GETTING TO GRIPS WITH THE TIDYVERSE

jumping rivers

JUMPINGRIVERS.COM

"THE TIDYVERSE IS AN OPINIONATED COLLECTION OF R PACKAGES DESIGNED FOR DATA SCIENCE. ALL PACKAGES SHARE AN UNDERLYING DESIGN PHILOSOPHY, GRAMMAR, AND DATA STRUCTURES."

THE TIDYVERSE WEBSITE.

"YOU HAD ONE JOB TO DO."

OCEAN'S ELEVEN.

Contents

1

*Introduction to the **tidyverse***

The tidyverse is an umbrella of R packages, with each package designed to give strength to a specific part of the data science workflow given in figure 1.1. With base R and many of the packages outside of the tidyverse, functions quite often have more than one use. Each function within the tidyverse is designed to do only one task and do it efficiently. It aims to make more involved manipulation easier by giving a collection of functions with consistent syntax where each function is aimed at doing one small task very well. Many people call this *Modern R*.

Many of these packages were developed by RStudio chief scientist Hadley Wickham[1], but are now being expanded at the hands of numerous other developers. The packages are not part of base R, and so must be installed separately.

There are numerous packages in the tidyverse. Essentially any R package that follows *tidy principals*. However, in saying that, there are core tidyverse packages[2].

[1] Hadley Wickham is the co-author of a fantastic book, R for Data Science, which explores how to wrangle, visualise, and explore data using the **tidyverse**. I fully recommend reading it.

[2] Of these we shall concentrate on **tibble**, **readr**, **tidyr**, **dplyr**, **lubridate** and **ggplot2**

What is "tidy data"?

Speak to anyone who works with data anywhere and they will tell you a large majority of their time is spent preparing the data for analysis, a.k.a. "*tidying*" the data. A decade or two ago the term "tidy data" had a different meaning between industries and even sectors within industries. This, in turn, made it difficult for workers of differing professions to act coherently. The principles of tidy data provide a standard way to organise data. This promotes cohesion between data science projects and simplifies the development of data science tools. Tidy data has the following characteristics:

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a table.

See `vignette("tidy-data")` for further details.

Components of the tidyverse

tidyverse: import

readr: This package[3] provides fantastic functionality for reading and

[3] **readr** contains drop in function for reading CSV files, e.g. `read_csv()`

writing rectangular data structures such as comma and tab separated value files.

readxl: One of the most common file types in industry and data science today is the Excel file[4]. **readxl** supplies smart ways to read and write Microsoft Excel sheets.

[4] As well as reading Excel files, we can select particular rows and columns.

haven: With **haven** you yield a set of powerful tools for reading and writing data from other programming languages such as SAS, SPSS and Stata.

tidyverse: tidy

Rather depressingly, this can form a large part of any analysis.

tibble: Tibbles are a modern take on data frames. Just as all base R uses data frames, all packages and functions within the **tidyverse** use and return tibbles. Hence knowledge about tibbles is essential for properly understanding the rest of the **tidyverse**.

tidyr: An essential part of data science is preparing the data for analysis, by making it *tidy*. In this chapter we will define what tidy data is and how **tidyr** makes tidying your data easy.

tidyverse: transform

dplyr: provides a consistent set of data manipulation functions that each do one job well. Here we will learn about a new operator, the piping operator, which makes combining functions from **dplyr** and all other **tidyverse** packages effortless.

forcats: Factors can be useful when dealing with categorical data[5]. **forcats** provides a set of tools that aim to solve common problems when dealing with factors in base R.

[5] A factor is something like `small`, `medium` and `large`

purrr: This package supplies a new type-stable set of tools for working with functions, vectors and lists. The aim is to replace for loops with code that is far more logical and explicit.

string: Character strings are a huge part of tidying and manipulating data in R[6]. The **stringr** provides a cohesive group of functions

[6] String manipulation is hard, as strings are so variable. This takes (some) of the pain away.

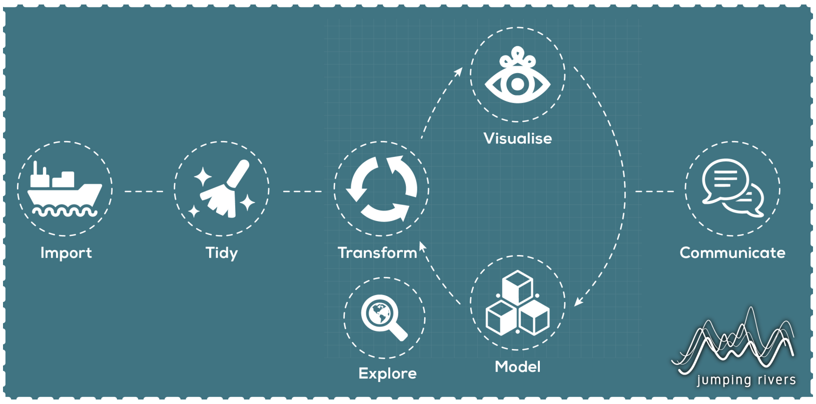

Figure 1.1: The tidyverse workflow.

designed to make manipulating strings simpler.

lubridate: The **lubridate** package makes working with dates and times easier. Although base R deal with dates/times, it is a bit tricky.

tidyverse: visualise

ggplot2 is a plotting system based on the grammar of graphics. It takes the good parts of base R graphics and throws away the old. With **ggplot2**, much of the hassle of creating graphics in base R is removed.

*The **tidyverse** R package*

The **tidyverse** R package is a convenient umbrella package that preloads some standard tidyverse packages. We can install the package in the usual way via

```
install.packages("tidyverse")
```

We can then load the functionality of the eight *core* tidyverse packages into our R session using

```
library("tidyverse")
#> -- Attaching packages ---- tidyverse 1.2.1 --
#> ✓ ggplot2 2.2.1.9000     ✓ purrr   0.2.4
#> ✓ tibble  1.4.2          ✓ dplyr   0.7.4
#> ✓ tidyr   0.8.0          ✓ stringr 1.3.0
#> ✓ readr   1.1.1          ✓ forcats 0.3.0
#> -- Conflicts ------- tidyverse_conflicts() --
#> x dplyr::filter() masks stats::filter()
#> x dplyr::lag()    masks stats::lag()
```

Other packages within the tidyverse that aren't part of the core tidyverse will have to be loaded with their own `library()` command. It is **recommended** that you only use `library("tidyverse")` for interactive data analysis.

In these notes and practicals, we'll be explicit with loading individual **tidyverse** packages to emphasis how the pieces fit together.

tidyverse: Conflicts

When running `library("tidyverse")` you may have noticed this message[7]

```
#> -- Conflicts ------- tidyverse_conflicts() --
#> x dplyr::filter() masks stats::filter()
#> x dplyr::lag()    masks stats::lag()
```

When you load a package, functions in the package that share names with any other functions that have already been loaded in, will replace those functions as the default function when you call their name. Here, that is the **dplyr** functions `filter()` and `lag()` have now overwritten

[7] With hindsight, this conflict is a bad design choice. But hindsight is a wonderful thing!

the base R functions `filter()` and `lag()`. To access these we'd have to use `::` to grab them directly from the package, i.e.[8]

```
stats::filter()
stats::lag()
```

You can run the `tidyverse_conflicts()` function anytime to determine if conflicts arise.

