Wrangling data with 'tidyverse'

Dan MacLean

5/1/22

Table of contents

Motivation

Working with data is pretty much the foundation of all science. Every field, lab, and scientist; every research tool, machine and data type seems to have it's own favourite way of storing, searching, analysing, and reporting data. For the individual scientist trying to analyse some work, this can be a practical nightmare.

If you've ever needed to work with any non-trivial data sets, or even tried to re-analyse an old one, you'll know that working with data is a massive ...

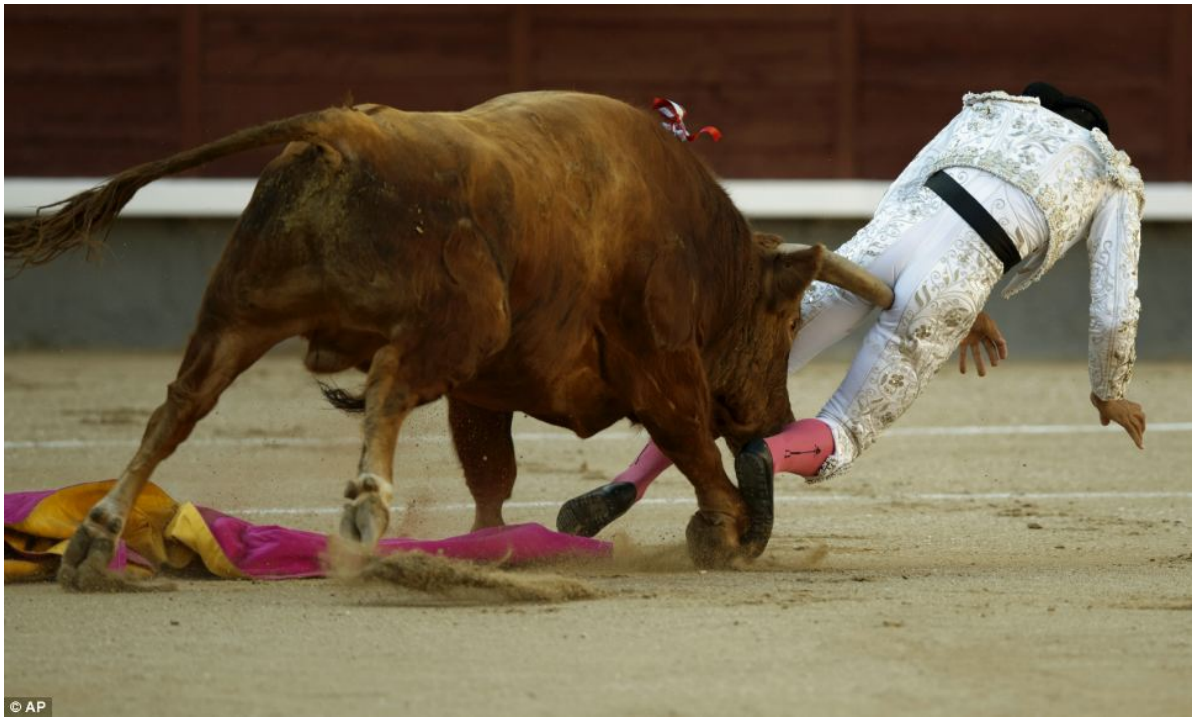


Figure 1: A significant pain in the ...

I think you know what I'm trying to say.

The aim of this book is to introduce you to some tools that can reduce the pain of data analysis. The techniques here won't get rid of all your problems - there's no magic bullet for that, but they will reduce them to a more manageable size.

The techniques themselves take a bit of getting used to - there is a learning curve. But over all they're a much smaller pain and are worth it especially if you're going to re-analyse growing datasets or do similar analysis over and over (and over) again.

Tidy data analysis

The method we'll look it is called `tidy` analysis. It's a set of ways of working that is supposed to make datasets easier to manipulate, analyse and visualise [1]. The tools for doing this in `R` are encapsulated in a set of packages collectively called the tidyverse. In reality this is the separate packages `dplyr, tidyr, readr, ggplot2, tibble, and purr`. We'll get to know *some* of them in this book.

Aim of this book

We don't aim to make you an expert in, or show you every trick in the tidyverse. In honesty, we're only going to scratch the surface of what is available in that package. But what you do learn should be enough to allow you a fighting chance to apply tidy, reproducible data analysis principles to all your data sets in the future.

Organization of this book

In the first part of this book we'll look at tools for rationally and reproducibly getting answers from table-like datasets. We'll look at the `dplyr` R package that helps you split datasets into sub-groups, compute new data based on the data you already have, filter and reduce the data to useful parts. This part will be especially useful for developing statistics to compare groups and replicates and for generating plots that compare different parts of the data. In this part we'll work with built-in data sets

In the second part of this book, we're going to look at ways of loading in data from external sources and making your own data consistent, in the hope that it will help you to organize your own data and then work on it in a reproducible, tidy fashion.

External Resources

Much of what is here is presented (sometimes in disparate fashion) in other resources. If you're looking for a different view on these techniques - this site R for data science by the inventor of the tidyverse is the best place to go. It's available as a print version too.

[1] See Hadley Wickham's tidy data paper - http://vita.had.co.nz/papers/tidy-data.html

Figure 2: A smaller pain in the …

Part I

Background

1 Tidy Data

1.1 About this chapter

1. Questions:

- What is tidy data?

2. Objectives:

- Understanding data type
- Understanding tidy data structures
- Explicitly describing and checking the data structure

3. Keypoints:

- Data needs to be in a particular format for tidy data principles to work

1.2 Tidy data

There are many ways to structure data. Here are two quite common ones.

treatment A

treatment B

John Smith

11

2

Jane Doe

16

11

Mary Johnson

3

1

John Smith

Jane Doe

Mary Johnson

treatment A

11

16

3

treatment B

2

11

1

source: Hadley Wickham

Tables contain two things, variables and values for those variables. In these two tables there are only three variables. `treatment` is one, with the values `a` and `b` . The second is 'name', with three values hidden in plain sight, and the third is `result` which is the value of the thing actually measured for each person and treatment.

For human reading purposes, we don't need to state the variables explicitly, we can see them by interpolating between the columns and rows and adding a bit of common sense. This mixing up of variables and values across tables like this has led some to call these tables 'messy'. A computer finds it hard to make sense of a messy table.

Working with R is made much less difficult if we get the data into a 'tidy' format. This format is distinct because each variable has its own column explicitly, like this

name

treatment

result

John Smith

a

11

Jane Doe

a

16

Mary Johnson

a

3

John Smith

b

2

Jane Doe

b

11

Mary Johnson

b

1

Now each variable has a column, and each seperate observation of the data has its own row. It is *much* more verbose for a human, but R can use this easily because we are now explicit about what is called what and how it relates to everything else.

More generally put, a tidy data set should look like this, schematically.

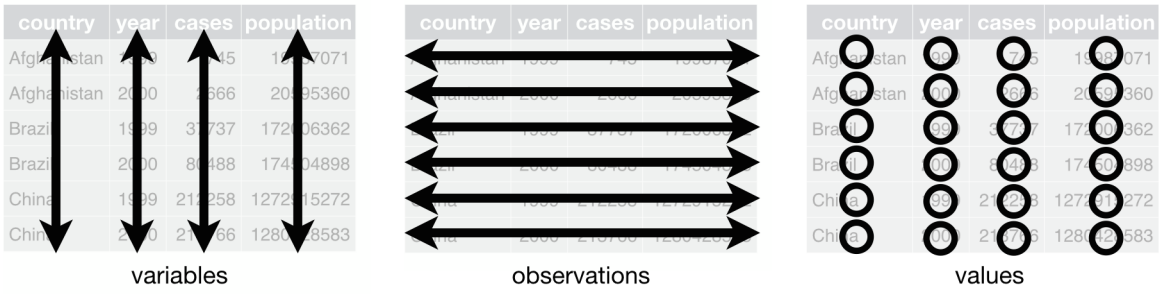


Figure 1.1: from Garret Grolemund - http://garrettgman.github.io/tidying/

1. Each variable is in its own column
2. Each observation is in its own row
3. The value of a variable in an observation is in a single cell.

1.3 A sample tidy data set

Let's use a tidy data set that comes with the tidyverse packages. The object `diamonds` is built in to `tidyr` and can be viewed by typing its name. We'll use the `head()` function to look at the top six rows only

```
library(tidyverse)
head(diamonds)
```

```
# A tibble: 6 x 10
  carat cut       color clarity depth table price     x     y     z
  <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31
4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75
6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
```

The output tells us that this is a thing called a `tibble` - this is just a table like object, more about these later. We can see the size of the tibble - 6 rows, 10 columns (this is truncated because of `head()` in reality its 53940 rows long). We can see the column headings and we can see the column type or, as this is called in R-speak, its class.

1.3.1 Class

Each of the columns has a particular type or class. Here class is either `<dbl>, <ord> or <int>`. This tells us what kind of data R is in that column. It's very important that you and R agree about what sort of data is in each column, otherwise the operations you run can go awry.

Thankfully there are only a few main classes to worry about

- `num` or `int` or `dbl` - number types
- `chr` - regular text
- `fctr` - A factor. A category or names for groups. Discrete values.
- `lgl` - `TRUE` or `FALSE` data. Can only have these two values.

Numeric, logical and character are pretty self explanatory. Factors need a bit more thinking about.

1.3.1.1 Factors

A factor is a variable that can only take pre-known values called levels. Often these will be experimental categories or groups. Usually you will know the values of the level before you even start an experiment. A treatment of a plant with different chemicals could be a factor. Its levels would be names for each treatment studied. E.G `GiberellicAcid`, `Jasmonate` or `Auxin`. Note a factor isn't restricted to describing inputs. In the same way, the sort of response of a plant to a treatment could be a factor, so `high`,`low`, `hypersensitive` could all be levels of an output factor variable in an infection assay.

A factor can have numeric-looking levels. Treatment or response can often be labelled `1`, `2`, `3` etc, but they are used as categories, not actual measurements or numbers in factors. If the values can be replaced by e.g `A`, `B`, `C` without loss of sense, then the variable is a category and should be encoded as a factor.

In our `diamonds` data set, the `cut`, `color` and `clarity` variables are factors - they just happen to be a particular sort of `ordered factor`.

Factors are what we will group and split our data sets by. We will do statistics, plots and comparisons based on numbers within factor levels.

1.3.1.2 Checking Class Explicitly

The `tibble` table-like object of our `diamonds` data does a good job of summarising type. R has some commands for this too.

