Contents

Introduction to R

These are course notes for the "Introduction to R" course given by the Monash Bioinformatics Platform[1] for the Proteomics Pre-Lorne workshop. Our teaching style is based on the style of The Carpentries[2]. This course is focussing on the modern Tidyverse[3] set of packages. We believe this is currently the quickest route to being productive in R.

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

During the workshop we will be using the RStudio Cloud to use R over the web:

- RStudio Cloud[6]

You can also install R on your own computer. There are two things to download and install:

- Download R[7]
- Download RStudio[8]

R is the language itself. RStudio provides a convenient environment in which to use R, either on your local computer or on a server.

Source code

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

[1] https://www.monash.edu/researchinfrastructure/bioinformatics
[2] https://carpentries.org/
[3] https://www.tidyverse.org/
[4] https://monashbioinformaticsplatform.github.io/Proteomics-Intro-R-Workshop-2020/r-intro.pdf
[5] https://monashbioinformaticsplatform.github.io/Proteomics-Intro-R-Workshop-2020/r-intro-files.zip
[6] https://rstudio.cloud/
[7] https://cran.rstudio.com/
[8] https://www.rstudio.com/products/rstudio/download/

- GitHub page[9]

Authors and copyright

This course is developed for the Monash Bioinformatics Platform by Paul Harrison, Anup Shah & Adele Barugahare.

[9] https://github.com/MonashBioinformaticsPlatform/Proteomics-Intro-R-Workshop-2020/

[10] http://creativecommons.org/licenses/by/4.0/

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

Chapter 1

Starting out in R

R is both a programming language and an interactive environment for data exploration and statistics. Today we will be concentrating on R as an *interactive environment*.

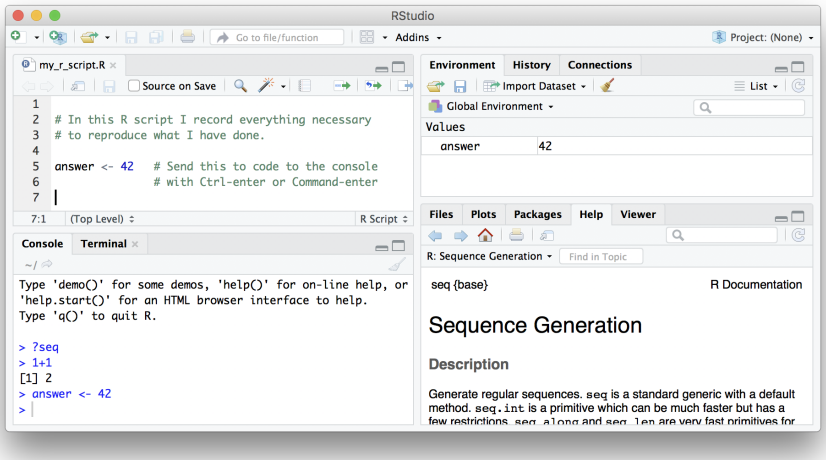
Working with R is primarily text-based. The basic mode of use for R is that the user types in a command in the R language and presses enter, and then R computes and displays the result.

We will be working in RStudio[1]. The easiest way to get started is to go to RStudio Cloud[2] and create a new project. Monash staff and students can log in using their Monash Google account.

The main way of working with R is the *console*, where you enter commands and view results. RStudio surrounds this with various conveniences. In addition to the console panel, RStudio provides panels containing:

- A text editor, where R commands can be recorded for future reference.
- A history of commands that have been typed on the console.
- An "environment" pane with a list of *variables*, which contain values that R has been told to save from previous commands.
- A file manager.
- Help on the functions available in R.
- A panel to show plots.

[1] https://www.rstudio.com/products/rstudio/download/
[2] https://rstudio.cloud/



Open RStudio, click on the "Console" pane, type `1+1` and press enter. R displays the result of the calculation. In this document, we will show such an interaction with R as below.

```
1+1
```

```
## [1] 2
```

`+` is called an operator. R has the operators you would expect for for basic mathematics: `+ - * / ^`. It also has operators that do more obscure things.

`*` has higher precedence than `+`. We can use brackets if necessary `( )`. Try `1+2*3` and `(1+2)*3`.

Spaces can be used to make code easier to read.

We can compare with `== < > <= >=`. This produces a *logical* value, `TRUE` or `FALSE`. Note the double equals, `==`, for equality comparison.

```
2 * 2 == 4
```

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

There are also character strings such as `"string"`. A character string must be surrounded by either single or double quotes.