[8] Technically the `::` operator allows us to directly access an exported function without loading the package.

2

*Data frames 2.0 with **tibbles***

What is a data frame?

A data frame is a core R data structure that we use for storing and manipulating data. It is so good, that it was copied by Python in the Pandas library. The easiest way of thinking about a data frame is as a sheet in an excel spreadsheet. However, data frames were created over twenty years ago and in the intervening years, a number of bad design decisions have come to light[1]. Unfortunately, if we just *fixed* data frames, we would break a lot of existing code.

[1] We'll see some examples below.

What is a tibble?

All packages in the **tidyverse** use tibbles. They are a modern take on data frames and are obtained from the **tibble** package.

The package is also loaded via library('tidyverse')

```
library("tibble")
```

Well, what's the difference then? The **tibble** documentation tells us

> They keep the features that have stood the test of time, and drop the features that used to be convenient but are now frustrating.

Typically[2], we would create a tibble by reading data in, either from a CSV, or Excel, or even a SQL database. So in this section, we'll just load tibbles from our package, in particular we'll use the following data sets

[2] We can also create tibbles by hand using the `tibble()` function or coerce a data frame via `as_tibble()`, e.g. `as_tibble("iris")`

```
data(GoT, package = "jrTidyverse")
data(GoT_df, package = "jrTidyverse")
```

The data set `GoT` contains audience numbers for the episodes of Games of Thrones as a tibble. It has three columns, `Season`, `Episode`, `Audience`. The `GoT_df` object contains exactly the same data, but stored as a data frame.

Why tibbles?

You might be wondering "OK, yes this is easy, but it's only as easy as a standard data frame in R, what are the benefits?". Tibbles have three main advantages over data frames: printing, recycling and sub-setting.

Printing

The most obvious benefit so far is the tidy print method. When you print a tibble, only the first 10 rows[3] and as many columns that will fit on the screen are printed. For example,

```
GoT
#> # A tibble: 60 x 3
#>   Season Episode Audience
#>   <chr>  <fct>      <dbl>
#> 1 1      1           2.22
#> 2 2      1           3.86
#> 3 3      1           4.37
#> 4 4      1           6.64
#> # ... with 56 more rows
```

Above each column is also an abbreviated description of the column type. In this case, we have a character, factor and double (a numeric value that allows decimal places).

We also have more flexibility with column names[4]

```
tibble(`Episode number` = 1)
#> # A tibble: 1 x 1
#>   `Episode number`
#>              <dbl>
#> 1             1.00
data.frame(`Episode number` = 1)
#>   Episode.number
#> 1              1
```

[3] It is possible to change the number of rows shown in the print method using `options()`. That is, `options(tibble.print_max = 10, tibble.print_min = 5)` would print so that if there is more than 10 rows, print only 5.

[4] Just because you can, doesn't mean you should!

Subsetting

Use of tibbles induces stricter rules for sub-setting. Using tibbles, sub-setting with single square brackets, `[ ]`, will *always* returns a tibble[5].

```
GoT[, 1]
#> # A tibble: 60 x 1
#>   Season
#>   <chr>
#> 1 1
#> 2 2
#> 3 3
#> 4 4
#> # ... with 56 more rows
```

Whereas if we removed the "tibbleness" and sub-setted

```
GoT_df[, 1]
```

```
#> [1] "1" "2" "3" "4" "5" "6"
```

we get a vector when we choose one column. But a data frame when we choose more than one column

[5] With data frames, sub-setting with single brackets, `[ ]`, sometimes returned a data frame and sometimes returned a vector.

```
GoT_df[,1:2]
```

```
#>   Season Episode
#> 1      1       1
#> 2      2       1
#> 3      3       1
#> 4      4       1
#> 5      5       1
#> 6      6       1
```

This seemingly minor change has actually large implications when interacting with packages outside the tidyverse. Since inside a package, the author may be relying on the conversion to a vector.

Another change, is extract columns. Something that isn't widely known, is that when extracting columns from a data frame, the `$` operator allows partial matching[6]. For example,

```
GoT_df$Sea
#>  [1] "1" "2" "3" "4" "5" "6" "1" "2" "3" "4"
#> [11] "5" "6" "1" "2" "3" "4" "5" "6" "1" "2"
#> [21] "3" "4" "5" "6" "1" "2" "3" "4" "5" "6"
#> [31] "1" "2" "3" "4" "5" "6" "1" "2" "3" "4"
#> [41] "5" "6" "1" "2" "3" "4" "5" "6" "1" "2"
#> [51] "3" "4" "5" "6" "1" "2" "3" "4" "5" "6"
```

Tibbles do **not** allow partial matching of column names and return a warning

```
GoT$Sea
#> Warning: Unknown or uninitialised column:
#> 'Sea'.
#> NULL
```

In my experience, this change is unlikely to break previous code[7] since most people would avoid partial matching.

[6] Now, this seems like a bad design choice. But R (and it's forerunner, S) were originally designed for statisticians to interact with relatively small data sets.

[7] Actually, it has broken my previous code, by highlighting bugs.

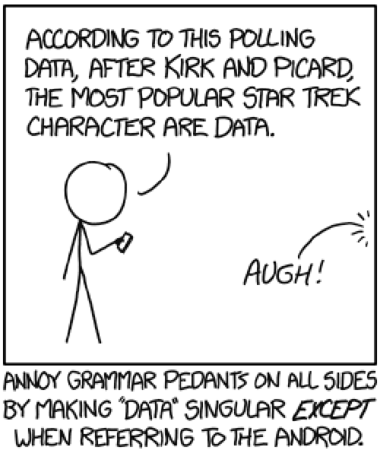

Figure 2.1: https://xkcd.com/1429/

3

Graphics with ***ggplot2***

Plot building

ggplot2 is a bit different from other graphics packages. It roughly follows the *philosophy* of Wilkinson, 1999.[1] Essentially, we think about plots as layers. By thinking of graphics in terms of layers it is easier for the user to iteratively add new components and for a developer to add new functionality.

[1] L Wilkinson. *The Grammar of Graphics*. Springer, 1st edition, 1999.

The package is loaded in the usual way via

```
library("ggplot2")
```

The basic plot object

To create an initial ggplot object, we use the `ggplot()` function. This function has two arguments:

- **data**: this must be a data frame (or tibble).
- an aesthetic **mapping**: this tells ggplot2 how to map data to the graphical elements.

