2

Getting Your Data into R

By now you must be quite excited that simply by typing some instructions, you can get R to do stuff. You may also be excited about making graphs in R. You have learned many new functions already. . . but there are crucial things to learn about data before we move on.

We can tell . . . you really want to get your data into R's brain. It is empty—R's brain, not yours. This chapter covers four things: how to prepare your data so they will be easy to import into R, and to work with in R; how to import your data into R; and how to check they were correctly imported. We also touch on some troubleshooting skills along the way. And we also provide some methods for how to deal with badly formatted data in the appendix to this chapter. Phew. Data time.

2.1 Getting data ready for R

One of the most frequent stumbling blocks for new students and seasoned researchers using R is actually just getting the data into R. This is really unfortunate, since it is also most people's first experience with R! Let's make this first step very easy.

As in many walks of life, careful preparation is the key to good results. So, hopefully you're reading this before you collected, acquired, or at least

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

Downloaded from https://academic.oup.com/book/27290/chapter/196933265 by University of Glasgow user on 25 September 2024

typed in the data you'll be working with. There's no disaster if you have already done so, however.

2.1.1 SETTING UP YOUR OWN DATA

R likes data in which there is one observation in a row, and each variable is represented by its own column. Some people call this 'long format'; others call it 'tidy data'. R likes data that look like this in the sense that many of the tools you use in R (the tools are *functions*) work on data that look like this.

Exactly what does it mean to have data with one observation per row and each variable in only one column? Imagine a dataset containing the heights of men and of women. This could be arranged in two columns, one containing the heights of the men, and another the heights of the women (Figure 2.1).

This is *not* what R likes! There are two observations in each row, the values of heights are in two columns, and the gender variable is represented in two columns, even though it is the same variable. The alternative, R-friendly arrangement is to generate one column called 'gender' and a second called 'height' (Figure 2.2). This arrangement has more rows

	A	B
1	Male.height	Female.height
2	138	115
3	161	132
4	183	149
5	136	158
6	183	111
7	186	158
8	174	127
9	167	143
10	191	114
11	147	168

Figure 2.1 An example of not such a good way to arrange data for them to be easily worked with in R. There are two columns that have the same type of information (heights), each row contains two observations.

Downloaded from https://academic.oup.com/book/27290/chapter/196933265 by University of Glasgow user on 25 September 2024

	A	B
1	Gender	Height
2	Male	138
3	Male	161
4	Male	183
5	Male	136
6	Male	183
7	Male	186
8	Male	174
9	Male	167
10	Male	191
11	Male	147
12	Female	115
13	Female	132
14	Female	149
15	Female	158
16	Female	111
17	Female	158
18	Female	127
19	Female	143
20	Female	114
21	Female	168

Figure 2.2 An example of a good way to arrange data for them to be easily worked with in R. The two columns represent each variable (gender, height). Each row contains only one observation, i.e. information about one person.

(twice as many if there are the same numbers of both genders), and this is why it's called 'long format', though a better name might be 'tall format'.

Here is another example gone wrong, and how to fix it. Imagine you recorded the heights of individuals through time. You could have a separate column for each date at which the height was recorded (Figure 2.3). R doesn't like this, and neither should you.

If anyone gives you data like this, they owe you a <insert preferred beverage>. If you give yourself data like this, do ten push-ups. The better arrangement is to specify Year as a variable—have a column called 'Year' (Figure 2.4). To make good sense, such a dataset would also have columns Person, Year, and of course Height.

Downloaded from https://academic.oup.com/book/27290/chapter/196933265 by University of Glasgow user on 25 September 2024

	A	B	C	D	E	F	G	H
1	Person	Height.year1	Height.year2	Height.year3	Height.year4	Height.year5	Height.year6	Height.year7
2	Ellie	138	142	145	150	154	157	162
3	Andrew	161	170	175	182	187	191	191
4	Noah	120	132	140	148	154	159	165
5	Darby	136	145	150	155	159	165	167

Figure 2.3 Ouch! It makes our eyes hurt, and will hurt R too. There are many columns with the same type of information (heights). Each row contains multiple observations.

	A	B	C
1	Person	Year	Height
2	Ellie	1	138
3	Andrew	1	161
4	Noah	1	120
5	Darby	1	136
6	Ellie	2	142
7	Andrew	2	170
8	Noah	2	132
9	Darby	2	145
10	Ellie	3	145
11	Andrew	3	175
12	Noah	3	140
13	Darby	3	150
14	Ellie	4	150
15	Andrew	4	182
16	Noah	4	148
17	Darby	4	155
18	Ellie	5	154
19	Andrew	5	187
20	Noah	5	154
21	Darby	5	159
22	Ellie	6	157
23	Andrew	6	191
24	Noah	6	159
25	Darby	6	165
26	Ellie	7	162
27	Andrew	7	191
28	Noah	7	165
29	Darby	7	167

Figure 2.4 Much better! R will like you if you arrange data like this. There is only one column for each type of information. Each row has only one observation.

