Chapter 9
Data Analysis

9.1 Introduction

So far, every example in this book has started with a nice dataset that's easy to plot. That's great for learning (because you don't want to struggle with data handling while you're learning visualisation), but in real life, datasets hardly ever come in exactly the right structure. To use ggplot2 in practice, you'll need to learn some data wrangling skills. Indeed, in my experience, visualisation is often the easiest part of the data analysis process: once you have the right data, in the right format, aggregated in the right way, the right visualisation is often obvious.

The goal of this part of the book is to show you how to integrate ggplot2 with other tools needed for a complete data analysis:

- In this chapter, you'll learn the principles of tidy data (Wickham, 2014), which help you organise your data in a way that makes it easy to visualise with ggplot2, manipulate with dplyr and model with the many modelling packages. The principles of tidy data are supported by the **tidyr** package, which helps you tidy messy datasets.
- Most visualisations require some data transformation whether it's creating a new variable from existing variables, or performing simple aggregations so you can see the forest for the trees. Chapter 10 will show you how to do this with the **dplyr** package.
- If you're using R, you're almost certainly using it for its fantastic modelling capabilities. While there's an R package for almost every type of model that you can think of, the results of these models can be hard to visualise. In Chap. 11, you'll learn about the **broom** package, by David Robinson, to convert models into tidy datasets so you can easily visualise them with ggplot2.

H. Wickham, *ggplot2*, Use R!, DOI 10.1007/978-3-319-24277-4_9

Tidy data is the foundation for data manipulation and visualising models. In the following sections, you'll learn the definition of tidy data, and the tools you need to make messy data tidy. The chapter concludes with two case studies that show how to apply the tools in sequence to work with real(istic) data.

9.2 Tidy Data

The principle behind tidy data is simple: storing your data in a consistent way makes it easier to work with it. Tidy data is a mapping between the statistical structure of a data frame (variables and observations) and the physical structure (columns and rows). Tidy data follows two main principles:

1. Variables go in columns.
2. Observations go in rows.

Tidy data is particularly important for ggplot2 because the job of ggplot2 is to map variables to visual properties: if your data isn't tidy, you'll have a hard time visualising it.

Sometimes you'll find a dataset that you have no idea how to plot. That's normally because it's not tidy. For example, take this data frame that contains monthly employment data for the United States:

```
ec2
#> Source: local data frame [12 x 11]
#>
#>    month  2006  2007  2008  2009  2010  2011  2012  2013  2014
#>    (dbl) (dbl) (dbl) (dbl) (dbl) (dbl) (dbl) (dbl) (dbl) (dbl)
#> 1      1   8.6   8.3   9.0  10.7  20.0  21.6  21.0  16.2  15.9
#> 2      2   9.1   8.5   8.7  11.7  19.9  21.1  19.8  17.5  16.2
#> 3      3   8.7   9.1   8.7  12.3  20.4  21.5  19.2  17.7  15.9
#> 4      4   8.4   8.6   9.4  13.1  22.1  20.9  19.1  17.1  15.6
#> 5      5   8.5   8.2   7.9  14.2  22.3  21.6  19.9  17.0  14.5
#> 6      6   7.3   7.7   9.0  17.2  25.2  22.3  20.1  16.6  13.2
#> ..   ...   ...   ...   ...   ...   ...   ...   ...   ...   ...
#> Variables not shown: 2015 (dbl)
```

(If it looks familiar, it's because it's derived from the `economics` dataset that we used earlier in the book.)

Imagine you want to plot a time series showing how unemployment has changed over the last 10 years. Can you picture the ggplot2 command you'd need to do it? What if you wanted to focus on the seasonal component of

unemployment by putting months on the x-axis and drawing one line for each year? It's difficult to see how to create those plots because the data is not tidy. There are three variables, month, year and unemployment rate, but each variable is stored in a different way:

- `month` is stored in a column.
- `year` is spread across the column names.
- `rate` is the value of each cell.

To make it possible to plot this data we first need to tidy it. There are two important pairs of tools:

- Spread & gather.
- Separate & unite.