These arguments set up the defaults for the various layers that are added to the plot and can be empty. For each plot layer, these arguments can be overwritten. The `data` argument is straightforward - it is a data frame[2] The `mapping` argument creates default aesthetic attributes. For example, the `GoT` dataset

[2] **ggplot2** is very strict regarding the `data` argument. It doesn't accept matrices or vectors, only data frames/tibbles.

```
data(GoT, package = "jrTidyverse")
```

is a tibble where each row is a particular Games of Thrones Episode. So we could *map* the `Season` and `Audience` columns to the `x` and `y` coordinates.

```
g = ggplot(data = GoT,
           mapping = aes(x = Season, y = Audience))
```

or equivalently[3],

[3] Unlike base graphics, we can store graphs as objects. In this case, the object `g`.

```
g = ggplot(GoT, aes(Season, Audience))
```

Running the above command generates a blank canvas. Now we'll look at adding layers.

The geom_* functions

The `geom_` functions are used to perform the actual rendering in a plot. For example, we have already seen that a line geom will create a line plot and a point geom creates a scatter plot. Each geom has a list of aesthetics that it expects. However, some geoms have unique elements. The error-bar geom requires arguments `ymax` and `ymin`. For a full list, see

`http://ggplot2.tidyverse.org/reference/`

Standard aesthetics are x, y, colour and size.

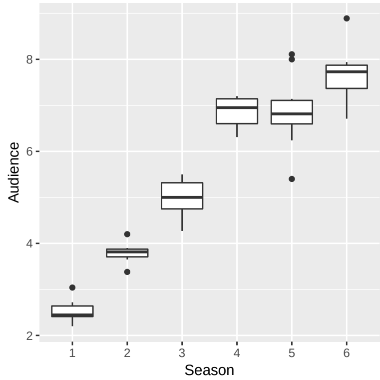

Figure 3.1: Audience numbers per season.

Example: combining geoms

Let's look at the `GoT` data set in more detail. We begin by creating a base ggplot object

```
g = ggplot(GoT, aes(Season, Audience))
```

Remember, the above piece of code just generates a blank canvas. To make the plot more interesting, we add *layers*. To start with, we'll add a boxplot layer

```
(g1 = g + geom_boxplot())
```

This produces figure 3.1. Notice that the default axis labels are the column headings of the associated data frame. We can have a more colourful boxplot using the `fill` aesthetic

```
## Try colour and group instead of fill
g2 = g + geom_boxplot(aes(fill = Season))
```

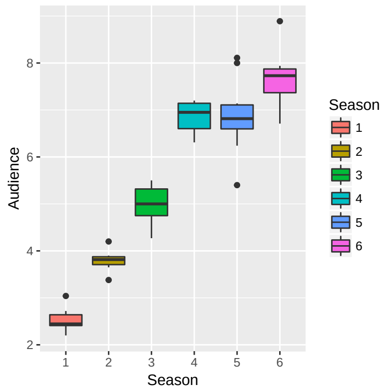

Figure 3.2: More colourful boxplots.

While not that useful in this situation, colour can be useful in boxplots to differentiate different experiment types (say).

Standard Plots

There are a few other standard `geom`'s that are particular useful:

- `geom_point()`: a scatter plot - see figure 3.3.
- `geom_bar()`: produces a standard barplot that counts the `x` values.
- `geom_histogram()`: produces a standard histogram.
- `geom_line()`: a line plot.
- `geom_text()`: adds labels to specified points. This has an additional (required) aesthetic: `label`. Other useful aesthetics, such as `hjust` and `vjust` control the horizontal and vertical position. The `angle` aesthetic controls the text angle.
- `geom_raster()`: a heat-map

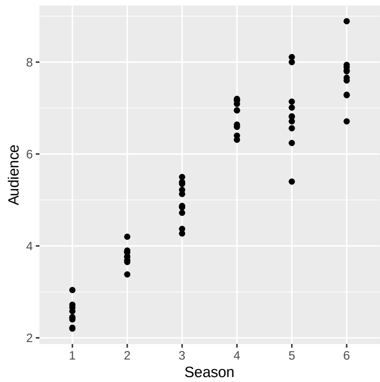

Figure 3.3: Scatter plot of Season vs Audience.

What is a geom_*?

A geom_* is a single layer that comprises of (at least) four elements:

- an aesthetic and data mapping;
- a statistical transformation (**stat**);

- a geometric object (**geom**);
- and a position adjustment, i.e. how should objects that overlap be handled.

When we use the function

```
geom_point(aes(colour = Episode))
```

this is actually a short cut for:

```
layer(
    data = GoT, #inherited
    mapping = aes(colour = Episode), #x,y are inherited
    stat = "identity", #Multiple by 1
    geom = "point",
    position = "identity", # If they overlap, who cares
    params = list(na.rm = FALSE)
)
```

In practice, we **never** use the layer function.

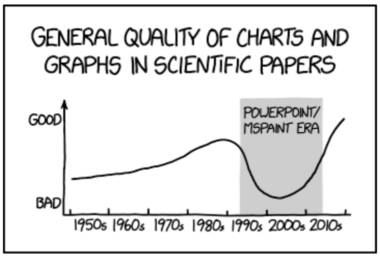

Figure 3.4: https://xkcd.com/1945/

4
Data manipulation: ***dplyr***

What is ***dplyr****?*

dplyr is a fantastic package for manipulating data frame structures. The package focuses on easy to read and easy to use functions, the motivation being that a user spends more time worrying about data than they do about writing code. As usual, we'll load the package via

```
library("dplyr")
```

In this chapter, we'll use a more complicated data set, from the IMDB data base

```
data(movies, package = "ggplot2movies")
```

The internet movie database is a website devoted to collecting movie data supplied by studios and fans.[1] It claims to be the biggest movie database on the web and is run by amazon. More information about IMDB can be found online at

[1] `http://imdb.com/`

`http://imdb.com/help/show_leaf?about`

Information about the data collection process is detailed at

IMDB makes their raw data available at http://uk.imdb.com/interfaces/.

`http://imdb.com/help/show_leaf?infosource`

Movies were selected for inclusion if they had a known length and had been rated by at least one imdb user. This gives 58788 films, where each film has twenty four associated variables. The first few rows are given in Table 4.1.

The dataset contains the following fields:

- Title. Title of the movie.
- Year. Year of release.
- Budget. Total budget in US dollars. If the budget isn't known, then it is stored as a missing value, `NA`.[2]
- Length. Length in minutes.
- Rating. Average IMDB user rating.
- Votes. Number of IMDB users who rated this movie.
- r1: Gives the percentage (to the nearest 10%) of users who rated this movie a 1.
- r2 – r10: Similar to r1.
- mpaa. The mpaa rating: PG, PG-13, R, NC-17.

[2] In R `NA` is used to represent missing data. More on this in chapter 2.