`class()` will give you the class(es) of a specific variable (we can use the `$` notation to get a single column out of a table-like object such as a `tibble`)

```
class(diamonds$cut)
```

```
[1] "ordered" "factor"
```

`levels()` will tell you all the levels of a factor

```
levels(diamonds$cut)
```

```
[1] "Fair"      "Good"      "Very Good" "Premium"   "Ideal"
```

`str()` will give you a summary of whole table-like objects

```
str(diamonds)
```

```
tibble [53,940 x 10] (S3: tbl_df/tbl/data.frame)
 $ carat  : num [1:53940] 0.23 0.21 0.23 0.29 0.31 0.24 0.24 0.26 0.22 0.23 ...
 $ cut    : Ord.factor w/ 5 levels "Fair"<"Good"<..: 5 4 2 4 2 3 3 3 1 3 ...
 $ color  : Ord.factor w/ 7 levels "D"<"E"<"F"<"G"<..: 2 2 2 6 7 7 6 5 2 5 ...
 $ clarity: Ord.factor w/ 8 levels "I1"<"SI2"<"SI1"<..: 2 3 5 4 2 6 7 3 4 5 ...
 $ depth  : num [1:53940] 61.5 59.8 56.9 62.4 63.3 62.8 62.3 61.9 65.1 59.4 ...
 $ table  : num [1:53940] 55 61 65 58 58 57 57 55 61 61 ...
 $ price  : int [1:53940] 326 326 327 334 335 336 336 337 337 338 ...
 $ x      : num [1:53940] 3.95 3.89 4.05 4.2 4.34 3.94 3.95 4.07 3.87 4 ...
 $ y      : num [1:53940] 3.98 3.84 4.07 4.23 4.35 3.96 3.98 4.11 3.78 4.05 ...
 $ z      : num [1:53940] 2.43 2.31 2.31 2.63 2.75 2.48 2.47 2.53 2.49 2.39 ...
```

1.4 Quiz

1. How many levels in the factor `color` in the `diamonds` data?
2. Is the table below 'tidy'?

country	year	type	count
Afghanistan	1999	cases	745
Afghanistan	1999	population	19987071
Afghanistan	2000	cases	2666
Afghanistan	2000	population	20595360
Brazil	1999	cases	37737
Brazil	1999	population	172006362
Brazil	2000	cases	80488
Brazil	2000	population	174504898
China	1999	cases	212258
China	1999	population	1272915272
China	2000	cases	213766
China	2000	population	1280428583

3. How many variables are contained in the table - how many columns should there be for it to be tidy?

Part II

Working with tidy data in `dplyr`

2 dplyr Verbs

2.1 About this chapter

1. Questions:

- How do I manipulate tidy data?

2. Objectives:

- Understanding the pipe syntax
- Working with the 6 main functions
- Overview of helper functions

3. Keypoints:

- Tidy data can be operated on with six main functions that can quickly split, apply summaries and combine sub-groups of data

2.2 dplyr

dplyr (data plier) is a tool for manipulating datasets. As part of the tidyverse it is loaded when you use `library(tidyverse)` but can be loaded on it's own with `library(dplyr)`. dplyr is set up as a small grammar, it has five main verbs that help you form small 'sentences' to get to your result.

The verbs are:

- `select()` picks variables based on their names.
- `filter()` picks cases based on their values.
- `mutate()` adds new variables that are functions of existing variables
- `summarise()` reduces multiple values down to a single summary.
- `arrange()` changes the ordering of the rows.

A sixth function `group_by()` allows you to operate on subsets of the data.

2.3 Pipe Syntax

The first argument to all these functions is the tidy table-like object (we'll start calling these data frames from here), so for our diamonds data set, then we use

```
filter(diamonds, ... ) # ... stands in for other arguments
```

If we want to perform more than one step in series, we end up in the situation of having to save our result with a new name and working from there. This gets cumbersome quickly…

```
diamonds2 <- filter(diamonds, ...)
diamonds3 <- select(diamonds2, ...)
diamonds4 <- mutate(diamonds3, ...)
```

To avoid this, there is a pipe operator - **%>%**. The purpose of the pipe is to take the thing on its left, and use it as the first argument of the thing on its right. So we can change that mess to

```
diamonds %>% filter( ... ) %>% select( ... ) %>% mutate( ... )
```

Which is much more readable. Further we can put the right hand side of the pipe on a new line, such that we can get a very easy to read pattern.

```
diamonds %>%
  filter( ... ) %>%
  select( ... ) %>%
  mutate( ... )
```

If you want to save the result, you'll just put the usual assign at the top

```
filtered_diamonds <-
diamonds %>%
  filter( ... ) %>%
  select( ... ) %>%
  mutate( ... )
```

2.4 select()

`select()` is probably the simplest verb. It lets you select whole columns from the dataframe, discarding others. This is most useful for working with huge datasets of many columns, or for extracting bits for ease of printing or presentation.

```
  diamonds %>%
    select(carat, cut)
```

```
# A tibble: 53,940 x 2
   carat cut
   <dbl> <ord>
 1  0.23 Ideal
 2  0.21 Premium
 3  0.23 Good
 4  0.29 Premium
 5  0.31 Good
 6  0.24 Very Good
 7  0.24 Very Good
 8  0.26 Very Good
 9  0.22 Fair
10  0.23 Very Good
# ... with 53,930 more rows
```

Shorthands include the `:` which lets you choose a range and `-` which can be read as `except` so leaves out the columns you state

```
  diamonds %>%
    select(depth:price)
```

```
# A tibble: 53,940 x 3
   depth table price
   <dbl> <dbl> <int>
 1  61.5    55   326
 2  59.8    61   326
 3  56.9    65   327
 4  62.4    58   334
 5  63.3    58   335
 6  62.8    57   336
 7  62.3    57   336
 8  61.9    55   337
 9  65.1    61   337
10  59.4    61   338
# ... with 53,930 more rows
```

```
diamonds %>%
  select( -x, -y, -z)
```

```
# A tibble: 53,940 x 7
   carat cut       color clarity depth table price
   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int>
 1  0.23 Ideal     E     SI2      61.5    55   326
 2  0.21 Premium   E     SI1      59.8    61   326
 3  0.23 Good      E     VS1      56.9    65   327
 4  0.29 Premium   I     VS2      62.4    58   334
 5  0.31 Good      J     SI2      63.3    58   335
 6  0.24 Very Good J     VVS2     62.8    57   336
 7  0.24 Very Good I     VVS1     62.3    57   336
 8  0.26 Very Good H     SI1      61.9    55   337
 9  0.22 Fair      E     VS2      65.1    61   337
10  0.23 Very Good H     VS1      59.4    61   338
# ... with 53,930 more rows
```

You can select columns with helpers,

- `starts_with()`
- `ends_with()`
- `contains()`
- `num_range()`

Here are examples.

```
diamonds %>%
  select( starts_with("c"))
```

```
# A tibble: 53,940 x 4
   carat cut       color clarity
   <dbl> <ord>     <ord> <ord>
 1  0.23 Ideal     E     SI2
 2  0.21 Premium   E     SI1
 3  0.23 Good      E     VS1
 4  0.29 Premium   I     VS2
 5  0.31 Good      J     SI2
 6  0.24 Very Good J     VVS2
 7  0.24 Very Good I     VVS1
 8  0.26 Very Good H     SI1
```

```
 9  0.22 Fair      E     VS2
10  0.23 Very Good H     VS1
# ... with 53,930 more rows
```

```
diamonds %>%
  select( ends_with("e"))
```

```
# A tibble: 53,940 x 2
   table price
   <dbl> <int>
 1    55   326
 2    61   326
 3    65   327
 4    58   334
 5    58   335
 6    57   336
 7    57   336
 8    55   337
 9    61   337
10    61   338
# ... with 53,930 more rows
```

```
diamonds %>%
  select( contains("l"))
```

```
# A tibble: 53,940 x 3
   color clarity table
   <ord> <ord>   <dbl>
 1 E     SI2        55
 2 E     SI1        61
 3 E     VS1        65
 4 I     VS2        58
 5 J     SI2        58
 6 J     VVS2       57
 7 I     VVS1       57
 8 H     SI1        55
 9 E     VS2        61
10 H     VS1        61
# ... with 53,930 more rows
```

2.4.1 rename()

Often when you're selecting columns to work on, you'll need to fix the names - `rename()` is useful for this. Let's fix that mis-spelled column

```
diamonds %>%
  rename( colour = color)
```

```
# A tibble: 53,940 x 10
   carat cut       colour clarity depth table price     x     y     z
   <dbl> <ord>     <ord>  <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
 1  0.23 Ideal     E      SI2      61.5    55   326  3.95  3.98  2.43
 2  0.21 Premium   E      SI1      59.8    61   326  3.89  3.84  2.31
 3  0.23 Good      E      VS1      56.9    65   327  4.05  4.07  2.31
 4  0.29 Premium   I      VS2      62.4    58   334  4.2   4.23  2.63
 5  0.31 Good      J      SI2      63.3    58   335  4.34  4.35  2.75
 6  0.24 Very Good J      VVS2     62.8    57   336  3.94  3.96  2.48
 7  0.24 Very Good I      VVS1     62.3    57   336  3.95  3.98  2.47
 8  0.26 Very Good H      SI1      61.9    55   337  4.07  4.11  2.53
 9  0.22 Fair      E      VS2      65.1    61   337  3.87  3.78  2.49
10  0.23 Very Good H      VS1      59.4    61   338  4     4.05  2.39
# ... with 53,930 more rows
```

2.5 filter()

The `filter()` function lets you select rows (observations) from your data frame based on criteria you specify. Here I'll look for all rows with a value of `G` for the `color` variable (I'll also pipe the output to the `head()` function to view just the top of the output.)

```
diamonds %>%
  filter( color == "G")
```

```
# A tibble: 11,292 x 10
   carat cut       color clarity depth table price     x     y     z
   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
 1  0.23 Very Good G     VVS2     60.4    58   354  3.97  4.01  2.41
 2  0.23 Ideal     G     VS1      61.9    54   404  3.93  3.95  2.44
 3  0.28 Ideal     G     VVS2     61.4    56   553  4.19  4.22  2.58
 4  0.31 Very Good G     SI1      63.3    57   553  4.33  4.3   2.73
 5  0.31 Premium   G     SI1      61.8    58   553  4.35  4.32  2.68
```

```
 6  0.24 Premium   G     VVS1     62.3    59   554  3.95  3.92  2.45
 7  0.7  Ideal     G     VS2      61.6    56  2757  5.7   5.67  3.5
 8  0.78 Very Good G     SI2      63.8    56  2759  5.81  5.85  3.72
 9  0.74 Ideal     G     SI1      61.6    55  2760  5.8   5.85  3.59
10  0.75 Premium   G     VS2      61.7    58  2760  5.85  5.79  3.59
# ... with 11,282 more rows
```

The syntax is fairly clear, just pass the column you want to think about and the condition to keep the rows. Multiple conditions can be used and all must be true to keep a row.

```
diamonds %>%
  filter( color == "G",
          cut == "Ideal" )
```

```
# A tibble: 4,884 x 10
   carat cut   color clarity depth table price     x     y     z
   <dbl> <ord> <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
 1  0.23 Ideal G     VS1      61.9    54   404  3.93  3.95  2.44
 2  0.28 Ideal G     VVS2     61.4    56   553  4.19  4.22  2.58
 3  0.7  Ideal G     VS2      61.6    56  2757  5.7   5.67  3.5
 4  0.74 Ideal G     SI1      61.6    55  2760  5.8   5.85  3.59
 5  0.75 Ideal G     SI1      62.2    55  2760  5.87  5.8   3.63
 6  0.71 Ideal G     VS2      62.4    54  2762  5.72  5.76  3.58
 7  0.64 Ideal G     VVS1     61.9    56  2766  5.53  5.56  3.43
 8  0.71 Ideal G     VS2      61.9    57  2771  5.73  5.77  3.56
 9  0.58 Ideal G     VVS1     61.5    55  2772  5.39  5.44  3.33
10  0.72 Ideal G     SI1      61.8    56  2776  5.72  5.75  3.55
# ... with 4,874 more rows
```

To make more complex queries, you'll need to combine comparisons and logical operators.