9.3 Spread and Gather

Take a look at the two tables below:

x	y	z
a	A	1
b	D	5
c	A	4
c	B	10
d	C	9

x	A	B	C	D
a	1	NA	NA	NA
b	NA	NA	NA	5
c	4	10	NA	NA
d	NA	NA	9	NA

If you study them for a little while, you'll notice that they contain the same data in different forms. I call the first form **indexed** data, because you look up a value using an index (the values of the `x` and `y` variables). I call the second form **Cartesian** data, because you find a value by looking at intersection of a row and a column. We can't tell if these datasets are tidy or not. Either form could be tidy depending on what the values "A", "B", "C", "D" mean.

(Also note the missing values: missing values that are explicit in one form may be implicit in the other. An `NA` is the presence of an absence; but sometimes a missing value is the absence of a presence.)

Tidying your data will often require translating Cartesian indexed forms, called **gathering**, and less commonly, indexed Cartesian, called **spreading**. The tidyr package provides the `spread()` and `gather()` functions to perform these operations, as described below.

(You can imagine generalising these ideas to higher dimensions.) However, data is almost always stored in 2d (rows & columns), so these generalisations are fun to think about, but not that practical. I explore the idea more in Wickham (2007).

9.3.1 Gather

`gather()` has four main arguments:

- `data`: the dataset to translate.
- `key` & `value`: the key is the name of the variable that will be created from the column names, and the value is the name of the variable that will be created from the cell values.
- `...`: which variables to gather. You can specify individually, `A, B, C, D`, or as a range `A:D`. Alternatively, you can specify which columns are *not* to be gathered with `-`: `-E, -F`.

To tidy the economics dataset shown above, you first need to identify the variables: `year`, `month` and `rate`. `month` is already in a column, but `year` and `rate` are in Cartesian form, and we want them in indexed form, so we need to use `gather()`. In this example, the key is `year`, the value is `unemp` and we want to select columns from `2006` to `2015`:

```
gather(ec2, key = year, value = unemp, `2006`:`2015`)
#> Source: local data frame [120 x 3]
#>
#>    month  year unemp
#>    (dbl) (chr) (dbl)
#> 1      1  2006   8.6
#> 2      2  2006   9.1
#> 3      3  2006   8.7
#> 4      4  2006   8.4
#> 5      5  2006   8.5
#> 6      6  2006   7.3
#> ..   ...   ...   ...
```

Note that the columns have names that are not standard variable names in R (they don't start with a letter). This means that we need to surround them in backticks, i.e. `‘2006‘` to refer to them.

Alternatively, we could gather all columns except `month`:

```
gather(ec2, key = year, value = unemp, -month)
#> Source: local data frame [120 x 3]
#>
#>    month  year unemp
```

```
#>    (dbl) (chr) (dbl)
#> 1      1  2006   8.6
#> 2      2  2006   9.1
#> 3      3  2006   8.7
#> 4      4  2006   8.4
#> 5      5  2006   8.5
#> 6      6  2006   7.3
#> ..   ...   ...   ...
```

To be most useful, we can provide two extra arguments:

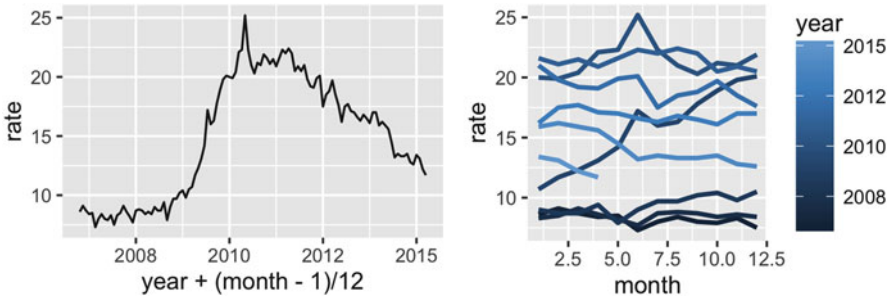
```
economics_2 <- gather(ec2, year, rate, `2006`:`2015`,
  convert = TRUE, na.rm = TRUE)
economics_2
#> Source: local data frame [112 x 3]
#>
#>    month  year  rate
#>    (dbl) (int) (dbl)
#> 1      1  2006   8.6
#> 2      2  2006   9.1
#> 3      3  2006   8.7
#> 4      4  2006   8.4
#> 5      5  2006   8.5
#> 6      6  2006   7.3
#> ..   ...   ...   ...
```

We use `convert = TRUE` to automatically convert the years from character strings to numbers, and `na.rm = TRUE` to remove the months with no data. (In some sense the data isn't actually missing because it represents dates that haven't occurred yet.)

