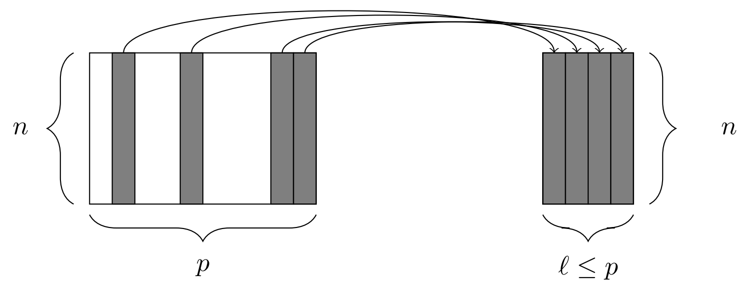

Figure 4.2: The `select()` function. At left, a data frame, from which we retrieve only a few of the columns. At right, the resulting data frame after selecting those columns.

Specifically, we will demonstrate the use of these functions on the `presidential` data frame (from the `ggplot2` package), which contains $p = 4$ variables about the terms of $n = 11$ recent U.S. Presidents.

```
library(mdsr)
presidential

# A tibble: 11  4
         name      start        end      party
        <chr>     <date>     <date>      <chr>
1  Eisenhower 1953-01-20 1961-01-20 Republican
2     Kennedy 1961-01-20 1963-11-22 Democratic
3     Johnson 1963-11-22 1969-01-20 Democratic
4       Nixon 1969-01-20 1974-08-09 Republican
5        Ford 1974-08-09 1977-01-20 Republican
6      Carter 1977-01-20 1981-01-20 Democratic
7      Reagan 1981-01-20 1989-01-20 Republican
8        Bush 1989-01-20 1993-01-20 Republican
9     Clinton 1993-01-20 2001-01-20 Democratic
10       Bush 2001-01-20 2009-01-20 Republican
11      Obama 2009-01-20 2017-01-20 Democratic
```

To retrieve only the names and party affiliations of these presidents, we would use `select()`. The first *argument* to the `select()` function is the data frame, followed by an arbitrarily long list of column names, separated by commas. Note that it is not necessary to wrap the column names in quotation marks.

```
select(presidential, name, party)

# A tibble: 11  2
         name      party
        <chr>      <chr>
1  Eisenhower Republican
2     Kennedy Democratic
3     Johnson Democratic
4       Nixon Republican
5        Ford Republican
6      Carter Democratic
7      Reagan Republican
8        Bush Republican
9     Clinton Democratic
10       Bush Republican
11      Obama Democratic
```

Similarly, the first argument to `filter()` is a data frame, and subsequent arguments are logical conditions that are evaluated on any involved columns. Thus, if we want to retrieve only those rows that pertain to Republican presidents, we need to specify that the value of the `party` variable is equal to `Republican`.

```
filter(presidential, party == "Republican")

# A tibble: 6  4
        name      start        end      party
       <chr>     <date>     <date>      <chr>
1 Eisenhower 1953-01-20 1961-01-20 Republican
2      Nixon 1969-01-20 1974-08-09 Republican
3       Ford 1974-08-09 1977-01-20 Republican
4     Reagan 1981-01-20 1989-01-20 Republican
5       Bush 1989-01-20 1993-01-20 Republican
6       Bush 2001-01-20 2009-01-20 Republican
```

Note that the `==` is a *test for equality*. If we were to use only a single equal sign here, we would be asserting that the value of `party` was `Republican`. This would cause all of the rows of `presidential` to be returned, since we would have overwritten the actual values of the `party` variable. Note also the quotation marks around `Republican` are necessary here, since `Republican` is a literal value, and not a variable name.

Naturally, combining the `filter()` and `select()` commands enables one to drill down to very specific pieces of information. For example, we can find which Democratic presidents served since Watergate.

```
select(filter(presidential, start > 1973 & party == "Democratic"), name)

# A tibble: 3  1
     name
    <chr>
1  Carter
2 Clinton
3   Obama
```

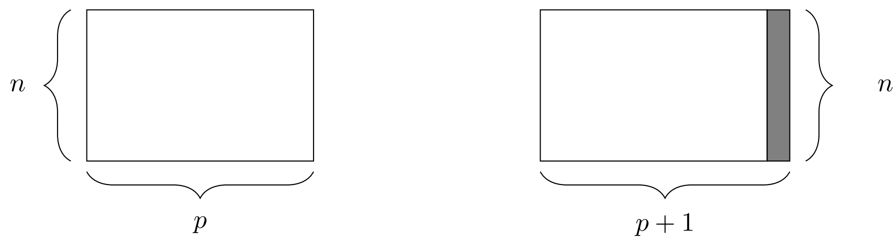

Figure 4.3: The mutate() function. At left, a data frame. At right, the resulting data frame after adding a new column.

In the syntax demonstrated above, the filter() operation is *nested* inside the select() operation. As noted above, each of the five verbs takes and returns a data frame, which makes this type of nesting possible. Shortly, we will see how these verbs can be chained together to make rather long expressions that can become very difficult to read. Instead, we recommend the use of the %>% (pipe) operator. Pipe-forwarding is an alternative to nesting that yields code that can be easily read from top to bottom. With the pipe, we can write the same expression as above in this more readable syntax.

```
presidential %>%
  filter(start > 1973 & party == "Democratic") %>%
  select(name)

# A tibble: 3  1
     name
    <chr>
1  Carter
2 Clinton
3   Obama
```

This expression is called a *pipeline*. Notice how the expression

```
dataframe %>% filter(condition)
```

is equivalent to `filter(dataframe, condition)`. In later examples we will see how this operator can make our code more readable and efficient, particularly for complex operations on large data sets.

4.1.2 mutate() and rename()

Frequently, in the process of conducting our analysis, we will create, re-define, and rename some of our variables. The functions mutate() and rename() provide these capabilities. A graphical illustration of the mutate() operation is shown in Figure 4.3.

While we have the raw data on when each of these presidents took and relinquished office, we don't actually have a numeric variable giving the length of each president's term. Of course, we can derive this information from the dates given, and add the result as a new column to our data frame. This date arithmetic is made easier through the use of the lubridate package, which we use to compute the number of exact years (eyears(1)()) that elapsed since during the interval() from the start until the end of each president's term.

In this situation, it is generally considered good style to create a new object rather than clobbering the one that comes from an external source. To preserve the existing `presidential` data frame, we save the result of `mutate()` as a new object called `mypresidents`.

```
library(lubridate)
mypresidents <- presidential %>%
  mutate(term.length = interval(start, end) / eyears(1))
mypresidents

# A tibble: 11  5
         name      start        end      party term.length
        <chr>     <date>     <date>      <chr>       <dbl>
1  Eisenhower 1953-01-20 1961-01-20 Republican        8.01
2     Kennedy 1961-01-20 1963-11-22 Democratic        2.84
3     Johnson 1963-11-22 1969-01-20 Democratic        5.17
4       Nixon 1969-01-20 1974-08-09 Republican        5.55
5        Ford 1974-08-09 1977-01-20 Republican        2.45
6      Carter 1977-01-20 1981-01-20 Democratic        4.00
7      Reagan 1981-01-20 1989-01-20 Republican        8.01
8        Bush 1989-01-20 1993-01-20 Republican        4.00
9     Clinton 1993-01-20 2001-01-20 Democratic        8.01
10       Bush 2001-01-20 2009-01-20 Republican        8.01
11      Obama 2009-01-20 2017-01-20 Democratic        8.01
```

The `mutate()` function can also be used to modify the data in an existing column. Suppose that we wanted to add to our data frame a variable containing the year in which each president was elected. Our first naïve attempt is to assume that every president was elected in the year before he took office. Note that `mutate()` returns a data frame, so if we want to modify our existing data frame, we need to overwrite it with the results.

```
mypresidents <- mypresidents %>% mutate(elected = year(start) - 1)
mypresidents

# A tibble: 11  6
         name      start        end      party term.length elected
        <chr>     <date>     <date>      <chr>       <dbl>   <dbl>
1  Eisenhower 1953-01-20 1961-01-20 Republican        8.01    1952
2     Kennedy 1961-01-20 1963-11-22 Democratic        2.84    1960
3     Johnson 1963-11-22 1969-01-20 Democratic        5.17    1962
4       Nixon 1969-01-20 1974-08-09 Republican        5.55    1968
5        Ford 1974-08-09 1977-01-20 Republican        2.45    1973
6      Carter 1977-01-20 1981-01-20 Democratic        4.00    1976
7      Reagan 1981-01-20 1989-01-20 Republican        8.01    1980
8        Bush 1989-01-20 1993-01-20 Republican        4.00    1988
9     Clinton 1993-01-20 2001-01-20 Democratic        8.01    1992
10       Bush 2001-01-20 2009-01-20 Republican        8.01    2000
11      Obama 2009-01-20 2017-01-20 Democratic        8.01    2008
```