2.5.1 Comparisons

R provides the following comparison operators

- `==` - strictly equal to
- `!=` - not equal to
- `>`, `<`, `>=`, `<=` - greater than, less than, greater or equal to, less or equal to

These all work as you might expect. Except for `==`. Trying to use `==` on numbers with a decimal point is tricky because of rounding errors in the computer. See this:

```
(1 / 49) * 49 == 1
```

```
[1] FALSE
```

This statement is asking 'is 1 divided by 49, multiplied by 49, equal to 1'. The computer says `FALSE` because the computer can't store infinite numbers of decimal places. The rounding error is extremely small (down to the last 16th decimal place) but it is there. To deal with this rounding error we use the `near()` function, which checks numbers are the same to about the 8th decimal place.

```
near( (1 / 49) * 49, 1)
```

```
[1] TRUE
```

2.5.2 Logical operators

`filter()` uses combinations of R logical operators, these are `&` for `and`, `|` for `or` and `!` for `not`. You can build filters with these. Let's modify our query to find `color G` or `cut ideal`.

```
diamonds %>%
  filter( color == "G" | cut == "Ideal" )
```

```
# A tibble: 27,959 x 10
   carat cut       color clarity depth table price     x     y     z
   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
 1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
 2  0.23 Ideal     J     VS1      62.8    56   340  3.93  3.9   2.46
 3  0.31 Ideal     J     SI2      62.2    54   344  4.35  4.37  2.71
 4  0.3  Ideal     I     SI2      62      54   348  4.31  4.34  2.68
 5  0.23 Very Good G     VVS2     60.4    58   354  3.97  4.01  2.41
 6  0.33 Ideal     I     SI2      61.8    55   403  4.49  4.51  2.78
 7  0.33 Ideal     I     SI2      61.2    56   403  4.49  4.5   2.75
 8  0.33 Ideal     J     SI1      61.1    56   403  4.49  4.55  2.76
 9  0.23 Ideal     G     VS1      61.9    54   404  3.93  3.95  2.44
10  0.32 Ideal     I     SI1      60.9    55   404  4.45  4.48  2.72
# ... with 27,949 more rows
```

Note that the computer doesn't read this like it's English. Consider this

```
diamonds %>%
  filter( color == "G" | "F")
```

You might consider this to read `filter rows with color column equal toGorF‘`. The computer doesn't read it like this. It needs more explicit statements

```
diamonds %>%
  filter( color == "G" | color == "F")
```

```
# A tibble: 20,834 x 10
   carat cut       color clarity depth table price     x     y     z
   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
 1  0.22 Premium   F     SI1      60.4    61   342  3.88  3.84  2.33
 2  0.23 Very Good G     VVS2     60.4    58   354  3.97  4.01  2.41
 3  0.23 Very Good F     VS1      60.9    57   357  3.96  3.99  2.42
 4  0.23 Very Good F     VS1      60      57   402  4     4.03  2.41
 5  0.23 Very Good F     VS1      59.8    57   402  4.04  4.06  2.42
 6  0.23 Good      F     VS1      58.2    59   402  4.06  4.08  2.37
 7  0.29 Premium   F     SI1      62.4    58   403  4.24  4.26  2.65
 8  0.24 Very Good F     SI1      60.9    61   404  4.02  4.03  2.45
 9  0.23 Ideal     G     VS1      61.9    54   404  3.93  3.95  2.44
10  0.28 Ideal     G     VVS2     61.4    56   553  4.19  4.22  2.58
# ... with 20,824 more rows
```

Which can be cumbersome if you want to filter on one of many possible values. For that reason we have **%in%** , which works like

```
diamonds %>%
  filter( color  %in% c("G", "F") )
```

```
# A tibble: 20,834 x 10
   carat cut       color clarity depth table price     x     y     z
   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
 1  0.22 Premium   F     SI1      60.4    61   342  3.88  3.84  2.33
 2  0.23 Very Good G     VVS2     60.4    58   354  3.97  4.01  2.41
 3  0.23 Very Good F     VS1      60.9    57   357  3.96  3.99  2.42
 4  0.23 Very Good F     VS1      60      57   402  4     4.03  2.41
 5  0.23 Very Good F     VS1      59.8    57   402  4.04  4.06  2.42
 6  0.23 Good      F     VS1      58.2    59   402  4.06  4.08  2.37
 7  0.29 Premium   F     SI1      62.4    58   403  4.24  4.26  2.65
 8  0.24 Very Good F     SI1      60.9    61   404  4.02  4.03  2.45
```

```
 9  0.23 Ideal     G     VS1      61.9    54   404  3.93  3.95  2.44
10  0.28 Ideal     G     VVS2     61.4    56   553  4.19  4.22  2.58
# ... with 20,824 more rows
```

You can select anything not in a list given to `%in%` with a judicious `!` (not), again this is a bit weird if you translate directly from English, as the not goes first.

```
diamonds %>%
  filter( ! color %in% c("G", "F") )
```

```
# A tibble: 33,106 x 10
   carat cut       color clarity depth table price     x     y     z
   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
 1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
 2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
 3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31
 4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
 5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75
 6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
 7  0.24 Very Good I     VVS1     62.3    57   336  3.95  3.98  2.47
 8  0.26 Very Good H     SI1      61.9    55   337  4.07  4.11  2.53
 9  0.22 Fair      E     VS2      65.1    61   337  3.87  3.78  2.49
10  0.23 Very Good H     VS1      59.4    61   338  4     4.05  2.39
# ... with 33,096 more rows
```

2.6 mutate()

The `mutate()` function lets you add new columns based on values in other columns. Note that doing so to this data set makes it too big to print, so I'll `select()` the appropriate columns.

```
diamonds %>%
  mutate(price_per_carat = price / carat) %>%
  select(price, carat, price_per_carat)
```

```
# A tibble: 53,940 x 3
   price carat price_per_carat
   <int> <dbl>           <dbl>
 1   326  0.23           1417.
 2   326  0.21           1552.
```

```
 3   327  0.23           1422.
 4   334  0.29           1152.
 5   335  0.31           1081.
 6   336  0.24           1400
 7   336  0.24           1400
 8   337  0.26           1296.
 9   337  0.22           1532.
10   338  0.23           1470.
# ... with 53,930 more rows
```

You can refer to columns straight after creating them, so you can minimise `mutate()`s

```
diamonds %>%
  mutate(price_per_carat = price / carat,
         depth_per_ppc = depth / price_per_carat) %>%
  select(depth_per_ppc, price_per_carat)
```

```
# A tibble: 53,940 x 2
   depth_per_ppc price_per_carat
           <dbl>           <dbl>
 1        0.0434           1417.
 2        0.0385           1552.
 3        0.0400           1422.
 4        0.0542           1152.
 5        0.0586           1081.
 6        0.0449           1400
 7        0.0445           1400
 8        0.0478           1296.
 9        0.0425           1532.
10        0.0404           1470.
# ... with 53,930 more rows
```

2.6.1 Functions in mutate()

You can create a new column with `mutate()` using pretty much any vectorized R function. It's a bit complicated to explain what I mean by 'vectorized' so let's start with some examples.

```
diamonds %>%
  mutate(log_price = log(price)) %>%
  select(-x, -y, -z)
```

```
# A tibble: 53,940 x 8
   carat cut       color clarity depth table price log_price
   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int>     <dbl>
 1  0.23 Ideal     E     SI2      61.5    55   326      5.79
 2  0.21 Premium   E     SI1      59.8    61   326      5.79
 3  0.23 Good      E     VS1      56.9    65   327      5.79
 4  0.29 Premium   I     VS2      62.4    58   334      5.81
 5  0.31 Good      J     SI2      63.3    58   335      5.81
 6  0.24 Very Good J     VVS2     62.8    57   336      5.82
 7  0.24 Very Good I     VVS1     62.3    57   336      5.82
 8  0.26 Very Good H     SI1      61.9    55   337      5.82
 9  0.22 Fair      E     VS2      65.1    61   337      5.82
10  0.23 Very Good H     VS1      59.4    61   338      5.82
# ... with 53,930 more rows
```

```
diamonds %>%
  mutate( total_price = sum(price)) %>%
  select( -x, -y, -z)
```

```
# A tibble: 53,940 x 8
   carat cut       color clarity depth table price total_price
   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int>       <int>
 1  0.23 Ideal     E     SI2      61.5    55   326   212135217
 2  0.21 Premium   E     SI1      59.8    61   326   212135217
 3  0.23 Good      E     VS1      56.9    65   327   212135217
 4  0.29 Premium   I     VS2      62.4    58   334   212135217
 5  0.31 Good      J     SI2      63.3    58   335   212135217
 6  0.24 Very Good J     VVS2     62.8    57   336   212135217
 7  0.24 Very Good I     VVS1     62.3    57   336   212135217
 8  0.26 Very Good H     SI1      61.9    55   337   212135217
 9  0.22 Fair      E     VS2      65.1    61   337   212135217
10  0.23 Very Good H     VS1      59.4    61   338   212135217
# ... with 53,930 more rows
```

Observe how the same number is in all the rows in the last example, this highlights how this 'vectorized' function idea works.

Briefly, vectorized functions work on whole columns at a time, not just single rows. So if it makes sense to treat each element of the column individually, the function will do that. Consider this column (here printed on its side) with the `log()` function.

```
log(c(1,2,3))
```

```
[1] 0.0000000 0.6931472 1.0986123
```

You get back a column of numbers the same length as you put in, each item logged. With the `sum()` function it makes sense to return the sum of all the numbers in the column.

```
sum(c(1,2,3))
```

```
[1] 6
```

So you get back a single number. The behaviour of `mutate()` then is akin to taking the whole column or columns you specify, apply whatever function you ask for and putting the resulting column in the dataframe. If the resulting column is of length one, that just gets repeated until it fits. That is why we get repeats of the same number in the `sum()` example and why that number is the sum of all the prices.

If the result from the function isn't the same length as the column or has length of one - the function will fail. Most common functions will work nicely though.

2.6.2 if_else()

One final vectorized function is `if_else()`, this is useful when you want to add a column that annotates the data with an arbitrary value based on values. Let's add a column called `cost` that can be `high` or `low` depending on the `price`.

```
diamonds %>%
  mutate( cost = if_else( price > 335, "high", "low")) %>%
  select( -x, -y, -z)
```

```
# A tibble: 53,940 x 8
   carat cut       color clarity depth table price cost
   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <chr>
 1  0.23 Ideal     E     SI2      61.5    55   326 low
 2  0.21 Premium   E     SI1      59.8    61   326 low
 3  0.23 Good      E     VS1      56.9    65   327 low
 4  0.29 Premium   I     VS2      62.4    58   334 low
 5  0.31 Good      J     SI2      63.3    58   335 low
 6  0.24 Very Good J     VVS2     62.8    57   336 high
```

```
 7  0.24 Very Good I     VVS1     62.3    57   336 high
 8  0.26 Very Good H     SI1      61.9    55   337 high
 9  0.22 Fair      E     VS2      65.1    61   337 high
10  0.23 Very Good H     VS1      59.4    61   338 high
# ... with 53,930 more rows
```

The `if_else()` function then just adds the first value ("high") if the condition is 'true' else it puts the second value.

2.7 summarize() and group_by()

The `summarize()` function is a reductive function. It reduces entire dataframes to a single row, and returns an entirely new dataframe.

```
diamonds %>%
  summarize( mean_price = mean(price) )
```

```
# A tibble: 1 x 1
  mean_price
       <dbl>
1      3933.
```

`summarize()` is best used with `group_by()` which helps split dataframes into subsets. The `summarize()` function will run once for each subset created by `group_by()`. Let's find the mean price for every colour of diamonds. We'll do this by grouping the `diamonds` dataframe on `color`, then summarising.

```
diamonds %>%
  group_by(color) %>%
  summarize(mean_price = mean(price) )
```

```
# A tibble: 7 x 2
  color mean_price
  <ord>      <dbl>
1 D          3170.
2 E          3077.
3 F          3725.
4 G          3999.
5 H          4487.
```

```
6 I          5092.
7 J          5324.
```

This is where the power of dplyr starts to be obvious. Once we've got our dataframe into shape with the `select()`, `filter()` and `mutate()` functions, we can start to compute new information with `group_by()` and `summarize()` and some of the helper functions.