When the data is in this form, it's easy to visualise in many different ways. For example, we can choose to emphasise either long term trend or seasonal variations:

```
ggplot(economics_2, aes(year + (month - 1) / 12, rate)) +
  geom_line()

ggplot(economics_2, aes(month, rate, group = year)) +
  geom_line(aes(colour = year), size = 1)
```



9.3.2 Spread

`spread()` is the opposite of `gather()`. You use it when you have a pair of columns that are in indexed form, instead of Cartesian form. For example, the following example dataset contains three variables (`day`, `rain` and `temp`), but `rain` and `temp` are stored in indexed form.

```
weather <- dplyr::data_frame(
  day = rep(1:3, 2),
  obs = rep(c("temp", "rain"), each = 3),
  val = c(c(23, 22, 20), c(0, 0, 5))
)
weather
#> Source: local data frame [6 x 3]
#>
#>     day   obs   val
#>   (int) (chr) (dbl)
#> 1     1  temp    23
#> 2     2  temp    22
#> 3     3  temp    20
#> 4     1  rain     0
#> 5     2  rain     0
#> 6     3  rain     5
```

Spread allows us to turn this messy indexed form into a tidy Cartesian form. It shares many of the arguments with `gather()`. You'll need to supply the `data` to translate, as well as the name of the `key` column which gives the variable names, and the `value` column which contains the cell values. Here the key is `obs` and the value is `val`:

```
spread(weather, key = obs, value = val)
#> Source: local data frame [3 x 3]
#>
#>     day  rain  temp
```

```
#>   (int) (dbl) (dbl)
#> 1     1     0    23
#> 2     2     0    22
#> 3     3     5    20
```

9.3.3 Exercises

1. How can you translate each of the initial example datasets into the other form?
2. How can you convert back and forth between the `economics` and `economics_long` datasets built into ggplot2?
3. Install the EDAWR package from https://github.com/rstudio/EDAWR. Tidy the `storms`, `population` and `tb` datasets.

9.4 Separate and Unite

Spread and gather help when the variables are in the wrong place in the dataset. Separate and unite help when multiple variables are crammed into one column, or spread across multiple columns.

For example, the following dataset stores some information about the response to a medical treatment. There are three variables (time, treatment and value), but time and treatment are jammed in one variable together:

```
trt <- dplyr::data_frame(
  var = paste0(rep(c("beg", "end"), each = 3), "_", rep(c("a", "b", "c"))),
  val = c(1, 4, 2, 10, 5, 11)
)
trt
#> Source: local data frame [6 x 2]
#>
#>     var   val
#>   (chr) (dbl)
#> 1 beg_a     1
#> 2 beg_b     4
#> 3 beg_c     2
#> 4 end_a    10
#> 5 end_b     5
#> 6 end_c    11
```

The `separate()` function makes it easy to tease apart multiple variables stored in one column. It takes four arguments:

- `data`: the data frame to modify.
- `col`: the name of the variable to split into pieces.

- `into`: a character vector giving the names of the new variables.
- `sep`: a description of how to split the variable apart. This can either be a regular expression, e.g. `_` to split by underscores, or `[^a-z]` to split by any non-letter, or an integer giving a position.

  In this case, we want to split by the `_` character:

```
separate(trt, var, c("time", "treatment"), "_")
#> Source: local data frame [6 x 3]
#>
#>    time treatment   val
#>   (chr)     (chr) (dbl)
#> 1   beg         a     1
#> 2   beg         b     4
#> 3   beg         c     2
#> 4   end         a    10
#> 5   end         b     5
#> 6   end         c    11
```

(If the variables are combined in a more complex form, have a look at `extract()`. Alternatively, you might need to create columns individually yourself using other calculations. A useful tool for this is `mutate()` which you'll learn about in the next chapter.)

`unite()` is the inverse of `separate()`—it joins together multiple columns into one column. This is less common, but it's useful to know about as the inverse of `separate()`.