Some aspects of this data set are wrong, because presidential elections are only held every four years. Lyndon Johnson assumed the office after President Kennedy was assassinated in 1963, and Gerald Ford took over after President Nixon resigned in 1974. Thus, there were no presidential elections in 1962 or 1973, as suggested in our data frame. We should overwrite these values with `NA`'s—which is how R denotes missing values. We can use the `ifelse()` function to do this. Here, if the value of `elected` is either 1962 or 1973, we overwrite that value with `NA`.[1] Otherwise, we overwrite it with the same value that it currently has. In this case, instead of checking to see whether the value of `elected` equals `1962` or `1973`, for brevity we can use the `%in%` operator to check to see whether the value of `elected` belongs to the vector consisting of `1962` and `1973`.

```
mypresidents <- mypresidents %>%
  mutate(elected = ifelse((elected %in% c(1962, 1973)), NA, elected))
mypresidents

# A tibble: 11  6
         name      start        end      party term.length elected
        <chr>     <date>     <date>      <chr>       <dbl>   <dbl>
1  Eisenhower 1953-01-20 1961-01-20 Republican        8.01    1952
2     Kennedy 1961-01-20 1963-11-22 Democratic        2.84    1960
3     Johnson 1963-11-22 1969-01-20 Democratic        5.17      NA
4       Nixon 1969-01-20 1974-08-09 Republican        5.55    1968
5        Ford 1974-08-09 1977-01-20 Republican        2.45      NA
6      Carter 1977-01-20 1981-01-20 Democratic        4.00    1976
7      Reagan 1981-01-20 1989-01-20 Republican        8.01    1980
8        Bush 1989-01-20 1993-01-20 Republican        4.00    1988
9     Clinton 1993-01-20 2001-01-20 Democratic        8.01    1992
10       Bush 2001-01-20 2009-01-20 Republican        8.01    2000
11      Obama 2009-01-20 2017-01-20 Democratic        8.01    2008
```

Finally, it is considered bad practice to use periods in the name of functions, data frames, and variables in R. Ill-advised periods could conflict with R's use of *generic* functions (i.e., R's mechanism for *method overloading*). Thus, we should change the name of the `term.length` column that we created earlier. In this book, we will use snake_case for function and variable names. We can achieve this using the `rename()` function.

Pro Tip: Don't use periods in the names of functions, data frames, or variables, as this can conflict with R's programming model.

```
mypresidents <- mypresidents %>% rename(term_length = term.length)
mypresidents

# A tibble: 11  6
         name      start        end      party term_length elected
        <chr>     <date>     <date>      <chr>       <dbl>   <dbl>
1  Eisenhower 1953-01-20 1961-01-20 Republican        8.01    1952
2     Kennedy 1961-01-20 1963-11-22 Democratic        2.84    1960
3     Johnson 1963-11-22 1969-01-20 Democratic        5.17      NA
4       Nixon 1969-01-20 1974-08-09 Republican        5.55    1968
5        Ford 1974-08-09 1977-01-20 Republican        2.45      NA
6      Carter 1977-01-20 1981-01-20 Democratic        4.00    1976
7      Reagan 1981-01-20 1989-01-20 Republican        8.01    1980
8        Bush 1989-01-20 1993-01-20 Republican        4.00    1988
```

[1] Incidentally, Johnson was elected in 1964 as an incumbent.

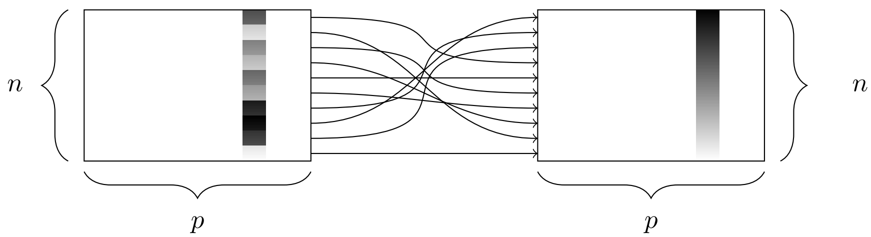

Figure 4.4: The arrange() function. At left, a data frame with an ordinal variable. At right, the resulting data frame after sorting the rows in descending order of that variable.

```
9     Clinton 1993-01-20 2001-01-20 Democratic        8.01    1992
10       Bush 2001-01-20 2009-01-20 Republican        8.01    2000
11      Obama 2009-01-20 2017-01-20 Democratic        8.01    2008
```

4.1.3 arrange()

The function sort() will sort a vector, but not a data frame. The function that will sort a data frame is called arrange(), and its behavior is illustrated in Figure 4.4.

In order to use arrange() on a data frame, you have to specify the data frame, and the column by which you want it to be sorted. You also have to specify the direction in which you want it to be sorted. Specifying multiple sort conditions will result in any ties being broken. Thus, to sort our presidential data frame by the length of each president's term, we specify that we want the column term_length in descending order.

```
mypresidents %>% arrange(desc(term_length))

# A tibble: 11  6
         name      start        end      party term_length elected
        <chr>     <date>     <date>      <chr>       <dbl>   <dbl>
1  Eisenhower 1953-01-20 1961-01-20 Republican        8.01    1952
2      Reagan 1981-01-20 1989-01-20 Republican        8.01    1980
3     Clinton 1993-01-20 2001-01-20 Democratic        8.01    1992
4        Bush 2001-01-20 2009-01-20 Republican        8.01    2000
5       Obama 2009-01-20 2017-01-20 Democratic        8.01    2008
6       Nixon 1969-01-20 1974-08-09 Republican        5.55    1968
7     Johnson 1963-11-22 1969-01-20 Democratic        5.17      NA
8      Carter 1977-01-20 1981-01-20 Democratic        4.00    1976
9        Bush 1989-01-20 1993-01-20 Republican        4.00    1988
10    Kennedy 1961-01-20 1963-11-22 Democratic        2.84    1960
11       Ford 1974-08-09 1977-01-20 Republican        2.45      NA
```

A number of presidents completed either one or two full terms, and thus have the exact same term length (4 or 8 years, respectively). To break these ties, we can further sort by party and elected.

```
mypresidents %>% arrange(desc(term_length), party, elected)

# A tibble: 11  6
```

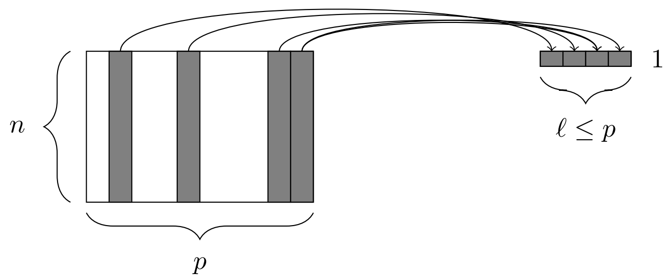

Figure 4.5: The summarize() function. At left, a data frame. At right, the resulting data frame after aggregating three of the columns.

```
          name      start        end      party term_length elected
         <chr>     <date>     <date>      <chr>       <dbl>   <dbl>
1      Clinton 1993-01-20 2001-01-20 Democratic        8.01    1992
2        Obama 2009-01-20 2017-01-20 Democratic        8.01    2008
3   Eisenhower 1953-01-20 1961-01-20 Republican        8.01    1952
4       Reagan 1981-01-20 1989-01-20 Republican        8.01    1980
5         Bush 2001-01-20 2009-01-20 Republican        8.01    2000
6        Nixon 1969-01-20 1974-08-09 Republican        5.55    1968
7      Johnson 1963-11-22 1969-01-20 Democratic        5.17      NA
8       Carter 1977-01-20 1981-01-20 Democratic        4.00    1976
9         Bush 1989-01-20 1993-01-20 Republican        4.00    1988
10     Kennedy 1961-01-20 1963-11-22 Democratic        2.84    1960
11        Ford 1974-08-09 1977-01-20 Republican        2.45      NA
```

Note that the default sort order is ascending order, so we do not need to specify an order if that is what we want.

