3

Data Management, Manipulation, and Exploration with *dplyr*

Exploring, manipulating, and graphing your data are fundamental parts of data analysis. In fact, we put huge emphasis on spending time developing familiarity with the data you've collected, culminating in a figure that reflects the questions you set out to ask when you actually collected the data. Our workflow starts with making a picture. Before you do any statistics. Because, surely, what you would love to see is that the patterns you *expected* to see in your data are actually there.

There are two fundamental toolsets for this initial effort: manipulation tools and graphing tools. In this chapter, we introduce the ***dplyr*** package to do various common data manipulations. This is extremely important learning, as we almost always need to do some data manipulation, such as subsetting or calculation of mean ± SE. You should do all your data manipulation in R (and none anywhere else), and this chapter gives you the tools to do this. It enables your entire workflow to be contained, as far as is humanly possible, in one place: your R script.

This chapter is definitely one you should work along with, building a script as you go. If you are not yet comfortable with libraries, importing data, and annotating a script, you will be by the end. Review the tips and

Getting Started with R Second Edition. Andrew Beckerman, Dylan Childs, & Owen Petchey: Oxford University Press (2017).
DOI 10.1093/oso/9780198787839.001.0001

tricks from the previous chapters about sending data from the script to R. As you go through this chapter, build a script, with copious annotations. Type the commands into the script, and submit them to the Console. Do not type them direct into the Console! Be strict with yourself about both these things. Building good habits now is really important.

We'll be working with the compensation data we imported in the previous chapter, so you should be ready to go. (The compensation data are available at `http://www.r4all.org/the-book/datasets`). You should have ***dplyr*** and ***ggplot*** installed already, so you can set up a script with some initial annotation, the **library**() function, and the brain-clearing **rm**(list = **ls**()). This should be followed by use of **read.csv**() to import the `compensation.csv` dataset.

In the Preface we described why we are teaching you to use ***dplyr*** and its functions, rather than the classic ways of managing and manipulating data. Nevertheless, you may be curious about the classic ways, perhaps because you have some experience with them. In an appendix to this chapter, we briefly make some comparisons between the classic ways and the newer ***dplyr*** ways.

3.1 Summary statistics for each variable

3.1.1 THE COMPENSATION DATA

Just so you have a frame of reference going forward, the compensation data are about the production of fruit (apples, kg) on rootstocks of different widths (mm; the tops are grafted onto rootstocks). Furthermore, some trees are in parts of the orchard that allow grazing by cattle, and others are in parts free from grazing. Grazing may reduce the amount of grass, which might compete with the apple trees for resources.

3.1.2 THE **summary**() FUNCTION

We covered a few handy tools in the previous chapter for looking at pieces of your data. While **names**(), **head**(), **dim**(), **str**(), **tbl_df**(), and **glimpse**() are good for telling you what your data look like (i.e. correct number of

rows and columns), they don't give much information about the particular values in each variable, or summary statistics of these. To get this information, use the **summary**() function on your data frame. Go ahead and apply the **summary**() function to these data:

```
compensation <- read.csv("compensation.csv")
glimpse(compensation) # just checkin'

# get summary statistics for the compensation variables
summary(compensation)
```

```
##       Root            Fruit            Grazing
##  Min.   : 4.426   Min.   : 14.73   Grazed  :20
##  1st Qu.: 6.083   1st Qu.: 41.15   Ungrazed:20
##  Median : 7.123   Median : 60.88
##  Mean   : 7.181   Mean   : 59.41
##  3rd Qu.: 8.510   3rd Qu.: 76.19
##  Max.   :10.253   Max.   :116.05
```

The **summary**() function in R gives us the median, mean, interquartile range, minimum, and maximum for all numeric columns (variables), and the 'levels' and sample size for each level of all categorical columns (variables). It's worth looking carefully at these summary statistics. They can tell you if there are unexpectedly extreme, implausible, or even impossible values—a good first pass at your data. Now let's continue to use these data to learn how ***dplyr*** can explore, subset, and manipulate data.