9.4.1 Exercises

1. Install the EDAWR package from https://github.com/rstudio/EDAWR. Tidy the `who` dataset.
2. Work through the demos included in the tidyr package (`demo(package = "tidyr")`)

9.5 Case Studies

For most real datasets, you'll need to use more than one tidying verb. There many be multiple ways to get there, but as long as each step makes the data tidier, you'll eventually get to the tidy dataset. That said, you typically apply the functions in the same order: `gather()`, `separate()` and `spread()` (although you might not use all three).

9.5.1 Blood Pressure

The first step when tidying a new dataset is always to identify the variables. Take the following simulated medical data. There are seven variables in this dataset: name, age, start date, week, systolic & diastolic blood pressure. Can you see how they're stored?

```
# Adapted from example by Barry Rowlingson,
# http://barryrowlingson.github.io/hadleyverse/
bpd <- readr::read_table(
"name age     start  week1  week2  week3
Anne  35 2014-03-27 100/80 100/75 120/90
 Ben  41 2014-03-09 110/65 100/65 135/70
Carl  33 2014-04-02 125/80   <NA>   <NA>
", na = "<NA>")
```

The first step is to convert from Cartesian to indexed form:

```
bpd_1 <- gather(bpd, week, bp, week1:week3)
bpd_1
#> Source: local data frame [9 x 5]
#>
#>     name   age      start  week     bp
#>    (chr) (int)     (date) (chr)  (chr)
#> 1   Anne    35 2014-03-27 week1 100/80
#> 2    Ben    41 2014-03-09 week1 110/65
#> 3   Carl    33 2014-04-02 week1 125/80
#> 4   Anne    35 2014-03-27 week2 100/75
#> 5    Ben    41 2014-03-09 week2 100/65
#> 6   Carl    33 2014-04-02 week2     NA
#> ..   ...   ...        ...   ...    ...
```

This is tidier, but we have two variables combined together in the `bp` variable. This is a common way of writing down the blood pressure, but analysis is easier if we break it into two variables. That's the job of separate:

```
bpd_2 <- separate(bpd_1, bp, c("sys", "dia"), "/")
bpd_2
#> Source: local data frame [9 x 6]
#>
#>     name   age      start  week   sys   dia
#>    (chr) (int)     (date) (chr) (chr) (chr)
#> 1   Anne    35 2014-03-27 week1   100    80
#> 2    Ben    41 2014-03-09 week1   110    65
#> 3   Carl    33 2014-04-02 week1   125    80
#> 4   Anne    35 2014-03-27 week2   100    75
#> 5    Ben    41 2014-03-09 week2   100    65
```

```
#> 6   Carl    33 2014-04-02 week2    NA    NA
#> ..   ...   ...        ...   ...   ...   ...
```

This dataset is now tidy, but we could do a little more to make it easier to use. The following code uses `extract()` to pull the week number out into its own variable (using regular expressions is beyond the scope of the book, but `\\d` stands for any digit). I also use `arrange()` (which you'll learn about in the next chapter) to order the rows to keep the records for each person together.

```
bpd_3 <- extract(bpd_2, week, "week", "(\\d)", convert = TRUE)
bpd_4 <- dplyr::arrange(bpd_3, name, week)
bpd_4
#> Source: local data frame [9 x 6]
#>
#>     name   age      start  week   sys   dia
#>    (chr) (int)     (date) (int) (chr) (chr)
#> 1   Anne    35 2014-03-27     1   100    80
#> 2   Anne    35 2014-03-27     2   100    75
#> 3   Anne    35 2014-03-27     3   120    90
#> 4    Ben    41 2014-03-09     1   110    65
#> 5    Ben    41 2014-03-09     2   100    65
#> 6    Ben    41 2014-03-09     3   135    70
#> ..   ...   ...        ...   ...   ...   ...
```

You might notice that there's some repetition in this dataset: if you know the name, then you also know the age and start date. This reflects a third condition of tidyness that I don't discuss here: each data frame should contain one and only one data set. Here there are really two datasets: information about each person that doesn't change over time, and their weekly blood pressure measurements. You can learn more about this sort of messiness in the resources mentioned at the end of the chapter.