Let's group by two things, `color` and `cut`, and get the mean and standard deviation of `price`.

```
diamonds %>%
  group_by(color, cut) %>%
  summarize(
    mean_price = mean(price),
    sd = sd(price)
    )
```

```
`summarise()` has grouped output by 'color'. You can override using the
`.groups` argument.
```

```
# A tibble: 35 x 4
# Groups:   color [7]
   color cut       mean_price    sd
   <ord> <ord>          <dbl> <dbl>
 1 D     Fair           4291. 3286.
 2 D     Good           3405. 3175.
 3 D     Very Good      3470. 3524.
 4 D     Premium        3631. 3712.
 5 D     Ideal          2629. 3001.
 6 E     Fair           3682. 2977.
 7 E     Good           3424. 3331.
 8 E     Very Good      3215. 3408.
 9 E     Premium        3539. 3795.
10 E     Ideal          2598. 2956.
# ... with 25 more rows
```

Note how every combination of `color` and `cut` is made into subsets.

2.7.1 Helpful summarize() functions

There are numerous helpful summary functions. We've already seen `mean()` , `sum()` and `sd()`.

The function `n()` counts the number of items in a group. It doesn't need a column name to work on.

```
diamonds %>%
  group_by(color) %>%
  summarize(
     count = n()
    )
```

```
# A tibble: 7 x 2
  color count
  <ord> <int>
1 D      6775
2 E      9797
3 F      9542
4 G     11292
5 H      8304
6 I      5422
7 J      2808
```

Related is `n_distinct()` which counts the number of unique values in a group. This one needs to know which column of things you want to use. Let's see how many observations there are for each `cut` and how many different `colors` are observed in each `cut`

```
diamonds %>%
  group_by(cut) %>%
  summarize(
     items = n(),
     unique_colors = n_distinct(color)
    )
```

```
# A tibble: 5 x 3
  cut       items unique_colors
  <ord>     <int>         <int>
1 Fair       1610             7
2 Good       4906             7
```

```
3 Very Good 12082             7
4 Premium   13791             7
5 Ideal     21551             7
```

There are other helpful summary functions, here's a non-exhaustive list

- `max()` or `min()` - maximum or minimum value in a column
- `median()` - median value in a column
- `IQR()` - interquartile range (distance) between 25th and 75th percentile
- `first()` or `last()` - first or last values in a column

2.8 arrange()

The `arrange()` function is a straightforward function that helps you arrange the final table from `summarize()` a bit more nicely. It simply orders the rows in a way that you specify.

```
diamonds %>%
  arrange(price)
```

```
# A tibble: 53,940 x 10
   carat cut       color clarity depth table price     x     y     z
   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
 1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
 2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
 3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31
 4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
 5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75
 6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
 7  0.24 Very Good I     VVS1     62.3    57   336  3.95  3.98  2.47
 8  0.26 Very Good H     SI1      61.9    55   337  4.07  4.11  2.53
 9  0.22 Fair      E     VS2      65.1    61   337  3.87  3.78  2.49
10  0.23 Very Good H     VS1      59.4    61   338  4     4.05  2.39
# ... with 53,930 more rows
```

To sort biggest first, use `desc()`

```
diamonds %>%
  arrange(desc(price))
```

```
# A tibble: 53,940 x 10
   carat cut       color clarity depth table price     x     y     z
   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
 1  2.29 Premium   I     VS2      60.8    60 18823  8.5   8.47  5.16
 2  2    Very Good G     SI1      63.5    56 18818  7.9   7.97  5.04
 3  1.51 Ideal     G     IF       61.7    55 18806  7.37  7.41  4.56
 4  2.07 Ideal     G     SI2      62.5    55 18804  8.2   8.13  5.11
 5  2    Very Good H     SI1      62.8    57 18803  7.95  8     5.01
 6  2.29 Premium   I     SI1      61.8    59 18797  8.52  8.45  5.24
 7  2.04 Premium   H     SI1      58.1    60 18795  8.37  8.28  4.84
 8  2    Premium   I     VS1      60.8    59 18795  8.13  8.02  4.91
 9  1.71 Premium   F     VS2      62.3    59 18791  7.57  7.53  4.7
10  2.15 Ideal     G     SI2      62.6    54 18791  8.29  8.35  5.21
# ... with 53,930 more rows
```

2.9 Missing Values

Many (many!) datasets will have some missing values at some points. These are encoded in R as `NA`. They need to be dealt with explicitly as they mess up lots of calculations.

Look at the toy dataframe `incomplete` below

```
incomplete
```

```
  group size
1     A 10.4
2     B   NA
3     C  8.0
4     A  6.0
5     B   NA
6     C   NA
```

When we try to `summarize()` we get stuck

```
incomplete %>%
  group_by( group ) %>%
  summarize(mean_size = mean(size))
```

```
# A tibble: 3 x 2
  group mean_size
```

```
  <chr>     <dbl>
1 A           8.2
2 B          NA
3 C          NA
```

The groups with any `NA` can't be calculated. We need to tell our helper function to remove `NA` before we work with it.

```
incomplete %>%
  group_by( group ) %>%
  summarize(mean_size = mean(size, na.rm = TRUE))
```

```
# A tibble: 3 x 2
  group mean_size
  <chr>     <dbl>
1 A           8.2
2 B         NaN
3 C           8
```

This works! Though because the group `B` only had `NA` in it the formula for mean fails because we can't divide by `0` and we get `NaN` (not a number).

You might think you can use `filter()` to get rid of any rows with NA in, but you get a weird result

```
incomplete %>%
  filter(size != NA) %>%
  group_by( group ) %>%
  summarize(mean_size = mean(size))
```

```
# A tibble: 0 x 2
# ... with 2 variables: group <chr>, mean_size <dbl>
```

By definition `NA` means `Not available`, which is a nice way of saying `don't know`, so, strictly, `x == NA` means "is `x` equal to something we don't know the value of?" To which the answer can only be `don't know`, for which R uses `NA`. The result is that any comparison with `NA` in it is `NA`. `filter()` doesn't know whether any row passes so throws it out. You get no rows for `group_by()` to group.

If you want to check something is an `NA`, you can use `is.na()`

```
  incomplete %>%
    filter(! is.na(size)) %>%
    group_by( group ) %>%
    summarize(mean_size = mean(size))
```

```
# A tibble: 2 x 2
  group mean_size
  <chr>     <dbl>
1 A           8.2
2 C           8
```

Note that you lose the information for the `B` group, which may be important.

You may want to pair these operations with an `n()` column to give you an idea of how many values you use to get your answer.

```
  incomplete %>%
    group_by( group ) %>%
    summarize(
      mean_size = mean(size, na.rm = TRUE),
      sample_size = n()
      )
```

```
# A tibble: 3 x 3
  group mean_size sample_size
  <chr>     <dbl>       <int>
1 A           8.2           2
2 B         NaN             2
3 C           8             2
```

```
  incomplete %>%
    filter(! is.na(size)) %>%
    group_by( group ) %>%
    summarize(mean_size = mean(size),
      sample_size = n()

      )
```

```
# A tibble: 2 x 3
  group mean_size sample_size
```

```
  <chr>     <dbl>       <int>
1 A           8.2           2
2 C           8             1
```

With a more complicated call, you can explicitly get the number of `NA`s.

```
incomplete %>%
  group_by( group ) %>%
  summarize(
    mean_size = mean(size, na.rm = TRUE),
    sample_size = n(),
    nas = sum(is.na(size))
    )
```

```
# A tibble: 3 x 4
  group mean_size sample_size   nas
  <chr>     <dbl>       <int> <int>
1 A           8.2           2     0
2 B         NaN             2     2
3 C           8             2     1
```

2.10 Quiz

1. Load the package `nycflights13` which is a tidy data set of flight information out of New York. Look at the `flights` table that has been loaded and note the column names and types.
2. The individual planes are identified by their `tailnum`. Which plane has the worst on-time record?
3. What time of day should you fly to avoid delays as much as possible?
4. For each destination compute the total minutes of delay?
5. Find all destinations that have at least two carriers.

(These exercises are taken from pg 75 of R for Data Science - check there for more challenges.)

3 Combining Datasets

3.1 About this chapter

1. Questions:

- How do I combine dataframes?

2. Objectives:

- Understanding keys
- Explore `join` functions

3. Keypoints:

- Dataframes get joined on key columns. The rows that are retained depends on the type of `join` performed

3.2 Joining

Often you will want to combine data contained in more than one dataset. In this section we will look at the functions that help you do that.

3.2.1 Key columns

The joining operation depends on the two datasets having some values in some column in common. The column in each dataset that allows you to combine columns is the key column. Consider these dataframes

```
band_members
```

```
# A tibble: 3 x 2
  name  band
  <chr> <chr>
1 Mick  Stones
2 John  Beatles
3 Paul  Beatles
```

```
band_instruments
```

```
# A tibble: 3 x 2
  name  plays
  <chr> <chr>
1 John  guitar
2 Paul  bass
3 Keith guitar
```

Note that the two dataframes have a column in common `name`.

3.3 Join functions

Join functions work to combine two dataframes side-by-side in some way. Usually they use one column as a base and add columns to that one from the other.

3.3.1 left_join()

The most common sort of join is the left join. This takes one dataframe, considers it to be on the left of the join and combines the second dataframe on to it, skipping rows in the right dataframe that have nowhere to join

```
left_join( band_members, band_instruments, )
```

```
Joining, by = "name"
```

```
# A tibble: 3 x 3
  name  band    plays
  <chr> <chr>   <chr>
1 Mick  Stones  <NA>
2 John  Beatles guitar
3 Paul  Beatles bass
```

Note how the column in common `name` is used as the key through which to join and that the `band_member Keith` goes missing because it isn't in the `left` dataframe, which is the reference.

3.3.2 right_join()

`right_join()` is the complementary function.

```
right_join( band_members, band_instruments)
```

```
Joining, by = "name"

# A tibble: 3 x 3
  name  band    plays
  <chr> <chr>   <chr>
1 John  Beatles guitar
2 Paul  Beatles bass
3 Keith <NA>    guitar
```

See how this time `Keith` is retained as we're joining to the right table as the base, but as he has no entry in the left table, an `NA` is used to fill the missing value.

3.3.3 inner_join()

`inner_join()` keeps only rows that are completely shared

```
inner_join( band_members, band_instruments)
```

```
Joining, by = "name"

# A tibble: 2 x 3
  name  band    plays
  <chr> <chr>   <chr>
1 John  Beatles guitar
2 Paul  Beatles bass
```

3.3.4 full_join()

`full_join()` joins all rows as well as possible, generating `NA` as appropriate.

```
  full_join( band_members, band_instruments)
```

```
Joining, by = "name"
```

```
# A tibble: 4 x 3
  name  band    plays
  <chr> <chr>   <chr>
1 Mick  Stones  <NA>
2 John  Beatles guitar
3 Paul  Beatles bass
4 Keith <NA>    guitar
```

3.3.5 Joins with no common column names

What can we do when there is no common column names? Consider this variant of `band instruments`

```
  band_instruments2
```

```
# A tibble: 3 x 2
  artist plays
  <chr>  <chr>
1 John   guitar
2 Paul   bass
3 Keith  guitar
```

The `name` column is called artist - we can join by explicitly stating the column to join by

```
  left_join( band_members, band_instruments2, by = c("name" = "artist"))
```

```
# A tibble: 3 x 3
  name  band    plays
  <chr> <chr>   <chr>
1 Mick  Stones  <NA>
2 John  Beatles guitar
3 Paul  Beatles bass
```

3.4 Binding operations

These allow you to paste dataframes together.