				Voting statistics					Movie genre							
Title	Year	Length	Budget	Rating	Votes	r1	...	r10	mpaa	Action	Animation	Comedy	Drama	Documentary	Romance	Short
A.k.a. Cassius	1970	85	-1	5.7	43	4.5	...	14.5	PG	0	0	0	0	1	0	0
AKA	2002	123	-1	6.0	335	24.5	...	1.5	R	0	0	0	1	0	0	0
Alien Vs. Pred	2004	102	45000000	5.4	14651	4.5	...	4.5	PG-13	1	0	0	0	0	0	0
Abandon	2002	99	25000000	4.7	2364	4.5	...	4.5	PG-13	0	0	0	1	0	0	0
Abendland	1999	146	-1	5.0	46	14.5	...	24.5	R	0	0	0	0	0	0	0
Aberration	1997	93	-1	4.8	149	14.5	...	4.5	R	0	0	0	0	0	0	0
Abilene	1999	104	-1	4.9	42	0.0	...	24.5	PG	0	0	0	1	0	0	0
Ablaze	2001	97	-1	3.6	98	24.5	...	14.5	R	1	0	0	1	0	0	0
Abominable Dr	1971	94	-1	6.7	1547	4.5	...	14.5	PG-13	0	0	0	0	0	0	0
About Adam	2000	105	-1	6.4	1303	4.5	...	4.5	R	0	0	1	0	0	1	0

Table 4.1: The first ten rows of the movie data set.

- Action, Animation, Comedy, Drama, Documentary, Romance, Short. Binary variables representing if movie was classified as belonging to that genre. A movie can belong to more one genre. See for example the film `Ablaze` in Table 4.1. (END)

Row partitioning: `filter()`

The **dplyr** package has a function `filter()` which we can use to perform data partitioning on each row, i.e we can select films that match certain criteria. For instance, if we wanted to filter the movies data set by only the movies with a rating greater than 9 would do so like this

```
sub_movies = filter(movies, rating > 9)
```

or if we wanted movies released in 1994 with a length less than the median length[3]

```
filter(movies, length < median(length) & year == 1994)
#> # A tibble: 425 x 24
#>   title      year length budget rating votes
#>   <chr>     <int>  <int>  <int>  <dbl> <int>
#> 1 06         1994     87     NA   6.40   238
#> 2 100 Year~  1994      9     NA   9.20    91
#> 3 89 mm od~  1994     12     NA   7.80    12
#> 4 Aapo       1994     55     NA   5.00    26
#> # ... with 421 more rows, and 18 more
#> #   variables: r1 <dbl>, r2 <dbl>, r3 <dbl>,
#> #   r4 <dbl>, r5 <dbl>, r6 <dbl>, r7 <dbl>,
#> #   r8 <dbl>, r9 <dbl>, r10 <dbl>,
#> #   mpaa <chr>, Action <int>,
#> #   Animation <int>, Comedy <int>,
#> #   Drama <int>, Documentary <int>,
#> #   Romance <int>, Short <int>
```

[3] Remember, `x == y` asks the question, “does x equal y?”. Whereas, “x = y”, states that “x equals y”.

The `filter()` function and most **dplyr** functions use non-standard evaluation (NSE). This is a catch-all term that means they don’t follow the usual R rules of evaluation. Instead, they capture the expression that you typed and evaluate it in a custom way.

Column Partitioning: `select()`

We can use the `select()` function to easily partition our data via the column names. At it's most basic, `select()` can grab one or more named columns[4]

```
select(movies, year, title)
#> # A tibble: 58,788 x 2
#>    year title
#>   <int> <chr>
#> 1  1971 $
#> 2  1939 $1000 a Touchdown
#> 3  1941 $21 a Day Once a Month
#> 4  1996 $40,000
#> # ... with 5.878e+04 more rows
```

[4] It can also remove columns, e.g. `select(movies, -year)` would remove the column `year`.

What if we wanted to grab all of the binary variables for the movie genre, and still keep the name? The sequence generation operator, `:`, in **dplyr** functions grabs all the columns between the two specified columns

This is a similar idea to using ':' with integers, i.e. '1:3' gives 1, 2, 3.

```
select(movies, title, Action:Drama)
#> # A tibble: 58,788 x 5
#>   title          Action Animation Comedy Drama
#>   <chr>           <int>     <int>  <int> <int>
#> 1 $                   0         0      1     1
#> 2 $1000 a Tou~        0         0      1     0
#> 3 $21 a Day O~        0         1      0     0
#> 4 $40,000             0         0      1     0
#> # ... with 5.878e+04 more rows
```

`select()` can also be combined with other functions to make selecting columns easier. For instance, to grab all the columns starting with `d`[5]

[5] The `starts_width()` function has an argument `ignore.case` whose default is `TRUE`.

```
select(movies, starts_with("d"))
#> # A tibble: 58,788 x 2
#>   Drama Documentary
#>   <int>       <int>
#> 1     1           0
#> 2     0           0
#> 3     0           0
#> 4     0           0
#> # ... with 5.878e+04 more rows
```

What about all the columns that start with `d` or end with `y`?

We can also specify columns by number range using `num_range(prefix = "r", range = 1:5)`.

```
select(movies, starts_with("d"), ends_with("y"))
#> # A tibble: 58,788 x 3
#>   Drama Documentary Comedy
#>   <int>       <int>  <int>
#> 1     1           0      1
#> 2     0           0      1
#> 3     0           0      0
```

```
#> 4     0            0      1
#> # ... with 5.878e+04 more rows
```

The `summarise()` function

Very often we want to apply summary statistics to variables in our data set. In order to apply a function which summarises one or more variables in our data to a single values (such as mean) we would use the `summarise()` function[6]. For example

[6] We'll come back to this function in more detail when we cover the pipe operator.

```
## Multiple summaries & rename
summarise(movies,
          avg_length = mean(length),
          med_rating = median(rating))
#> # A tibble: 1 x 2
#>   avg_length med_rating
#>        <dbl>      <dbl>
#> 1       82.3       6.10
```

Variants of `summarise()`

The `summarise()` function has three sister functions useful for summarising more than one column in a quicker fashion. `summarise_all()` applies the summary function to all columns

```
sub_movies = select(movies, r1:r10)
summarise_all(sub_movies, mean)
#> # A tibble: 1 x 10
#>      r1    r2    r3    r4    r5    r6    r7
#>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1  7.01  4.02  4.72  6.37  9.80  13.0  15.5
#> # ... with 3 more variables: r8 <dbl>,
#> #   r9 <dbl>, r10 <dbl>
```

`summarise_at()` can be used to apply a function to a specific set of columns. This uses the same variable name-based helpers as `select()` to select variables and summarise the in one quick line of code. For instance, the above two lines of code would become

```
summarise_at(movies, vars(r1:r10), mean)
#> # A tibble: 1 x 10
#>      r1    r2    r3    r4    r5    r6    r7
#>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1  7.01  4.02  4.72  6.37  9.80  13.0  15.5
#> # ... with 3 more variables: r8 <dbl>,
#> #   r9 <dbl>, r10 <dbl>
```