3.2 *dplyr* verbs

We are now going to introduce to you five 'verbs' that are also functions in ***dplyr***—a package focused on manipulating data. The verbs are **select**(), **slice**(), **filter**(), **arrange**(), and **mutate**(). **select**() is for selecting columns, and **slice**() is for selecting rows. **filter**() is for selecting subsets of rows, conditional upon values in a column. **arrange**() is for sorting rows and **mutate**() is for creating new variables in the data frame. These are core functions for core activities centred on grabbing pieces of, or

subsetting, your data (e.g. **select**(), **slice**(), **filter**()) transforming variables (e.g. **mutate**()), or sorting (e.g. **arrange**()). Of course, there are more . . . a quick Google search on 'cheatsheet ***dplyr***' or following the RStudio -> Help Menu -> Cheatsheets trail will take you to lovely places.

The key to using ***dplyr*** is to remember that the first argument to ALL ***dplyr*** functions is the data frame. You might try saying this 25 times. The first argument for ***dplyr*** functions is . . .

3.3 Subsetting

Three verbs form the core of subsetting data: they get columns, rows, or subsets of rows.

3.3.1 select()

select() grabs columns. Of course, it helps to know the name of the columns, so if you need to, use **names**(compensation) first. Here is how we can use it to get the Fruit column (we have truncated the output; you will see more rows than we have printed):

```
select(compensation, Fruit)  # use the Fruit column
```

```
## Source: local data frame [40 x 1]
##
##    Fruit
##    (dbl)
## 1  59.77
## 2  60.98
## 3  14.73
## 4  19.28
## 5  34.25
## 6  35.53
## 7  87.73
## 8  63.21
## 9  24.25
## 10 64.34
## ..   ...
```

Note: If you get an error `Error: could not find function "select"` then you have either not put **library**(dplyr) at the top of your script, or have not run that line of code.

So, let's work through some details about how ***dplyr*** works. First, you can see that it is quite easy to use. If we want a column, we tell ***dplyr*** which dataset to look in, and which column to grab. Super.

The astute observer will recognize something else very interesting. **select**(), as a ***dplyr*** verb, uses a data frame *and* returns a data frame. If you scroll up to see the top of the output, you will see the column name 'Fruit'. You have asked for part of a data frame, and you get one back—in this case a one-column data frame. Not all R functions act like this. The appendix to this chapter provides some detail on base (classic) R functions that do similar things to ***dplyr*** functions, but can return different types of object.

Finally, you may also notice that **select**() seems to do one thing only. This is totally true. All ***dplyr*** functions do one thing, and one thing *very* fast and *very* effectively.

select() can also be used to select all columns *except* one. For example, if we wanted to leave out the Root column, leaving only the Fruit and Grazing columns:

```
select(compensation, -Root)  # that is a minus sign
```

```
## Source: local data frame [40 x 2]
##
##    Fruit  Grazing
##    (dbl)   (fctr)
## 1  59.77 Ungrazed
## 2  60.98 Ungrazed
## 3  14.73 Ungrazed
## 4  19.28 Ungrazed
## 5  34.25 Ungrazed
## 6  35.53 Ungrazed
## 7  87.73 Ungrazed
## 8  63.21 Ungrazed
## 9  24.25 Ungrazed
## 10 64.34 Ungrazed
## ..   ...      ...
```

3.3.2 slice()

slice() grabs rows. It works by returning specific row numbers you ask for. You can ask for one row, a sequence, or a discontinuous set. For example, to get the second row, we use

```
slice(compensation, 2)

##    Root Fruit  Grazing
## 1 6.487 60.98 Ungrazed
```

If we want the second to the tenth, we can invoke the : to generate the sequence:

```
slice(compensation, 2:10)

##    Root Fruit  Grazing
## 1 6.487 60.98 Ungrazed
## 2 4.919 14.73 Ungrazed
## 3 5.130 19.28 Ungrazed
## 4 5.417 34.25 Ungrazed
## 5 5.359 35.53 Ungrazed
## 6 7.614 87.73 Ungrazed
## 7 6.352 63.21 Ungrazed
## 8 4.975 24.25 Ungrazed
## 9 6.930 64.34 Ungrazed
```

And discontinuous sets are easy, but we need to *collect* the row numbers using another helper function in R called c():

```
slice(compensation, c(2, 3, 10))

##    Root Fruit  Grazing
## 1 6.487 60.98 Ungrazed
## 2 4.919 14.73 Ungrazed
## 3 6.930 64.34 Ungrazed
```

One thing you may notice about slice() is that it also returns a data frame, but it does not return the row number identity found in the original data. You have new, continuous row numbers. Just be aware.