4.1.4 summarize() with group_by()

Our last of the five verbs for single-table analysis is summarize(), which is nearly always used in conjunction with group_by(). The previous four verbs provided us with means to manipulate a data frame in powerful and flexible ways. But the extent of the analysis we can perform with these four verbs alone is limited. On the other hand, summarize() with group_by() enables us to make comparisons.

When used alone, summarize() collapses a data frame into a single row. This is illustrated in Figure 4.5. Critically, we have to specify *how* we want to reduce an entire column of data into a single value. The method of aggregation that we specify controls what will appear in the output.

```
mypresidents %>%
  summarize(
    N = n(), first_year = min(year(start)), last_year = max(year(end)),
    num_dems = sum(party == "Democratic"),
    years = sum(term_length),
    avg_term_length = mean(term_length))

# A tibble: 1   6
```

```
      N first_year last_year num_dems years avg_term_length
  <int>      <dbl>     <dbl>    <int> <dbl>           <dbl>
1    11       1953      2017        5    64            5.82
```

The first argument to `summarize()` is a data frame, followed by a list of variables that will appear in the output. Note that every variable in the output is defined by operations performed on *vectors*—not on individual values. This is essential, since if the specification of an output variable is not an operation on a vector, there is no way for R to know how to collapse each column.

In this example, the function `n()` simply counts the number of rows. This is almost always useful information.

Pro Tip: To help ensure that data aggregation is being done correctly, use `n()` every time you use `summarize()`.

The next two variables determine the first year that one of these presidents assumed office. This is the smallest year in the `start` column. Similarly, the most recent year is the largest year in the `end` column. The variable `num_dems` simply counts the number of rows in which the value of the `party` variable was `Democratic`. Finally, the last two variables compute the sum and average of the `term_length` variable. Thus, we can quickly see that 5 of the 11 presidents who served from 1953 to 2017 were Democrats, and the average term length over these 64 years was about 5.8 years.

This begs the question of whether Democratic or Republican presidents served a longer average term during this time period. To figure this out, we can just execute `summarize()` again, but this time, instead of the first argument being the data frame `mypresidents`, we will specify that the rows of the `mypresidents` data frame should be grouped by the values of the `party` variable. In this manner, the same computations as above will be carried out for each party separately.

```
mypresidents %>%
  group_by(party) %>%
  summarize(
    N = n(), first_year = min(year(start)), last_year = max(year(end)),
    num_dems = sum(party == "Democratic"),
    years = sum(term_length),
    avg_term_length = mean(term_length))

# A tibble: 2  7
       party     N first_year last_year num_dems years avg_term_length
       <chr> <int>      <dbl>     <dbl>    <int> <dbl>           <dbl>
1 Democratic     5       1961      2017        5    28             5.6
2 Republican     6       1953      2009        0    36             6.0
```

This provides us with the valuable information that the six Republican presidents served an average of 6 years in office, while the five Democratic presidents served an average of only 5.6. As with all of the `dplyr` verbs, the final output is a data frame.

Pro Tip: In this chapter we are using the `dplyr` package. The most common way to extract data from data tables is with SQL (structured query language). We'll introduce SQL in Chapter 12. The `dplyr` package provides a new interface that fits more smoothly into an overall data analysis workflow and is, in our opinion, easier to learn. Once you understand data wrangling with dplyr, it's straightforward to learn SQL if needed. And dplyr can work as an interface to many systems that use SQL internally.

4.2 Extended example: Ben's time with the Mets

In this extended example, we will continue to explore Sean Lahman's historical baseball database, which contains complete seasonal records for all players on all Major League Baseball teams going back to 1871. These data are made available in R via the Lahman package [80]. Here again, while domain knowledge may be helpful, it is not necessary to follow the example. To flesh out your understanding, try reading the Wikipedia entry on Major League Baseball.

```
library(Lahman)
dim(Teams)

[1] 2805   48
```

The `Teams` table contains the seasonal results of every major league team in every season since 1871. There are 2805 rows and 48 columns in this table, which is far too much to show here, and would make for a quite unwieldy spreadsheet. Of course, we can take a peek at what this table looks like by printing the first few rows of the table to the screen with the `head()` command, but we won't print that on the page of this book.

Ben worked for the New York Mets from 2004 to 2012. How did the team do during those years? We can use `filter()` and `select()` to quickly identify only those pieces of information that we care about.

```
mets <- Teams %>% filter(teamID == "NYN")
myMets <- mets %>% filter(yearID %in% 2004:2012)
myMets %>% select(yearID, teamID, W, L)

  yearID teamID  W  L
1   2004    NYN 71 91
2   2005    NYN 83 79
3   2006    NYN 97 65
4   2007    NYN 88 74
5   2008    NYN 89 73
6   2009    NYN 70 92
7   2010    NYN 79 83
8   2011    NYN 77 85
9   2012    NYN 74 88
```

Notice that we have broken this down into three steps. First, we filter the rows of the `Teams` data frame into only those teams that correspond to the New York Mets.[2] There are 54 of those, since the Mets joined the National League in 1962.

```
nrow(mets)

[1] 54
```

[2]The `teamID` value of `NYN` stands for the **N**ew **Y**ork **N**ational League club.

Next, we filtered these data so as to include only those seasons in which Ben worked for the team—those with `yearID` between 2004 and 2012. Finally, we printed to the screen only those columns that were relevant to our question: the year, the team's ID, and the number of wins and losses that the team had.

While this process is logical, the code can get unruly, since two ancillary data frames (`mets` and `myMets`) were created during the process. It may be the case that we'd like to use data frames later in the analysis. But if not, they are just cluttering our workspace, and eating up memory. A more streamlined way to achieve the same result would be to *nest* these commands together.

```
select(filter(mets, teamID == "NYN" & yearID %in% 2004:2012),
  yearID, teamID, W, L)

  yearID teamID  W  L
1   2004    NYN 71 91
2   2005    NYN 83 79
3   2006    NYN 97 65
4   2007    NYN 88 74
5   2008    NYN 89 73
6   2009    NYN 70 92
7   2010    NYN 79 83
8   2011    NYN 77 85
9   2012    NYN 74 88
```

This way, no additional data frames were created. However, it is easy to see that as we nest more and more of these operations together, this code could become difficult to read. To maintain readability, we instead *chain* these operations, rather than nest them (and get the same exact results).

```
Teams %>%
  select(yearID, teamID, W, L) %>%
  filter(teamID == "NYN" & yearID %in% 2004:2012)
```

This *piping* syntax (introduced in Section 4.1.1) is provided by the `dplyr` package. It retains the step-by-step logic of our original code, while being easily readable, and efficient with respect to memory and the creation of temporary data frames. In fact, there are also performance enhancements under the hood that make this the most efficient way to do these kinds of computations. For these reasons we will use this syntax whenever possible throughout the book. Note that we only have to type `Teams` once—it is implied by the pipe operator (`%>%`) that the subsequent command takes the previous data frame as its first argument. Thus, `df %>% f(y)` is equivalent to `f(df, y)`.

We've answered the simple question of how the Mets performed during the time that Ben was there, but since we are data scientists, we are interested in deeper questions. For example, some of these seasons were subpar—the Mets had more losses than wins. Did the team just get unlucky in those seasons? Or did they actually play as badly as their record indicates?