Note, we have to wrap the variable functions in `vars()` when using the sequence generation operator `:`. Alternatively, if the columns to be summarised are not consecutive, we can pass a vector of variable names.

```
summarise_at(movies, c("length","rating"), mean)
#> # A tibble: 1 x 2
#>   length rating
#>    <dbl>  <dbl>
#> 1   82.3   5.93
```

Similarly we could use this function to apply many functions to many variables using `funs()` to wrap the collection of function names.

```
summarise_at(movies, vars(r1:r3), funs(mean,sd))
#> # A tibble: 1 x 6
#>   r1_mean r2_mean r3_mean r1_sd r2_sd r3_sd
#>     <dbl>   <dbl>   <dbl> <dbl> <dbl> <dbl>
#> 1    7.01    4.02    4.72  10.9  5.96  6.45
```

`summarise_if()` will apply a summary function to all columns matching a predicate[7]. For example we could use this to calculate the mean for all numeric columns of the movies data frame. In the example below the function `is.numeric()` returns `TRUE` whenever a variable contains numeric values. `mean()` is the summary function to be passed and the additional argument `na.rm = TRUE` is passed through to the summary function. In this case we are removing missing values before calculating the mean.[8]

[7] A function or test which returns a logical value.

[8] See `?mean` for more info on the `na.rm` argument.

```
summarise_if(movies, is.numeric, mean, na.rm = TRUE)
#> # A tibble: 1 x 22
#>    year length   budget rating votes    r1
#>   <dbl>  <dbl>    <dbl>  <dbl> <dbl> <dbl>
#> 1  1976   82.3 13412513   5.93   632  7.01
#> # ... with 16 more variables: r2 <dbl>,
#> #   r3 <dbl>, r4 <dbl>, r5 <dbl>, r6 <dbl>,
#> #   r7 <dbl>, r8 <dbl>, r9 <dbl>, r10 <dbl>,
#> #   Action <dbl>, Animation <dbl>,
#> #   Comedy <dbl>, Drama <dbl>,
#> #   Documentary <dbl>, Romance <dbl>,
#> #   Short <dbl>
```

The Pipe operator

All **dplyr** functions follow a consistent syntax

The pipe operator is actually part of the **magrittr** package. **dplyr** just imports it.

```
function_name(data set, variable conditions)
```

where the input and output are always tibbles.[9] Apart from making the structure consistent this has another useful consequence. That is we can make use of an operator called the pipe operator `%>%` which feeds forward a result as the first argument of the function.

[9] Strictly they are data frame structures, but most often used with other tidyverse packages such as `tibble`.

To try to make clear how this operates, the following two lines of code perform the exact same calculation

```
# pass the value 1:5 on the left as the first argument to mean
1:5 %>% mean(na.rm = TRUE)
#> [1] 3
```

```
# explicitly pass 1:5 into the function
mean(1:5, na.rm = TRUE)
#> [1] 3
```

The motivation here is legibility when chaining together multiple commands into a data manipulation “pipeline”. For example we could first `filter()` then `summarise()`

```
movies %>%
  filter(rating > 9) %>%
  summarise(mean_length = mean(length))
#> # A tibble: 1 x 1
#>   mean_length
#>         <dbl>
#> 1        64.2
```

Because `%>%` passes forward the result as the first argument of the next function we don’t need to store intermediate results, or refer to the data frame after the beginning. The idea is that this is “easy to read” as “take the movies data, filter it this way then summarise it that way”.

If I wanted the mean and standard deviation of movie ratings for all PG rated films I could

```
movies %>%
  filter(mpaa == "PG") %>%
  summarise(mean = mean(rating), sd = sd(rating))
#> # A tibble: 1 x 2
#>    mean    sd
#>   <dbl> <dbl>
#> 1  5.61  1.47
```

group_by()

Given the desire to calculate mean and standard deviation for one subset of films based on certifation, it stands to reason that I often want the same summaries for all similar subsets. In this example we also want means and standard deviations of all the other film certifications. We could run similar code for “PG”, “PG-13” …, but this quickly becomes tedious[10]. Instead we use `group_by()`

[10] A general rule of thumb, if you copy and paste three times, you’re doing it wrong.

```
movies %>%
  group_by(mpaa)
```

Notice that `group_by()` by itself doesn’t change the appearance of our data, all that it has done is create a categorical structure behind the scenes. We can see the fruits of our labour when applying the `summarise()` function to the grouped data.

```
movies %>%
  group_by(mpaa) %>%
  summarise(mean_length = mean(length))
#> # A tibble: 5 x 2
```

```
#>   mpaa  mean_length
#>   <chr>       <dbl>
#> 1 ""           80.6
#> 2 NC-17       110
#> 3 PG           97.4
#> 4 PG-13       105
#> # ... with 1 more row
```

Everything we do after the `group_by` statement (using **dplyr** functions) will happen to each unique group according to that variable or combination of variables.[11]

[11] This is not limited to just `summarise()`, the same holds true for the other **dplyr** functions. For example grouped filtering is often desirable.

This chaining together of commands, even with the few functions that we have seen so far, can make it much easier to do fairly advanced data manipulation. For example let's say that I want to know about films that are a reasonable length, say less than 2.5 hours, for which we know information on the budgets, and I want to know how the average ratings, lengths and budgets all compare for Action and non Action films among the four film certification ratings.

We can take the steps and put them into some order:

The `is.na()` function determines if a variable is a `NA`, i.e. a missing value.

1. keep only films for where we know about the budget and it fits length criteria (`filter()`),
2. group our data into the relevant subsets (`group_by()`),
3. calculate summary statistics (`summarise()`).

```
movies %>%
  filter(length < 150 & !is.na(budget)) %>%
  group_by(mpaa, Action) %>%
  summarise(m_length = mean(length),
            m_rating = mean(rating),
            m_budget = mean(budget))
#> # A tibble: 9 x 5
#> # Groups:   mpaa [?]
#>   mpaa  Action m_length m_rating m_budget
#>   <chr>  <int>    <dbl>    <dbl>    <dbl>
#> 1 ""         0     85.1     6.27  4335842
#> 2 ""         1     96.7     5.88 12204388
#> 3 NC-17      0     99.7     5.95  7897833
#> 4 PG         0     98.2     5.80 31858683
#> # ... with 5 more rows
```

This is something that would be very laborious to do using only the base R functions.

Merging and Joining data

In many real situations, data isn't stored in a single place. Typically when this is the case we want to join this data together in a smart way, perhaps by matching unique IDs of similar. **dplyr** currently supports a number of join functions which merge this data in slightly different ways.