3.3.3 filter()

filter() is super-powerful subsetting. It requires some basic knowledge of logical operators and boolean operators in R. Let's first work through those, and then learn how to apply them via filter().

Logical operators and booleans

R has a complete set of logical operators. In Table 3.1, we provide some insight into common logical and boolean operators, with examples of their use in **filter**().

One of the tricks you can use to understand how this works can be seen in the next few snippets of code. Let's see how R interprets > first:

```
with(compensation, Fruit > 80)

##   [1] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
##   [8] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [15] FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE
##  [22] FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE
##  [29]  TRUE  TRUE FALSE FALSE FALSE FALSE  TRUE
##  [36] FALSE FALSE FALSE FALSE  TRUE
```

First, **with**() is a handy function . . . it says to R, 'LOOK in this data frame, and do what comes next, and then stop looking'. Second, you will notice that the > symbol, a logical, on its own produces a sequence of TRUE and FALSE, identifying *where* in the `Fruit` vector it is TRUE that the value of `Fruit` is > 80. This is handy. Other R functions can use this set of TRUE and FALSE values to retrieve or omit data. This set of TRUE and FALSE values is the information passed to **filter**() . . . and this is what **filter**() can act on and return to you.

Using filter()

Let's imagine we are interested in all of the trees producing a large amount of fruit. We see from the **summary**() output above that big fruit production means > 80 kg. As with all ***dplyr*** functions, we first supply the data frame, and then the condition by which we judge whether to return rows (Table 3.1):

```
# find the rows where it is true that Fruit is >80 return
# them as a data frame
filter(compensation, Fruit > 80)

##     Root  Fruit  Grazing
## 1  7.614  87.73 Ungrazed
```

Table 3.1 Some of the more common logical operators and boolean operators, with examples of use in filter().

R logical or boolean	Meaning	Example	Note
"=="	Equals	filter(compensation, Fruit ==80)	The == finds in which rows it is TRUE that the condition is met.
"!="	Does not equal	filter(compensation, Fruit !=80)	The !=finds in which rows it is TRUE that the condition is NOT met.
<, >, >=, <=	Less than, greater than, equal to or greater than, equal to or less than	filter(compensation, Fruit <=80)	
|	OR	filter(compensation, Fruit >80 | Fruit < 20)	OR: in which rows is it TRUE that FRUIT is > 80 OR FRUIT is < 20? It will return all of these rows.
&	AND	filter(compensation, Fruit >80 & Root < 2.3)	AND: BOTH conditions must be true. This will return rows where it is true that both conditions, from two variables, are met.

```
## 2  7.001  80.64 Ungrazed
## 3 10.253 116.05   Grazed
## 4  9.039  84.37   Grazed
## 5  8.988  80.31   Grazed
## 6  8.975  82.35   Grazed
## 7  9.844 105.07   Grazed
## 8  9.351  98.47   Grazed
## 9  8.530  83.03   Grazed
```

We can easily select rows according to multiple conditions. For example, to keep only rows with Fruit > 80 OR less than 20, we employ the boolean *or* symbol |:

```
filter(compensation, Fruit > 80 | Fruit < 20)

##      Root  Fruit  Grazing
## 1   4.919  14.73 Ungrazed
## 2   5.130  19.28 Ungrazed
## 3   7.614  87.73 Ungrazed
## 4   7.001  80.64 Ungrazed
## 5   4.426  18.89 Ungrazed
## 6  10.253 116.05   Grazed
## 7   9.039  84.37   Grazed
## 8   6.106  14.95   Grazed
## 9   8.988  80.31   Grazed
## 10  8.975  82.35   Grazed
## 11  9.844 105.07   Grazed
## 12  9.351  98.47   Grazed
## 13  8.530  83.03   Grazed
```