In order to answer this question, we need a model for *expected winning percentage*. It turns out that one of the most widely used contributions to the field of baseball analytics (courtesy of Bill James) is exactly that. This model translates the number of runs [3] that

[3] In baseball, a team scores a run when a player traverses the bases and return to home plate. The team with the most runs in each game wins, and no ties are allowed.

a team scores and allows *over the course of an entire season* into an expectation for how many games they should have won. The simplest version of this model is this:

$$\widehat{WPct} = \frac{1}{1 + \left(\frac{RA}{RS}\right)^2},$$

where RA is the number of runs the team allows, RS is the number of runs that the team scores, and $\widehat{WPct}$ is the team's expected winning percentage. Luckily for us, the runs scored and allowed are present in the `Teams` table, so let's grab them and save them in a new data frame.

```
metsBen <- Teams %>% select(yearID, teamID, W, L, R, RA) %>%
  filter(teamID == "NYN" & yearID %in% 2004:2012)
metsBen

  yearID teamID  W  L   R  RA
1   2004    NYN 71 91 684 731
2   2005    NYN 83 79 722 648
3   2006    NYN 97 65 834 731
4   2007    NYN 88 74 804 750
5   2008    NYN 89 73 799 715
6   2009    NYN 70 92 671 757
7   2010    NYN 79 83 656 652
8   2011    NYN 77 85 718 742
9   2012    NYN 74 88 650 709
```

First, note that the runs-scored variable is called `R` in the `Teams` table, but to stick with our notation we want to rename it `RS`.

```
metsBen <- metsBen %>% rename(RS = R)    # new name = old name
metsBen

  yearID teamID  W  L  RS  RA
1   2004    NYN 71 91 684 731
2   2005    NYN 83 79 722 648
3   2006    NYN 97 65 834 731
4   2007    NYN 88 74 804 750
5   2008    NYN 89 73 799 715
6   2009    NYN 70 92 671 757
7   2010    NYN 79 83 656 652
8   2011    NYN 77 85 718 742
9   2012    NYN 74 88 650 709
```

Next, we need to compute the team's actual winning percentage in each of these seasons. Thus, we need to add a new column to our data frame, and we do this with the `mutate()` command.

```
metsBen <- metsBen %>% mutate(WPct = W / (W + L))
metsBen

  yearID teamID  W  L  RS  RA  WPct
1   2004    NYN 71 91 684 731 0.438
```

```
2   2005    NYN 83 79 722 648 0.512
3   2006    NYN 97 65 834 731 0.599
4   2007    NYN 88 74 804 750 0.543
5   2008    NYN 89 73 799 715 0.549
6   2009    NYN 70 92 671 757 0.432
7   2010    NYN 79 83 656 652 0.488
8   2011    NYN 77 85 718 742 0.475
9   2012    NYN 74 88 650 709 0.457
```

We also need to compute the model estimates for winning percentage.

```
metsBen <- metsBen %>% mutate(WPct_hat = 1 / (1 + (RA/RS)^2))
metsBen

  yearID teamID  W  L  RS  RA  WPct WPct_hat
1   2004    NYN 71 91 684 731 0.438    0.467
2   2005    NYN 83 79 722 648 0.512    0.554
3   2006    NYN 97 65 834 731 0.599    0.566
4   2007    NYN 88 74 804 750 0.543    0.535
5   2008    NYN 89 73 799 715 0.549    0.555
6   2009    NYN 70 92 671 757 0.432    0.440
7   2010    NYN 79 83 656 652 0.488    0.503
8   2011    NYN 77 85 718 742 0.475    0.484
9   2012    NYN 74 88 650 709 0.457    0.457
```

The expected number of wins is then equal to the product of the expected winning percentage times the number of games.

```
metsBen <- metsBen %>% mutate(W_hat = WPct_hat * (W + L))
metsBen

  yearID teamID  W  L  RS  RA  WPct WPct_hat W_hat
1   2004    NYN 71 91 684 731 0.438    0.467  75.6
2   2005    NYN 83 79 722 648 0.512    0.554  89.7
3   2006    NYN 97 65 834 731 0.599    0.566  91.6
4   2007    NYN 88 74 804 750 0.543    0.535  86.6
5   2008    NYN 89 73 799 715 0.549    0.555  90.0
6   2009    NYN 70 92 671 757 0.432    0.440  71.3
7   2010    NYN 79 83 656 652 0.488    0.503  81.5
8   2011    NYN 77 85 718 742 0.475    0.484  78.3
9   2012    NYN 74 88 650 709 0.457    0.457  74.0
```

In this case, the Mets' fortunes were better than expected in three of these seasons, and worse than expected in the other six.

```
filter(metsBen, W >= W_hat)

  yearID teamID  W  L  RS  RA  WPct WPct_hat W_hat
1   2006    NYN 97 65 834 731 0.599    0.566  91.6
2   2007    NYN 88 74 804 750 0.543    0.535  86.6
3   2012    NYN 74 88 650 709 0.457    0.457  74.0
```

```
filter(metsBen, W < W_hat)

  yearID teamID  W  L  RS  RA  WPct WPct_hat W_hat
1   2004    NYN 71 91 684 731 0.438    0.467  75.6
2   2005    NYN 83 79 722 648 0.512    0.554  89.7
3   2008    NYN 89 73 799 715 0.549    0.555  90.0
4   2009    NYN 70 92 671 757 0.432    0.440  71.3
5   2010    NYN 79 83 656 652 0.488    0.503  81.5
6   2011    NYN 77 85 718 742 0.475    0.484  78.3
```

Naturally, the Mets experienced ups and downs during Ben's time with the team. Which seasons were best? To figure this out, we can simply sort the rows of the data frame.

```
arrange(metsBen, desc(WPct))

  yearID teamID  W  L  RS  RA  WPct WPct_hat W_hat
1   2006    NYN 97 65 834 731 0.599    0.566  91.6
2   2008    NYN 89 73 799 715 0.549    0.555  90.0
3   2007    NYN 88 74 804 750 0.543    0.535  86.6
4   2005    NYN 83 79 722 648 0.512    0.554  89.7
5   2010    NYN 79 83 656 652 0.488    0.503  81.5
6   2011    NYN 77 85 718 742 0.475    0.484  78.3
7   2012    NYN 74 88 650 709 0.457    0.457  74.0
8   2004    NYN 71 91 684 731 0.438    0.467  75.6
9   2009    NYN 70 92 671 757 0.432    0.440  71.3
```

In 2006, the Mets had the best record in baseball during the regular season and nearly made the World Series. But how do these seasons rank in terms of the team's performance relative to our model?

```
metsBen %>%
  mutate(Diff = W - W_hat) %>%
  arrange(desc(Diff))

  yearID teamID  W  L  RS  RA  WPct WPct_hat W_hat    Diff
1   2006    NYN 97 65 834 731 0.599    0.566  91.6  5.3840
2   2007    NYN 88 74 804 750 0.543    0.535  86.6  1.3774
3   2012    NYN 74 88 650 709 0.457    0.457  74.0  0.0199
4   2008    NYN 89 73 799 715 0.549    0.555  90.0 -0.9605
5   2009    NYN 70 92 671 757 0.432    0.440  71.3 -1.2790
6   2011    NYN 77 85 718 742 0.475    0.484  78.3 -1.3377
7   2010    NYN 79 83 656 652 0.488    0.503  81.5 -2.4954
8   2004    NYN 71 91 684 731 0.438    0.467  75.6 -4.6250
9   2005    NYN 83 79 722 648 0.512    0.554  89.7 -6.7249
```

So 2006 was the Mets' most fortunate year—since they won five more games than our model predicts—but 2005 was the least fortunate—since they won almost seven games fewer than our model predicts. This type of analysis helps us understand how the Mets performed in individual seasons, but we know that any randomness that occurs in individual years is likely to average out over time. So while it is clear that the Mets performed well in some seasons and poorly in others, what can we say about their overall performance?

We can easily summarize a single variable with the favstats() command from the mosaic package.

```
favstats(~ W, data = metsBen)

 min Q1 median Q3 max mean  sd n missing
  70 74     79 88  97 80.9 9.1 9       0
```

This tells us that the Mets won nearly 81 games on average during Ben's tenure, which corresponds almost exactly to a 0.500 winning percentage, since there are 162 games in a regular season. But we may be interested in aggregating more than one variable at a time. To do this, we use summarize().

```
metsBen %>%
  summarize(
    num_years = n(), total_W = sum(W), total_L = sum(L),
    total_WPct = sum(W) / sum(W + L), sum_resid = sum(W - W_hat))

  num_years total_W total_L total_WPct sum_resid
1         9     728     730      0.499     -10.6
```

In these nine years, the Mets had a combined record of 728 wins and 730 losses, for an overall winning percentage of .499. Just one extra win would have made them exactly 0.500! (If we could pick which game, we would definitely pick the final game of the 2007 season. A win there would have resulted in a playoff berth.) However, we've also learned that the team under-performed relative to our model by a total of 10.6 games over those nine seasons.

