Introduction to Power BI

Contents

Introduction

These are course notes for the "Introduction to Power BI" course given by the Monash Bioinformatics Platform[1] for the Monash Data Fluency[2] initiative. Our teaching style is based on the style of The Carpentries[3].

- PDF version for printing[4]
- ZIP of data files used in this workshop[5]

During this workshop we will be using Power BI Desktop installed on your computer. There are several ways to download Power BI Desktop, depending on which system you use.

1. Windows User

- Power BI website You can download Power BI Desktop from the website and install it as an application on your computer. Monash machine, My software, contact eSolutions to gain admin access(link to eSolution)

After the setup process, you will be able to see the following Start Screen.

- Windows Store Or you can visit Windows Store to get the Power BI Desktop app and install it. Note that the system requirements is Windows 10 version 14393.0 or higher.

- Power BI service You can also download it from the Power BI service by clicking the download button in the upper right and selecting Power BI Desktop. To use Power BI service, you may need a Microsoft account.

2. Mac User

Power BI Desktop is not available on Macs. There are two main options. Dual BootCamp The first is to run a Windows session on your Mac via BootCamp or something similar. This is probably a longer term solution until Microsoft release a Mac version. MoVE: TODO

After installing Power BI Desktop, you can sign up for Power BI using your Monash account here By signing in the Power BI Desktop, you will be able to save your work and later publish it to the Power BI service.

[1] https://www.monash.edu/researchinfrastructure/bioinformatics
[2] https://monashdatafluency.github.io/
[3] https://carpentries.org/
[4] https://monashdatafluency.github.io/powerbi/powerbi-intro.pdf
[5] https://monashdatafluency.github.io/powerbi-intro/powerbi-files.zip

3. Linux Users

TODO

Data

Please download the data file here for the course.

Source code

This book was created in R using the `rmarkdown` and `bookdown` packages!

- GitHub page[6]

Authors and copyright

This course is developed for the Monash Data Fluency Team.

Data files are derived from Gapminder, which has a CC BY-4 license. The attribution is “Free data from www.gapminder.org”. The data is given here in a form designed to teach various points about the R language. Refer to the Gapminder site[8] for the original form of the data if using it for other uses.

[6] https://github.com/MonashDataFluency/r-intro-2
[7] http://creativecommons.org/licenses/by/4.0/
[8] https://www.gapminder.org

Chapter 1

Introduction to PowerBI

1.1 Overview of Power BI

Microsoft Power BI is a collection of apps, software services and connectors that come together to turn the unrelated data into visually impressive and interactive insights. Power BI can work with the simplest of data sources like Microsoft Excel and the more complicated ones like a collection of cloud-based or on-premises hybrid Data warehouses. Power BI has the capabilities to easily connect to your data sources, visualise and share and publish your findings with anyone and everyone.

Power BI can be simple and fast enough to connect to an Excel workbook or a local database or it can be robust and enterprise-grade, ready for extensive modeling and real time analytics and also for custom development. Hence, it can be a personal report and vis tool but can also act as the analytics and decision engine behind group projects, divisions, or entire corporations.

1.2 The parts of Power BI

Power BI constitutes of a Microsoft Windows desktop application called Power BI Desktop, an online SaaS (Software as a Service) called Power BI Service and a mobile Power BI apps that can be accessed from Windows phones and tablets, and are also available on Apple iOS and Google Android devices.

These three elements— **Desktop**, the **service**, and **Mobile** apps - are the backbone of the Power BI system and lets users create, share and consume the actionable insights in the most effective way.

1.3 Use of Power BI and roles

The use of Power BI could depend a lot on the role that you are in. For example: if you are the stakeholder of a project, then you might want to use **Power BI Service** or the Mobile **app** to have a glance at how the business is performing. But on the other hand, if you are a developer, you would be using **Power BI Desktop** extensively and then publish Power BI desktop reports to the Power BI Service.

In the upcoming modules we would be discussing about these three components - **Desktop**, **Service** and **Mobile** apps - in more detail.

1.4 Power BI Flow

In the most general way, the flow starts at the Power BI Desktop, where a report is created. This created report can be published to the Power BI Service and finally shared so that the users can use it from the Mobile apps.

Its not always the case that this flow happens, but more often or not it is. We will stick to this flow for this entire tutorial to help learn the different aspects of Power BI.

1.5 Use Power BI:

The **common** flow of activity in Power BI looks like this: 1. Bring data into Power BI Desktop, and create a report. 2. Publish to the Power BI service, where you can create new visualizations or build dashboards. 3. Share dashboards with others, especially people who are on the go. 4. View and interact with shared dashboards and reports in Power BI Mobile apps.

As mentioned earlier, depending on the user role, the user might spend its most of the time in one of the three components than the other.

1.6 Building blocks of Power BI:

The basic building blocks in Power BI are: * Visualizations

- Datasets
- Reports
- Dashboards
- Tiles

1.6.1 Visualizations

A visualization is a representation of data in a visual format. It could be a line chart, a bar graph, a color coded map or anything interesting to present the data.

–Picture of a final visualisation–

Visualizations can be simple as a number representing something significant or it could be quite complex like multiple stacked chart showing the proportion users participating in a survey. The prime idea of visualisation is to show the data in a way that it tells the story that's lying underneath it. Like its said, a picture says a thousand words.

1.6.2 Datasets:

A **dataset** is a collection of data that Power BI uses to create its visualizations. You can have a simple dataset that's based on a single table from a Microsoft Excel workbook, similar to what's shown in the following image.

–Picture of Dataset–

Dataset can also be a combination of many different sources, which can be filtered using Power BI and combine into one to use.

For eg: One of the data could be the countries and its central location in the form of Latitude and Longitude and other data could be the demographics of the countries like, population, GDP etc. Power BI can combine these two data and make one dataset out of it to be used for visualizations.

An important feature of Power BI is the ability of it to connect to various data sources using its connectors. Whether the data you want is in Excel or a Microsoft SQL Server database, in Azure or Oracle, or in a service like Facebook, Salesforce, or MailChimp, Power BI has built-in data connectors that let you easily connect to that data, filter it if necessary, and bring it into your dataset.

After you have a dataset, you can begin creating visualizations that show different portions of it in different ways, and gain insights based on what you see. That's where reports come in.

1.6.3 Reports:

In Power BI, a **report** is a collection of visualizations that appear together on one or more pages. A report in Power BI is a collection of items that are related to each other. The following image shows a report that you would be creating by the end of the session. You can also create reports in the Power BI service.

– Picture of report–

Reports let us create many visualizations and possibly on multiple pages based on the way the developer wants to tell the story.

1.6.4 Dashboards:

A Power BI dashboard is a collection of visuals from a single page that you can share with others. Often, it's a selected group of visuals that provide quick insight into the data or story you're trying to present.

A dashboard must fit on a single page, often called a canvas (the canvas is the blank backdrop in Power BI Desktop or the service, where you put visualizations). Think of it like the canvas that an artist or painter uses—a workspace where you create, combine, and rework interesting and compelling visuals. You can share dashboards with other users or groups, who can then interact with your dashboards when they're in the Power BI service or on their mobile device.

1.6.5 Tiles:

TODO

1.7 Power BI Services:

1.7.1 Overview of Power BI Desktop

Power BI Desktop is a free application for PCs that lets you gather, transform, and visualize your data. In this module, you'll learn how to find and collect data from different sources and how to clean or transform it. You'll also learn tricks to make data-gathering easier. Power BI Desktop and the Power BI Service work together. You can create your reports and dashboards in Power BI Desktop, and then publish them to the Power BI Service for others to consume.

–Picture of desktop view–

1. **Ribbon** - Displays common tasks that are associated with reports and visualizations.
2. **Report view, or canvas** - Where visualizations are created and arranged. You can switch between **Report**, **Data**, and **Model** views by selecting the icons in the left column.
3. **Pages tab** - Located along the bottom of the page, this area is where you would select or add a report page.
4. **Visualizations pane** - Where you can change visualizations, customize colors or axes, apply filters, drag fields, and more.
5. **Fields pane** - Where query elements and filters can be dragged onto the **Report** view or dragged to the **Filters** area of the Visualizations pane.

Chapter 2

Data frames

Data frame is R's name for tabular data. We generally want each row in a data frame to represent a unit of observation, and each column to contain a different type of information about the units of observation. Tabular data in this form is called "tidy data"[1].

Today we will be using a collection of modern packages collectively known as the Tidyverse[2]. R and its predecessor S have a history dating back to 1976. The Tidyverse fixes some dubious design decisions baked into "base R", including having its own slightly improved form of data frame, which is called a *tibble*. Sticking to the Tidyverse where possible is generally safer, Tidyverse packages are more willing to generate errors rather than ignore problems.

2.1 Setting up

Our first step is to download the files we need and to install the Tidyverse. This is the one step where we ask you to copy and paste some code:

```
# Download files for this workshop
download.file(
  "https://monashdatafluency.github.io/r-intro-2/r-intro-2-files.zip",
  destfile="r-intro-2-files.zip")
unzip("r-intro-2-files.zip")

# Install Tidyverse
install.packages("tidyverse")
```

If using RStudio Cloud, you might need to switch to R version 3.5.3 to successfully install Tidyverse. Use the drop-down in the top right corner of the page.

People also sometimes have problems installing all the packages in Tidyverse on Windows machines. If you run into problems you may have more success

[1]http://vita.had.co.nz/papers/tidy-data.html
[2]https://www.tidyverse.org/

installing individual packages.

```
install.packages(c("dplyr","readr","tidyr","ggplot2"))
```

We need to load the `tidyverse` package in order to use it.

```
library(tidyverse)

# OR
library(dplyr)
library(readr)
library(tidyr)
library(ggplot2)
```

The `tidyverse` package loads various other packages, setting up a modern R environment. In this section we will be using functions from the `dplyr`, `readr` and `tidyr` packages.