3.3.4 MAKING SURE YOU CAN *use* THE SUBSET OF DATA

At the moment, you've been asking R to do stuff, and report the outcome of doing it in the Console. However, very often, you will want to use the results in subsequent jocularities. As you will recall from the previous chapters, the assignment operator (<-) is what you need to use. If we want the low and high Fruit-producing trees for some other activity, we have to assign the result to an object:

```
lo_hi_fruit <- filter(compensation, Fruit > 80 | Fruit < 20)
# now look at it
lo_hi_fruit

##      Root  Fruit  Grazing
## 1   4.919  14.73 Ungrazed
## 2   5.130  19.28 Ungrazed
## 3   7.614  87.73 Ungrazed
```

```
## 4   7.001  80.64 Ungrazed
## 5   4.426  18.89 Ungrazed
## 6  10.253 116.05   Grazed
## 7   9.039  84.37   Grazed
## 8   6.106  14.95   Grazed
## 9   8.988  80.31   Grazed
## 10  8.975  82.35   Grazed
## 11  9.844 105.07   Grazed
## 12  9.351  98.47   Grazed
## 13  8.530  83.03   Grazed
```

Commit this to memory: *assign the values returned by subsetting using **filter** to an object (word) if you want to use them again.*

3.3.5 WHAT SHOULD MY SCRIPT LOOK LIKE NOW?

At this point, we'd like to show you what your script should (or might) look like if you were following along. More annotation is always good. Note that annotation can go above, to the right of, or below chunks of code. Note too that we have white space between chunks of code:

```
# my first dplyr script

# clear R's brain
rm(list=ls())

# libraries I need (no need to install...)
library(dplyr)
library(ggplot2)

# get the data
compensation <- read.csv('compensation.csv')

# quick summary
summary(compensation)

# using dplyr; always takes and gives a data frame

# columns
select(compensation, Fruit) # gets the Fruit column
select(compensation, -Root) # take Root column out from data

# rows
slice(compensation, c(2,3,10)) # get 2nd, 3rd & 10th rows

# gets rows for each condition, and assigns to an object
lo_hi_fruit <- filter(compensation, Fruit > 80 | Fruit < 20)

# run this to see what the above line 'saved' for later use.
lo_hi_fruit
```

3.4 Transforming

Transforming columns of your data is a common practice in many biological fields. For example, it is common to log-transform variables for graphing and data analysis. You may also find yourself in a situation where you want to present or analyse a variable that is function of other variables in your data. For example, if you have data on the total number of observations and on the number that were blue, you might want to present the proportion of observations that were blue. Here, we reveal the basic approach to using **mutate**() to achieve these ends.

3.4.1 mutate()

As with all ***dplyr*** functions . . . **mutate**() starts with the data frame in which the variables reside, and then designates a new column name and the transformation. For example, let's log-transform Fruit, and call it logFruit. We will make this new column appear in our working data frame by employing a neat trick, assigning the values returned by **mutate**() to an object of the *same* name as the original data. We are essentially overwriting the data frame! We will also use **head**() to limit the number of rows we see . . . just for clarity:

```
# what does compensation look like now?
head(compensation)

##    Root Fruit  Grazing
## 1 6.225 59.77 Ungrazed
## 2 6.487 60.98 Ungrazed
## 3 4.919 14.73 Ungrazed
## 4 5.130 19.28 Ungrazed
## 5 5.417 34.25 Ungrazed
## 6 5.359 35.53 Ungrazed

# use mutate
# log(Fruit) is in the column logFruit
# all of which gets put into the object compensation
compensation <- mutate(compensation, logFruit = log(Fruit))

# first 6 rows of the new compensation
head(compensation)

##    Root Fruit  Grazing logFruit
## 1 6.225 59.77 Ungrazed 4.090504
```

```
## 2 6.487 60.98 Ungrazed 4.110546
## 3 4.919 14.73 Ungrazed 2.689886
## 4 5.130 19.28 Ungrazed 2.959068
## 5 5.417 34.25 Ungrazed 3.533687
## 6 5.359 35.53 Ungrazed 3.570377
```

Just to drive home a very nice point about working with R and a script, ask yourself whether you've changed anything in your safe, backed-up, securely stored `.csv` file on your computer. Have you? Nope. We are working with a copy of our data inside R, manipulating these data, but at no time are we altering the original raw data. You can always go back to those if you need.