To demonstrate this we'll first load two example data sets

```
data("stock_price", package = "jrTidyverse")
data("stock_change", package = "jrTidyverse")
stock_price
#> # A tibble: 3 x 3
#>   Name         Price   Est
#>   <chr>        <dbl> <dbl>
#> 1 Bitcoin       6000  2000
#> 2 Rolls Royce    827  1904
#> 3 Sainsbury's    242  1869
stock_change
#> # A tibble: 3 x 2
#>   Name         Change
#>   <chr>         <dbl>
#> 1 Bitcoin      -50.0
#> 2 Rolls Royce    1.00
#> 3 Aviva          2.00
```

- `left_join(x,y)` - This will join two datasets, x and y, by keeping all the rows in x, all the columns from x and y then merging the data where necessary. This is best demonstrated with an example

  ```
  left_join(x = stock_price, y = stock_change, by = "Name")
  ```

  ```
  #> # A tibble: 3 x 4
  #>   Name         Price   Est Change
  #>   <chr>        <dbl> <dbl>  <dbl>
  #> 1 Bitcoin       6000  2000 -50.0
  #> 2 Rolls Royce    827  1904   1.00
  #> 3 Sainsbury's    242  1869  NA
  ```

  Essentially we are keeping the information we have in x. Then, if any of the values of the variable `Name` match in y and x, add the new information about those variables found in y, to x. Here, that is adding the variable `Change` to the `Name`'s Bitcoin and Rolls Royce.

- `right_join(x,y)` - This will join two datasets, x and y, by keeping all the rows in y and all the columns from x and y then merging the data where necessary.[12]

```
right_join(x = stock_price, y = stock_change,
           by = "Name")
#> # A tibble: 3 x 4
#>   Name         Price   Est Change
#>   <chr>        <dbl> <dbl>  <dbl>
#> 1 Bitcoin       6000  2000 -50.0
#> 2 Rolls Royce    827  1904   1.00
#> 3 Aviva           NA    NA   2.00
```

[12] Essentially the reverse of `left_join()`. `left_join(x,y) == right_join(y,x)`

- `inner_join()`[13] - This will join two datasets, x and y, by keeping only the rows that exist in both x and y in the joining variable, keeping all columns from both

```
inner_join(x = stock_price, y = stock_change,
           by = "Name")

#> # A tibble: 2 x 4
#>   Name         Price   Est Change
#>   <chr>        <dbl> <dbl>  <dbl>
#> 1 Bitcoin       6000  2000 -50.0
#> 2 Rolls Royce    827  1904   1.00
```

Here, because only two of the `Name` variable are in both x and y, we have only kept those two rows, but returns all columns in both data sets. If there is a more than one match per row of x, `inner_join()` will return all combinations of the matches. [14]

- `full_join()` - This will join two data sets, x and y, by returning all rows and columns from both and merging the data where necessary.

```
full_join(x = stock_price, y = stock_change,
          by = "Name")

#> # A tibble: 4 x 4
#>   Name         Price   Est Change
#>   <chr>        <dbl> <dbl>  <dbl>
#> 1 Bitcoin       6000  2000 -50.0
#> 2 Rolls Royce    827  1904   1.00
#> 3 Sainsbury's    242  1869  NA
#> 4 Aviva           NA    NA   2.00
```

Whilst it can be a bit tricky getting to grips with thinking about data manipulation using **dplyr**, once you get the hang of it, it's definitely worth the effort.

There are numerous **dplyr** functions[15] many of which we do not have time to explore here. These functions tend to all look and work in the same sort of way. See table 4.2 for a short (certainly not exhaustive) list.[16]

[13] The opposite of `inner_join()` is `anti_join()`, where two data sets, x and y, are joined by by removing rows from x where x and y have matching values in the variable specified

[14] `semi_join()` is a version of `inner_join()` that will not duplicate rows of x.

[15] 231 to be exact.

[16] Another bonus of getting used to **dplyr** and tibbles is that it allows you to interact with other types of data. For example you can connect directly to SQL data bases and retain all of the nice R syntax and data manipulation, but act on your relational data.

Function	Purpose
`filter()`	Extracting subsets of data
`arrange()`	Re–order your data by sorting on (a) given column(s)
`select()`	Choose specific columns from the data
`rename()`	Rename specific columns from the data
`distinct()`	Keep only unique rows of the data
`mutate()`	Add a new column to the data, can be formed from calculation
`sample_n()`	Take a random sample from your data
`count()`	Tally the number of observations in each group of data

Table 4.2: Useful dplyr functions

5

Dates/times with ***lubridate***

> Lubridate makes it easier to do the things R does with date-times and possible to do the things R does not.
>
> `http://lubridate.tidyverse.org/`

Time is a measurement system. When we think of a date, say January 1st 2015, we know that it is 2015 years from 0 AD. R applies this principle in representing date and time objects. It stores date-times as either:

- `POSIXct` objects: these represent the (signed) number of seconds since the beginning of 1970 as a numeric vector.
- `POSIXlt` objects: this is a named list of vectors representing `sec` (0 - 61[1]), `min` (0 - 59), `hour` (0 - 23), `mday` (day of month, 1 - 31), `mon` (month, 0 - 11), `year` (years since 1900), `wday` (day of week, 0 - 6), `yday` (day of year, 0 - 365) and `isdst` (flag for “is daylight saving time”).

[1] This facilitates representation of leap seconds.

If your data contains dates then storing them as date-time objects, rather than numbers or strings, has a number of advantages. For example, it allows easy extraction of information, manipulation and arithmetic within a consistent, principled framework.
We load the **lubridate** package in the usual way[2]

[2] Note **lubridate** is not part of the **tidyverse** package but is part of the **tidyverse**.

```
library("lubridate")
```

At it’s most basic we can parse character strings to dates. To parse a date simply identify the order in which the day (d), month (m) and year (y) appear and arrange the letters `d`, `m` and `y` in that order. This gives the name of the function which you can use to parse the date[3]

[3] The functions automatically recognise commonly used separators such as “-”, “/”, “ ” and no separator.

```
ydm("19993001")
#> [1] "1999-01-30"
mdy("01/30/1999")
#> [1] "1999-01-30"
dmy("30 January, 1999")
#> [1] "1999-01-30"
```

If we wanted to parse character strings for times, then we use `h`, `m`, and `s`

```
hms("12:30:45")
#> [1] "12H 30M 45S"
hm("12:30")
#> [1] "12H 30M 0S"
```

You can also parse character strings containing date-times, by combining any of the date functions with any of the time functions with a _

```
ymd_hms("19990130 12:30:45")
#> [1] "1999-01-30 12:30:45 UTC"
ydm_hm("1999, 30 January 12:30")
#> [1] "1999-01-30 12:30:00 UTC"
```

By default, **lubridate** sets the timezone to Universal Coordinated Time Zone (UTC), but we can change this using the `tz` argument. For instance

```
ymd_hms("19990130 12:30:45", tz = "America/Los_Angeles")
#> [1] "1999-01-30 12:30:45 PST"
```

We can grab set components of date-time objects using accessor functions. These are given in table 5.1. For instance

```
world_cup = dmy("01/07/1966")
wday(world_cup)    # weekday
#> [1] 6
yday(world_cup)    # year day
#> [1] 182
```

Table 5.1: Accessor functions for specific date/time components, applied to a POSIXct object d.