R is a language with mini-languages within it that solve specific problem domains. `dplyr` is such a mini-language, a set of "verbs" (functions) that work well together. `dplyr`, with the help of `tidyr` for some more complex operations, provides a way to perform most manipulations on a data frame that you might need.

2.2 Loading data

We will use the `read_csv` function from `readr` to load a data set. (See also `read.csv` in base R.) CSV stands for Comma Separated Values, and is a text format used to store tabular data. The first few lines of the file we are loading are shown below. Conventionally the first line contains column headings.

```
name,region,oecd,g77,lat,long,income2017
Afghanistan,asia,FALSE,TRUE,33,66,low
Albania,europe,FALSE,FALSE,41,20,upper_mid
Algeria,africa,FALSE,TRUE,28,3,upper_mid
Andorra,europe,FALSE,FALSE,42.50779,1.52109,high
Angola,africa,FALSE,TRUE,-12.5,18.5,lower_mid
```

```
geo <- read_csv("r-intro-2-files/geo.csv")
```

```
## Parsed with column specification:
## cols(
##   name = col_character(),
##   region = col_character(),
##   oecd = col_logical(),
##   g77 = col_logical(),
##   lat = col_double(),
##   long = col_double(),
##   income2017 = col_character()
## )
```

```
geo
```

```
## # A tibble: 196 x 7
##    name                region   oecd  g77     lat   long income2017
##    <chr>               <chr>    <lgl> <lgl> <dbl>  <dbl> <chr>
##  1 Afghanistan         asia     FALSE TRUE   33     66    low
##  2 Albania             europe   FALSE FALSE  41     20    upper_mid
##  3 Algeria             africa   FALSE TRUE   28      3    upper_mid
##  4 Andorra             europe   FALSE FALSE  42.5    1.52 high
##  5 Angola              africa   FALSE TRUE  -12.5   18.5  lower_mid
##  6 Antigua and Barbuda americas FALSE TRUE   17.0  -61.8  high
##  7 Argentina           americas FALSE TRUE  -34    -64    upper_mid
##  8 Armenia             europe   FALSE FALSE  40.2   45    lower_mid
##  9 Australia           asia     TRUE  FALSE -25    135    high
## 10 Austria             europe   TRUE  FALSE  47.3   13.3  high
## # ... with 186 more rows
```

`read_csv` has guessed the type of data each column holds:

- `<chr>` - character strings
- `<dbl>` - numerical values. Technically these are "doubles", which is a way of storing numbers with 15 digits precision.
- `<lgl>` - logical values, `TRUE` or `FALSE`.

We will also encounter:

- `<int>` - integers, a fancy name for whole numbers.
- `<fct>` - factors, categorical data. We will get to this shortly.

You can also see this data frame referring to itself as "a tibble". This is the Tidyverse's improved form of data frame. Tibbles present themselves more conveniently than base R data frames. Base R data frames don't show the type of each column, and output every row when you try to view them.

Tip

A data frame can also be created from vectors, with the `tibble` function. (See also `data.frame` in base R.) For example:

```r
tibble(foo=c(10,20,30), bar=c("a","b","c"))
```

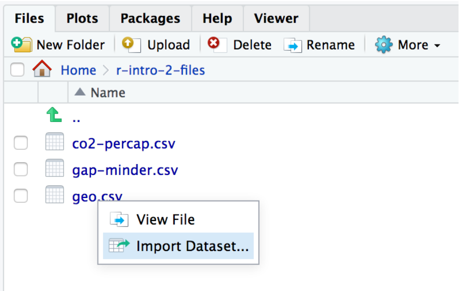
```
## # A tibble: 3 x 2
##     foo bar
##   <dbl> <chr>
## 1    10 a
## 2    20 b
## 3    30 c
```

The argument names become column names in the data frame.

Tip

The *path* to the file on our server is `"r-intro-2-files/geo.csv"`. This says, starting from your working directory, look in the directory `r-intro-2-files` for the file `geo.csv`. The steps in the path are separated by `/`. Your working directory is shown at the top of the console pane. The path needed might be different on your own computer, depending where you downloaded the file.

One way to work out the correct path is to find the file in the file browser pane, click on it and select "Import Dataset…".



2.3 Exploring

The `View` function gives us a spreadsheet-like view of the data frame.

```
View(geo)
```

`print` with the `n` argument can be used to show more than the first 10 rows on the console.

```
print(geo, n=200)
```

We can extract details of the data frame with further functions:

```
nrow(geo)
```

```
## [1] 196
```

```
ncol(geo)
```

```
## [1] 7
```

```
colnames(geo)
```

```
## [1] "name"       "region"     "oecd"       "g77"        "lat"
## [6] "long"       "income2017"
```

```
summary(geo)
```

```
##      name              region              oecd              g77
##  Length:196         Length:196         Mode :logical    Mode :logical
##  Class :character   Class :character   FALSE:165        FALSE:65
##  Mode  :character   Mode  :character   TRUE :31         TRUE :131
##
##
##
##       lat               long            income2017
```

```
##  Min.   :-42.00   Min.   :-175.000   Length:196
##  1st Qu.:  4.00   1st Qu.:  -5.625   Class :character
##  Median : 17.42   Median :  21.875   Mode  :character
##  Mean   : 19.03   Mean   :  23.004
##  3rd Qu.: 39.82   3rd Qu.:  51.892
##  Max.   : 65.00   Max.   : 179.145
```

2.4 Indexing data frames

Data frames can be subset using `[row,column]` syntax.

```
geo[4,2]
```

```
## # A tibble: 1 x 1
##   region
##   <chr>
## 1 europe
```

Note that while this is a single value, it is still wrapped in a data frame. (This is a behaviour specific to Tidyverse data frames.) More on this in a moment.

Columns can be given by name.

```
geo[4,"region"]
```

```
## # A tibble: 1 x 1
##   region
##   <chr>
## 1 europe
```

The column or row may be omitted, thereby retrieving the entire row or column.

```
geo[4,]
```

```
## # A tibble: 1 x 7
##   name    region oecd  g77     lat  long income2017
##   <chr>   <chr>  <lgl> <lgl> <dbl> <dbl> <chr>
## 1 Andorra europe FALSE FALSE  42.5  1.52 high
```

```
geo[,"region"]
```

```
## # A tibble: 196 x 1
##    region
##    <chr>
##  1 asia
##  2 europe
##  3 africa
##  4 europe
##  5 africa
##  6 americas
##  7 americas
##  8 europe
```

```
##  9 asia
## 10 europe
## # ... with 186 more rows
```

Multiple rows or columns may be retrieved using a vector.

```
rows_wanted <- c(1,3,5)
geo[rows_wanted,]
```

```
## # A tibble: 3 x 7
##   name        region oecd  g77     lat  long income2017
##   <chr>       <chr>  <lgl> <lgl> <dbl> <dbl> <chr>
## 1 Afghanistan asia   FALSE TRUE   33    66   low
## 2 Algeria     africa FALSE TRUE   28     3   upper_mid
## 3 Angola      africa FALSE TRUE  -12.5  18.5 lower_mid
```

Vector indexing can also be written on a single line.

```
geo[c(1,3,5),]
```

```
## # A tibble: 3 x 7
##   name        region oecd  g77     lat  long income2017
##   <chr>       <chr>  <lgl> <lgl> <dbl> <dbl> <chr>
## 1 Afghanistan asia   FALSE TRUE   33    66   low
## 2 Algeria     africa FALSE TRUE   28     3   upper_mid
## 3 Angola      africa FALSE TRUE  -12.5  18.5 lower_mid
```

```
geo[1:7,]
```

```
## # A tibble: 7 x 7
##   name                region   oecd  g77     lat   long income2017
##   <chr>               <chr>    <lgl> <lgl> <dbl>  <dbl> <chr>
## 1 Afghanistan         asia     FALSE TRUE   33    66    low
## 2 Albania             europe   FALSE FALSE  41    20    upper_mid
## 3 Algeria             africa   FALSE TRUE   28     3    upper_mid
## 4 Andorra             europe   FALSE FALSE  42.5   1.52 high
## 5 Angola              africa   FALSE TRUE  -12.5  18.5  lower_mid
## 6 Antigua and Barbuda americas FALSE TRUE   17.0 -61.8  high
## 7 Argentina           americas FALSE TRUE  -34   -64    upper_mid
```

2.5 Columns are vectors

Ok, so how do we actually get data out of a data frame?