An exercise in data preparation

This section tells you how to prepare a data sheet and enter data. You may not have any now, but it's worth reading through to see how it would be done and, specifically, for some conventions for data entry (e.g. about missing values and file formats).

Downloaded from https://academic.oup.com/book/27290/chapter/196933265 by University of Glasgow user on 25 September 2024

Now that you know how R likes data arranged, make a blank data sheet in Excel (or some other spreadsheet software) and print it out ready for you to fill it in (or keep it on your techno device if you plan to fill it in directly there). Keep column names informative, brief, and simple. Try not to use spaces or special symbols, which R can deal with but may not do so particularly nicely.

Also, if you have categorical variables, such as sex (male, female), use the *actual* category names rather than codes for them (e.g. don't use 1 = male, 2 = female). R can easily deal with variables that contain words, and this will make subsequent tasks in R much more efficient and reliable.

Now . . . in between making this sheet and coming back to R, you will have done some science and filled in your data sheet. Rest assured you can continue using the book. We will provide you with a dataset to carry on.

Top Tip 1. As you get to the data entry point of your work, *remember*, all cells in the data sheet should have an entry, even if it is something saying 'no observation was made'. The expected placeholder for missing data in R is NA. Other options are possible, but be careful. A blank cell is ambiguous: perhaps you just forgot to make the observation, or perhaps it really couldn't be made. Good practice is to look over the data sheet every hour or so, checking for and adding entries in empty cells.

Once you've entered the data on the data sheet, type your data into a spreadsheet program. Once complete, print your entered data onto paper, and check that this copy of your data matches the data on your original data sheets. Correct any mistakes you made when you typed your data in.

Top Tip 2. We don't advocate saving your file as an `.xls`, `.xlsx`, `.oo`, or `.numbers` file. Instead, we actively argue for using a 'comma-separated values' file (a `.csv` file). A `.csv` file is easily transported with a low memory and small size footprint among computing platforms. In Excel, Open Office, or Numbers, after you click Save As . . . you can change the format of the file to 'comma-separated values', then press Save. Excel might then, or when you close the Excel file, ask if you're sure you'd like to save your data in this format. Yes, you are sure!

At this point in our workflow, you have your original paper data sheets, a digital copy of the data in a `.csv` file, and a printed copy of the data

Downloaded from https://academic.oup.com/book/27290/chapter/196933265 by University of Glasgow user on 25 September 2024

from the `.csv` file. One of the remarkable things about R is that once a copy of your 'raw data' is established, the use of R for data visualization and analysis will never require you to alter the original file (unless you collect more data!). Therefore keep it very safe! In many statistical and graphing programs, your data spreadsheet has columns added to it, or you manipulate columns or rows of the spreadsheet before doing some analysis or graphic. With R, your original data file can and must always remain unmanipulated. If any adjustments do occur, they occur in an R copy only and only via your explicit instructions harboured in the script.

2.1.2 SOMEONE ELSE'S DATA?

And what if you already have your data, and they're not in the arrangement R likes? Or perhaps your data came from someone else and you didn't have a chance to tell them how R likes data arranged? Or your data were recorded by a machine, and you therefore didn't have much choice about how they were recorded? (Fortunately, most machines seem to know how R likes data to be arranged, and therefore adhere to the 'one observation per row and a variable per column' rule.) One thing you can do is send it back, saying how you need it! When you think such a request might not go down too well, you're going to have to do the work yourself.

You can rearrange the data in Excel, though this can cause errors, can be awfully time-consuming, and is tedious and boring for big datasets. The alternative is to let R do the hard work: let R rearrange the data. We provide details on how to do this in the appendix at the end of this chapter.

2.2 Getting your data into R

Brilliant . . . you now have your beautiful data ready for R. If you just happened to miss our emphasis above, it should be a long/tall-format `.csv` file. But how do you get the data in this `.csv` file into R? How do you go from making a dataset in a `.csv` file to having a working copy in R's brain to use for your analysis?

Downloaded from https://academic.oup.com/book/27290/chapter/196933265 by University of Glasgow user on 25 September 2024

First things first . . . you will need the datasets that we use in this book. You can get these from `http://www.r4all.org/the-book/datasets`. Don't forget to unzip them and put them somewhere nice! Part of keeping your data safe is putting it somewhere safe. But where? We like to be organized, and we want you to be organized too.

Make a new folder in the main location/folder in which you store your research information—perhaps you have a `Projects` folder in `MyDocuments` (old PC) or `Documents` (new PC and Mac)—create a folder inside this location. Give the folder an informative name—perhaps '`MyFirstAnalysis`'. It should really go without saying that you should be able to find this folder, i.e. you should be able to navigate to it within Explorer (Windows) or Finder (Mac), or whatever the equivalent is in your favourite Linux.

Now, inside this folder you just made, make another folder called `Analyses` (i.e. '`MyFirstAnalysis/analyses`'). And then another: `Datasets`. At this point, you should move the datasets for the book to the Datasets folder you just made.