`bind_rows()` sticks them together top-to-bottom.

```
  bind_rows(band_members, band_members)
```

```
# A tibble: 6 x 2
  name  band
  <chr> <chr>
1 Mick  Stones
2 John  Beatles
3 Paul  Beatles
4 Mick  Stones
5 John  Beatles
6 Paul  Beatles
```

Note the column names need not be identical for this to work. `NAs` are propogated as required.

```
  bind_rows(band_members, band_instruments)
```

```
# A tibble: 6 x 3
  name  band    plays
  <chr> <chr>   <chr>
1 Mick  Stones  <NA>
2 John  Beatles <NA>
3 Paul  Beatles <NA>
4 John  <NA>    guitar
5 Paul  <NA>    bass
6 Keith <NA>    guitar
```

`bind_cols()` sticks dataframes together side-by-side/

```
  bind_cols(band_members, band_instruments)
```

```
New names:
* `name` -> `name...1`
* `name` -> `name...3`
```

```
# A tibble: 3 x 4
  name...1 band    name...3 plays
  <chr>    <chr>   <chr>    <chr>
1 Mick     Stones  John     guitar
2 John     Beatles Paul     bass
3 Paul     Beatles Keith    guitar
```

Note how it doesn't do any sensible matching - it's just pasting them together. Repeated column names get modified. What happens if the dataframes aren't of equal length?

```
data_4_rows <- tibble( names = letters[1:4], values = 1:4)
bind_cols(band_members, data_4_rows)
```

```
Error in `bind_cols()`:
! Can't recycle `..1` (size 3) to match `..2` (size 4).
```

3.5 Quiz

The quiz for this section is mixed in with the quiz for section 5. When you get that far, do the quiz there.

4 dplyr and ggplot

1. Questions:

- How do I explore data graphically?

2. Objectives:

- Using `filter()` and `group_by()` to subset and scale data
- Using `summarize()` to get graphical summaries

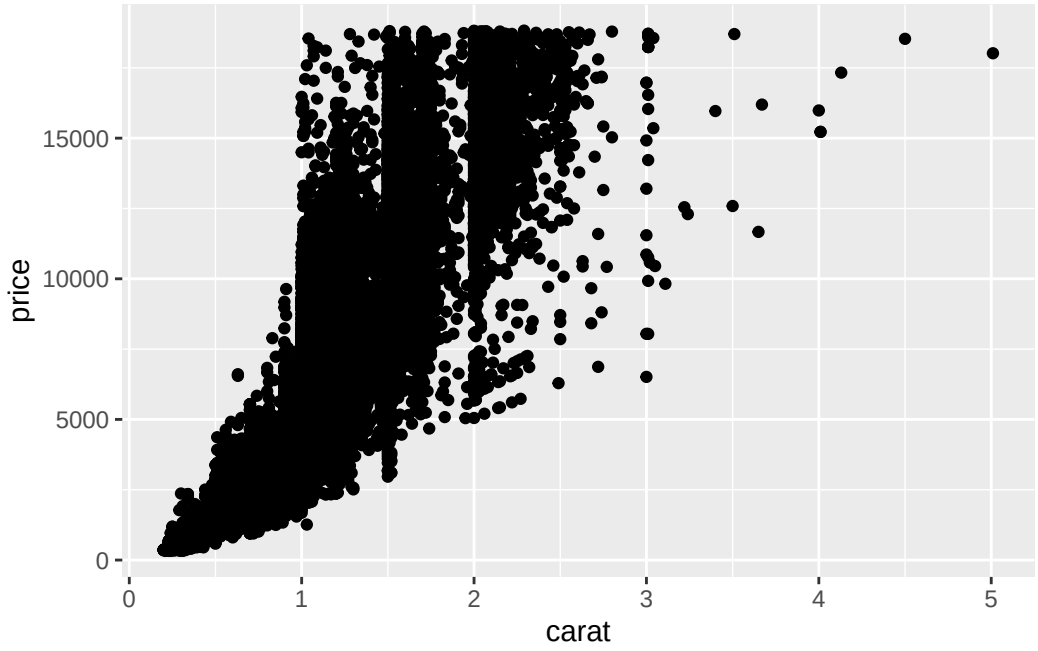
3. Keypoints:

- Piping filtered or summarized data to `ggplot2()` can give quick graphical readouts of data.

4.1 Piping to `ggplot()`

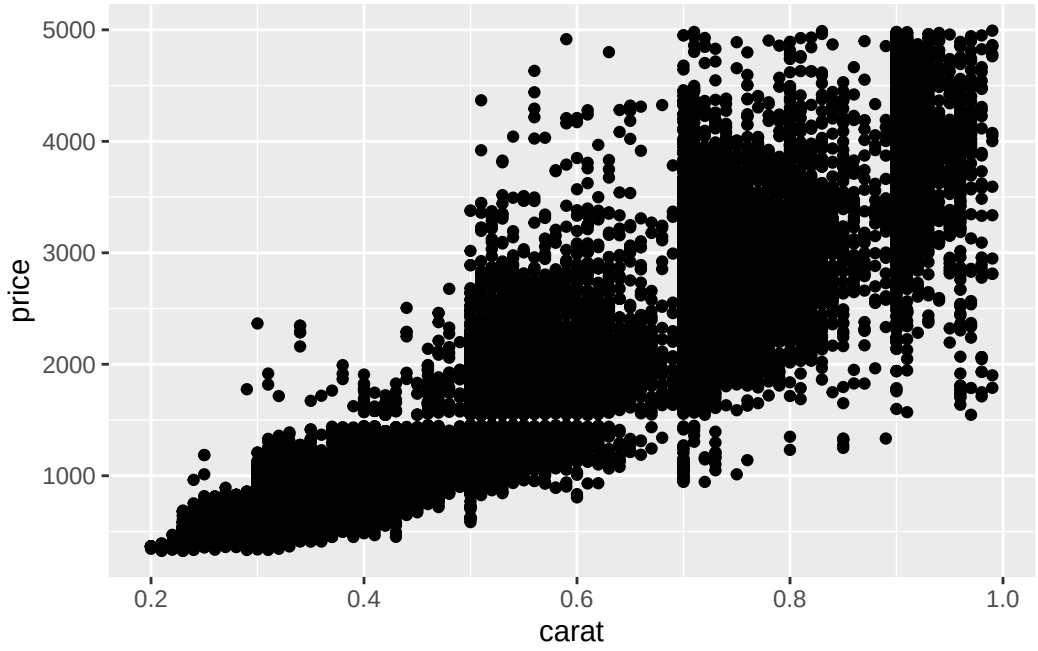
The pipe operator `%>%` we used earlier can be used with `ggplot()` commands to give us a graphical view of our data. Lets try the diamonds data

```
diamonds %>%
  ggplot() + aes(x = carat, y = price) + geom_point()
```



So, let's look at that messy bottom left corner, from 0 - 1 in `carat` and less than 5000 in price. A `filter()` should sort us out

```
diamonds %>%
  filter(carat < 1, price < 5000) %>%
  ggplot() + aes(x = carat, y = price) + geom_point()
```



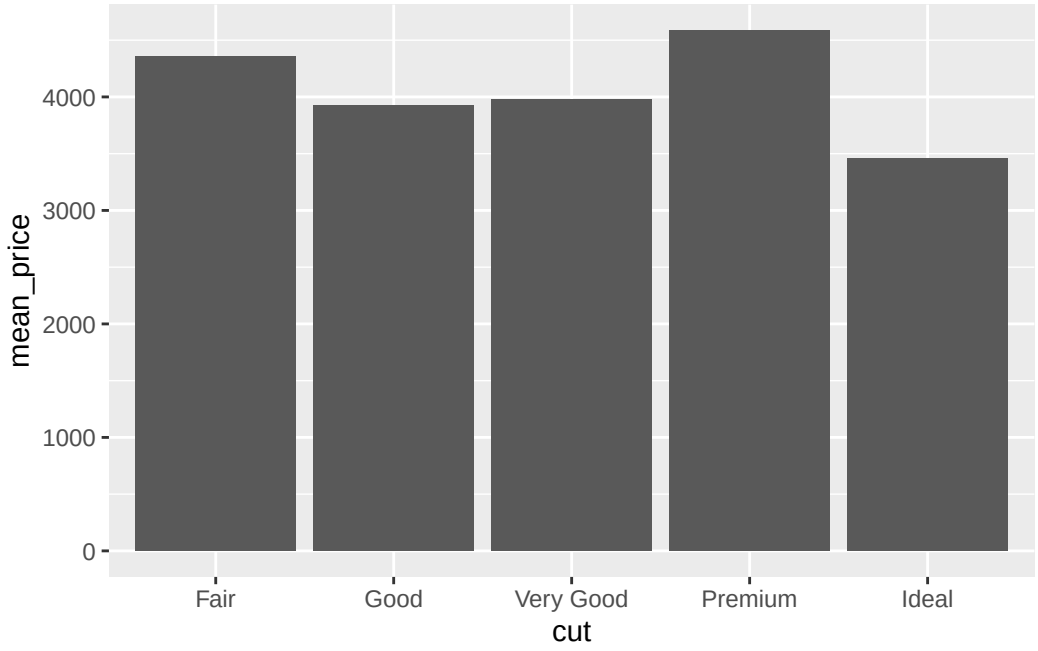
It's easier to do this sort of zooming and filtering with `dplyr` than it is by setting `ggplot` axes.

4.1.1 Quick bar charts

`dplyr` also provides a quick way to make bar charts in `ggplot`. Although bar charts are generally far less use than jitter or scatter plots, lots of supervisors like them. Which is a shame.

We need to run `group_by()` and `summary()` and send it to `ggplot`'s `geom_bar()`.

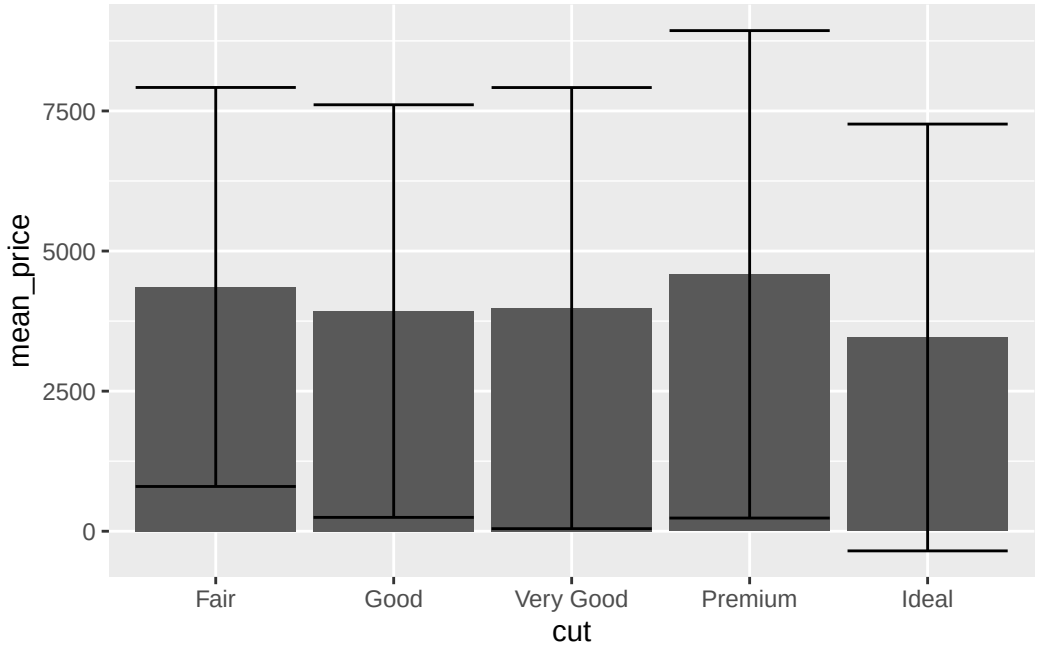
```
diamonds %>%
  group_by(cut) %>%
  summarize(mean_price = mean(price)) %>%
  ggplot() + aes(x = cut, y = mean_price) + geom_bar(stat="identity")
```



The `stat = "identity"` call tells `ggplot` to use the values passed (default behaviour is to do a count, which is a bit unexpected).