Under the hood, a data frame is a list of column vectors. We can use `$` to retrieve columns. Occasionally it is also useful to use `[[ ]]` to retrieve columns, for example if the column name we want is stored in a variable.

```
head( geo$region )
```

```
## [1] "asia"     "europe"   "africa"   "europe"   "africa"   "americas"
```

```
head( geo[["region"]] )
```

```
## [1] "asia"      "europe"    "africa"    "europe"    "africa"    "americas"
```

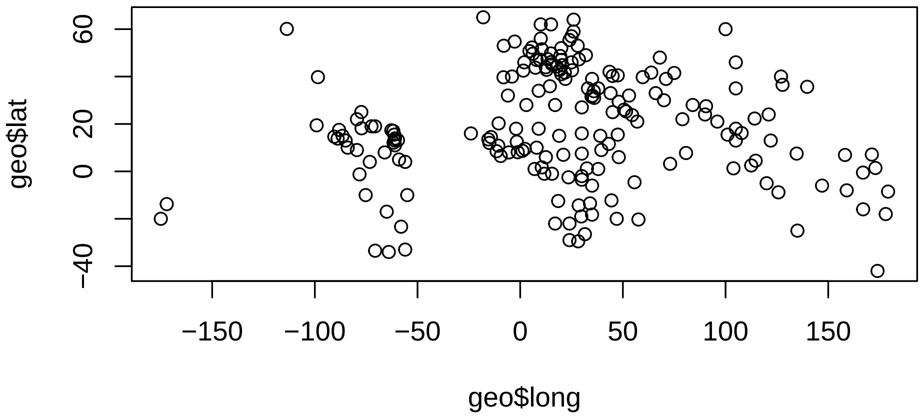
To get the “region” value of the 4th row as above, but unwrapped, we can use:

```
geo$region[4]
```

```
## [1] "europe"
```

For example, to plot the longitudes and latitudes we could use:

```
plot(geo$long, geo$lat)
```



2.6 Logical indexing

A method of indexing that we haven't discussed yet is logical indexing. Instead of specifying the row number or numbers that we want, we can give a logical vector which is `TRUE` for the rows we want and `FALSE` otherwise. This can also be used with vectors.

We will first do this in a slightly verbose way in order to understand it, then learn a more concise way to do this using the `dplyr` package.

Southern countries have latitude less than zero.

```
is_southern <- geo$lat < 0

head(is_southern)
```

```
## [1] FALSE FALSE FALSE FALSE  TRUE FALSE
```

```
sum(is_southern)
```

```
## [1] 40
```

`sum` treats TRUE as 1 and FALSE as 0, so it tells us the number of TRUE elements in the vector.

We can use this logical vector to get the southern countries from `geo`:

```
geo[is_southern,]
```

```
## # A tibble: 40 x 7
##    name              region   oecd  g77     lat  long income2017
##    <chr>             <chr>    <lgl> <lgl> <dbl> <dbl> <chr>
##  1 Angola            africa   FALSE TRUE  -12.5  18.5 lower_mid
##  2 Argentina         americas FALSE TRUE  -34   -64   upper_mid
##  3 Australia         asia     TRUE  FALSE -25   135   high
##  4 Bolivia           americas FALSE TRUE  -17   -65   lower_mid
##  5 Botswana          africa   FALSE TRUE  -22    24   upper_mid
##  6 Brazil            americas FALSE TRUE  -10   -55   upper_mid
##  7 Burundi           africa   FALSE TRUE   -3.5  30   low
##  8 Chile             americas TRUE  TRUE  -33.5 -70.6 high
##  9 Comoros           africa   FALSE TRUE  -12.2  44.4 low
## 10 Congo, Dem. Rep. africa   FALSE TRUE   -2.5  23.5 low
## # ... with 30 more rows
```

Comparison operators available are:

- `x == y` – "equal to"
- `x != y` – "not equal to"
- `x < y` – "less than"
- `x > y` – "greater than"
- `x <= y` – "less than or equal to"
- `x >= y` – "greater than or equal to"

More complicated conditions can be constructed using logical operators:

- `a & b` – "and", TRUE only if both `a` and `b` are TRUE.
- `a | b` – "or", TRUE if either `a` or `b` or both are TRUE.
- `! a` – "not" , TRUE if `a` is FALSE, and FALSE if `a` is TRUE.

The `oecd` column of `geo` tells which countries are in the Organisation for Economic Co-operation and Development, and the `g77` column tells which countries are in the Group of 77 (an alliance of developing nations). We could see which OECD countries are in the southern hemisphere with:

```
southern_oecd <- is_southern & geo$oecd

geo[southern_oecd,]
```

```
## # A tibble: 3 x 7
##   name        region   oecd  g77     lat  long income2017
##   <chr>       <chr>    <lgl> <lgl> <dbl> <dbl> <chr>
## 1 Australia   asia     TRUE  FALSE -25   135   high
## 2 Chile       americas TRUE  TRUE  -33.5 -70.6 high
## 3 New Zealand asia     TRUE  FALSE -42   174   high
```

`is_southern` seems like it should be kept within our `geo` data frame for future use. We can add it as a new column of the data frame with:

```
geo$southern <- is_southern

geo
```

```
## # A tibble: 196 x 8
##    name                region  oecd  g77     lat   long income2017 southern
##    <chr>               <chr>   <lgl> <lgl> <dbl>  <dbl> <chr>      <lgl>
##  1 Afghanistan         asia    FALSE TRUE   33    66    low        FALSE
##  2 Albania             europe  FALSE FALSE  41    20    upper_mid  FALSE
##  3 Algeria             africa  FALSE TRUE   28     3    upper_mid  FALSE
##  4 Andorra             europe  FALSE FALSE  42.5   1.52 high       FALSE
##  5 Angola              africa  FALSE TRUE  -12.5  18.5  lower_mid  TRUE
##  6 Antigua and Barbuda americ~ FALSE TRUE   17.0 -61.8  high       FALSE
##  7 Argentina           americ~ FALSE TRUE  -34   -64    upper_mid  TRUE
##  8 Armenia             europe  FALSE FALSE  40.2  45    lower_mid  FALSE
##  9 Australia           asia    TRUE  FALSE -25   135    high       TRUE
## 10 Austria             europe  TRUE  FALSE  47.3  13.3  high       FALSE
## # ... with 186 more rows
```

Challenge: logical indexing

1. Which country is in both the OECD and the G77?
2. Which countries are in neither the OECD nor the G77?
3. Which countries are in the Americas? These have longitudes between -150 and -40.

2.6.1 A `dplyr` shorthand

The above method is a little laborious. We have to keep mentioning the name of the data frame, and there is a lot of punctuation to keep track of. `dplyr` provides a slightly magical function called `filter` which lets us write more concisely. For example:

```
filter(geo, lat < 0 & oecd)
```

```
## # A tibble: 3 x 8
##   name        region   oecd  g77     lat  long income2017 southern
##   <chr>       <chr>    <lgl> <lgl> <dbl> <dbl> <chr>      <lgl>
## 1 Australia   asia     TRUE  FALSE -25   135   high       TRUE
## 2 Chile       americas TRUE  TRUE  -33.5 -70.6 high       TRUE
## 3 New Zealand asia     TRUE  FALSE -42   174   high       TRUE
```

In the second argument, we are able to refer to columns of the data frame as though they were variables. The code is beautiful, but also opaque. It's important to understand that under the hood we are creating and combining logical vectors.

2.7 Factors

The `count` function from `dplyr` can help us understand the contents of some of the columns in `geo`. `count` is also *magical*, we can refer to columns of the data frame directly in the arguments to `count`.

```
count(geo, region)
```

```
## # A tibble: 4 x 2
##   region       n
##   <chr>    <int>
## 1 africa      54
## 2 americas    35
## 3 asia        59
## 4 europe      48
```

```
count(geo, income2017)
```

```
## # A tibble: 4 x 2
##   income2017     n
##   <chr>      <int>
## 1 high          58
## 2 low           31
## 3 lower_mid     52
## 4 upper_mid     55
```

One annoyance here is that the different categories in `income2017` aren't in a sensible order. This comes up quite often, for example when sorting or plotting categorical data. R's solution is a further type of vector called a *factor* (think a factor of an experimental design). A factor holds categorical data, and has an associated ordered set of *levels*. It is otherwise quite similar to a character vector.

Any sort of vector can be converted to a factor using the `factor` function. This function defaults to placing the levels in alphabetical order, but takes a `levels` argument that can override this.

```
head( factor(geo$income2017, levels=c("low","lower_mid","upper_mid","high")) )
```

```
## [1] low       upper_mid upper_mid high      lower_mid high
## Levels: low lower_mid upper_mid high
```

We should modify the `income2017` column of the `geo` table in order to use this:

```
geo$income2017 <- factor(geo$income2017, levels=c("low","lower_mid","upper_mid","high"))
```

`count` now produces the desired order of output:

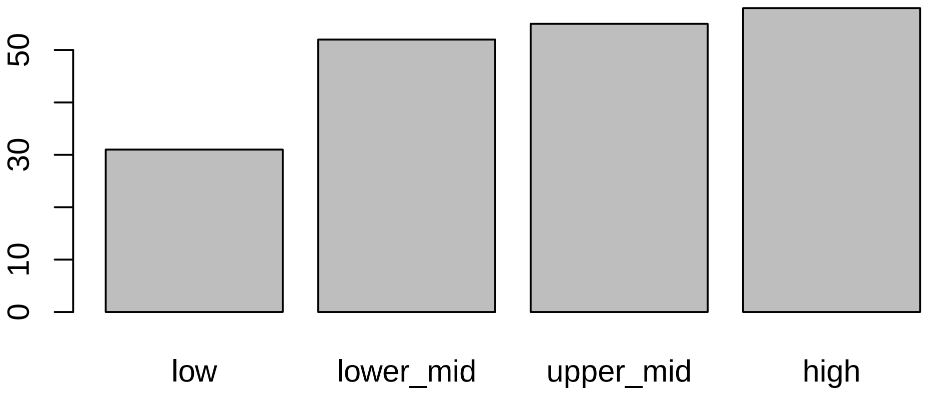
```
count(geo, income2017)
```

```
## # A tibble: 4 x 2
##   income2017     n
##   <fct>      <int>
## 1 low           31
```

```
## 2 lower_mid      52
## 3 upper_mid      55
## 4 high           58
```