If you're working on a full project, you might perhaps make another folder called `Manuscript` and perhaps another one called `Important PDFs`. We think you might be getting the point: use folders to organize separate parts of your project. Your file of instructions for R—your *script* file—will go in the `Analyses` folder. We'll build one soon. Be patient. ☺

2.2.1 IMPORTING DATA, PREPARATION

For R to import data (i.e. to get your data into R's brain), you need to tell R the location of your data. The location, or address, for information on a computer is known as the *path*, and these are often rather long, difficult to remember, and even harder to type accurately. And if you make one little mistake while typing your path, R will be lost. So never type your path. Let RStudio do the hard work.

First, open Rstudio and open a new script file. Put the preliminary information (remember this from the previous chapter?) at the start of your

Downloaded from https://academic.oup.com/book/27290/chapter/196933265 by University of Glasgow user on 25 September 2024

script— some annotation, the libraries, and the brain clearer. Go ahead and save this. Perhaps call it `DataImportExample.R`?

We don't expect you to use your own data for this. In the next few sections and also the next chapter, we will work with the `compensation.csv` dataset.

There are at least four really easy ways to get R to import the data, without having to do anything tricky like type the path to your data file.

2.2.2 METHOD 1: THE IMPORT DATASET TOOL

Rstudio provides menu-based import functionality via the *`Import Dataset`* tab in the upper right pane of RStudio. This will allow you to navigate to your data file, select it, and click the *Open* button. When you attempt to use this for the first time, RStudio may ask you to install or update some packages; just say yes! These are packages that make data import faster and simpler.

Given our focus on .csv files, select *From CSV* as your option. There are clearly others. The dialogue box will open revealing a very handy tool (Figure 2.5).

First, click the *Browse* button and navigate to your data file in the window that pops up. Click *Open* and you will see how R is interpreting your data (it is not yet into R, you're just getting a preview). At this point, you could fiddle with some of the import options in the lower left part of the window (e.g., the type of delimiter / separator). Often you don't have to change anything, and can just click the *Import* button.

Before you do anything more, copy the first two lines in the code preview box (example shown in Fig 2.6). Then, click *Import*. RStudio will run all of the code in the preview box. This imports the data using the **read_csv**() function from the readr library. It also opens a window showing the data. Close that window - you don't need it open.

Now, paste the two lines of code into your script. Having done this, you don't have to use the *`Import Dataset`* tool again. You just run these lines of code and your data are read in (assuming you don't move the data . . .). Your script now contains the information about which dataset is being used and where it is on your computer. You run this code each time you work with the script.

Downloaded from https://academic.oup.com/book/27290/chapter/196933265 by University of Glasgow user on 25 September 2024

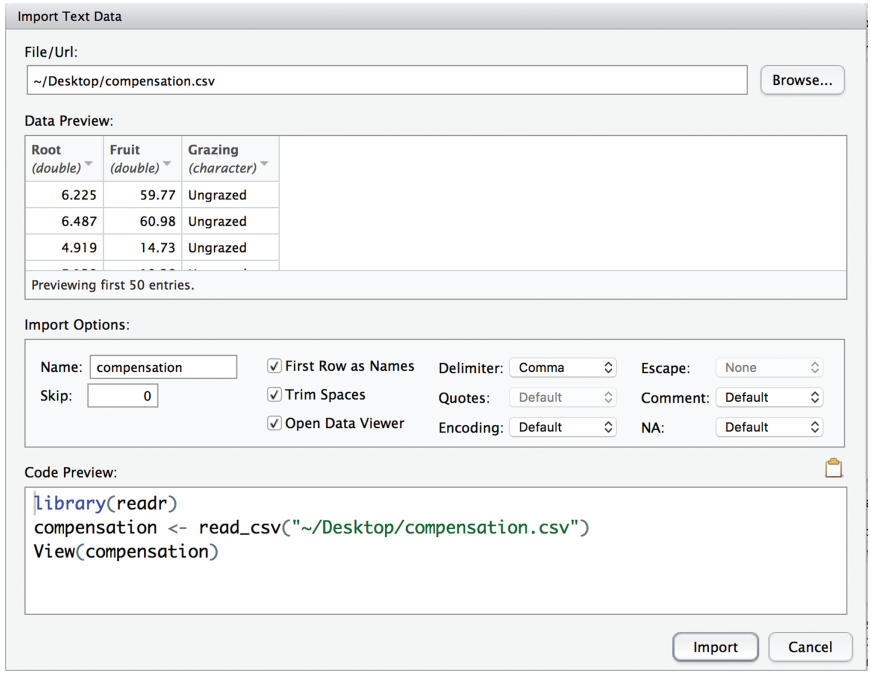

Figure 2.5 The *Import Dataset* tab produces this dialogue box, with details about the format, delimiter (e.g. comma), and values being used for NA (the na.string).

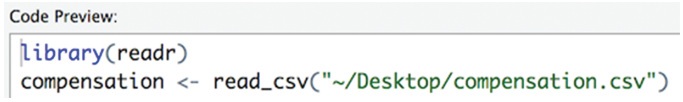

Figure 2.6 An example of the two lines of instructions produced by the *Import Dataset* tool in Rstudio.