Error bars can be added by calculating the error on each group and adding it in to the table, then using the `geom_errorbar()` geom from `ggplot`. We'll use the standard deviation from the `sd()` function, though any error can be used.

```
diamonds %>%
  group_by(cut) %>%
  summarise(
    mean_price = mean(price),
    sd_price = sd(price)
    ) %>%
    ggplot() +
  aes(x = cut, y = mean_price) +
  geom_bar(stat="identity") +
  geom_errorbar(
      aes(
        ymin = mean_price - sd_price,
        ymax = mean_price + sd_price
        )
    )
```



It is helpful to look at the output from the `dplyr` bit to see what's going on here.

```
diamonds %>%
  group_by(cut) %>%
  summarise(
    mean_price = mean(price),
    sd_price = sd(price)
    )
```

```
# A tibble: 5 x 3
  cut       mean_price sd_price
  <ord>          <dbl>    <dbl>
1 Fair           4359.    3560.
2 Good           3929.    3682.
3 Very Good      3982.    3936.
4 Premium        4584.    4349.
5 Ideal          3458.    3808.
```

The `dplyr group_by()` and `summarise()` returns a table with two new columns. These are the values `ggplot` uses. The `mean_price` for the bar height, the `mean_price - sd_price` for the lower extent of each error bar, and `mean_price + sd_price` for the higher extent of each error bar.

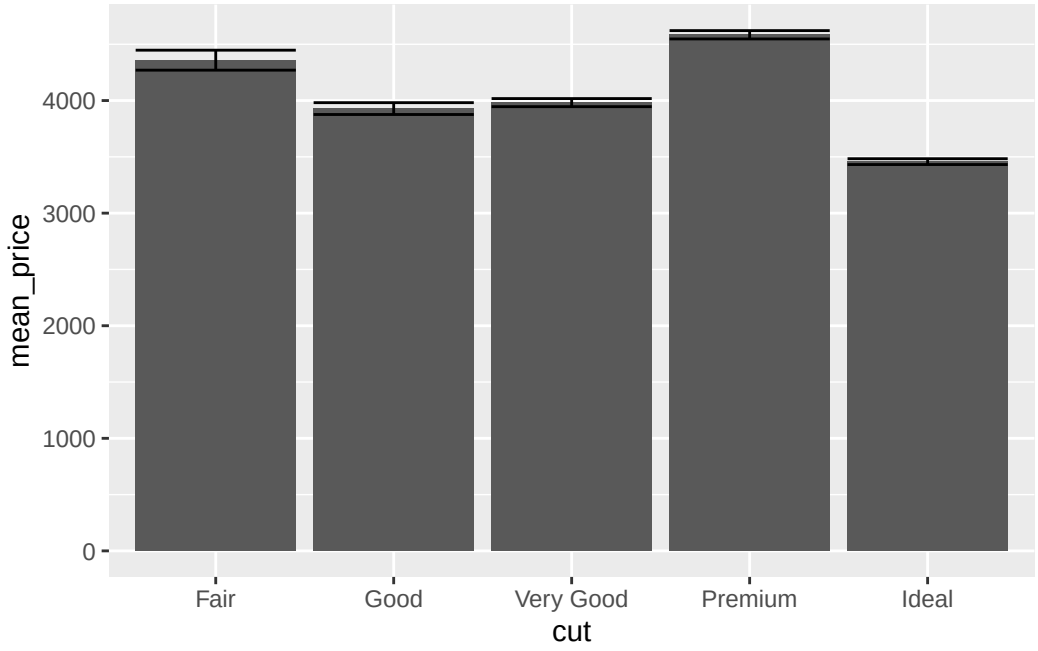
So how do we get standard error added onto the bars? Recalling that standard error is just the standard deviation divided by the square root of the sample size, and the sample size for a group would be the same as the number of things in it, which we get from the `n()` function we can use

```
diamonds %>%
  group_by(cut) %>%
  summarise(
    mean_price = mean(price),
    se_price = sd(price) / sqrt(n())
    )
```

```
# A tibble: 5 x 3
  cut       mean_price se_price
  <ord>          <dbl>    <dbl>
1 Fair           4359.     88.7
2 Good           3929.     52.6
3 Very Good      3982.     35.8
4 Premium        4584.     37.0
5 Ideal          3458.     25.9
```

These can be worked with as before

```
diamonds %>%
  group_by(cut) %>%
  summarise(
    mean_price = mean(price),
    se_price = sd(price)  / sqrt(n())
    ) %>%
    ggplot() +
  aes(x = cut, y = mean_price) +
  geom_bar(stat="identity") +
  geom_errorbar(
      aes(
        ymin = mean_price - se_price,
        ymax = mean_price + se_price
        )
    )
```



The figure shows how a large sample size really distorts error bar calculations! An interesting view of difference of price is given by using standard error and interpreting lack of overlap as a proxy for significance with such large sample sizes.

Part III

Making messy data tidy with `tidyr`

5 Tidying data

5.1 About this chapter

1. Questions:

- How do I go from non-tidy to tidy data structures?

2. Objectives:

- Manipulating dataframes with the `tidyr` verbs

3. Keypoints:

- The `tidyr` package contains functions that affect the layout and structure of dataframes.

5.2 tidyr

tidyr is a tool for manipulating layout of datasets. As part of the tidyverse it is loaded when you use `library(tidyverse)` but can be loaded on its own with `library(tidyr)`. tidyr has two main functions - `spread()` and `gather()`.

5.2.1 Sample tidy datasets

Let's look at five sample tables that show the same data in different ways, only one of which counts as tidy.

```
table1
```

```
# A tibble: 6 x 4
  country      year  cases population
  <chr>       <int>  <int>      <int>
1 Afghanistan  1999    745   19987071
2 Afghanistan  2000   2666   20595360
3 Brazil       1999  37737  172006362
```

```
4 Brazil       2000  80488  174504898
5 China        1999 212258 1272915272
6 China        2000 213766 1280428583
```

```
table2
```

```
# A tibble: 12 x 4
   country      year type            count
   <chr>       <int> <chr>           <int>
 1 Afghanistan  1999 cases             745
 2 Afghanistan  1999 population    19987071
 3 Afghanistan  2000 cases            2666
 4 Afghanistan  2000 population    20595360
 5 Brazil       1999 cases           37737
 6 Brazil       1999 population  172006362
 7 Brazil       2000 cases           80488
 8 Brazil       2000 population  174504898
 9 China        1999 cases          212258
10 China        1999 population 1272915272
11 China        2000 cases          213766
12 China        2000 population 1280428583
```

```
table3
```

```
# A tibble: 6 x 3
  country      year rate
* <chr>       <int> <chr>
1 Afghanistan  1999 745/19987071
2 Afghanistan  2000 2666/20595360
3 Brazil       1999 37737/172006362
4 Brazil       2000 80488/174504898
5 China        1999 212258/1272915272
6 China        2000 213766/1280428583
```

```
table4a
```

```
# A tibble: 3 x 3
  country     `1999` `2000`
* <chr>        <int>  <int>
```

```
1 Afghanistan    745   2666
2 Brazil       37737  80488
3 China       212258 213766
```

```
table4b
```

```
# A tibble: 3 x 3
  country         `1999`     `2000`
* <chr>            <int>      <int>
1 Afghanistan   19987071   20595360
2 Brazil       172006362  174504898
3 China       1272915272 1280428583
```

The tidy data is in `table1`.

- `table2` is not tidy because not every variable has its own column. The `count` column has values for `cases` and `population` and these are two different things. The `type` column *could* be a variable on its own, but as used here its a way to mix up the count variable unneccesarily.
- `table3` is not tidy because `rate` is currently a function of two variables - literally its printed as `cases/population`. The column `rate` should contain the result of `cases/population` and if we wanted to retain the `case` and `population` information it should be in its own column, like in `table1`
- `table4a` and `table4b` aren't tidy, because the data are split over two tables and in each table the values of the `year` variable are split over multiple columns.

Let's work with each of these non-tidy datasets in turn to get them tidy.

5.3 pivot_longer()

The most common problem is that in `table4a`, where the values of a variable are split over multiple columns.

```
table4a
```

```
# A tibble: 3 x 3
  country     `1999` `2000`
* <chr>        <int>  <int>
1 Afghanistan    745   2666
2 Brazil       37737  80488
3 China       212258 213766
```

To tidy this, we can use `pivot_longer()`, which increases the number of rows and decreases the number of columns. It needs three bits of information:

1. The column(s) to keep un-pivoted - these are columns that are likely already tidy. All other columns will be pivoted
2. The name of a new column in which to put the old column names
3. The name of a new column in which to put the old values

The function call looks like this:

```
table4a %>%
  pivot_longer(-country, names_to = "year", values_to = "cases")
```

```
# A tibble: 6 x 3
  country     year   cases
  <chr>       <chr>  <int>
1 Afghanistan 1999     745
2 Afghanistan 2000    2666
3 Brazil      1999   37737
4 Brazil      2000   80488
5 China       1999  212258
6 China       2000  213766
```

Note how we use the `-country` syntax to mean 'pivot everything but `country`'. The `names_to` argument tells `pivot_longer()` where to put the names, and the `values_to` argument specifies where the numbers should go.

Note too how the columns we pivoted (`1999` and `2000`) have been dropped from the table completely. This little table is now tidy.

We can do the same with `table4b` but this one has the value `population`

```
table4b %>%
  pivot_longer(-country, names_to = "year", values_to = "population")
```

```
# A tibble: 6 x 3
  country     year  population
  <chr>       <chr>      <int>
1 Afghanistan 1999    19987071
2 Afghanistan 2000    20595360
3 Brazil      1999   172006362
4 Brazil      2000   174504898
```

```
5 China       1999  1272915272
6 China       2000  1280428583
```

To combine these together we can use `left_join()`.

```
t4a <- table4a %>%
  pivot_longer(-country, names_to = "year", values_to = "cases")
t4b <- table4b %>%
  pivot_longer(-country, names_to = "year", values_to = "population")

left_join(t4a, t4b)
```

```
Joining, by = c("country", "year")
```

```
# A tibble: 6 x 4
  country     year   cases population
  <chr>       <chr>  <int>      <int>
1 Afghanistan 1999     745   19987071
2 Afghanistan 2000    2666   20595360
3 Brazil      1999   37737  172006362
4 Brazil      2000   80488  174504898
5 China       1999  212258 1272915272
6 China       2000  213766 1280428583
```

Note we don't need to specify the `by =` argument, since the two tables have column names in common - `left_join()` works that out and does the join automatically.