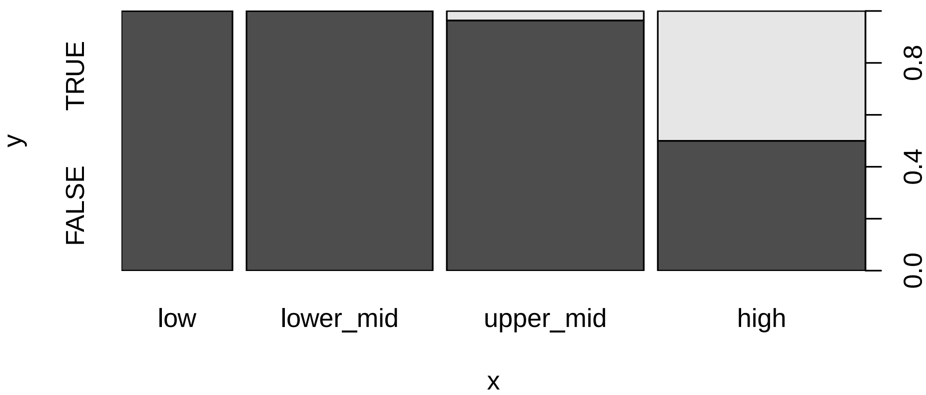
When `plot` is given a factor, it shows a bar plot:

```
plot(geo$income2017)
```



When given two factors, it shows a mosaic plot:

```
plot(geo$income2017, factor(geo$oecd))
```



Similarly we can count two categorical columns at once.

```
count(geo, income2017, oecd)
```

```
## # A tibble: 6 x 3
##   income2017 oecd      n
##   <fct>      <lgl> <int>
## 1 low        FALSE    31
## 2 lower_mid  FALSE    52
## 3 upper_mid  FALSE    53
## 4 upper_mid  TRUE      2
## 5 high       FALSE    29
## 6 high       TRUE     29
```

2.8 Readability vs tidyness

The counts we obtained counting `income2017` vs `oecd` were properly tidy in the sense of containing a single unit of observation per row. However to view the data, it would be more convenient to have income as columns and OECD membership as rows. We can use the `spread` function from `tidyr` to achieve this.

```
counts <- count(geo, income2017, oecd)
spread(counts, key=income2017, value=n, fill=0)
```

```
## # A tibble: 2 x 5
##   oecd    low lower_mid upper_mid  high
##   <lgl> <dbl>     <dbl>     <dbl> <dbl>
## 1 FALSE    31        52        53    29
## 2 TRUE      0         0         2    29
```

Here:

- The `key` column became column names.
- The `value` column became the values in the new columns.
- The `fill` value is used to fill in any missing values.

Tip

Tidying is often the first step when exploring a data-set. The tidyr[3] package contains a number of useful functions that help tidy (or un-tidy!) data. We've just seen `spread` which spreads two columns into multiple columns. The inverse of `spread` is `gather`, which gathers multiple columns into two columns: a column of column names, and a column of values.

Challenge: counting

Investigate how many OECD and non-OECD nations come from the northern and southern hemispheres.

1. Using `count`.
2. By making a mosaic plot.

Remember you may need to convert columns to factors for `plot` to work, and that a `southern` column could be added to `geo` with:

```
geo$southern <- geo$lat < 0
```

2.9 Sorting

Data frames can be sorted using the `arrange` function in `dplyr`.

[3] http://tidyr.tidyverse.org/

```
arrange(geo, lat)
```

```
## # A tibble: 196 x 8
##    name         region   oecd  g77     lat  long income2017 southern
##    <chr>        <chr>    <lgl> <lgl> <dbl> <dbl> <fct>      <lgl>
##  1 New Zealand  asia     TRUE  FALSE -42   174   high       TRUE
##  2 Argentina    americas FALSE TRUE  -34   -64   upper_mid  TRUE
##  3 Chile        americas TRUE  TRUE  -33.5 -70.6 high       TRUE
##  4 Uruguay      americas FALSE TRUE  -33   -56   high       TRUE
##  5 Lesotho      africa   FALSE TRUE  -29.5  28.2 lower_mid  TRUE
##  6 South Africa africa   FALSE TRUE  -29    24   upper_mid  TRUE
##  7 Swaziland    africa   FALSE TRUE  -26.5  31.5 lower_mid  TRUE
##  8 Australia    asia     TRUE  FALSE -25   135   high       TRUE
##  9 Paraguay     americas FALSE TRUE  -23.3 -58   upper_mid  TRUE
## 10 Botswana     africa   FALSE TRUE  -22    24   upper_mid  TRUE
## # ... with 186 more rows
```

Numeric columns are sorted in numeric order. Character columns will be sorted in alphabetical order. Factor columns are sorted in order of their levels. The `desc` helper function can be used to sort in descending order.

```
arrange(geo, desc(name))
```

```
## # A tibble: 196 x 8
##    name           region   oecd  g77     lat   long income2017 southern
##    <chr>          <chr>    <lgl> <lgl> <dbl>  <dbl> <fct>      <lgl>
##  1 Zimbabwe       africa   FALSE TRUE  -19    29.8  low        TRUE
##  2 Zambia         africa   FALSE TRUE  -14.3  28.5  lower_mid  TRUE
##  3 Yemen          asia     FALSE TRUE   15.5  47.5  lower_mid  FALSE
##  4 Vietnam        asia     FALSE TRUE   16.2 108.   lower_mid  FALSE
##  5 Venezuela      americas FALSE TRUE    8   -66    upper_mid  FALSE
##  6 Vanuatu        asia     FALSE TRUE  -16   167    lower_mid  TRUE
##  7 Uzbekistan     asia     FALSE FALSE  41.7  63.8  lower_mid  FALSE
##  8 Uruguay        americas FALSE TRUE  -33   -56    high       TRUE
##  9 United States  americas TRUE  FALSE  39.8 -98.5  high       FALSE
## 10 United Kingdom europe   TRUE  FALSE  54.8  -2.70 high       FALSE
## # ... with 186 more rows
```

2.10 Joining data frames

Let's move on to a larger data set. This is from the Gapminder[4] project and contains information about countries over time.

```
gap <- read_csv("r-intro-2-files/gap-minder.csv")
gap
```

```
## # A tibble: 4,312 x 5
##    name                   year population gdp_percap life_exp
```

[4] https://www.gapminder.org

```
##    <chr>                  <dbl>      <dbl>      <dbl>    <dbl>
##  1 Afghanistan             1800    3280000        603     28.2
##  2 Albania                 1800     410445        667     35.4
##  3 Algeria                 1800    2503218        715     28.8
##  4 Andorra                 1800       2654       1197     NA
##  5 Angola                  1800    1567028        618     27.0
##  6 Antigua and Barbuda     1800      37000        757     33.5
##  7 Argentina               1800     534000       1507     33.2
##  8 Armenia                 1800     413326        514     34
##  9 Australia               1800     351014        814     34.0
## 10 Austria                 1800    3205587       1847     34.4
## # ... with 4,302 more rows
```

Quiz

What is the unit of observation in this new data frame?

It would be useful to have general information about countries from `geo` available as columns when we use this data frame. `gap` and `geo` share a column called `name` which can be used to match rows from one to the other.

```
gap_geo <- left_join(gap, geo, by="name")
gap_geo
```

```
## # A tibble: 4,312 x 12
##    name   year population gdp_percap life_exp region oecd  g77     lat
##    <chr> <dbl>      <dbl>      <dbl>    <dbl> <chr>  <lgl> <lgl> <dbl>
##  1 Afgh~  1800    3280000        603     28.2 asia   FALSE TRUE   33
##  2 Alba~  1800     410445        667     35.4 europe FALSE FALSE  41
##  3 Alge~  1800    2503218        715     28.8 africa FALSE TRUE   28
##  4 Ando~  1800       2654       1197     NA   europe FALSE FALSE  42.5
##  5 Ango~  1800    1567028        618     27.0 africa FALSE TRUE  -12.5
##  6 Anti~  1800      37000        757     33.5 ameri~ FALSE TRUE   17.0
##  7 Arge~  1800     534000       1507     33.2 ameri~ FALSE TRUE  -34
##  8 Arme~  1800     413326        514     34   europe FALSE FALSE  40.2
##  9 Aust~  1800     351014        814     34.0 asia   TRUE  FALSE -25
## 10 Aust~  1800    3205587       1847     34.4 europe TRUE  FALSE  47.3
## # ... with 4,302 more rows, and 3 more variables: long <dbl>,
## #   income2017 <fct>, southern <lgl>
```

The output contains all ways of pairing up rows by `name`. In this case each row of `geo` pairs up with multiple rows of `gap`.

The "left" in "left join" refers to how rows that can't be paired up are handled. `left_join` keeps all rows from the first data frame but not the second. This is a good default when the intent is to attaching some extra information to a data frame. `inner_join` discard all rows that can't be paired up. `full_join` keeps all rows from both data frames.

2.11 Further reading

We've covered the fundamentals of dplyr and data frames, but there is much more to learn. Notably, we haven't covered the use of the pipe `%>%` to chain `dplyr` verbs together. The "R for Data Science" book[5] is an excellent source to learn more. The Monash Data Fluency "Programming and Tidy data analysis in R" course[6] also covers this.

[5] http://r4ds.had.co.nz/
[6] https://monashdatafluency.github.io/r-progtidy/

Chapter 3

Plotting with ggplot2

We already saw some of R's built in plotting facilities with the function `plot`. A more recent and much more powerful plotting library is `ggplot2`. `ggplot2` is another mini-language within R, a language for creating plots. It implements ideas from a book called "The Grammar of Graphics"[1]. The syntax can be a little strange, but there are plenty of examples in the online documentation[2].