Usually, when we are summarizing a data frame like we did above, it is interesting to consider different groups. In this case, we can discretize these years into three chunks: one for each of the three general managers under whom Ben worked. Jim Duquette was the Mets' general manager in 2004, Omar Minaya from 2005 to 2010, and Sandy Alderson from 2011 to 2012. We can define these eras using two nested ifelse() functions (the case_when() function in the dplyr package is helpful in such a setting).

```
metsBen <- metsBen %>%
  mutate(
    gm = ifelse(yearID == 2004, "Duquette",
         ifelse(yearID >= 2011, "Alderson", "Minaya")))
```

Next, we use the gm variable to define these groups with the group_by() operator. The combination of summarizing data by groups can be very powerful. Note that while the Mets were far more successful during Minaya's regime (i.e., many more wins than losses), they did not meet expectations in any of the three periods.

```
metsBen %>%
  group_by(gm) %>%
  summarize(
    num_years = n(), total_W = sum(W), total_L = sum(L),
    total_WPct = sum(W) / sum(W + L), sum_resid = sum(W - W_hat)) %>%
  arrange(desc(sum_resid))

# A tibble: 3  6
```

```
        gm num_years total_W total_L total_WPct sum_resid
     <chr>     <int>   <int>   <int>      <dbl>     <dbl>
1 Alderson         2     151     173      0.466     -1.32
2 Duquette         1      71      91      0.438     -4.63
3   Minaya         6     506     466      0.521     -4.70
```

The full power of the chaining operator is revealed below, where we do all the analysis at once, but retain the step-by-step logic.

```
Teams %>%
  select(yearID, teamID, W, L, R, RA) %>%
  filter(teamID == "NYN" & yearID %in% 2004:2012) %>%
  rename(RS = R) %>%
  mutate(
    WPct = W / (W + L), WPct_hat = 1 / (1 + (RA/RS)^2),
    W_hat = WPct_hat * (W + L),
    gm = ifelse(yearID == 2004, "Duquette",
         ifelse(yearID >= 2011, "Alderson", "Minaya"))) %>%
  group_by(gm) %>%
  summarize(
    num_years = n(), total_W = sum(W), total_L = sum(L),
    total_WPct = sum(W) / sum(W + L), sum_resid = sum(W - W_hat)) %>%
  arrange(desc(sum_resid))

# A tibble: 3  6
        gm num_years total_W total_L total_WPct sum_resid
     <chr>     <int>   <int>   <int>      <dbl>     <dbl>
1 Alderson         2     151     173      0.466     -1.32
2 Duquette         1      71      91      0.438     -4.63
3   Minaya         6     506     466      0.521     -4.70
```

Even more generally, we might be more interested in how the Mets performed relative to our model, in the context of all teams during that nine year period. All we need to do is remove the `teamID` filter and group by franchise (`franchID`) instead.

```
Teams %>% select(yearID, teamID, franchID, W, L, R, RA) %>%
  filter(yearID %in% 2004:2012) %>%
  rename(RS = R) %>%
  mutate(
    WPct = W / (W + L), WPctHat = 1 / (1 + (RA/RS)^2),
    WHat = WPctHat * (W + L)) %>%
  group_by(franchID) %>%
  summarize(
    numYears = n(), totalW = sum(W), totalL = sum(L),
    totalWPct = sum(W) / sum(W + L), sumResid = sum(W - WHat)) %>%
  arrange(sumResid) %>%
  print(n = 6)

# A tibble: 30  6
  franchID numYears totalW totalL totalWPct sumResid
    <fctr>    <int>  <int>  <int>     <dbl>    <dbl>
```

```
1      TOR        9    717    740     0.492    -29.2
2      ATL        9    781    677     0.536    -24.0
3      COL        9    687    772     0.471    -22.7
4      CHC        9    706    750     0.485    -14.5
5      CLE        9    710    748     0.487    -13.9
6      NYM        9    728    730     0.499    -10.6
# ... with 24 more rows
```

We can see now that only five other teams fared worse than the Mets,[4] relative to our model, during this time period. Perhaps they are cursed!

4.3 Combining multiple tables

In the previous section, we illustrated how the five verbs can be chained to perform operations on a single table. This single table is reminiscent of a single well-organized spreadsheet. But in the same way that a workbook can contain multiple spreadsheets, we will often work with multiple tables. In Chapter 12, we will describe how multiple tables related by unique identifiers called *keys* can be organized into a *relational database management system*.

It is more efficient for the computer to store and search tables in which "like is stored with like." Thus, a database maintained by the Bureau of Transportation Statistics on the arrival times of U.S. commercial flights will consist of multiple tables, each of which contains data about different things. For example, the `nycflights13` package contains one table about `flights`—each row in this table is a single flight. As there are many flights, you can imagine that this table will get very long—hundreds of thousands of rows per year. But there are other related kinds of information that we will want to know about these flights. We would certainly be interested in the particular airline to which each flight belonged. It would be inefficient to store the complete name of the airline (e.g., `American Airlines Inc.`) in every row of the flights table. A simple code (e.g., `AA`) would take up less space on disk. For small tables, the savings of storing two characters instead of 25 is insignificant, but for large tables, it can add up to noticeable savings both in terms of the size of data on disk, and the speed with which we can search it. However, we still want to have the full names of the airlines available if we need them. The solution is to store the data *about airlines* in a separate table called `airlines`, and to provide a *key* that links the data in the two tables together.

4.3.1 `inner_join()`

If we examine the first few rows of the `flights` table, we observe that the `carrier` column contains a two-character string corresponding to the airline.

```
library(nycflights13)
head(flights, 3)

# A tibble: 3  19
   year month   day dep_time sched_dep_time dep_delay arr_time
  <int> <int> <int>    <int>          <int>     <dbl>    <int>
1  2013     1     1      517            515         2      830
2  2013     1     1      533            529         4      850
```

[4]Note that whereas the `teamID` that corresponds to the Mets is `NYN`, the value of the `franchID` variable is `NYM`.

```
3  2013     1     1      542            540         2      923
# ... with 12 more variables: sched_arr_time <int>, arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
#   time_hour <dttm>
```

In the `airlines` table, we have those same two-character strings, but also the full names of the airline.

```
head(airlines, 3)

# A tibble: 3  2
  carrier                   name
    <chr>                  <chr>
1      9E      Endeavor Air Inc.
2      AA American Airlines Inc.
3      AS   Alaska Airlines Inc.
```

In order to retrieve a list of flights and the full names of the airlines that managed each flight, we need to match up the rows in the `flights` table with those rows in the `airlines` table that have the corresponding values for the `carrier` column in *both* tables. This is achieved with the function `inner_join()`.

```
flightsJoined <- flights %>%
  inner_join(airlines, by = c("carrier" = "carrier"))
glimpse(flightsJoined)

Observations: 336,776
Variables: 20
$ year           <int> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013,...
$ month          <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,...
$ day            <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,...
$ dep_time       <int> 517, 533, 542, 544, 554, 554, 555, 557, 557, 55...
$ sched_dep_time <int> 515, 529, 540, 545, 600, 558, 600, 600, 600, 60...
$ dep_delay      <dbl> 2, 4, 2, -1, -6, -4, -5, -3, -3, -2, -2, -2, -2...
$ arr_time       <int> 830, 850, 923, 1004, 812, 740, 913, 709, 838, 7...
$ sched_arr_time <int> 819, 830, 850, 1022, 837, 728, 854, 723, 846, 7...
$ arr_delay      <dbl> 11, 20, 33, -18, -25, 12, 19, -14, -8, 8, -2, -...
$ carrier        <chr> "UA", "UA", "AA", "B6", "DL", "UA", "B6", "EV",...
$ flight         <int> 1545, 1714, 1141, 725, 461, 1696, 507, 5708, 79...
$ tailnum        <chr> "N14228", "N24211", "N619AA", "N804JB", "N668DN...
$ origin         <chr> "EWR", "LGA", "JFK", "JFK", "LGA", "EWR", "EWR"...
$ dest           <chr> "IAH", "IAH", "MIA", "BQN", "ATL", "ORD", "FLL"...
$ air_time       <dbl> 227, 227, 160, 183, 116, 150, 158, 53, 140, 138...
$ distance       <dbl> 1400, 1416, 1089, 1576, 762, 719, 1065, 229, 94...
$ hour           <dbl> 5, 5, 5, 5, 6, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 5,...
$ minute         <dbl> 15, 29, 40, 45, 0, 58, 0, 0, 0, 0, 0, 0, 0, 0, ...
$ time_hour      <dttm> 2013-01-01 05:00:00, 2013-01-01 05:00:00, 2013...
$ name           <chr> "United Air Lines Inc.", "United Air Lines Inc....
```

Notice that the `flightsJoined` data frame now has an additional variable called `name`.