5.4 pivot_wider()

The inverse function to `pivot_longer()` is `pivot_wider()` which increases column number and decreases row count. This function needs two pieces of information

1. The column from which to get the new row names
2. The columns from which to get the values

This is useful for dealing with the `table2` case.

```
table2 %>%
  pivot_wider(names_from = type, values_from = count)
```

```
# A tibble: 6 x 4
  country      year  cases population
  <chr>       <int>  <int>      <int>
1 Afghanistan  1999    745   19987071
2 Afghanistan  2000   2666   20595360
3 Brazil       1999  37737  172006362
4 Brazil       2000  80488  174504898
5 China        1999 212258 1272915272
6 China        2000 213766 1280428583
```

5.5 separate()

The `seperate()` function turns one column into many by splitting the value whenever a particular character appears. Remember `table3`

```
table3
```

```
# A tibble: 6 x 3
  country      year rate
* <chr>       <int> <chr>
1 Afghanistan  1999 745/19987071
2 Afghanistan  2000 2666/20595360
3 Brazil       1999 37737/172006362
4 Brazil       2000 80488/174504898
5 China        1999 212258/1272915272
6 China        2000 213766/1280428583
```

We can separate that `rate` column into two - `cases` and `population`

```
table3 %>%
  separate(rate, into = c("cases", "population"))
```

```
# A tibble: 6 x 4
  country      year cases  population
  <chr>       <int> <chr>  <chr>
1 Afghanistan  1999 745    19987071
2 Afghanistan  2000 2666   20595360
3 Brazil       1999 37737  172006362
4 Brazil       2000 80488  174504898
5 China        1999 212258 1272915272
```

```
6 China        2000 213766 1280428583
```

By default `separate()` splits things on any non-numeric character. But we can be explicit with the `sep` argument.

```
table3 %>%
  separate(rate, into = c("cases", "population"), sep = "/")
```

```
# A tibble: 6 x 4
  country      year cases  population
  <chr>       <int> <chr>  <chr>
1 Afghanistan  1999 745    19987071
2 Afghanistan  2000 2666   20595360
3 Brazil       1999 37737  172006362
4 Brazil       2000 80488  174504898
5 China        1999 212258 1272915272
6 China        2000 213766 1280428583
```

This works just as well, but is useful if the computer makes a bad guess.

Note the column types of `cases` and `population`, they're down as `chr`. By default the type of the parent column is retained, but you can make `separate()` guess what type the new column is with the `convert` argument.

```
table3 %>%
  separate(rate, into = c("cases", "population"), sep = "/", convert = TRUE)
```

```
# A tibble: 6 x 4
  country      year  cases population
  <chr>       <int>  <int>      <int>
1 Afghanistan  1999    745   19987071
2 Afghanistan  2000   2666   20595360
3 Brazil       1999  37737  172006362
4 Brazil       2000  80488  174504898
5 China        1999 212258 1272915272
6 China        2000 213766 1280428583
```

And now, we're back to tidy data.

5.6 unite()

The `unite()` function is the inverse of `separate()`, and combines multiple columns into a single one.

To demonstrate `unite()` we can make a new table, `table5` by using the `separate()` function on the `year` column in `table3`. Passing `sep` a number, tells it just to split that many characters into the string

Here's `table5`

```
table5 <- table3 %>%
  separate(year, into = c("century", "year"), sep = 2, convert = TRUE)
table5
```

```
# A tibble: 6 x 4
  country     century  year rate
  <chr>         <int> <int> <chr>
1 Afghanistan      19    99 745/19987071
2 Afghanistan      20     0 2666/20595360
3 Brazil           19    99 37737/172006362
4 Brazil           20     0 80488/174504898
5 China            19    99 212258/1272915272
6 China            20     0 213766/1280428583
```

We can now re- `unite()` `table5`. The arguments for this function are just the name of the new column, and the columns to join

```
table5 %>%
  unite(new, century, year)
```

```
# A tibble: 6 x 3
  country     new   rate
  <chr>       <chr> <chr>
1 Afghanistan 19_99 745/19987071
2 Afghanistan 20_0  2666/20595360
3 Brazil      19_99 37737/172006362
4 Brazil      20_0  80488/174504898
5 China       19_99 212258/1272915272
6 China       20_0  213766/1280428583
```

Here the default is to use an underscore _ to join the values, but we can be explicit and use nothing with the `sep` argument

```
table5 %>%
  unite(new, century, year, sep="")
```

```
# A tibble: 6 x 3
  country     new   rate
  <chr>       <chr> <chr>
1 Afghanistan 1999  745/19987071
2 Afghanistan 200   2666/20595360
3 Brazil      1999  37737/172006362
4 Brazil      200   80488/174504898
5 China       1999  212258/1272915272
6 China       200   213766/1280428583
```

5.7 Quiz

1. Examine the `table1` and `table4a` datasets. Combine `table4a` to `table1` to create two new columns. Ensure the columns make sense and retain data integrity.
2. Tidying data is hard! And it needs you to know your data quite well, which naturally takes time. Rather than quiz questions here, a worked example will give good benefit, so let's try one of those. Do the Case Study on page 163 of R for Data Science. If you don't have the print edition it is available online here Case Study for Tidy Data.

6 Loading data from files

6.1 About this chapter

1. Questions:

- How do I get my data into R?

2. Objectives:

- Loading a '.csv' file
- Checking column contents
- Dealing with headers and column names
- Loading a specific sheet from a '.xlsx' file.

3. Keypoints:

- The `readr` and `readxl` packages contain functions for loading data from .csv and .xslx files. These functions help you to ensure that your data is loaded as you expect.

6.2 readr

readr is a tool for loading data into R. As part of the tidyverse it is loaded when you use `library(tidyverse)` but can be loaded on its own with `library(readr)`. We will use `readr` to load in data from a 'flat' `.csv` file. Most

6.2.1 read_csv()

The main function is `read_csv()` which can read a standard comma separated values file from disk into an R dataframe. There are a few variants of `read_csv()` which may be appropriate for different sorts of `.csv` file, but they all work the same.

- `read_csv2()` - reads semi-colon delimited files, which are commonly used where a comma is used as a decimal separator
- `read_tsv()` - reads tab delimited files
- `read_delim()` - reads files delimited by an arbitrary character

The first argument to `read_csv()` is the path to the file to read. Here I'll read a file on my Desktop that contains the diamonds data we've been using.

```
read_csv("~/Desktop/diamonds.csv")
```

```
Rows: 53940 Columns: 10
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (3): cut, color, clarity
dbl (7): carat, depth, table, price, x, y, z

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
# A tibble: 53,940 x 10
   carat cut       color clarity depth table price     x     y     z
   <dbl> <chr>     <chr> <chr>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
 2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
 3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31
 4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
 5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75
 6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
 7  0.24 Very Good I     VVS1     62.3    57   336  3.95  3.98  2.47
 8  0.26 Very Good H     SI1      61.9    55   337  4.07  4.11  2.53
 9  0.22 Fair      E     VS2      65.1    61   337  3.87  3.78  2.49
10  0.23 Very Good H     VS1      59.4    61   338  4     4.05  2.39
# ... with 53,930 more rows
```

On loading we see a column specification, `read_csv()` has guessed at what the columns should be and made those types. Its fine for the most part, but some of those columns we'd prefer to be factors. We can set our own column specification to force the column types on loading. We only have to do the ones that `read_csv()` gets wrong. Specifically, lets fix `cut` and `color` to a `factor`. We can do that with the `col_types` argument.

```
read_csv("~/Desktop/diamonds.csv",
    col_types = cols(
      cut = col_factor(NULL),
      color = col_factor(NULL)
    )
)
```

```
# A tibble: 53,940 x 10
   carat cut       color clarity depth table price     x     y     z
   <dbl> <fct>     <fct> <chr>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
 2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
 3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31
 4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
 5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75
 6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
 7  0.24 Very Good I     VVS1     62.3    57   336  3.95  3.98  2.47
 8  0.26 Very Good H     SI1      61.9    55   337  4.07  4.11  2.53
 9  0.22 Fair      E     VS2      65.1    61   337  3.87  3.78  2.49
10  0.23 Very Good H     VS1      59.4    61   338  4     4.05  2.39
# ... with 53,930 more rows
```

6.2.2 Parser functions

This works by assigning a parser function that returns a specific type to each column, here it's `col_factor()`. There are parser functions for all types of data, and all of them can be used if `read_csv()` doesn't guess your data properly. We won't go into detail of all of them, just remember that if your numbers or dates or stuff won't load properly, there's a parser function that can help.

The parser functions all have their own arguments, so we can manipulate those. We can see the `NULL` argument being passed to `col_factor()` above, which means 'all values found should be used as levels of the factor'. This is a great default setting, but if we have a large file, it won't help us find unexpected values.

Consider a situation where we are certain we should only have the values `Fair`, `Good` and `Very Good` for `cut` in our `diamonds` data. We can make the parser function check this for us and give a warning if it finds anything else.

```
read_csv("~/Desktop/diamonds.csv",
    col_types = cols(
      cut = col_factor(levels = c( "Fair" ,"Good", "Very Good")),
      color = col_factor(NULL)
    )
)
```

```
Warning: One or more parsing issues, call `problems()` on your data frame for details,
e.g.:
  dat <- vroom(...)
```

```
  problems(dat)
```

```
# A tibble: 53,940 x 10
   carat cut       color clarity depth table price     x     y     z
   <dbl> <fct>     <fct> <chr>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1  0.23 <NA>      E     SI2      61.5    55   326  3.95  3.98  2.43
 2  0.21 <NA>      E     SI1      59.8    61   326  3.89  3.84  2.31
 3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31
 4  0.29 <NA>      I     VS2      62.4    58   334  4.2   4.23  2.63
 5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75
 6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
 7  0.24 Very Good I     VVS1     62.3    57   336  3.95  3.98  2.47
 8  0.26 Very Good H     SI1      61.9    55   337  4.07  4.11  2.53
 9  0.22 Fair      E     VS2      65.1    61   337  3.87  3.78  2.49
10  0.23 Very Good H     VS1      59.4    61   338  4     4.05  2.39
# ... with 53,930 more rows
```

This time, we get a large number of warnings. Though the output is quite cryptic at first glance, `read_csv()` is complaining that it found values for cut that were not in the list we passed to the parser function.

Hence we can use parsers to ensure we are loading in the data we expect and generate errors if not.