`ggplot2` is part of the Tidyverse, so loadinging the `tidyverse` package will load `ggplot2`.

```
library(tidyverse)
```

We continue with the Gapminder dataset, which we loaded with:

```
geo <- read_csv("r-intro-2-files/geo.csv")
gap <- read_csv("r-intro-2-files/gap-minder.csv")
gap_geo <- left_join(gap, geo, by="name")
```

3.1 Elements of a ggplot

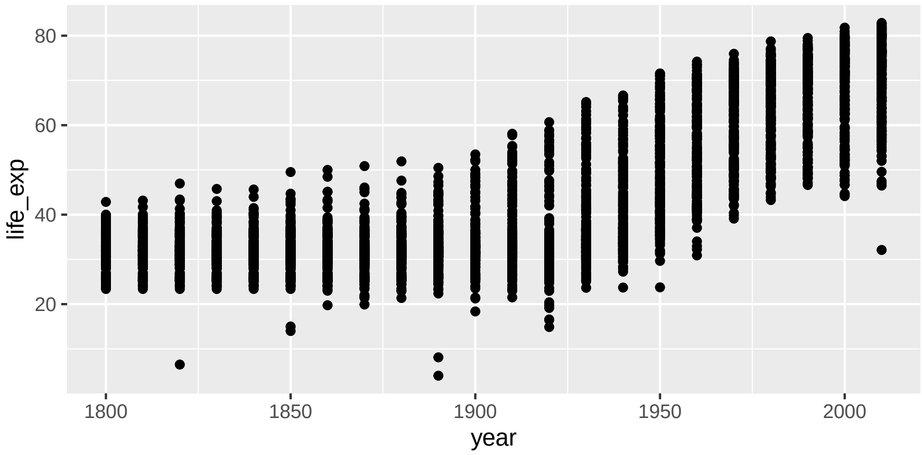
Producing a plot with `ggplot2`, we must give three things:

1. A data frame containing our data.
2. How the columns of the data frame can be translated into positions, colors, sizes, and shapes of graphical elements ("aesthetics").
3. The actual graphical elements to display ("geometric objects").

Let's make our first ggplot.

```
ggplot(gap_geo, aes(x=year, y=life_exp)) +
    geom_point()
```

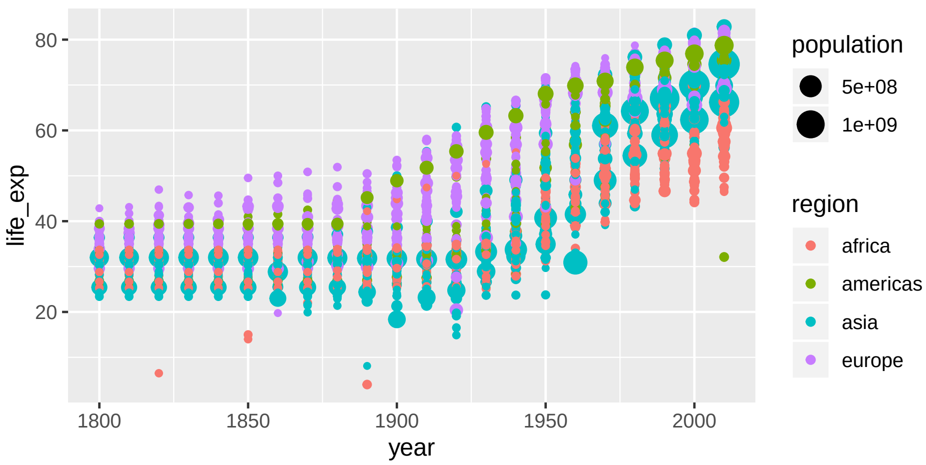
[1] https://www.amazon.com/Grammar-Graphics-Statistics-Computing/dp/0387245448
[2] http://ggplot2.tidyverse.org/reference/



The call to `ggplot` and `aes` sets up the basics of how we are going to represent the various columns of the data frame. `aes` defines the "aesthetics", which is how columns of the data frame map to graphical attributes such as x and y position, color, size, etc. `aes` is another example of magic "non-standard evaluation", arguments to `aes` may refer to columns of the data frame directly. We then literally add layers of graphics ("geoms") to this.

Further aesthetics can be used. Any aesthetic can be either numeric or categorical, an appropriate scale will be used.

```
ggplot(gap_geo, aes(x=year, y=life_exp, color=region, size=population)) +
    geom_point()
```



3.1.1 Challenge: make a ggplot

This R code will get the data from the year 2010:

```
gap2010 <- filter(gap_geo, year == 2010)
```

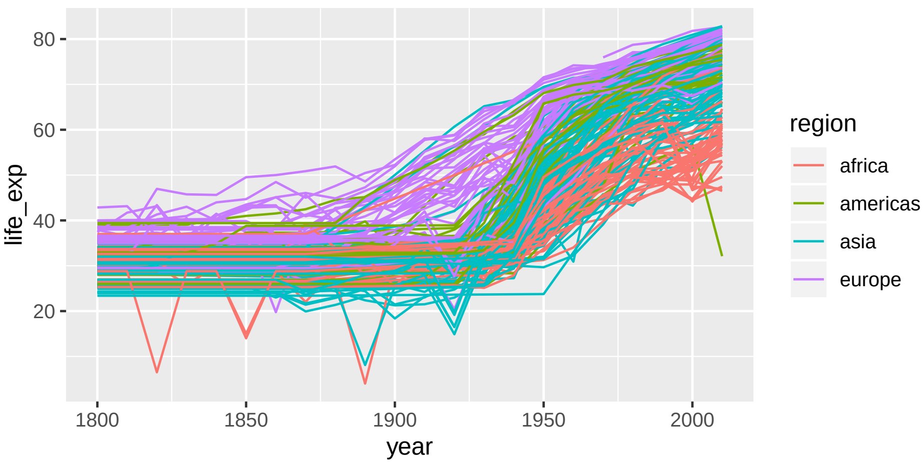
Create a ggplot of this with:

- `gdp_percap` as x.
- `life_exp` as y.
- `population` as the size.
- `region` as the color.

3.2 Further geoms

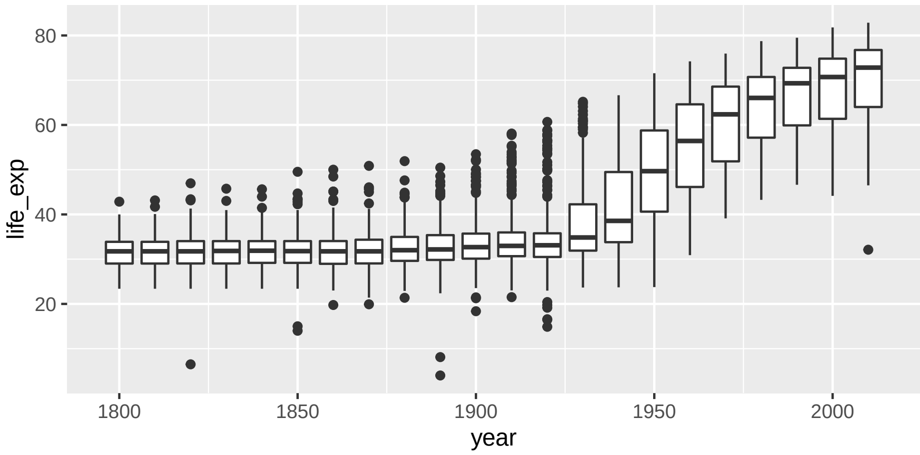
To draw lines, we need to use a "group" aesthetic.

```
ggplot(gap_geo, aes(x=year, y=life_exp, group=name, color=region)) +
    geom_line()
```



A wide variety of geoms are available. Here we show Tukey box-plots. Note again the use of the "group" aesthetic, without this ggplot will just show one big box-plot.

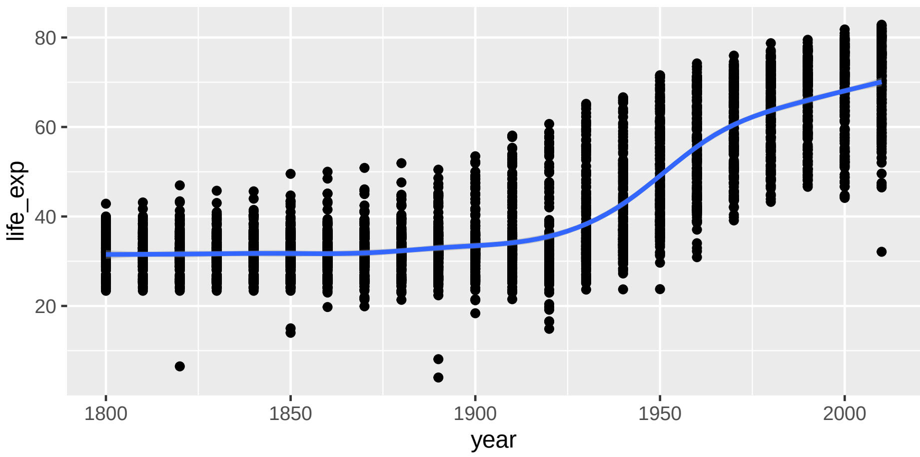
```
ggplot(gap_geo, aes(x=year, y=life_exp, group=year)) +
    geom_boxplot()
```



`geom_smooth` can be used to show trends.

```
ggplot(gap_geo, aes(x=year, y=life_exp)) +
    geom_point() +
    geom_smooth()
```