9.5.2 Test Scores

Imagine you're interested in the effect of an intervention on test scores. You've collected the following data. What are the variables?

```
# Adapted from http://stackoverflow.com/questions/29775461
scores <- dplyr::data_frame(
  person = rep(c("Greg", "Sally", "Sue"), each = 2),
  time   = rep(c("pre", "post"), 3),
  test1  = round(rnorm(6, mean = 80, sd = 4), 0),
  test2  = round(jitter(test1, 15), 0)
)
```

```
scores
#> Source: local data frame [6 x 4]
#>
#>   person  time test1 test2
#>    (chr) (chr) (dbl) (dbl)
#> 1   Greg   pre    84    83
#> 2   Greg  post    76    75
#> 3  Sally   pre    80    78
#> 4  Sally  post    78    77
#> 5    Sue   pre    83    80
#> 6    Sue  post    76    75
```

I think the variables are person, test, pre-test score and post-test score. As usual, we start by converting columns in Cartesian form (`test1` and `test2`) to indexed form (`test` and `score`):

```
scores_1 <- gather(scores, test, score, test1:test2)
scores_1
#> Source: local data frame [12 x 4]
#>
#>    person  time  test score
#>     (chr) (chr) (chr) (dbl)
#> 1    Greg   pre test1    84
#> 2    Greg  post test1    76
#> 3   Sally   pre test1    80
#> 4   Sally  post test1    78
#> 5     Sue   pre test1    83
#> 6     Sue  post test1    76
#> ..    ...   ...   ...   ...
```

Now we need to do the opposite: `pre` and `post` should be variables, not values, so we need to spread `time` and `score`:

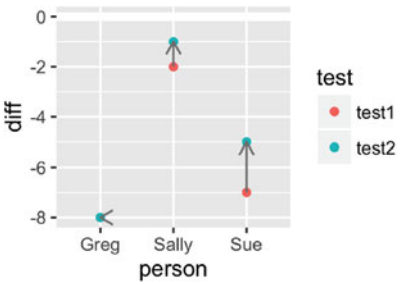
```
scores_2 <- spread(scores_1, time, score)
scores_2
#> Source: local data frame [6 x 4]
#>
#>   person  test  post   pre
#>    (chr) (chr) (dbl) (dbl)
#> 1   Greg test1    76    84
#> 2   Greg test2    75    83
#> 3  Sally test1    78    80
#> 4  Sally test2    77    78
#> 5    Sue test1    76    83
#> 6    Sue test2    75    80
```

A good indication that we have made a tidy dataset is that it's now easy to calculate the statistic of interest: the difference between pre- and post-intervention scores:

```
scores_3 <- mutate(scores_2, diff = post - pre)
scores_3
#> Source: local data frame [6 x 5]
#>
#>   person  test  post   pre  diff
#>    (chr) (chr) (dbl) (dbl) (dbl)
#> 1   Greg test1    76    84    -8
#> 2   Greg test2    75    83    -8
#> 3  Sally test1    78    80    -2
#> 4  Sally test2    77    78    -1
#> 5    Sue test1    76    83    -7
#> 6    Sue test2    75    80    -5
```

And it's similarly easy to plot:

```
ggplot(scores_3, aes(person, diff, color = test)) +
  geom_hline(size = 2, colour = "white", yintercept = 0) +
  geom_point() +
  geom_path(aes(group = person), colour = "grey50",
    arrow = arrow(length = unit(0.25, "cm")))
```



(Again, you'll learn about `mutate()` in the next chapter.)

9.6 Learning More

Data tidying is a big topic and this chapter only scratches the surface. I recommend the following references which go into considerably more depth on this topic:

- The tidyr documentation. I've described the most important arguments, but most functions have other arguments that help deal with less common situations. If you're struggling, make sure to read the documentation to see if there's an argument that might help you.
- "Tidy data (http://www.jstatsoft.org/v59/i10/)", an article in the *Journal of Statistical Software*. It describes the ideas of tidy data in more depth and shows other types of messy data. Unfortunately the paper was written before tidyr existed, so to see how to use tidyr instead of reshape2, consult the tidyr vignette (http://cran.r-project.org/web/packages/tidyr/vignettes/tidy-data.html).
- The data wrangling cheatsheet (http://rstudio.com/cheatsheets) by RStudio, includes the most common tidyr verbs in a form designed to jog your memory when you're stuck.

References

Wickham H (2007) Reshaping data with the reshape package. J Stat Soft 21(12). http://www.jstatsoft.org/v21/i12/paper

Wickham H (2014) Tidy data. J Stat Softw 59. http://www.jstatsoft.org/v59/i10/