This is the column from `airlines` that is now attached to our combined data frame. Now we can view the full names of the airlines instead of the cryptic two-character codes.

```
flightsJoined %>%
  select(carrier, name, flight, origin, dest) %>%
  head(3)

# A tibble: 3  5
  carrier                   name flight origin  dest
    <chr>                  <chr>  <int>  <chr> <chr>
1      UA  United Air Lines Inc.   1545    EWR   IAH
2      UA  United Air Lines Inc.   1714    LGA   IAH
3      AA American Airlines Inc.   1141    JFK   MIA
```

In an `inner_join()`, the result set contains only those rows that have matches in both tables. In this case, all of the rows in `flights` have exactly one corresponding entry in `airlines`, so the number of rows in `flightsJoined` is the same as the number of rows in `flights` (this will not always be the case).

```
nrow(flights)

[1] 336776

nrow(flightsJoined)

[1] 336776
```

Pro Tip: It is always a good idea to carefully check that the number of rows returned by a join operation is what you expected. In particular, you often want to check for rows in one table that matched to more than one row in the other table.

4.3.2 `left_join()`

Another commonly used type of join is a `left_join()`. Here the rows of the first table are *always* returned, regardless of whether there is a match in the second table.

Suppose that we are only interested in flights from the NYC airports to the West Coast. Specifically, we're only interested in airports in the Pacific Time Zone. Thus, we filter the `airports` data frame to only include those 152 airports.

```
airportsPT <- filter(airports, tz == -8)
nrow(airportsPT)

[1] 152
```

Now, if we perform an `inner_join()` on `flights` and `airportsPT`, matching the destinations in `flights` to the FAA codes in `airports`, we retrieve only those flights that flew to our airports in the Pacific Time Zone.

```
nycDestsPT <- flights %>% inner_join(airportsPT, by = c("dest" = "faa"))
nrow(nycDestsPT)

[1] 46324
```

However, if we use a left_join() with the same conditions, we retrieve all of the rows of flights. NA's are inserted into the columns where no matched data was found.

```
nycDests <- flights %>% left_join(airportsPT, by = c("dest" = "faa"))
nrow(nycDests)

[1] 336776

sum(is.na(nycDests$name))

[1] 290452
```

Left joins are particularly useful in databases in which *referential integrity* is broken (not all of the *keys* are present—see Chapter 12).

4.4 Extended example: Manny Ramirez

In the context of baseball and the Lahman package, multiple tables are used to store information. The batting statistics of players are stored in one table (Batting), while information about people (most of whom are players) is in a different table (Master).

Every row in the Batting table contains the statistics accumulated by a single player during a single stint for a single team in a single year. Thus, a player like Manny Ramirez has many rows in the Batting table (21, in fact).

```
manny <- filter(Batting, playerID == "ramirma02")
nrow(manny)

[1] 21
```

Using what we've learned, we can quickly tabulate Ramirez's most common career offensive statistics. For those new to baseball, some additional background may be helpful. A hit (H) occurs when a batter reaches base safely. A home run (HR) occurs when the ball is hit out of the park or the runner advances through all of the bases during that play. Barry Bonds has the record for most home runs (762) hit in a career. A player's batting average (BA) is the ratio of the number of hits to the number of eligible at-bats. The highest career batting average in major league baseball history of 0.366 was achieved by Ty Cobb—season averages above 0.300 are impressive. Finally, runs batted in (RBI) is the number of runners (including the batter in the case of a home run) that score during that batter's at-bat. Hank Aaron has the record for most career RBIs with 2,297.

```
manny %>% summarize(
  span = paste(min(yearID), max(yearID), sep = "-"),
  numYears = n_distinct(yearID), numTeams = n_distinct(teamID),
  BA = sum(H)/sum(AB), tH = sum(H), tHR = sum(HR), tRBI = sum(RBI))

       span numYears numTeams    BA   tH tHR tRBI
1 1993-2011       19        5 0.312 2574 555 1831
```

Notice how we have used the paste() function to combine results from multiple variables into a new variable, and how we have used the n_distinct() function to count the number of distinct rows. In his 19-year career, Ramirez hit 555 home runs, which puts him in the top 20 among all Major League players.

However, we also see that Ramirez played for five teams during his career. Did he perform equally well for each of them? Breaking his statistics down by team, or by league, is as easy as adding an appropriate `group_by()` command.

```
manny %>%
  group_by(teamID) %>%
  summarize(
    span = paste(min(yearID), max(yearID), sep = "-"),
    numYears = n_distinct(yearID), numTeams = n_distinct(teamID),
    BA = sum(H)/sum(AB), tH = sum(H), tHR = sum(HR), tRBI = sum(RBI)) %>%
  arrange(span)

# A tibble: 5  8
  teamID      span numYears numTeams     BA    tH   tHR  tRBI
  <fctr>     <chr>    <int>    <int>  <dbl> <int> <int> <int>
1    CLE 1993-2000        8        1 0.3130  1086   236   804
2    BOS 2001-2008        8        1 0.3117  1232   274   868
3    LAN 2008-2010        3        1 0.3224   237    44   156
4    CHA 2010-2010        1        1 0.2609    18     1     2
5    TBA 2011-2011        1        1 0.0588     1     0     1
```

While Ramirez was very productive for Cleveland, Boston, and the Los Angeles Dodgers, his brief tours with the Chicago White Sox and Tampa Bay Rays were less than stellar. In the pipeline below, we can see that Ramirez spent the bulk of his career in the American League.

```
manny %>%
  group_by(lgID) %>%
  summarize(
    span = paste(min(yearID), max(yearID), sep = "-"),
    numYears = n_distinct(yearID), numTeams = n_distinct(teamID),
    BA = sum(H)/sum(AB), tH = sum(H), tHR = sum(HR), tRBI = sum(RBI)) %>%
  arrange(span)

# A tibble: 2  8
    lgID      span numYears numTeams    BA    tH   tHR  tRBI
  <fctr>     <chr>    <int>    <int> <dbl> <int> <int> <int>
1     AL 1993-2011       18        4 0.311  2337   511  1675
2     NL 2008-2010        3        1 0.322   237    44   156
```

If Ramirez played in only 19 different seasons, why were there 21 rows attributed to him? Notice that in 2008, he was traded from the Boston Red Sox to the Los Angeles Dodgers, and thus played for both teams. Similarly, in 2010 he played for both the Dodgers and the Chicago White Sox. When summarizing data, it is critically important to understand exactly how the rows of your data frame are organized. To see what can go wrong here, suppose we were interested in tabulating the number of seasons in which Ramirez hit at least 30 home runs. The simplest solution is:

```
manny %>%
  filter(HR >= 30) %>%
  nrow()

[1] 11
```

But this answer is wrong, because in 2008, Ramirez hit 20 home runs for Boston before being traded and then 17 more for the Dodgers afterwards. Neither of those rows were counted, since they were *both* filtered out. Thus, the year 2008 does not appear among the 11 that we counted in the previous pipeline. Recall that each row in the `manny` data frame corresponds to one stint with one team in one year. On the other hand, the question asks us to consider each year, *regardless of team*. In order to get the right answer, we have to aggregate the rows by team. Thus, the correct solution is:

```
manny %>%
  group_by(yearID) %>%
  summarize(tHR = sum(HR)) %>%
  filter(tHR >= 30) %>%
  nrow()

[1] 12
```

Note that the `filter()` operation is applied to `tHR`, the total number of home runs in a season, and not `HR`, the number of home runs in a single stint for a single team in a single season. (This distinction between filtering the rows of the original data versus the rows of the aggregated results will appear again in Chapter 12.)