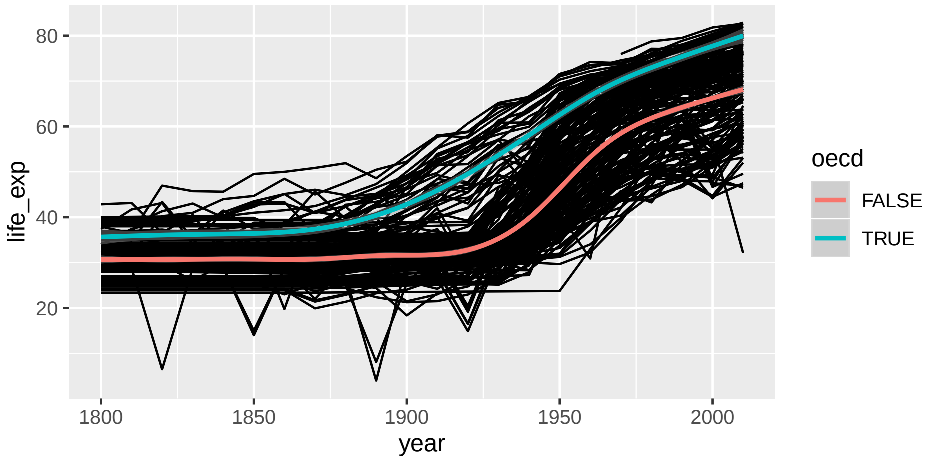
```
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```



Aesthetics can be specified globally in `ggplot`, or as the first argument to individual geoms. Here, the "group" is applied only to draw the lines, and "color" is used to produce multiple trend lines:

```
ggplot(gap_geo, aes(x=year, y=life_exp)) +
    geom_line(aes(group=name)) +
    geom_smooth(aes(color=oecd))
```

```
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```

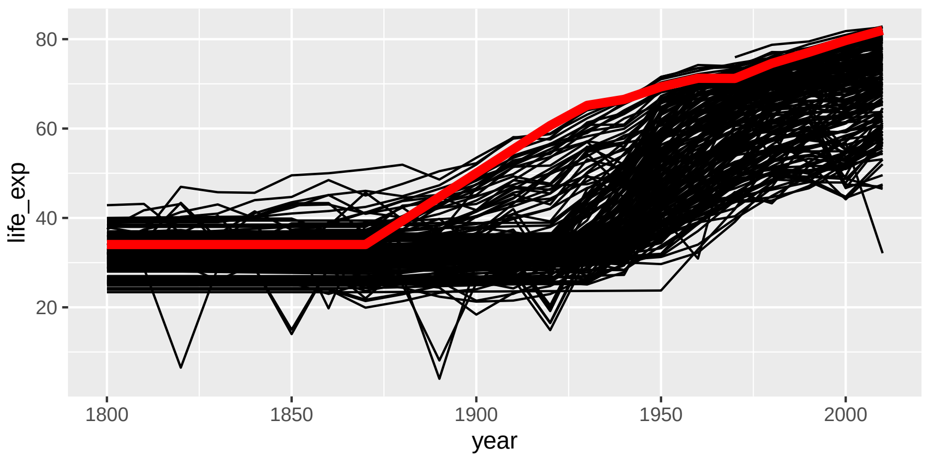


3.3 Highlighting subsets

Geoms can be added that use a different data frame, using the `data=` argument.

```
gap_australia <- filter(gap_geo, name == "Australia")

ggplot(gap_geo, aes(x=year, y=life_exp, group=name)) +
    geom_line() +
    geom_line(data=gap_australia, color="red", size=2)
```

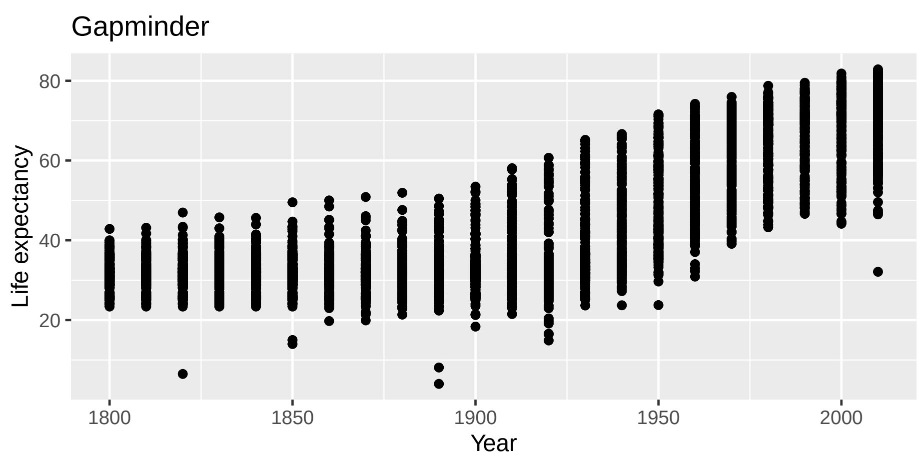


Notice also that the second `geom_line` has some further arguments controlling its appearance. These are **not** aesthetics, they are not a mapping of data to appearance, but rather a direct specification of the appearance. There isn't an associated scale as when color was an aesthetic.

3.4 Fine-tuning a plot

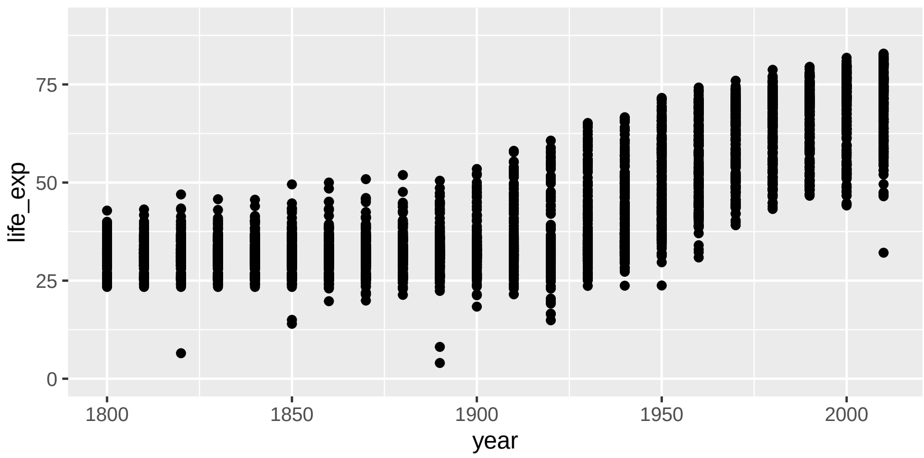
Adding `labs` to a ggplot adjusts the labels given to the axes and legends. A plot title can also be specified.

```
ggplot(gap_geo, aes(x=year, y=life_exp)) +
    geom_point() +
    labs(x="Year", y="Life expectancy", title="Gapminder")
```



`coord_cartesian` can be used to set the limits of the x and y axes. Suppose we want our y-axis to start at zero.

```
ggplot(gap_geo, aes(x=year, y=life_exp)) +
    geom_point() +
    coord_cartesian(ylim=c(0,90))
```



Type `scale_` and press the tab key. You will see functions giving fine-grained controls over various scales (x, y, color, etc). These allow transformations (eg log10), and manually specified breaks (labelled values). Very fine grained control is possible over the appearance of ggplots, see the ggplot2 documentation for details and further examples.

3.4.1 Challenge: refine your ggplot

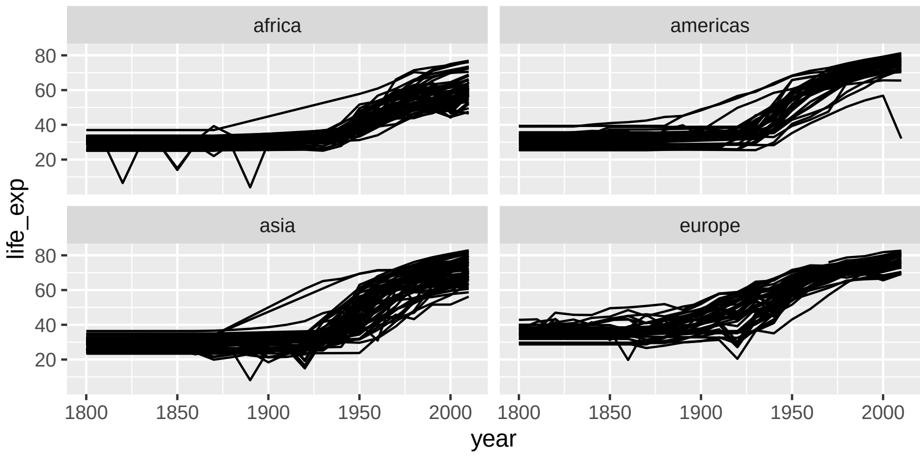
Continuing with your scatter-plot of the 2010 data, add axis labels to your plot.

Give your x axis a log scale by adding `scale_x_log10()`.

3.5 Faceting

Faceting lets us quickly produce a collection of small plots. The plots all have the same scales and the eye can easily compare them.

```
ggplot(gap_geo, aes(x=year, y=life_exp, group=name)) +
    geom_line() +
    facet_wrap(~ region)
```



Note the use of `~`, which we've not seen before. `~` syntax is used in R to specify dependence on some set of variables, for example when specifying a linear model. Here the information in each plot is dependent on the continent.

3.5.1 Challenge: facet your ggplot

Let's return again to your scatter-plot of the 2010 data.

Adjust your plot to now show data from all years, with each year shown in a separate facet, using `facet_wrap(~ year)`.

Advanced: Highlight Australia in your plot.

3.6 Saving ggplots

The act of plotting a ggplot is actually triggered when it is printed. In an interactive session we are automatically printing each value we calculate, but if you are using it with a programming construct such as a for loop or function you might need to explcitly `print( )` the plot.