Top Tip. Do not rely on using the `Import Dataset` tool each time you work on an analysis. This would be inefficient but, more importantly, not pasting in the R code snippet leaves you with an incomplete script, and if you keep reading your data in via `Import Dataset` you may end up working with the wrong file.

2.2.3 METHOD 2: THE file.choose() FUNCTION

If you don't use Rstudio, or even if you do, you can use the **file.choose**() function to get your path and filename into R without typing it. In the

Downloaded from https://academic.oup.com/book/27290/chapter/196933265 by University of Glasgow user on 25 September 2024

```
> file.choose()
[1] "/Users/owenpetchey/work/0 research/5.published/Dilpetus/Dileptus data/dileptus
expt data.csv"
```

Figure 2.7 An example of the instruction produced when you use the **file.choose**() function to find a data file.

Console, type **file.choose**() and press enter. A dialogue box will open, implicitly inviting you to navigate to your data file, to select it, and to press open. In the Console, you will then see the path and filename. Figure 2.7 gives an example of the Console after you've done this.

Select the double quotes and everything inside them (and not the [1]), copy this, and paste it into your script. There, you have your path and filename. Easy, eh! Now you need to give this information to the function that reads in the data file . . . this function is called **read.csv**() because, no surprises, it reads a `csv` file into R. So put the path and filename inside the brackets of **read.csv**() and use the assignment arrow (<-) to save the imported data into an object with a name of your choice. In the example below, this object is called compensation. Remember that this instruction should now be in your script file, and not typed directly into the Console. As above, for this script/project, and via this method, you don't have to use the **file.choose**() function again.

2.2.4 METHOD 3: DANCING WITH THE IMPORT DEVIL

There is another solution. It DOES require that you know where you are working within the hierarchy of your computer. In RStudio, you can set the working directory for a session of coding: `Session -> Set Working Directory -> Choose Directory`. This will point RStudio's eyes *directly* at a folder in a specific location, the folder where the data are.

Once RStudio is looking there, the use of **read.csv**() becomes 'path free'; you need only supply the full name of the dataset, in quotation marks, for example compensation <- **read.csv**("compensation.csv"). The advantage of doing things this way is that if you move your work, or collaborate with someone who keeps the project in a different location on their machine,

Downloaded from https://academic.oup.com/book/27290/chapter/196933265 by University of Glasgow user on 25 September 2024

you don't have to update your file paths. Just set the working directory and away you go.

As we note above, this can be dicey for some of you (and some of us). But if you organized your files in folders that are sensibly named and located, there will be no trouble. In fact, after a few uses of `Import Dataset` via RStudio, you will know more about the path than ever before.

2.2.5 METHOD 4: PUT YOUR DATA IN THE SAME PLACE AS YOUR SCRIPT

If you put your data and script in the same folder (which may or may not be such a good idea) and you double-click on the script, and this causes RStudio to start (because it wasn't already running), then the default place RStudio will look for data is in the folder with the script. In this case, you don't need the path at all. It's a bit risky to rely on this, since it only works if RStudio was *not* already running. Also, you often don't want your data in the same folder as your script. The solution is then to use relative paths, but that is beyond what we want to go into here. After you get comfortable with important data and paths, you can maybe take a moment to Google for 'using relative paths R' if you're interested.

2.3 Checking that your data are your data

A very sensible next step is to make sure the data you just asked R to commit to its brain are actually the data you wanted. Never trust anyone, let alone yourself, or R. Some basic things to check are:

- The correct number of rows are in R.
- The correct names and number of variables are in R.
- The variable are of the correct type (e.g. R recognizes numeric variables to be numeric).
- Variables describing types of things (e.g. gender) have the correct number of categories (levels).

Downloaded from https://academic.oup.com/book/27290/chapter/196933265 by University of Glasgow user on 25 September 2024

If you have imported the compensation data, you should have an object called compensation. A few very sensible functions to commit to your memory are presented below, and we cover what they produce. You should feel free to add these to your script:

```
names(compensation)

## [1] "Root"    "Fruit"   "Grazing"

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

dim(compensation)

## [1] 40  3

str(compensation)

## 'data.frame':    40 obs. of  3 variables:
##  $ Root   : num  6.22 6.49 4.92 5.13 5.42 ...
##  $ Fruit  : num  59.8 61 14.7 19.3 34.2 ...
##  $ Grazing: Factor w/ 2 levels "Grazed",
               "Ungrazed": 2 2 2 2 2 2 2 2 2 2 ...
```

Not surprisingly, **names**() tells you the names you assigned each column (i.e. your variable names, the column names in your `.csv` files, which you typed into Excel or wherever). **head**() returns the first six rows of the dataset (guess what **tail**() does?). **dim**() tells you the numbers of rows and columns—the *dimension*—of the dataset. Finally, **str**() returns the structure of the dataset, combining nearly all of the previous functions into one handy function.