Date component	Accessor function
Year	`year(d)`
Month	`month(d)`
Week	`week(d)`
Day of year	`yday(d)`
Day of month	`mday(d)`
Day of week	`wday(d)`
Hour	`hour(d)`
Minute	`minute(d)`
Second	`second(d)`
Time zone	`tz(d)`

We can use this to reassign parts of the date to other values

```
world_cup
#> [1] "1966-07-01"
year(world_cup) = 2018 # Hopefully!!!!
world_cup
#> [1] "2018-07-01"
```

It's true, the English can't get over 1966. This example was constructed independently by my English colleague. He never mentioned 1967 though!

"Hopefully", it depends on where you come from!

The update function will update more than one element at once

```
update(world_cup, year = 2022, month = 06, day = 31)
#> [1] "2022-07-01"
```

It is also possible to grab the current date or the current date-time

```
today()
#> [1] "2018-02-20"
now()
#> [1] "2018-02-20 21:05:23 GMT"
```

The functions `with_tz()` will give us a time as it appears in another timezone, i.e.

```
with_tz(world_cup, tz = "America/Los_Angeles")
#> [1] "2018-07-01"
```

We can use the function `interval()` will create a date-time interval between two dates/times

```
(int = interval(world_cup, today()))
#> [1] 2018-07-01 UTC--2018-02-20 UTC
```

The function `as.period()` will then tell us the length of the interval

```
as.period(int)
#> [1] "-4m -9d 0H 0M 0S"
```

Binning dates

lubridate provides three useful functions for rounding dates

- `round_date(d, unit)`: rounds a date/time object `d` to the nearest whole-numbered value of the specified time `unit`;
- `floor_date(d, unit)`: rounds a date/time object `d` down to the nearest whole-numbered value of the specified time `unit`;
- `ceiling_date(d, unit)`: rounds a date/time object `d` up to the nearest whole-numbered value of the specified time `unit`.

In each of the above functions, `unit` should be one of `"second"`, `"minute"`, `"hour"`, `"day"`, `"week"`, `"month"`, or `"year"`. To illustrate, we start with the date world cup winning date 1st July 1966

```
world_cup
#> [1] "2018-07-01"
```

To get the date of the *nearest* Sunday (which has the numeric value 1), the *previous* Sunday or the *next* Sunday we would use, respectively,

```
# return date of nearest Sunday (label=1)
round_date(world_cup, "week")
#> [1] "2018-07-01"
floor_date(world_cup, "week")   # previous Sunday
#> [1] "2018-07-01"
ceiling_date(world_cup, "week") # next Sunday
#> [1] "2018-07-08"
```

Similarly, to get the date of the *previous* 1st of the month, we would use

```
# return date of previous 1st of month
floor_date(world_cup, "month")
#> [1] "2018-07-01"
```

And to get the date of the *next* 1st of January, we would use

```
# return date of next 1st of January
ceiling_date(world_cup, "year")
#> [1] "2019-01-01"
```

Figure 5.1: https://xkcd.com/1179/

6

*Tidy data & **tidyr***

*The **tidyr** package*

The goal of the **tidyr** package is to get data into the structure laid out above. As usual, we'll begin by loading the package

```
library("tidyr")
```

There are five fundamental functions in tidyr: `gather()`, `spread()`, `seperate()`, `unite()` and `extract()`[1].

[1] `extract()`, used for extracting data values from columns, requires knowledge of regular expressions, which we do not have time to cover. See this link for a reason why: stackoverflow.com/q/201323/203420

`gather()` & `spread()`

`gather()` takes data from a wide format into a long format. Let's start with an example. We're going to take a sample of data from a school, containing the name of the child and their respective scores in English, Maths and French

```
data(School, package = "jrTidyverse")
School
#> # A tibble: 3 x 4
#>   Name      English Maths French
#>   <chr>       <dbl> <dbl>  <dbl>
#> 1 Billie       74.0  58.0   60.0
#> 2 Rosezella    69.0  43.0   70.0
#> 3 Calyn        58.0  92.0   84.0
```

Here we have three variables: name, type of class and score. Sticking to the rules of "tidy data", all 3 should have their own column. However, only name currently has it's own column. We can use `gather()` to do this

```
School_tidy = School %>%
  gather(key = Class, value = Score, English:French)
```

Here, we are gathering the columns from `English` to `French` into a column pairing of `Class` and `Score`, where the current column names will become the first variable given, `Class`, and the values in the current columns will become the second variable given, `Score`. If we wanted to exclude a column, say `Maths`, from the gathering process then we'd use the `-` operator inside `gather()`

```
School %>%
  gather(key = Class, value = Score, English:French, -Maths)
```

```
#> # A tibble: 6 x 4
#>   Name      Maths Class   Score
#>   <chr>     <dbl> <chr>   <dbl>
#> 1 Billie     58.0 English  74.0
#> 2 Billie     58.0 French   60.0
#> 3 Calyn      92.0 English  58.0
#> 4 Calyn      92.0 French   84.0
#> 5 Rosezella  43.0 English  69.0
#> 6 Rosezella  43.0 French   70.0
```

This is useful for larger data sets with lots of variables and comparing one particular aspect of a variable with the rest of the data.

`spread()` is the compliment of gather, effectively making long data wider. It takes a pair of columns and spreads them to multiple columns. Let's revisit the tidied school data, `School_tidy`. To return this into it's original wide format we would do as so

```
School_tidy %>%
  spread(Class, Score)
#> # A tibble: 3 x 4
#>   Name      English French Maths
#>   <chr>       <dbl>  <dbl> <dbl>
#> 1 Billie       74.0   60.0  58.0
#> 2 Calyn        58.0   84.0  92.0
#> 3 Rosezella    69.0   70.0  43.0
```

Here we are spreading the first column, `Class`, in to separate variable with the column `Score` becoming the column values.

separate() & unique()

`separate()` does as it says on the tin and is also a little easier to follow than gather and spread. It takes a single character column and turns it into multiple columns. Let's say that our original School data had an extra column for the predicted grade of the children

```
School = mutate(School, PredGrade = c("C-D", "B-D", "A-B"))
```

To split this column into an upper and lower bound for the predicted grade, we could use `seperate()`

```
(School_sep = separate(School, PredGrade,
                       c("Upper", "Lower"), sep = "-"))
#> # A tibble: 3 x 6
#>   Name      English Maths French Upper Lower
#>   <chr>       <dbl> <dbl>  <dbl> <chr> <chr>
#> 1 Billie       74.0  58.0   60.0 C     D
#> 2 Rosezella    69.0  43.0   70.0 B     D
#> 3 Calyn        58.0  92.0   84.0 A     B
```

Here, we have split the column `PredGrade`, in to 2 further columns, `Lower` and `Upper`, by the separation character `-`. Notice we have to supply our new variable names as a vector of characters.