3.5 Sorting

3.5.1 arrange()

Sometimes it's important or desirable to put the observations (rows) of our data in a particular order, i.e. to sort them. It may simply be that you'd like to look at the dataset, and prefer a particular ordering of rows. For example, we might want to see the compensation data in order of increasing `Fruit` production. We can use the **arrange**() function:

```
arrange(compensation, Fruit)
```

```
##    Root Fruit  Grazing logFruit
## 1 4.919 14.73 Ungrazed 2.689886
## 2 6.106 14.95   Grazed 2.704711
## 3 4.426 18.89 Ungrazed 2.938633
## 4 5.130 19.28 Ungrazed 2.959068
## 5 4.975 24.25 Ungrazed 3.188417
## 6 5.451 32.35 Ungrazed 3.476614
```

Another reason for arranging rows in increasing order is if we'd like to perform analyses that need a specific order. For example, some types of time series analyses need the data in the correct temporal order (and may not themselves ensure this). In this case, we would be very well advised to do the sorting ourselves, and check it carefully. (Look at the help file for **arrange**() to see how to sort by multiple variables.)

3.6 Mini-summary and two top tips

So, that was a rapid introduction to five key verbs from ***dplyr***. ***dplyr*** functions are fast and consistent, and do one thing well. On this latter note, here is *Top Tip 1*: you can use more than one ***dplyr*** function in one line of code! Imagine you want fruit production > 80, and the rootstock widths ONLY. That's a **filter**() and a **select**() agenda, if we've ever heard one:

```
# Root values from Fruit > 80 subset
select(filter(compensation, Fruit > 80), Root)

##      Root
## 1   7.614
## 2   7.001
## 3  10.253
## 4   9.039
## 5   8.988
## 6   8.975
## 7   9.844
## 8   9.351
## 9   8.530
```

Reading this from the inside out helps. Here we've asked for ***dplyr*** to **filter**() the data first, then take the data frame from **filter**(), and use **select**() to get the `Root` column only. Nice.

But this leads us to *Top Tip 2*. Built into ***dplyr*** is a very special kind of magic, provided by Stefan Milton Bache and Hadley Wickham in the ***magrittr*** package. This gets installed when you install ***dplyr***, so you don't need to get it yourself. The magic is found in a symbol called a *pipe*. In R, the pipe command is %>%. You can read this like 'put the answer of the left-hand command into the function on the right'.

Some of you may find this so logical[1] and so much fun that you never stop. We haven't stopped.

Let's translate the two-function code above into 'piped' commands. The art of piping with ***dplyr*** is to remember to always start with the data frame . . .

[1] Some of you may know about the Unix/Linux pipe command '|'. This is modelled on that; the '|' is used in other places in R.

```
# Root values from Fruit > 80 subset
# Via piping
compensation %>%
  filter(Fruit > 80) %>%
    select(Root)

##     Root
## 1  7.614
## 2  7.001
## 3 10.253
## 4  9.039
## 5  8.988
## 6  8.975
## 7  9.844
## 8  9.351
## 9  8.530
```

Read from left to right, top to bottom, this says (1) work with the compensation data, (2) **filter**() it based on the fruit column, getting all rows where it is true that Fruit > 80, and then pass this data frame to (3) **select**() and return *only* the `Root` column as the final data frame. Sweet.

There are a few reasons we like this. First, it may be nicer and easier to read than putting functions inside functions. Second, this gets even more valuable when more than two functions can potentially be used inside each other. Third, um, well, it is just so cool!

3.7 Calculating summary statistics about groups of your data

At this point, you should be feeling pretty confident. You can import and explore the structure of your data, and you can access and even manipulate various parts of it using five verbs from ***dplyr***, including **filter**() and **select**(). What next? In this section, we introduce to you functions that help you generate custom summaries of your data. We will continue working with the compensation data.

In this data frame we have a categorical variable, Grazing. It has two levels, *Grazed* and *Ungrazed*. This structure to the data means we might be able to calculate the *mean* fruit production in each category. Whenever you

have structure, or groups, you can generate with R some rather amazing and fast summary information.

The two key ***dplyr*** functions for this are **group_by**() and **summarise**(). We also introduce the functions **mean**() and **sd**() (standard deviation).

3.7.1 OVERVIEW OF SUMMARIZATION

Summarization is accomplished in a series of steps. The core idea, using ***dplyr***, is to:

1. Declare the data frame and what the grouping variable is.
2. Provide some kind of maths function with which to summarize the data (e.g. **mean**() or **sd**()).
3. Provide a nice name for the values returned.
4. Make R use all of this information.

We offer two methods for making this happen.