The function **str**() returns several pieces of information about objects. For a *data frame* (spreadsheet-like dataset), it returns in its first line a statement saying that you have a data frame type of object, and the number of observations (rows) and variables (columns). Then there is a line for each of your variables, giving the variable name, variable type (numeric, factors,

Downloaded from https://academic.oup.com/book/27290/chapter/196933265 by University of Glasgow user on 25 September 2024

or integers, for example), and the first few values of that variable. Thus, **str**() it is an outstanding way to ensure that the data you have imported are what you intended to import, and to remind yourself about features of your data.

Of course, you could also just type the name of the dataset—`compensation`. This can be pretty fun/infuriating if you forget that there are 10,000+ rows in your dataset.

2.3.1 YOUR FIRST INTERACTION WITH *dplyr*

In the previous chapter, we had you install the package ***dplyr***. If you have been working along, and you've run the script examples from Chapter 1, ***dplyr*** should be in and ready to use. Two functions from ***dplyr*** are also useful for looking at the data you've just imported. One, **glimpse**(), provides a horizontal view of the data. The second, **tbl_df**(), provides a vertical view. Both show the column names and the type of data.

We will emphasize this over and over again as we move on in the book: the functions in ***dplyr*** and ***ggplot2***, and in fact all packages from Hadley Wickham, all have the SAME first argument—the data frame. We'll introduce more ***dplyr*** functions in Chapter 3, all of which take the data frame as their first argument.

Let's see what **glimpse**() and **tbl_df**() do:

```
# dplyr viewing of data
library(dplyr)

#glimpse and tbl_df
glimpse(compensation)

## Observations: 40
## Variables: 3
## $ Root    (dbl) 6.225, 6.487, 4.919, 5.130, 5.417, 5.35...
## $ Fruit   (dbl) 59.77, 60.98, 14.73, 19.28, 34.25, 35.5...
## $ Grazing (fctr) Ungrazed, Ungrazed, Ungrazed, Ungrazed...

tbl_df(compensation)

## Source: local data frame [40 x 3]
##
##     Root Fruit  Grazing
```

Downloaded from https://academic.oup.com/book/27290/chapter/196933265 by University of Glasgow user on 25 September 2024

```
##    (dbl) (dbl)   (fctr)
## 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
## 7  7.614 87.73 Ungrazed
## 8  6.352 63.21 Ungrazed
## 9  4.975 24.25 Ungrazed
## 10 6.930 64.34 Ungrazed
## ..   ...   ...      ...
```

Nice. We get two simple summaries of the data inside `compensation`. We are going to use **glimpse**() from now on (mostly).

2.4 Basic troubleshooting while importing data

At this point you know how to create a `.csv` file of your data, import these data, and make sure the data in R are the data you wanted, using several functions. Of course, along the way, bad things will happen. The magic will fail. Something just won't work. Here are a few that we find happening to new R people all the time.

Problem. *My imported data has more columns and/or rows than it should! (and there are many, many NA's!)* This likely to be caused by Excel, which has saved some extra columns and/or rows (goodness knows why!). To see if this is the problem, open your data file (`.csv` file) using Notepad (Windows) or TextEdit (Mac). If you see any extra commas at the end of rows, or lines of commas at the bottom of your data, this is the problem. To fix this, open your `.csv` file in Excel, select only the data you want, copy this, and paste this into a new Excel file. Save this new file as before (and delete the old one). It should not have the extra rows/columns.

Problem. *There's only one column in the imported data, but definitely more than one in my data file!* This is probably caused by your file not being 'comma separated', but R expecting it to be. Most often this happens when

Downloaded from https://academic.oup.com/book/27290/chapter/196933265 by University of Glasgow user on 25 September 2024

Excel decides it wants to save a `.csv` file with semicolon (;) separation, instead of comma! There are several options. On the Excel side, try and ensure that Excel is using commas for `.csv` files (though this is sometimes easier said than done). On the R side, try using the *Import Dataset* facility in Rstudio; during the process, you can see the raw data and the imported data and look at the raw data to see what the separator is, and then select this in the dialogue box (see Figure 2.5).

Problem. *R gives this error:*

```
## Warning in file(file, "rt"): cannot open file 'blah.csv': No
such file or directory
## Error in file(file, "rt"): cannot open the connection
```

Read the first line of that error message: 'No such file or directory'. Did you type, rather than copy and paste, the path or filename? If so, you probably made a mistake. Or perhaps you moved or renamed the file? This is a very common error associated with trying to find your data. Use the *Import Dataset* tool in Rstudio, or the **file.choose**() function again, and you will fix this error.

Problem: *My dataset contains a column of dates, but I think R is not recognizing or treating them as dates!* You are justifiably concerned, because R won't, by default, read in your dates as real dates. Look at the example in the appendix about tidying data in this chapter, for how to get R to recognize a variable as containing dates.

2.5 Summing up

You just learned how to prepare your data for R, how to import your data, how to check they are properly in R, and quite a bit about what to do if things go wrong. These tasks are the basic first steps in any process of quantitative problem solving, and they are the foundation. They are not optional steps. Forget these, get these wrong, and everything else will be wrong also. So focus on having rock-solid data in R.