Ggplots can be saved using `ggsave`.

```
# Plot created but not shown.
p <- ggplot(gap_geo, aes(x=year, y=life_exp)) + geom_point()

# Only when we try to look at the value p is it shown
p

# Alternatively, we can explicitly print it
print(p)

# To save to a file
ggsave("test.png", p)


# This is an alternative method that works with "base R" plots as well:
png("test.png")
print(p)
dev.off()
```

3.6.1 Tip about sizing

Figures in papers tend to be quite small. This means text must be proportionately larger than we usually show on screen. Dots should also be proportionately larger, and lines proportionately thicker. The way to achieve this using `ggsave` is to specify a small width and height, given in inches. To ensure the output also has good resolution, specify a high dots-per-inch, or use a vector-graphics format such as PDF or SVG.

```
ggsave("test2.png", p, width=3, height=3, dpi=600)
```

Chapter 4

Summarizing data

Having loaded and thoroughly explored a data set, we are ready to distill it down to concise conclusions. At its simplest, this involves calculating summary statistics like counts, means, and standard deviations. Beyond this is the fitting of models, and hypothesis testing and confidence interval calculation. R has a huge number of packages devoted to these tasks and this is a large part of its appeal, but is beyond the scope of today.

Loading the data as before, if you have not already done so:

```
library(tidyverse)

geo <- read_csv("r-intro-2-files/geo.csv")
gap <- read_csv("r-intro-2-files/gap-minder.csv")
gap_geo <- left_join(gap, geo, by="name")
```

4.1 Summary functions

R has a variety of functions for summarizing a vector, including: `sum`, `mean`, `min`, `max`, `median`, `sd`.

```
mean( c(1,2,3,4) )
```

```
## [1] 2.5
```

We can use these on the Gapminder data.

```
gap2010 <- filter(gap_geo, year == 2010)
sum(gap2010$population)
```

```
## [1] 6949495061
```

```
mean(gap2010$life_exp)
```

```
## [1] NA
```

4.2 Missing values

Why did `mean` fail? The reason is that `life_exp` contains missing values (`NA`).

```
gap2010$life_exp
```

```
##   [1] 56.20 76.31 76.55 82.66 60.08 76.85 75.82 73.34 81.98 80.50 69.13
##  [12] 73.79 76.03 70.39 76.68 70.43 79.98 71.38 61.82 72.13 71.64 76.75
##  [23] 57.06 74.19 77.08 73.86 57.89 57.73 66.12 57.25 81.29 72.45 47.48
##  [34] 56.49 79.12 74.59 76.44 65.93 57.53 60.43 80.40 56.34 76.33 78.39
##  [45] 79.88 77.47 79.49 63.69 73.04 74.60 76.72 70.52 74.11 60.93 61.66
##  [56] 76.00 61.30 65.28 80.00 81.42 62.86 65.55 72.82 80.09 62.16 80.41
##  [67] 71.34 71.25 57.99 55.65 65.49 32.11 71.58 82.61 74.52 82.03 66.20
##  [78] 69.90 74.45 67.24 80.38 81.42 81.69 74.66 82.85 75.78 68.37 62.76
##  [89] 60.73 70.10 80.13 78.20 68.45 63.80 73.06 79.85 46.50 60.77 76.10
## [100]    NA 73.17 81.35 74.01 60.84 53.07 74.46 77.91 59.46 80.28 63.72
## [111] 68.23 73.42 75.47 65.38 69.74    NA 66.18 76.36 73.55 54.48 66.84
## [122] 58.60    NA 68.26 80.73 80.90 77.36 58.78 60.53 81.04 76.09 65.33
## [133]    NA 77.85 58.70 74.07 77.92 69.03 76.30 79.84 79.52 73.66 69.24
## [144] 64.59    NA 75.48 71.64 71.46    NA 68.91 75.13 64.01 74.65 73.38
## [155] 55.05 82.69 75.52 79.45 61.71 53.13 54.27 81.94 74.42 66.29 70.32
## [166] 46.98 81.52 82.21 76.15 79.19 69.61 59.30 76.57 71.10 58.74 69.86
## [177] 72.56 76.89 78.21 67.94    NA 56.81 70.41 76.51 80.34 78.74 76.36
## [188] 68.77 63.02 75.41 72.27 73.07 67.51 52.02 49.57 58.13
```

R will not ignore these unless we explicitly tell it to with `na.rm=TRUE`.

```
mean(gap2010$life_exp, na.rm=TRUE)
```

```
## [1] 70.34005
```

Ideally we should also use `weighted.mean` here, to take population into account.

```
weighted.mean(gap2010$life_exp, gap2010$population, na.rm=TRUE)
```

```
## [1] 70.96192
```

`NA` is a special value. If we try to calculate with `NA`, the result is `NA`

```
NA + 1
```

```
## [1] NA
```

`is.na` can be used to detect `NA` values, or `na.omit` can be used to directly remove rows of a data frame containing them.

```
is.na( c(1,2,NA,3) )
```

```
## [1] FALSE FALSE  TRUE FALSE
```

```
cleaned <- filter(gap2010, !is.na(life_exp))
weighted.mean(cleaned$life_exp, cleaned$population)
```

```
## [1] 70.96192
```

4.3 Grouped summaries

The `summarize` function in `dplyr` allows summary functions to be applied to data frames.

```
summarize(gap2010, mean_life_exp=weighted.mean(life_exp, population, na.rm=TRUE))
```

```
## # A tibble: 1 x 1
##   mean_life_exp
##           <dbl>
## 1          71.0
```

So far unremarkable, but `summarize` comes into its own when the `group_by` "adjective" is used.

```
summarize(
    group_by(gap_geo, year),
    mean_life_exp=weighted.mean(life_exp, population, na.rm=TRUE))
```

```
## # A tibble: 22 x 2
##     year mean_life_exp
##    <dbl>         <dbl>
##  1  1800          30.9
##  2  1810          31.1
##  3  1820          31.2
##  4  1830          31.4
##  5  1840          31.4
##  6  1850          31.6
##  7  1860          30.3
##  8  1870          31.5
##  9  1880          32.0
## 10  1890          32.5
## # ... with 12 more rows
```

Challenge: summarizing

What is the total population for each year? Plot the result.

Advanced: What is the total GDP for each year? For this you will first need to calculate GDP per capita times the population of each country.

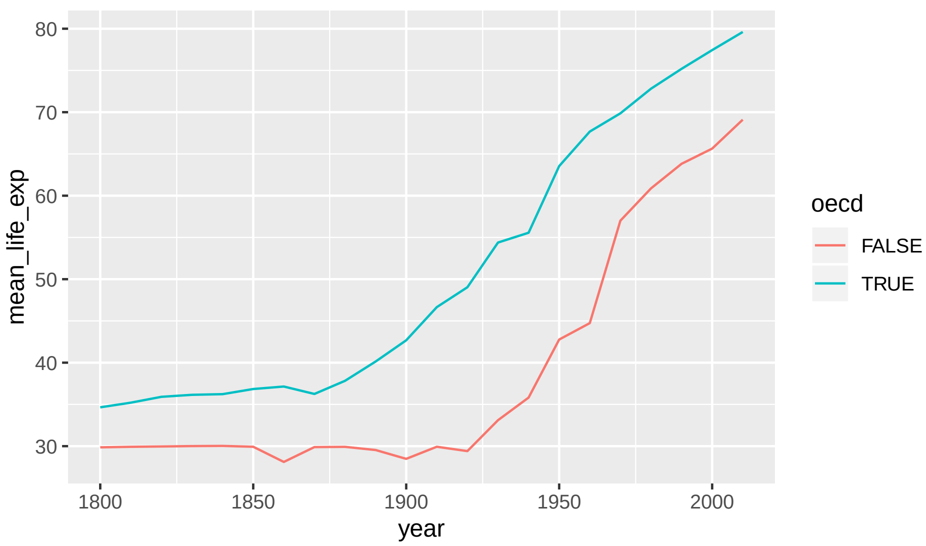
`group_by` can be used to group by multiple columns, much like `count`. We can use this to see how the rest of the world is catching up to OECD nations in terms of life expectancy.

```
result <- summarize(
    group_by(gap_geo,year,oecd),
```

```
    mean_life_exp=weighted.mean(life_exp, population, na.rm=TRUE))
result
```

```
## # A tibble: 44 x 3
## # Groups:   year [22]
##     year oecd  mean_life_exp
##    <dbl> <lgl>         <dbl>
##  1  1800 FALSE          29.9
##  2  1800 TRUE           34.7
##  3  1810 FALSE          29.9
##  4  1810 TRUE           35.2
##  5  1820 FALSE          30.0
##  6  1820 TRUE           35.9
##  7  1830 FALSE          30.0
##  8  1830 TRUE           36.2
##  9  1840 FALSE          30.0
## 10  1840 TRUE           36.2
## # ... with 34 more rows
```

```
ggplot(result, aes(x=year,y=mean_life_exp,color=oecd)) + geom_line()
```



A similar plot could be produced using `geom_smooth`. Differences here are that we have full control over the summarization process so we were able to use the exact summarization method we want (`weighted.mean` for each year), and we have access to the resulting numeric data as well as the plot. We have reduced a large data set down to a smaller one that distills out one of the stories present in this data. However the earlier visualization and exploration activity using `ggplot2` was essential. It gave us an idea of what sort of variability was present in the data, and any unexpected issues the data might have.

4.4 t-test

We will finish this section by demonstrating a t-test. The main point of this section is to give a flavour of how statistical tests work in R, rather than the details of what a t-test does.