The complement to `separate()` is `unite()`. If we wanted to return our data to it's original state we would run

```
unite(School_sep, PredGrade,
      c("Upper", "Lower"), sep = "-")
#> # A tibble: 3 x 5
#>   Name      English Maths French PredGrade
#>   <chr>       <dbl> <dbl>  <dbl> <chr>
#> 1 Billie       74.0  58.0   60.0 C-D
#> 2 Rosezella    69.0  43.0   70.0 B-D
#> 3 Calyn        58.0  92.0   84.0 A-B
```

This works in reverse to `seperate()` by combining the two columns given (`c("Upper", "Lower")`) into the new column (`PredGrade`) and separating them by the given character string (`sep = "-"`).

Figure 6.1: https://xkcd.com/1267/

7

Data I/O: ***readr***, ***readxl*** and ***haven***

readr: *Reading directly from CSV files*

The easiest way of getting data in and out of R is to use a comma separated value file (CSV). Software packages like Excel allow you save a *single* sheet as a CSV file. If you go to

`https://www.jumpingrivers.com/data/movie.txt`

you will see a raw CSV file for the movie data set.[1]

The **readr** package contains the function `read_csv()` which reads in CSV files

```
library("readr")
## Could be any path
url = "https://www.jumpingrivers.com/data/movie.txt"
movies = read_csv(url)
#> Parsed with column specification:
#> cols(
#>   .default = col_double(),
#>   Title = col_character(),
#>   Year = col_integer(),
#>   Length = col_integer(),
#>   Votes = col_integer(),
#>   mpaa = col_character(),
#>   Action = col_integer(),
#>   Animation = col_integer(),
#>   Comedy = col_integer(),
#>   Drama = col_integer(),
#>   Documentary = col_integer(),
#>   Romance = col_integer(),
#>   Short = col_integer()
#> )
#> See spec(...) for full column specifications.
```

A few points/tips:

- The `col_names` argument indicates if your table has a header row. Default: `TRUE`.
- We can also read directly from compressed CSV files (.zip, .gz, .bz2, .xz) and urls (`http://`, `https://`, `ftp://`, ftps://).

[1] In order to stop Windows automatically opening the file with Excel, I've used the **.txt** file extension.

- If your data is separated by something other than a comma, then you can use `read_delim()` and specify the separator using the `delim` argument. See the associated help page for a full description.

As this is the **tidyverse**, the `read_csv()` function returns a tibble object. This has several benefits over it's base R alternative `read.csv()`:

1. `read_csv()` is more efficient
2. `read_csv()` does will not convert character variables to factors.[2]
3. The function contains a few other handy arguments, such as `col_types` for explicitly specifying the column types and `skip` to skip the first few lines. See the associated help page `?read_csv` for full details.

[2] Factors are useful, but typically not when you first read in data.

Example: Game of Thrones

Within the **jrTidyverse** package, we have a CSV file containing the Game of Thrones data set. You can get the exact path using

```
library("jrTidyverse")
(fname = get_got_path())
#> [1] "/home/jamie/R/x86_64-pc-linux-gnu-library/3.4/jrTidyverse/datasets/GoT.csv"
```

We can then read in the data using `read_csv()`

```
(got_csv = read_csv(fname))
#> # A tibble: 60 x 3
#>   Season Episode Audience
#>    <int>   <int>    <dbl>
#> 1      1       1     2.22
#> 2      2       1     3.86
#> 3      3       1     4.37
#> 4      4       1     6.64
#> # ... with 56 more rows
```

Notice that by **readr** has interpreted the first two columns as integers. We can change this behaviour using the `col_types` argument[3]

[3] Other types include, `l`: logical, `D`: Date, `t`: time, `i`: integer.

```
# c denotes character, d denotes as double
read_csv(fname, col_types = "ccn")
#> # A tibble: 60 x 3
#>   Season Episode Audience
#>   <chr>  <chr>      <dbl>
#> 1 1      1           2.22
#> 2 2      1           3.86
#> 3 3      1           4.37
#> 4 4      1           6.64
#> # ... with 56 more rows
```

readxl: Reading directly from Excel files

It is possible to read a **.xls** or **.xlsx** file directly, using the `read_excel()`[4] function. This has a few handy arguments:

[4] **readxl** comes with two other functions, `read_xls()` and `read_xlsx()`, to read `.xls` and `.xlsx` files respectively. `read_excel()` reads both types of file.

- `sheet` - specify the sheet by number or name

```
read_excel("dataset.xls", sheet = 2)
read_excel("dataset.xls", sheet = "experiment 2")
```

- `range` - specify range of cells

```
read_excel("dataset.xlsx", range = "A3:D7")
```

  Here, we are reading data contained in the rectangle defined by the top left corner of `A3` and the bottom right corner of `D7`. Just like `read_csv()`, `read_excel()` reads data in as a tibble. There are plenty more arguments, see `?read_excel` for more arguments.

readr: Exporting CSV files

If you have carried out some data analysis using R and you want to export your data to some other piece of software, the easiest way is via a CSV file. If your data is in a data frame or tibble, we just use the `write_csv()`[5] command. For example,

```
write_csv(got_csv, path = "got.csv")
```

[5] Or we could use `write_delim()` and specify the between column separator.

haven: Reading directly from SAS, SPSS & Stata[6]

The **haven** package comes with an excellent array of methods for reading and writing data to and from other programming languages such as SAS, SPSS and Stata. Table 7.1 provides a summary of the functions used for each program.

[6] It is possible to read a minitab portable worksheet using the function `read.minitab()` from the **foreign** package (not part of the **tidyverse**). This returns a list with one component for each column, matrix or constant stored in the Minitab worksheet: `read.minitab("filename.mtp")`

	Program		
Task	SAS	SPSS	Stata
Reading	`read_sas()`	`read_spss()`	`read_dta()`
	`read_xpt()`	`read_por()`	
Writing	`write_xpt()`	`write_sav()`	`write_dta()`

Table 7.1: **haven** functions for reading and writing data.

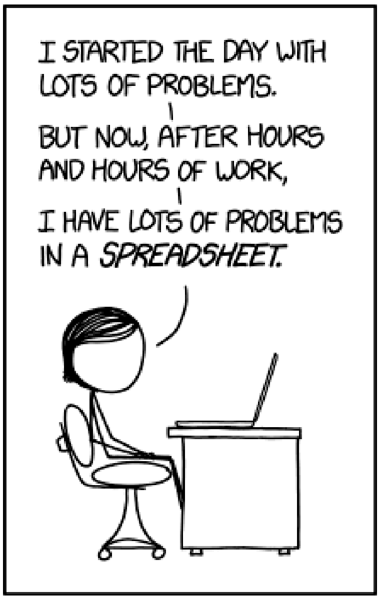

Figure 7.1: https://xkcd.com/1906/