Downloaded from https://academic.oup.com/book/27290/chapter/196933265 by University of Glasgow user on 25 September 2024

Appendix Advanced activity: dealing with untidy data

Health warning: this section is not necessary if you're working through the book for the first time. It's also a bit complex, so think about skipping it, saving it for when you're happy to take a bit of a challenge (great if that is now, however . . . go for it!).

This section uses several new packages that you will need to install first (e.g. download from CRAN). It uses many functions we have not introduced, and we don't introduce many in depth. If you prepare your data well, this section is unnecessary. But you will encounter messy data. So, as we said, feel free to come back to it.

WHAT ARE UNTIDY DATA?

There are lots of ways data can be untidy, and it's not appropriate (or possible) to deal with them all here. So we show a somewhat typical example in which observations of the same thing (here bacterial density) have been recorded on multiple dates, and the observation from each date is placed in a separate column. Rows of the data correspond to experimental units (defined here by the protist species consuming the bacteria, and the environmental temperature) (another variable, Bottle, is a unique identifier for the experimental unit). Let's import these data, check them, and then tidy them:

```
nasty.format <- read.csv("nasty format.csv")
```

```
str(nasty.format)

## 'data.frame':    37 obs. of  11 variables:
##  $ Species : Factor w/ 5 levels "",
    "Colpidium",..: 3 3 3 3 3 3 3 3 3 5 ...
##  $ Bottle  : Factor w/ 37 levels "","10-C.s","11-
    C.s",..: 35 36 37 11 12 13 23 24 25 5 ...
##  $ Temp    : int  22 22 22 20 20 20 15 15 15 22 ...
##  $ X1.2.13 : num  100 62.5 75 75 50 87.5 75 50 75 37.5 ...
##  $ X2.2.13 : num  58.8 71.3 72.5 73.8 NA NA NA
    NA NA 52.5 ...
##  $ X3.2.13 : num  67.5 67.5 62.3 76.3 81.3 62.5 90
```

Downloaded from https://academic.oup.com/book/27290/chapter/196933265 by University of Glasgow user on 25 September 2024

```
      78.8 78.3 23.8 ...
##  $ X4.2.13 : num  6.8 7.9 7.9 31.3 32.5 28.8 72.5
      92.5 77.5 1.25 ...
##  $ X6.2.13 : num  0.93 0.9 0.88 3.12 3.75 ...
##  $ X8.2.13 : num  0.39 0.36 0.25 1.01 1.06 1 67.5
      72.5 60 0.96 ...
##  $ X10.2.13: num  0.19 0.16 0.23 0.56 0.49 0.41
      37.5 52.5 60 0.33 ...
##  $ X12.2.13: num  0.46 0.34 0.31 0.5 0.38 ...
```

First things first. The data frame in R has 37 observations and 11 variables. The experiment involved only 36 experimental units, however. Looking at the data in R, by clicking on the `nasty.format` text in the Rstudio Environment pane, and scrolling down in the displayed data, shows there is a 37th row containing no data. This is Excel trying to trip us up (actually, it's usually caused by something odd *we* did in Excel). Let's remove that row. We do this by looking at the row and seeing what is unique about it, compared with all the others. Importantly, it lacks an entry in the Bottle variable. So we tell R to keep only rows that have an entry in the Bottle variable, using the **filter**() function in the ***dplyr*** package (more about this add-on package, and **filter**(), in the next chapter):

```
library(dplyr)
nasty.format <- filter(nasty.format, Bottle != "")
glimpse(nasty.format)

## Observations: 36
## Variables: 11
## $ Species  (fctr) P.caudatum, P.caudatum, P.caudatum, P...
## $ Bottle   (fctr) 7-P.c, 8-P.c, 9-P.c, 22-P.c, 23-P.c, ...
## $ Temp     (int) 22, 22, 22, 20, 20, 20, 15, 15, 15, 22...
## $ X1.2.13  (dbl) 100.0, 62.5, 75.0, 75.0, 50.0, 87.5, 7...
## $ X2.2.13  (dbl) 58.8, 71.3, 72.5, 73.8, NA, NA, NA, NA...
## $ X3.2.13  (dbl) 67.5, 67.5, 62.3, 76.3, 81.3, 62.5, 90...
## $ X4.2.13  (dbl) 6.80, 7.90, 7.90, 31.30, 32.50, 28.80,...
## $ X6.2.13  (dbl) 0.93, 0.90, 0.88, 3.12, 3.75, 3.12, 10...
## $ X8.2.13  (dbl) 0.39, 0.36, 0.25, 1.01, 1.06, 1.00, 67...
## $ X10.2.13 (dbl) 0.19, 0.16, 0.23, 0.56, 0.49, 0.41, 37...
## $ X12.2.13 (dbl) 0.46, 0.34, 0.31, 0.50, 0.38, 0.46, 41...
```

Downloaded from https://academic.oup.com/book/27290/chapter/196933265 by University of Glasgow user on 25 September 2024

TIDYING WITH GATHER()