Has life expectancy increased from 2000 to 2010?

```
gap2000 <- filter(gap_geo, year == 2000)
gap2010 <- filter(gap_geo, year == 2010)

t.test(gap2010$life_exp, gap2000$life_exp)
```

```
##
##  Welch Two Sample t-test
##
## data:  gap2010$life_exp and gap2000$life_exp
## t = 3.0341, df = 374.98, p-value = 0.002581
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  1.023455 4.792947
## sample estimates:
## mean of x mean of y
##  70.34005  67.43185
```

Statistical routines often have many ways to tweak the details of their operation. These are specified by further arguments to the function call, to override the default behaviour. By default, `t.test` performs an unpaired t-test, but these are repeated observations of the same countries. We can specify `paired=TRUE` to `t.test` to perform a paired sample t-test and gain some statistical power. Check this by looking at the help page with `?t.test`.

It's important to first check that both data frames are in the same order.

```
all(gap2000$name == gap2010$name)
```

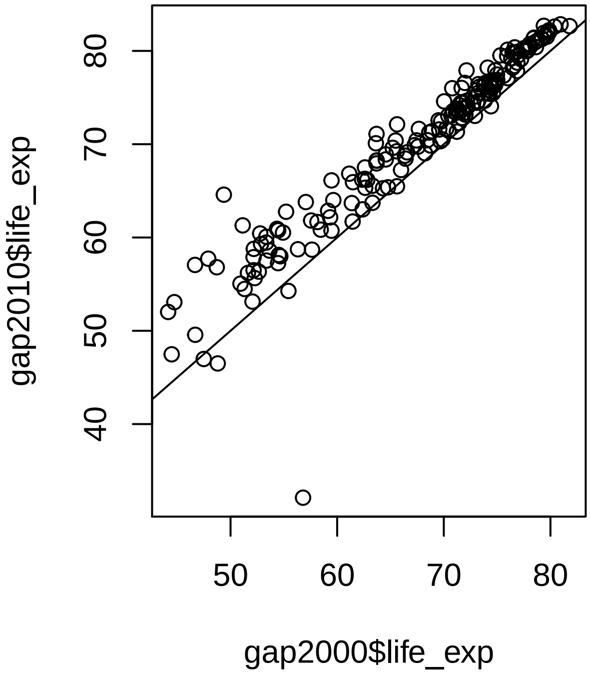
```
## [1] TRUE
```

```
t.test(gap2010$life_exp, gap2000$life_exp, paired=TRUE)
```

```
##
##  Paired t-test
##
## data:  gap2010$life_exp and gap2000$life_exp
## t = 13.371, df = 188, p-value < 2.2e-16
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  2.479153 3.337249
## sample estimates:
## mean of the differences
##                2.908201
```

When performing a statistical test, it's good practice to visualize the data to make sure there is nothing funny going on.

```
plot(gap2000$life_exp, gap2010$life_exp)
abline(0,1)
```



This is a visual confirmation of the t-test result. If there were no difference between the years then points would lie approximately evenly above and below the diagonal line, which is clearly not the case. However the outlier may warrant investigation.

Chapter 5

Thinking in R

The result of a t-test is actually a value we can manipulate further. Two functions help us here. `class` gives the "public face" of a value, and `typeof` gives its underlying type, the way R thinks of it internally. For example numbers are "numeric" and have some representation in computer memory, either "integer" for whole numbers only, or "double" which can hold fractional numbers (stored in memory in a base-2 version of scientific notation).

```
class(42)
```

```
## [1] "numeric"
```

```
typeof(42)
```

```
## [1] "double"
```

Let's look at the result of a t-test:

```
result <- t.test(gap2010$life_exp, gap2000$life_exp, paired=TRUE)

class(result)
```

```
## [1] "htest"
```

```
typeof(result)
```

```
## [1] "list"
```

```
names(result)
```

```
## [1] "statistic"   "parameter"   "p.value"     "conf.int"    "estimate"
## [6] "null.value"  "alternative" "method"      "data.name"
```

```
result$p.value
```

```
## [1] 4.301261e-29
```

In R, a t-test is just another function returning just another type of data, so it can also be a building block. The value it returns is a special type of vector called a "list", but with a public face that presents itself nicely. This is a common pattern in R. Besides printing to the console nicely, this public face may alter the behaviour of generic functions such as `plot` and `summary`.

Similarly a data frame is a list of vectors that is able to present itself nicely.

5.1 Lists

Lists are vectors that can hold anything as elements (even other lists!). It's possible to create lists with the `list` function. This becomes especially useful once you get into the programming side of R. For example writing your own function that needs to return multiple values, it could do so in the form of a list.

```
mylist <- list(hello=c("Hello","world"), numbers=c(1,2,3,4))
mylist
```

```
## $hello
## [1] "Hello" "world"
##
## $numbers
## [1] 1 2 3 4
```

```
class(mylist)
```

```
## [1] "list"
```

```
typeof(mylist)
```

```
## [1] "list"
```

```
names(mylist)
```

```
## [1] "hello"   "numbers"
```

Accessing lists can be done by name with `$` or by position with `[[ ]]`.

```
mylist$hello
```

```
## [1] "Hello" "world"
```

```
mylist[[2]]
```

```
## [1] 1 2 3 4
```

5.2 Other types not covered here

Matrices are another tabular data type. These come up when doing more mathematical tasks in R. They are also commonly used in bioinformatics, for example to represent RNA-Seq count data. A matrix, as compared to a data frame:

- contains only one type of data, usually numeric (rather than different types in different columns).

- commonly has `rownames` as well as `colnames`. (Base R data frames can have `rownames` too, but it is easier to have any unique identifier as a normal column instead.)
- has individual cells as the unit of observation (rather than rows).

Matrices can be created using `as.matrix` from a data frame, `matrix` from a single vector, or using `rbind` or `cbind` with several vectors.

You may also encounter "S4 objects", especially if you use Bioconductor[1] packages. The syntax for using these is different again, and uses `@` to access elements.

5.3 Programming

Once you have a useful data analysis, you may want to do it again with different data. You may have some task that needs to be done many times over. This is where programming comes in:

- Writing your own functions[2].
- For-loops[3] to do things multiple times.
- If-statements[4] to make decisions.

The "R for Data Science" book[5] is an excellent source to learn more. Monash Data Fluency "Programming and Tidy data analysis in R" course[6] also covers this.

[1] http://bioconductor.org/
[2] http://r4ds.had.co.nz/functions.html
[3] http://r4ds.had.co.nz/iteration.html
[4] http://r4ds.had.co.nz/functions.html#conditional-execution
[5] http://r4ds.had.co.nz/
[6] https://monashdatafluency.github.io/r-progtidy/

Chapter 6

Next steps

6.1 Deepen your understanding

Our number one recommendation is to read the book "R for Data Science"[1] by Garrett Grolemund and Hadley Wickham.

Also, statistical tasks such as model fitting, hypothesis testing, confidence interval calculation, and prediction are a large part of R, and one we haven't demonstrated fully today. Linear models, and the linear model formula syntax `~`, are core to much of what R has to offer statistically. Many statistical techniques take linear models as their starting point, including limma[2] for differential gene expression, `glm` for logistic regression (etc), survival analysis with `coxph`, and mixed models to characterize variation within populations.

- "Statistical Models in S" by J.M. Chambers and T.J. Hastie is the primary reference for this, although there are some small differences between R and its predecessor S.
- "An Introduction to Statistical Learning"[3] by G. James, D. Witten, T. Hastie and R. Tibshirani can be seen as further development of the ideas in "Statistical Models in S", and is available online. It has more of a machine learning than a statistics flavour to it (the distinction is fuzzy!).
- "Modern Applied Statistics with S" by W.N. Venable and B.D. Ripley is a well respected reference covering R and S.
- "Linear Models with R" and "Extending the Linear Model with R" by J. Faraway[4] cover linear models, with many practical examples.

[1] http://r4ds.had.co.nz/
[2] https://bioconductor.org/packages/release/bioc/html/limma.html
[3] http://www-bcf.usc.edu/~gareth/ISL/
[4] http://www.maths.bath.ac.uk/~jjf23/

6.2 Expand your vocabulary

Have a look at these cheat sheets to see what is possible with R.

- RStudio's collection of cheat sheets[5] cover newer packages in R.
- An old-school cheat sheet[6] for dinosaurs and people wishing to go deeper.
- A Bioconductor cheat sheet[7] for biological data.

The R Manuals[8] are the place to look if you need a precise definition of how R behaves.

6.3 Join the community

Join the Data Fluency community at Monash[9].

- Mailing list for workshop and event announcements.
- Slack for discussion.
- Monthly seminars on Data Science topics.
- Drop-in sessions on Friday afternoon.

Meetups in Melbourne:

- MelbURN[10]
- R-Ladies[11]

The Carpentries[12] run intensive two day workshops on scientific computing and data science topics worldwide. The style of this present workshop is very much based on theirs. For bioinformatics, COMBINE[13] is an Australian student and early career researcher organization, and runs Carpentries workshops and similar.

[5] https://www.rstudio.com/resources/cheatsheets/
[6] https://cran.r-project.org/doc/contrib/Short-refcard.pdf
[7] https://github.com/mikelove/bioc-refcard/blob/master/README.Rmd
[8] https://cran.r-project.org/manuals.html
[9] https://www.monash.edu/data-fluency
[10] https://www.meetup.com/en-AU/MelbURN-Melbourne-Users-of-R-Network/
[11] https://www.meetup.com/en-AU/R-Ladies-Melbourne/
[12] https://carpentries.org/
[13] https://combine.org.au/