3.7.2 METHOD 1: NESTED, NO PIPE

In the nested approach, we construct everything as follows. A good test of knowing your data is asking what you might expect. Here, using one grouping variable with two levels and asking for the means, we expect a data frame to be returned with two numbers, a mean for *Grazed* and a mean for *Ungrazed* `Fruit` production:

```
summarise(
  group_by(compensation, Grazing),
           meanFruit = mean(Fruit))

## Source: local data frame [2 x 2]
##
##     Grazing meanFruit
##      (fctr)     (dbl)
## 1   Grazed   67.9405
## 2 Ungrazed   50.8805
```

The second line of code has some good stuff on the 'inside'. The **group_by**() function works with the data frame and declares Grazing as

our grouping variable. Of course, if we have more than one grouping variable, we can add them with commas in between. It's that easy.

The third line is where we ask for the mean to be calculated for the `Fruit` column. We can do this, and R knows where to look, because the **group_by**() function has set it all up. The 'word' meanFruit is some formatting for the output, as you can see in the output.

Beautiful, eh! We get the mean of each of the `Grazing` treatments, just as expected. And don't forget, if you want to use the means, you must use the <- symbol and assign the result to a new object. Perhaps you'll call it mean.fruit:

```
mean.fruit <- summarise(
  group_by(compensation, Grazing),
           meanFruit = mean(Fruit))
```

3.7.3 METHOD 2: PIPE, NO NESTING

Some of you may have already figured out the piping method. It is perhaps more logical in flow. As above, we always start by declaring the data frame. One big difference is that **summarise**() is now third in the list, rather than on the 'outside'. Start with the data, divide it into groups, and calculate the mean of the fruit data in each group. That's the order:

```
compensation %>%
  group_by(Grazing) %>%
    summarise(meanFruit = mean(Fruit))
```

3.7.4 SUMMARIZING AND EXTENDING SUMMARIZATION

group_by() and **summarise**() are wonderful functions. You can **group_by**() whichever categorical variables you have, and calculate whatever summary statistics you like, including **mean**(), **sd**(), **median**(), and even functions that you make yourself.

In fact, with very few changes to the above code, you can ask for more than one 'statistic' or metric:

```
compensation %>%
  group_by (Grazing) %>%
    summarise(
      meanFruit = mean(Fruit),
      sdFruit = sd(Fruit)
```

This makes creating fast and efficient summaries of your data over many dimensions a doddle.

3.8 What have you learned . . . lots

From Chapter 1 to now, you've gained a great deal of understanding of how R does maths, works with objects, and can make your life easy and *very* organized by allowing you to work in a script. You've learned about tidy data and, importantly, how to import data into R and now know to explore those data. You've gained new skills via ***dplyr***, letting you work with columns, rows, and subsets of your data. Not to mention creating new transformations of your data, and rearranging them. Oh, and that magical pipe.

Furthermore, you've calculated some basic statistics, and in doing so learned several new tricks about working with groups of your data. You are beginning to have the capacity to see the patterns in your data, patterns that you expected. Most importantly, from an R data analysis/data management perspective, you have a *permanent, repeatable, annotated, shareable, cross-platform, executable record* of what you are doing—the script.

Have you saved your script?

We think you are ready for some serious fun now. That serious fun comes from moving from just working with your data, which is super-important, to making figures. Pictures of your data that have meaning, reflect theory, and, if you are lucky, show you the answer to the questions you set out to ask when you were collecting the data. This is what Chapter 4 is all about. See you there.

Appendix 3a Comparing classic methods and *dplyr*

To be clear, the following is 'possibly nice-to-know' information, and probably not 'need-to-know'. One instance in which you might benefit from it is if you have access to scripts written in your lab long ago, or

are working with seasoned R veterans who've not themselves embraced fully the Hadleyverse. A summary comparison of some 'base' methods and ***dplyr*** methods is given in Table 3.2. A brief description of this comparison follows.

Using ***dplyr***, we use the functions **select**(), **slice**(), and **filter**() to get subsets of data frames. The classic method for doing this often involves something called *indexing*, and this is accomplished with square brackets and a comma, with the rows we want before the comma and the columns/variables we want after the comma: something like `mydata[rows,columns]`. There are lots of ways of specifying the rows and columns, including by number, name, or logical operator. It's very flexible, quick, and convenient in many cases. Selecting rows can also be done with the **subset**() base R function, which actually possesses the combined functionality of **filter**() and **select**() from ***dplyr***. Ordering rows or columns can be achieved by a combination of indexing and the base R **order**() function.