Now we make the data tidy. We need a new variable that contains the date on which observations were made, and a new variable containing the observations of bacterial abundance that are currently stored in columns 4 to 11. And then we need to move the data into these new columns. This is all very straightforward, thanks to the **gather**() function in the ***tidyr*** package. Use it like this:

```
library(tidyr)
tidy_data <- gather(nasty.format, Date, Abundance, 4:11)
```

This first argument of **gather** is the data frame to work on: nasty.format. The second is the name of the new variable that will contain the dates (we call it Date). The third is the name of the new variable that will contain the bacterial abundances (we call it Abundance). The fourth argument is the location in the nasty.format data frame of the observations of bacterial abundance that are to be put into the new Abundance variable. Use **str**() to see if it worked:

```
glimpse(tidy_data)

## Observations: 288
## Variables: 5
## $ Species   (fctr) P.caudatum, P.caudatum, P.caudatum, ...
## $ Bottle    (fctr) 7-P.c, 8-P.c, 9-P.c, 22-P.c,
   23-P.c,...
## $ Temp      (int) 22, 22, 22, 20, 20, 20, 15, 15, 15, 2...
## $ Date      (chr) "X1.2.13", "X1.2.13", "X1.2.13", "X1....
## $ Abundance (dbl) 100.0, 62.5, 75.0, 75.0, 50.0, 87.5, ...
```

Yes, super! We have 288 observations (36 experimental units (bottles), each observed on eight dates). We have the new Date variable, a `factor`-type variable with eight levels, and the new Abundance variable, which is numeric. (Give yourself a pat on the back if you wonder why there are still 37 levels in the Bottle variable; though it isn't too much of a problem to correct at present.)

CLEANING THE DATES

The data are now officially *tidy*. The Date variable still needs a bit of work, however. We need to remove the X at the beginning, and make R recognize

Downloaded from https://academic.oup.com/book/27290/chapter/196933265 by University of Glasgow user on 25 September 2024

that these are dates (i.e. change the variable type from Factor to a date-type variable.). First we remove the X, using the **sub_str**() function in the stringr add-on package:

```
library(stringr)
tidy_data <- mutate(tidy_data, Date = substr(Date, 2, 20))
```

We tell the **sub_str**() function the variable to work on, and the character in that variable at which to start keeping characters (i.e. keep from the second character on). The third variable we have made 20, which is past the end of the date, so we don't discard any later characters. We do all this within the **mutate**() function of the ***dplyr*** add-on package, which provides a neat way of changing (or adding) variables in a data frame (much more about this in the next chapter).

Now we need to change the variable into one that R recognizes as containing dates. Technically, we are going to *parse* the information in the date variable so that R regards it as a date. This will allow us to use the Date variable as a continuous variable, for example to make a graph with the date on the *x*-axis, or even to calculate the number of days, months, or years between two or more dates.

We will use a function in the ***lubridate*** package. This package contains the functions **ymd**(), **ydm**(), **dym**(), **dmy**(), **myd**(), and **mdy**(), among others. Which of these functions should we use for our data? Take a look at the date values:

```
unique(tidy_data$Date)

## [1] "1.2.13"  "2.2.13"  "3.2.13"  "4.2.13"  "6.2.13"
## [6] "8.2.13"  "10.2.13" "12.2.13"
```

(Note that we've used a bit of classic R here, the dollar sign, to refer to a variable inside a data frame. Sometimes the classic way is still very useful!)

Not too much sleuthing shows that our date has the format day.month.year. So we use the function **dmy**(). This is a quite intelligent function, able to deal with different separators (ours is a dot but others work), and dates that include leading zeros (ours does):

Downloaded from https://academic.oup.com/book/27290/chapter/196933265 by University of Glasgow user on 25 September 2024

```
library(lubridate)
tidy_data <- mutate(tidy_data, Date = dmy(Date))
```

And look at the type of variable the dates are now:

```
glimpse(tidy_data)

## Observations: 288
## Variables: 5
## $ Species   (fctr) P.caudatum, P.caudatum, P.caudatum, ...
## $ Bottle    (fctr) 7-P.c, 8-P.c, 9-P.c, 22-P.c,
   23-P.c,...
## $ Temp      (int) 22, 22, 22, 20, 20, 20, 15, 15, 15, 2...
## $ Date      (date) 2013-02-01, 2013-02-01, 2013-
   02-01, ...
## $ Abundance (dbl) 100.0, 62.5, 75.0, 75.0, 50.0, 87.5, ...
```

It is a `POSIXct`-type variable if you use **str**() or it is a `date` variable if you use **glimpse**(). Look `POSIXct` up in Google if you care to know exactly what it means. It's enough, however, to know that this means that R knows this variable contains dates.

NOW SEE WHAT WE CAN DO! . . .