6.2.3 Headers and column names

By default `read_csv()` uses the first line of the file for column names. Consider this toy example.

```
toy_csv <-
"a,b,c
1,2,3
4,5,6"
read_csv(toy_csv)
```

```
Rows: 2 Columns: 3
-- Column specification --------------------------------------------------------
Delimiter: ","
dbl (3): a, b, c

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
# A tibble: 2 x 3
      a     b     c
  <dbl> <dbl> <dbl>
1     1     2     3
2     4     5     6
```

The first line of the toy file becomes the column headings. This may not be appropriate, since there could be some metadata in the file

```
toy_csv <-
"some info about stuff
a,b,c
1,2,3
4,5,6"
read_csv(toy_csv)
```

```
Warning: One or more parsing issues, call `problems()` on your data frame for details,
e.g.:
  dat <- vroom(...)
  problems(dat)

Rows: 3 Columns: 1
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (1): some info about stuff

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.

# A tibble: 3 x 1
  `some info about stuff`
  <chr>
1 a,b,c
2 1,2,3
3 4,5,6
```

The loaded data gets really messed up, so we can skip a set number of lines if needed

```
toy_csv <-
"some info about stuff
a,b,c
```

```
  1,2,3
  4,5,6"
  read_csv(toy_csv, skip = 1)
```

```
Rows: 2 Columns: 3
-- Column specification --------------------------------------------------------
Delimiter: ","
dbl (3): a, b, c

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
# A tibble: 2 x 3
      a     b     c
  <dbl> <dbl> <dbl>
1     1     2     3
2     4     5     6
```

Alternatively, you might have comments that begin with a particular character. You can use the `comment` argument to skip those lines

```
  toy_csv <-
  "#some info about stuff
  #some more info
  #goodness, lots of INFO
  a,b,c
  1,2,3
  4,5,6"
  read_csv(toy_csv, comment = "#")
```

```
Rows: 2 Columns: 3
-- Column specification --------------------------------------------------------
Delimiter: ","
dbl (3): a, b, c

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
# A tibble: 2 x 3
      a     b     c
```

```
  <dbl> <dbl> <dbl>
1     1     2     3
2     4     5     6
```

The data might not have any column names at all, the first row may data. This situation is handled with `col_names` argument

```
toy_csv <-
"1,2,3
4,5,6"
read_csv(toy_csv, col_names = FALSE)
```

```
Rows: 2 Columns: 3
-- Column specification --------------------------------------------------------
Delimiter: ","
dbl (3): X1, X2, X3

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.

# A tibble: 2 x 3
     X1    X2    X3
  <dbl> <dbl> <dbl>
1     1     2     3
2     4     5     6
```

So, `read_csv()` sets up arbitrary column names. We can specify column names if we wish

```
toy_csv <-
"a,b,c
1,2,3
4,5,6"
read_csv(toy_csv, col_names = c("x", "y", "z"))
```

```
Rows: 3 Columns: 3
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (3): x, y, z

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
# A tibble: 3 x 3
  x     y     z
  <chr> <chr> <chr>
1 a     b     c
2 1     2     3
3 4     5     6
```

6.2.4 Missing values

There are many different ways of encoding missing values, you can tell `read_csv()` which character represents a missing value explicitly with the `na` argument. These values will all be loaded as proper `NA`.

```
toy_csv <-
"a,b,c
1,_,3
4,5,_"
read_csv(toy_csv, na = "_")
```

```
Rows: 2 Columns: 3
-- Column specification --------------------------------------------------------
Delimiter: ","
dbl (3): a, b, c

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.

# A tibble: 2 x 3
      a     b     c
  <dbl> <dbl> <dbl>
1     1    NA     3
2     4     5    NA
```

6.3 Writing Files

A complementary function to `read_csv()` `write_csv()` allows you to write a dataframe out to a '.csv' file. The convention is straightforward, you need the name of the dataframe and the name of the file and path to write to.

```
write_csv(diamonds, "~/Desktop/my_data.csv")
```

6.4 readxl

The `readxl` package is installed as part of the tidyverse `install.packages()` command, but it is not part of the core, so `library(tidyverse)` does not load it. You must do it explicitly with `library(readxl)`.

6.4.1 read_xlsx()

The main function is `read_xlsx()`, it's similar to `read_csv()`.

```
library(readxl)
read_xlsx("~/Desktop/datasets.xlsx")
```

```
# A tibble: 150 x 5
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
          <dbl>       <dbl>        <dbl>       <dbl> <chr>
 1          5.1         3.5          1.4         0.2 setosa
 2          4.9         3            1.4         0.2 setosa
 3          4.7         3.2          1.3         0.2 setosa
 4          4.6         3.1          1.5         0.2 setosa
 5          5           3.6          1.4         0.2 setosa
 6          5.4         3.9          1.7         0.4 setosa
 7          4.6         3.4          1.4         0.3 setosa
 8          5           3.4          1.5         0.2 setosa
 9          4.4         2.9          1.4         0.2 setosa
10          4.9         3.1          1.5         0.1 setosa
# ... with 140 more rows
```

By default it loads the first worksheet, you can examine the sheets available with `excel_sheets()`

```
excel_sheets("~/Desktop/datasets.xlsx")
```

```
[1] "iris"     "chickwts" "mtcars"   "quakes"
```

Then load in the one you want.

```
read_xlsx("~/Desktop/datasets.xlsx", sheet = "chickwts")
```

```
# A tibble: 71 x 2
   weight feed
    <dbl> <chr>
 1    179 horsebean
 2    160 horsebean
 3    136 horsebean
 4    227 horsebean
 5    217 horsebean
 6    168 horsebean
 7    108 horsebean
 8    124 horsebean
 9    143 horsebean
10    140 horsebean
# ... with 61 more rows
```

Loading then follows the same pattern as for `read_csv()`, with a difference in the column specifications - in this package its much simpler. You can only specify type columnwise and the specification can only be one of "skip", "guess", "logical", "numeric", "date", "text" or "list" - meaning you can't do the advanced parsing as for `read_csv()`.

A sample spec might look like

```
read_xlsx(
  "~/Desktop/datasets.xlsx",
  sheet = "chickwts",
  col_types = c("numeric", "text")
  )
```

```
# A tibble: 71 x 2
   weight feed
    <dbl> <chr>
 1    179 horsebean
 2    160 horsebean
 3    136 horsebean
 4    227 horsebean
 5    217 horsebean
 6    168 horsebean
 7    108 horsebean
 8    124 horsebean
 9    143 horsebean
10    140 horsebean
# ... with 61 more rows
```

Prerequisites

No specific knowledge is assumed for this book, though you will need to install some software.

1. R
2. RStudio
3. The `tidyverse` packages

You'll also need the following files diamonds.csv and datasets.xlsx

Installing R

Follow this link and install the right version for your operating system https://www.stats.bris.ac.uk/R/

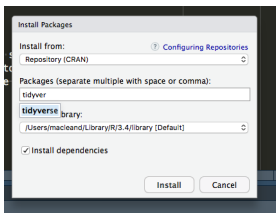
Installing RStudio

Follow this link and install the right version for your operating system https://www.rstudio.com/products/rstudio/download/

Installing R packages in RStudio

To install all the `tidyverse` packages in one go start RStudio and use the Packages tab in the lower-right panel. Click the install button (top left of the panel) and enter 'tidyverse', then click install as in this picture

To complete the quizzes you'll need a package called `nycflights13`. Install that in the same way.



R Fundamentals

About this chapter

1. Questions:

- How do I use R?

2. Objectives:

- Become familiar with R syntax
- Understand the concepts of objects and assignment
- Get exposed to a few functions

3. Keypoints:

- R's capabilities are provided by functions
- R users call functions and get results

Working with R

In this workshop we'll use R in the extremely useful RStudio software For the most part we'll work interactively, meaning we'll type stuff straight into the R console in RStudio (Usually this is a window on the left or lower left) and get our results there too (usually in the consoled or in a window on the right). That's what you see in panels like the ones below - first the thing to type into R, and below it, the calculated result from R. Let's look at how R works by using it for it's most basic job - as a calculator:

```
3 + 5
```

```
[1] 8
```

```
12 * 2
```

```
[1] 24
```

```
1 / 3
```

```
[1] 0.3333333
```

```
12 * 2
```

```
[1] 24
```

```
 3 / 0
```

```
[1] Inf
```

Fairly straightforward, we type in the expression and we get a result. That's how this whole book will work, you type the stuff in, and get answers out. It'll be easiest to learn if you go ahead and copy the examples one by one. Try to resist the urge to use copy and paste. Typing longhand really encourages you to look at what you're entering.

As far as the R ouput itself goes, it's really straightforward - its just the answer with a `[1]` stuck on the front. This `[1]` tells us how far through the output we are. Often R will return long lists of numbers and it can be helpful to have this extra information

Variables

We can save the output of operations for later use by giving it a name using the assignment symbol `<-`. Read this symbol as 'gets', so `x <- 5` reads as 'x gets 5'. These names are called variables, because the value they are associated with can change.

Let's give five a name, `x` then refer to the value 5 by it's name. We can then use the name in place of the value. In the jargon of computing we say we are assigning a value to a variable.

```
x <- 5
x
```

```
[1] 5
```

```
x * 2
```

```
[1] 10
```

```
y <- 3
x * y
```

```
[1] 15
```

This is of course of limited value with just numbers but is of great value when we have large datasets, as the whole thing can be referred to by the variable.

Using objects and functions

At the top level, R is a simple language with two types of thing: functions and objects. As a user you will use functions to do stuff, and get back objects as an answer. Functions are easy to spot, they are a name followed by a pair of brackets like `mean()` is the function for calculating a mean. The options (or arguments) for the function go inside the brackets:

```
sqrt(16)
```

```
[1] 4
```

Often the result from a function will be more complicated than a simple number object, often it will be a vector (simple list), like from the `rnorm()` function that returns lists of random numbers

```
rnorm(100)
```

```
  [1]  1.520700076  1.036295473 -0.874599369 -0.260136251 -0.056805311
  [6]  0.749202701 -0.096622995 -0.066419309  0.252636265 -0.886850139
 [11] -1.938269917 -0.792701681 -1.363333540 -1.362584355 -0.648782720
 [16]  1.939155384 -1.314069238 -0.008326479  0.441398889 -1.107396158
 [21] -1.155586431 -0.402659726 -1.227652192  0.937668935  0.272135223
 [26] -0.084738579 -1.393393079 -0.895853294 -1.247726973  0.207936924
 [31]  0.405781234 -0.697056048  2.508132859 -0.900891178  0.788282000
 [36]  1.550095361  1.278454358  0.281136570 -1.990978556  0.616745489
 [41]  1.210177857  0.319485078  1.531433602 -0.356557177  0.285202059
 [46] -0.255506914  2.058455279 -0.188188467 -0.537531336 -0.882161159
 [51]  1.598926240 -0.924468708  0.207928789  0.136033386 -0.061600532
 [56]  0.008107545  0.281493956 -0.664740240 -0.017240637 -0.592897451
 [61] -0.650489689 -0.021639955  0.577086388  0.170213713  1.565463050
 [66]  1.269155178 -1.044526533  0.297361521 -1.205617689  1.070891663
```

```
 [71] -0.686640154  0.617385075 -0.626533459  0.164793000 -0.533990925
 [76] -1.212021336  0.187127581 -2.201308449 -0.120977051  0.291965233
 [81]  1.205023626 -0.701046187 -1.196396786 -0.542822385  0.149667506
 [86]  1.090811850  1.626486885 -0.880739549 -0.804580498 -2.591347290
 [91] -0.229750150  0.410049189  0.754869422 -0.282827877  0.331979313
 [96]  1.034427787  0.116049658 -2.326052584  1.068990597  0.118103577
```

We can combine objects, variables and functions to do more complex stuff in R, here's how we get the mean of 100 random numbers.

```
numbers <- rnorm(100)
mean(numbers)
```

```
[1] 0.04819538
```

Here we created a vector object with `rnorm(100)` and assigned it to the variable `numbers`. We than used the `mean()` function, passing it the variable `numbers`. The `mean()` function returned the mean of the hundred random numbers.

Bracket notation in this document

I'm going to use the following descriptions for the symbols `()`, `[]` and `{}`:

`()` are brackets, `[]` are square brackets `{}` are curly brackets

Quiz

1. Create two variables, `a` and `b`: Add them. What happens if we change a and then re-add a and b?
2. We can also assign `a + b` to a new variable, `c`. How would you do this?
3. Try some R functions: `round()`, `c()`, `range()`, `plot()` hint: Get help on a function by typing `?function_name` e.g `?c()`. Use the `mean()` function to calculate the average age of everyone in your house (Invent a housemate if you have to).

Acknowledgements