We began this exercise by filtering the `Batting` table for the player with `playerID` equal to `ramirma02`. How did we know to use this identifier? This player ID is known as a *key*, and in fact, `playerID` is the *primary key* defined in the `Master` table. That is, every row in the `Master` table is uniquely identified by the value of `playerID`. Thus there is exactly one row in that table for which `playerID` is equal to `ramirma02`.

But how did we know that this ID corresponds to Manny Ramirez? We can search the `Master` table. The data in this table include characteristics about Manny Ramirez that do not change across multiple seasons (with the possible exception of his weight).

```
Master %>% filter(nameLast == "Ramirez" & nameFirst == "Manny")

   playerID birthYear birthMonth birthDay birthCountry        birthState
1 ramirma02      1972          5       30         D.R. Distrito Nacional
      birthCity deathYear deathMonth deathDay deathCountry deathState
1 Santo Domingo        NA         NA       NA         <NA>       <NA>
  deathCity nameFirst nameLast        nameGiven weight height bats throws
1      <NA>     Manny  Ramirez Manuel Aristides    225     72    R      R
       debut  finalGame  retroID   bbrefID deathDate  birthDate
1 1993-09-02 2011-04-06 ramim002 ramirma02      <NA> 1972-05-30
```

The `playerID` column forms a primary key in the `Master` table, but it does not in the `Batting` table, since as we saw previously, there were 21 rows with that `playerID`. In the `Batting` table, the `playerID` column is known as a *foreign key*, in that it references a primary key in another table. For our purposes, the presence of this column in both tables allows us to link them together. This way, we can combine data from the `Batting` table with data in the `Master` table. We do this with `inner_join()` by specifying the two tables that we want to join, and the corresponding columns in each table that provide the link. Thus, if we want to display Ramirez's name in our previous result, as well as his age, we must join the `Batting` and `Master` tables together.

```
Batting %>%
  filter(playerID == "ramirma02") %>%
  inner_join(Master, by = c("playerID" = "playerID")) %>%
  group_by(yearID) %>%
  summarize(
    Age = max(yearID - birthYear), numTeams = n_distinct(teamID),
    BA = sum(H)/sum(AB), tH = sum(H), tHR = sum(HR), tRBI = sum(RBI)) %>%
  arrange(yearID)

# A tibble: 19  7
   yearID   Age numTeams     BA    tH   tHR  tRBI
    <int> <int>    <int>  <dbl> <int> <int> <int>
1    1993    21        1 0.1698     9     2     5
2    1994    22        1 0.2690    78    17    60
3    1995    23        1 0.3079   149    31   107
4    1996    24        1 0.3091   170    33   112
5    1997    25        1 0.3280   184    26    88
6    1998    26        1 0.2942   168    45   145
7    1999    27        1 0.3333   174    44   165
8    2000    28        1 0.3508   154    38   122
9    2001    29        1 0.3062   162    41   125
10   2002    30        1 0.3486   152    33   107
11   2003    31        1 0.3251   185    37   104
12   2004    32        1 0.3081   175    43   130
13   2005    33        1 0.2924   162    45   144
14   2006    34        1 0.3207   144    35   102
15   2007    35        1 0.2961   143    20    88
16   2008    36        2 0.3315   183    37   121
17   2009    37        1 0.2898   102    19    63
18   2010    38        2 0.2981    79     9    42
19   2011    39        1 0.0588     1     0     1
```

Pro Tip: Always specify the `by` argument that defines the join condition. Don't rely on the defaults.

Notice that even though Ramirez's age is a constant for each season, we have to use a vector operation (i.e., `max()`) in order to reduce any potential vector to a single number.

Which season was Ramirez's best as a hitter? One relatively simple measurement of batting prowess is OPS, or On-Base Plus Slugging Percentage, which is the simple sum of two other statistics: On-Base Percentage (OBP) and Slugging Percentage (SLG). The former basically measures the percentage of time that a batter reaches base safely, whether it comes via a hit (`H`), a base on balls (`BB`), or from being hit by the pitch (`HBP`). The latter measures the average number of bases advanced per at-bat (`AB`), where a single is worth one base, a double (`X2B`) is worth two, a triple (`X3B`) is worth three, and a home run (`HR`) is worth four. (Note that every hit is exactly one of a single, double, triple, or home run.) Let's add this statistic to our results and use it to rank the seasons.

```
mannyBySeason <- Batting %>%
  filter(playerID == "ramirma02") %>%
  inner_join(Master, by = c("playerID" = "playerID"))  %>%
```

```
  group_by(yearID) %>%
  summarize(
    Age = max(yearID - birthYear), numTeams = n_distinct(teamID),
    BA = sum(H)/sum(AB), tH = sum(H), tHR = sum(HR), tRBI = sum(RBI),
    OBP = sum(H + BB + HBP) / sum(AB + BB + SF + HBP),
    SLG = sum(H + X2B + 2*X3B + 3*HR) / sum(AB)) %>%
  mutate(OPS = OBP + SLG) %>%
  arrange(desc(OPS))
mannyBySeason

# A tibble: 19  10
   yearID   Age numTeams     BA    tH   tHR  tRBI    OBP    SLG   OPS
    <int> <int>    <int>  <dbl> <int> <int> <int>  <dbl>  <dbl> <dbl>
1    2000    28        1 0.3508   154    38   122 0.4568 0.6970 1.154
2    1999    27        1 0.3333   174    44   165 0.4422 0.6628 1.105
3    2002    30        1 0.3486   152    33   107 0.4498 0.6468 1.097
4    2006    34        1 0.3207   144    35   102 0.4391 0.6192 1.058
5    2008    36        2 0.3315   183    37   121 0.4297 0.6014 1.031
6    2003    31        1 0.3251   185    37   104 0.4271 0.5870 1.014
7    2001    29        1 0.3062   162    41   125 0.4048 0.6087 1.014
8    2004    32        1 0.3081   175    43   130 0.3967 0.6127 1.009
9    2005    33        1 0.2924   162    45   144 0.3877 0.5939 0.982
10   1996    24        1 0.3091   170    33   112 0.3988 0.5818 0.981
11   1998    26        1 0.2942   168    45   145 0.3771 0.5989 0.976
12   1995    23        1 0.3079   149    31   107 0.4025 0.5579 0.960
13   1997    25        1 0.3280   184    26    88 0.4147 0.5383 0.953
14   2009    37        1 0.2898   102    19    63 0.4176 0.5312 0.949
15   2007    35        1 0.2961   143    20    88 0.3884 0.4928 0.881
16   1994    22        1 0.2690    78    17    60 0.3571 0.5207 0.878
17   2010    38        2 0.2981    79     9    42 0.4094 0.4604 0.870
18   1993    21        1 0.1698     9     2     5 0.2000 0.3019 0.502
19   2011    39        1 0.0588     1     0     1 0.0588 0.0588 0.118
```

We see that Ramirez's OPS was highest in 2000. But 2000 was the height of the steroid era, when many sluggers were putting up tremendous offensive numbers. As data scientists, we know that it would be more instructive to put Ramirez's OPS in context by comparing it to the league average OPS in each season—the resulting ratio is often called OPS+. To do this, we will need to compute those averages. Because there is missing data in some of these columns in some of these years, we need to invoke the `na.rm` argument to ignore that data.

```
mlb <- Batting %>%
  filter(yearID %in% 1993:2011) %>%
  group_by(yearID) %>%
  summarize(lgOPS =
    sum(H + BB + HBP, na.rm = TRUE) / sum(AB + BB + SF + HBP, na.rm = TRUE) +
    sum(H + X2B + 2*X3B + 3*HR, na.rm = TRUE) / sum(AB, na.rm = TRUE))
```