We just tidied our data, and made the dates into dates. Maybe that seemed a lot of effort, so we'll take a moment to demonstrate one example of what this allows us to do. Imagine we want to view the dynamics of bacterial abundance in each bottle. We can now do this with very little code:

```
library(ggplot2)
ggplot(data = tidy_data, aes(x=Date, y=Abundance)) +
  geom_point() +
  facet_wrap(~Bottle)
```

First, if you're not familiar with making graphs using **ggplot**(), go to Chapter 4, where we explain everything. We tell **ggplot**() the data frame to look in for variables, the *x*- and *y*-variables, to plot points, and then ask for on graph (i.e. facet) for each of the bottles. The resulting panel of graphs (Figure 2.8) has correct dates on the *x*-axis, and is really useful for data exploration, and even publication after some polishing (e.g. making the *x*-axis tick labels legible). A panel of graphs like this would have been

Downloaded from https://academic.oup.com/book/27290/chapter/196933265 by University of Glasgow user on 25 September 2024

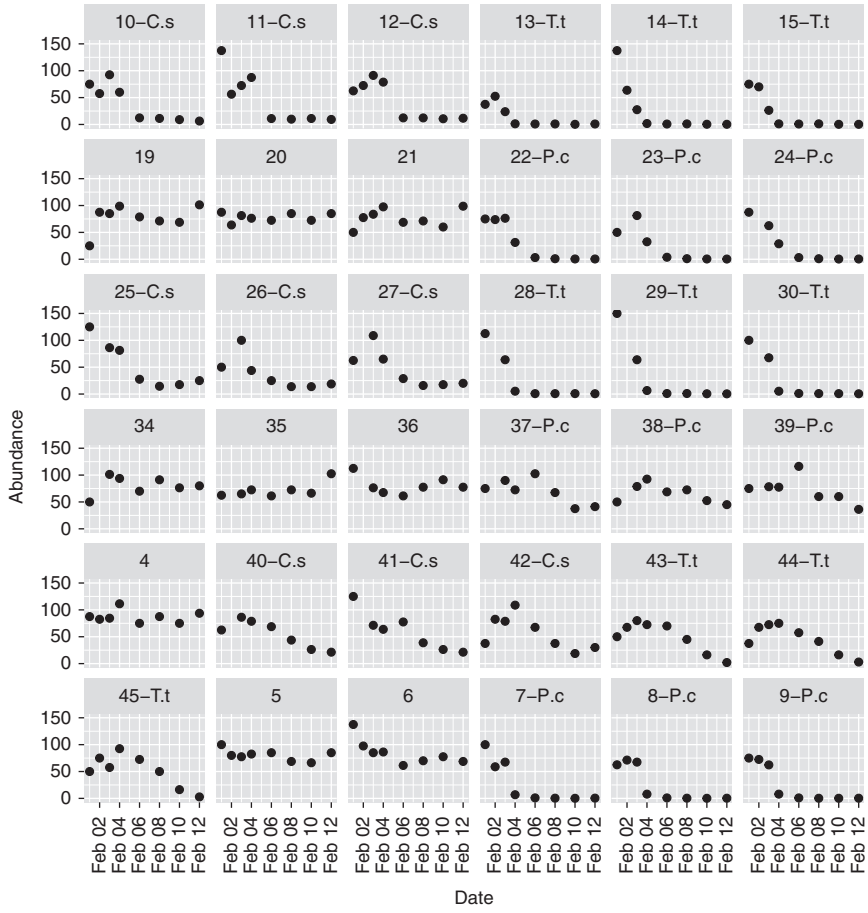

Figure 2.8 Figures like this are a doddle if one has tidy data (just three short lines of code was used to make the graph).

considerably more hassle to make if the data were not tidy, i.e. if the data were left as they were in the data file, with one variable for each date.

Why not rearrange the data in Excel (or any spreadsheet program)?

1. We have better things to do with our time, like poking ourselves in the eye with a red-hot skewer.
2. We will make mistakes, eventually, and probably won't know.
3. We'll have to do it all over again if we change the dataset.

Downloaded from https://academic.oup.com/book/27290/chapter/196933265 by University of Glasgow user on 25 September 2024

As we mentioned, there are many ways in which data can be untidy and messy. Fortunately, there are lots of nice functions for dealing with untidy and messy data. In the ***base***, ***tidyr***, and ***dplyr*** packages there are:

- **spread**(): does the opposite of **gather**. Useful for preparing data for multivariate methods.
- **separate**(): separates information present in one column into multiple new columns.
- **unite**(): puts information from several columns into one column.
- **rename**(): renames the columns of your data.
- **rbind**(): puts two datasets with exactly the same columns together (i.e. it binds rows together).
- **cbind**(): puts two datasets with exactly the same rows together (i.e. it binds columns together). (often better to use the next function . . .)
- **join**(): a suite of functions, such as **full_join**(), which allows joining together two datasets with one or more columns in common. (Same as the **merge**() function in the ***base*** package.)

You will probably find that tidying data, especially data that you didn't prepare, can take a lot of time and effort. However, it's investment well made, as having tidy and clean data makes everything following ten times easier.

We've used functions from several add-on packages: ***tidyr***, ***stringr***, ***lubridate***, ***dplyr***, and ***ggplot2***. It would be well worth your while to review these, and make notes about which functions we used from each of these packages, why we used them, and the arguments we used.

Downloaded from https://academic.oup.com/book/27290/chapter/196933265 by University of Glasgow user on 25 September 2024