Adding a new variable, which might be some transformation of existing ones, is also very similar between base R, using the **transform**() function, and ***dplyr***, using the **mutate**() function. People often add columns by using a dollar sign followed by the new variable, for example mydata$new_variable <- mydata$old_variable (see Table 3.2).

The 'old skool' and still useful methods for getting information about groups of data use functions like **aggregate**() and **tapply**()—both of these were covered in detail in the previous edition of this book. These functions have separate `arguments` that specify the groups and the summary statistics. In ***dplyr*** the groups are specified by the **group_by**() function and the summary statistics by **summarise**().

Appendix 3b Advanced *dplyr*

The wonderful depths of ***dplyr*** are beyond the scope of an introductory R text such as this book. One of our justifications for teaching ***dplyr*** was,

Table 3.2 Comparison (using the compensation data) of common data manipulation methods using the classic way (base R) and the modern way, with functions in the ***dplyr*** package. The placement of the comma, when using [ ]'s, is quite important and subtle. The help files for `Extract` (for [ ]), `subset`, `order`, `aggregate`, and `tapply` are worth consulting should you run into these functions.

Operation	Base R example	*dplyr* function	*dplyr* example
Select some rows	compensation[c(2,3,10),]	slice()	slice(compensation, c(2,3,10))
Select some columns	compensation[,1:2] OR compensation[,c("Root", "Fruit")]	select()	select(compensation, Root, Fruit)
Subset	compensation[compensation $Fruit>80,] OR subset(compensation, Fruit>80)	filter()	filter(compensation, Fruit>80)
Order the rows	compensation[order(compensation $Fruit),]	arrange()	arrange(compensation, Fruit)
Add a column	compensation$logFruit <- log(compensation$Fruit) OR transform(compensation, logFruit=log(Fruit)	mutate()	mutate(compensation, logFruit = log(Fruit)
Define groups of data	-	group_by()	compensation %>% group_by(Grazing)
Summarize the data	aggregate(Fruit ~ Grazing, data = compensation, FUN = mean) OR tapply(compensation$Fruit, list(compensation$Grazing), mean)	summarise() AND group_by()	compensation %>% group_by(Grazing) %>% meanFruit = mean(Fruit)

however, those depths. So we mention a few of them here. And probably not too long after you finish with this book, you'll benefit from knowing a bit about them.

First, an easy one: you can merge together two datasets using one of the **join**() functions. There are a few versions, because there are a few different ways to merge two datasets, such as keeping all rows, even ones that aren't common between the two datasets, or keeping just the common rows.

Second, and this one is absolutely fantastic, you can apply transformations to groups of your data. You already know how to summarize groups of your data, but this is different. Let's say, for example, you want to subtract the mean of each group of your data from each value in the corresponding group. To do this, you first declare the groups, just as before, using the **group_by**() function. Then you use the **mutate**() function, which you already know as the function for transforming your data. The following code does this, subtracting the mean fruit production for each grazing treatment from the values for the corresponding grazing treatment, using piping, of course:

```
compensation_mean_centred <- compensation %>%
  group_by(Grazing) %>%
    mutate(Fruit_minus_mean = Fruit - mean(Fruit))
```

Magic, eh? Now check that the mean of each group of the Fruit_minus_mean variable is equal to zero.

Third, and finally, you can actually ask for more or less anything to be done to a group of your data. For example, do a linear model for each group. The new function for this is, surprise surprise, **do**(). Here's how we would do a linear model for each grazing treatment:

```
library(broom)
compensation_lms <- compensation %>%
  group_by(Grazing) %>%
    do(tidy(lm(Fruit ~ Root, data=.)))
```

Note that we used a function called **tidy**() from the package ***broom*** to tidy up the output of the linear model function **lm**(). Otherwise, things are rather messy.

There is much, much more one can do with ***dplyr***. That is just a taster. Just keep it in the back of your mind. And have a play with one of your own datasets, using that and your scientific question, to drive (constrain) your learning. Otherwise, you may get completely and wonderfully lost in the depths of ***dplyr***.