Next, we need to match these league average OPS values to the corresponding entries for Ramirez. We can do this by joining these tables together, and computing the ratio of Ramirez's OPS to that of the league average.

```
mannyRatio <- mannyBySeason %>%
  inner_join(mlb, by = c("yearID" = "yearID")) %>%
  mutate(OPSplus = OPS / lgOPS) %>%
  select(yearID, Age, OPS, lgOPS, OPSplus) %>%
  arrange(desc(OPSplus))
mannyRatio

# A tibble: 19  5
   yearID   Age   OPS lgOPS OPSplus
    <int> <int> <dbl> <dbl>   <dbl>
1    2000    28 1.154 0.782   1.475
2    2002    30 1.097 0.748   1.466
3    1999    27 1.105 0.778   1.420
4    2006    34 1.058 0.768   1.377
5    2008    36 1.031 0.749   1.376
6    2003    31 1.014 0.755   1.344
7    2001    29 1.014 0.759   1.336
8    2004    32 1.009 0.763   1.323
9    2005    33 0.982 0.749   1.310
10   1998    26 0.976 0.755   1.292
11   1996    24 0.981 0.767   1.278
12   1995    23 0.960 0.755   1.272
13   2009    37 0.949 0.751   1.264
14   1997    25 0.953 0.756   1.261
15   2010    38 0.870 0.728   1.194
16   2007    35 0.881 0.758   1.162
17   1994    22 0.878 0.763   1.150
18   1993    21 0.502 0.736   0.682
19   2011    39 0.118 0.720   0.163
```

In this case, 2000 still ranks as Ramirez's best season relative to his peers, but notice that his 1999 season has fallen from 2nd to 3rd. Since by definition a league batter has an OPS+ of 1, Ramirez posted 17 consecutive seasons with an OPS that was at least 15% better than the average across the major leagues—a truly impressive feat.

Finally, not all joins are the same. An `inner_join()` requires corresponding entries in *both* tables. Conversely, a `left_join()` returns at least as many rows as there are in the first table, regardless of whether there are matches in the second table. Thus, an `inner_join()` is bidirectional, whereas in a `left_join()`, the order in which you specify the tables matters.

Consider the career of Cal Ripken, who played in 21 seasons from 1981 to 2001. His career overlapped with Ramirez's in the nine seasons from 1993 to 2001, so for those, the league averages we computed before are useful.

```
ripken <- Batting %>% filter(playerID == "ripkeca01")
nrow(inner_join(ripken, mlb, by = c("yearID" = "yearID")))

[1] 9

nrow(inner_join(mlb, ripken, by = c("yearID" = "yearID"))) #same

[1] 9
```

For seasons when Ramirez did not play, `NA`'s will be returned.

```
ripken %>%
  left_join(mlb, by = c("yearID" = "yearID")) %>%
  select(yearID, playerID, lgOPS) %>%
  head(3)

  yearID  playerID lgOPS
1   1981 ripkeca01    NA
2   1982 ripkeca01    NA
3   1983 ripkeca01    NA
```

Conversely, by reversing the order of the tables in the join, we return the 19 seasons for which we have already computed the league averages, regardless of whether there is a match for Ripken (results not displayed).

```
mlb %>%
  left_join(ripken, by = c("yearID" = "yearID")) %>%
  select(yearID, playerID, lgOPS)
```

4.5 Further resources

Hadley Wickham is an enormously influential innovator in the field of statistical computing. Along with his colleagues at RStudio and other organizations, he has made significant contributions to improve data wrangling in R. These packages are sometimes called the "Hadleyverse" or the "*tidyverse*," and are now manageable through a single `tidyverse` [231] package. His papers and vignettes describing widely used packages such as `dplyr` [234] and `tidyr` [230] are highly recommended reading. In particular, his paper on tidy data [218] builds upon notions of normal forms—common to database designers from computer science—to describe a process of thinking about how data should be stored and formatted. Finzer [77] writes of a "data habit of mind" that needs to be inculcated among data scientists. The RStudio data wrangling cheat sheet is a useful reference.

Sean Lahman, a self-described "database journalist," has long curated his baseball data set, which feeds the popular website baseball-reference.com. Michael Friendly maintains the `Lahman` R package [80]. For the baseball enthusiast, Cleveland Indians analyst Max Marchi and Jim Albert have written an excellent book on analyzing baseball data in R [140]. Albert has also written a book describing how baseball can be used as a motivating example for teaching statistics [2].

4.6 Exercises

Exercise 4.1

Each of these tasks can be performed using a single data verb. For each task, say which verb it is:

1. Find the average of one of the variables.
2. Add a new column that is the ratio between two variables.
3. Sort the cases in descending order of a variable.

4. Create a new data table that includes only those cases that meet a criterion.
5. From a data table with three categorical variables A, B, and C, and a quantitative variable X, produce a data frame that has the same cases but only the variables A and X.

Exercise 4.2

Use the `nycflights13` package and the `flights` data frame to answer the following questions: What month had the highest proportion of cancelled flights? What month had the lowest? Interpret any seasonal patterns.

Exercise 4.3

Use the `nycflights13` package and the `flights` data frame to answer the following question: What plane (specified by the `tailnum` variable) traveled the most times from New York City airports in 2013? Plot the number of trips per week over the year.

Exercise 4.4

Use the `nycflights13` package and the `flights` and `planes` tables to answer the following questions: What is the oldest plane (specified by the `tailnum` variable) that flew from New York City airports in 2013? How many airplanes that flew from New York City are included in the `planes` table?

Exercise 4.5

Use the `nycflights13` package and the `flights` and `planes` tables to answer the following questions: How many planes have a missing date of manufacture? What are the five most common manufacturers? Has the distribution of manufacturer changed over time as reflected by the airplanes flying from NYC in 2013? (Hint: you may need to recode the manufacturer name and collapse rare vendors into a category called `Other`.)

Exercise 4.6

Use the `nycflights13` package and the `weather` table to answer the following questions: What is the distribution of temperature in July, 2013? Identify any important outliers in terms of the `wind_speed` variable. What is the relationship between `dewp` and `humid`? What is the relationship between `precip` and `visib`?

Exercise 4.7

Use the `nycflights13` package and the `weather` table to answer the following questions: On how many days was there precipitation in the New York area in 2013? Were there differences in the mean visibility (`visib`) based on the day of the week and/or month of the year?

Exercise 4.8

Define two new variables in the `Teams` data frame from the `Lahman` package: batting average (BA) and slugging percentage (SLG). Batting average is the ratio of hits (`H`) to at-bats (`AB`), and slugging percentage is total bases divided by at-bats. To compute total bases, you get 1 for a single, 2 for a double, 3 for a triple, and 4 for a home run.

Exercise 4.9

Plot a time series of SLG since 1954 conditioned by lgID. Is slugging percentage typically higher in the American League (AL) or the National League (NL)? Can you think of why this might be the case?

Exercise 4.10

Display the top 15 teams ranked in terms of slugging percentage in MLB history. Repeat this using teams since 1969.

Exercise 4.11

The Angels have at times been called the California Angels (CAL), the Anaheim Angels (ANA), and the Los Angeles Angels of Anaheim (LAA). Find the 10 most successful seasons in Angels history. Have they ever won the World Series?

Exercise 4.12

Create a factor called election that divides the yearID into four-year blocks that correspond to U.S. presidential terms. During which term have the most home runs been hit?

Exercise 4.13

Name every player in baseball history who has accumulated at least 300 home runs (HR) and at least 300 stolen bases (SB).

Exercise 4.14

Name every pitcher in baseball history who has accumulated at least 300 wins (W) and at least 3,000 strikeouts (SO).

Exercise 4.15

Identify the name and year of every player who has hit at least 50 home runs in a single season. Which player had the lowest batting average in that season?

Exercise 4.16

The Relative Age Effect is an attempt to explain anomalies in the distribution of birth month among athletes. Briefly, the idea is that children born just after the age cut-off for participation will be as much as 11 months older than their fellow athletes, which is enough of a disparity to give them an advantage. That advantage will then be compounded over the years, resulting in notably more professional athletes born in these months. Display the distribution of birth months of baseball players who batted during the decade of the 2000s. How are they distributed over the calendar year? Does this support the notion of a relative age effect?

Exercise 4.17

The Violations data set in the mdsr package contains information regarding the outcome of health inspections of restaurants in New York City. Use these data to calculate the median violation score by zip code for zip codes in Manhattan with 50 or more inspections. What pattern do you see between the number of inspections and the median score?

Exercise 4.18

Download data on the number of deaths by firearm from the Florida Department of Law Enforcement. Wrangle these data and use ggplot2 to re-create Figure 6.1.