1.1 Variables

A variable is a name for a value. We can create a new variable by assigning a value to it using `<-`.

```
width <- 5
```

RStudio helpfully shows us the variable in the "Environment" pane. We can also print it by typing the name of the variable and hitting enter. In general, R

will print to the console any object returned by a function or operation *unless* we assign it to a variable.

```
width
```

```
## [1] 5
```

Examples of valid variables names: `hello`, `subject_id`, `subject.ID`, `x42`. Spaces aren't ok *inside* variable names. Dots (`.`) are ok in R, unlike in many other languages. Numbers are ok, except as the first character. Punctuation is not allowed, with two exceptions: `_` and `..`

We can do arithmetic with the variable:

```
# Area of a square
width * width
```

```
## [1] 25
```

and even save the result in another variable:

```
# Save area in "area" variable
area <- width * width
```

We can also change a variable's value by assigning it a new value:

```
width <- 10
width
```

```
## [1] 10
```

```
area
```

```
## [1] 25
```

Notice that the value of `area` we calculated earlier hasn't been updated. Assigning a new value to one variable does not change the values of other variables. This is different to a spreadsheet, but usual for programming languages.

1.2 Saving code in an R script

Once we've created a few variables, it becomes important to record how they were calculated so we can reproduce them later.

The usual workflow is to save your code in an R script (".R file"). Go to "File/New File/R Script" to create a new R script. Code in your R script can be sent to the console by selecting it or placing the cursor on the correct line, and then pressing **Control-Enter** (**Command-Enter** on a Mac).

Tip

Add comments to code, using lines starting with the `#` character. This makes it easier for others to follow what the code is doing (and also for us the next time we come back to it).

Challenge: using variables

1. Re-write this calculation so that it *doesn't* use variables:

```
a <- 4*20
b <- 7
a+b
```

2. Re-write this calcuation over multiple lines, using a variable:

```
2*2+2*2+2*2
```

1.3 Vectors

A *vector* of numbers is a collection of numbers. "Vector" means different things in different fields (mathematics, geometry, biology), but in R it is a fancy name for a collection of numbers. We call the individual numbers *elements* of the vector.

We can make vectors with `c( )`, for example `c(1,2,3)`. `c` means "combine". R is obsesssed with vectors, in R even single numbers are vectors of length one. Many things that can be done with a single number can also be done with a vector. For example arithmetic can be done on vectors as it can be on single numbers.

```
myvec <- c(10,20,30,40,50)
myvec
```

```
## [1] 10 20 30 40 50
```

```
myvec + 1
```

```
## [1] 11 21 31 41 51
```

```
myvec + myvec
```

```
## [1]  20  40  60  80 100
```

```
length(myvec)
```

```
## [1] 5
```

```
c(60, myvec)
```

```
## [1] 60 10 20 30 40 50
```

```
c(myvec, myvec)
```

```
##  [1] 10 20 30 40 50 10 20 30 40 50
```

When we talk about the length of a vector, we are talking about the number of numbers in the vector.

1.4 Types of vector

We will also encounter vectors of character strings, for example `"hello"` or `c("hello","world")`. Also we will encounter "logical" vectors, which contain `TRUE` and `FALSE` values. R also has "factors", which are categorical vectors, and behave much like character vectors (think the factors in an experiment).

Challenge: mixing types

Sometimes the best way to understand R is to try some examples and see what it does.

What happens when you try to make a vector containing different types, using `c( )`? Make a vector with some numbers, and some words (eg. character strings like `"test"`, or `"hello"`).

Why does the output show the numbers surrounded by quotes `" "` like character strings are?

Because vectors can only contain one type of thing, R chooses a lowest common denominator type of vector, a type that can contain everything we are trying to put in it. A different language might stop with an error, but R tries to soldier on as best it can. A number can be represented as a character string, but a character string can not be represented as a number, so when we try to put both in the same vector R converts everything to a character string.

1.5 Indexing vectors

Access elements of a vector with `[ ]`, for example `myvec[1]` to get the first element. You can also assign to a specific element of a vector.

```
myvec[1]
```

```
## [1] 10
```

```
myvec[2]
```

```
## [1] 20
```

```
myvec[2] <- 5
myvec
```

```
## [1] 10  5 30 40 50
```

Can we use a vector to index another vector? Yes!

```
myind <- c(4,3,2)
myvec[myind]
```

```
## [1] 40 30  5
```

We could equivalently have written:

```
myvec[c(4,3,2)]
```

```
## [1] 40 30  5
```

Challenge: indexing

We can create and index character vectors as well. A cafe is using R to create their menu.

```
items <- c("spam", "eggs", "beans", "bacon", "sausage")
```

1. What does `items[-3]` produce? Based on what you find, use indexing to create a version of `items` without `"spam"`.
2. Use indexing to create a vector containing spam, eggs, sausage, spam, and spam.
3. Add a new item, "lobster", to `items`.

1.6 Sequences

Another way to create a vector is with `:`:

```
1:10
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

This can be useful when combined with indexing:

```
items[1:4]
```

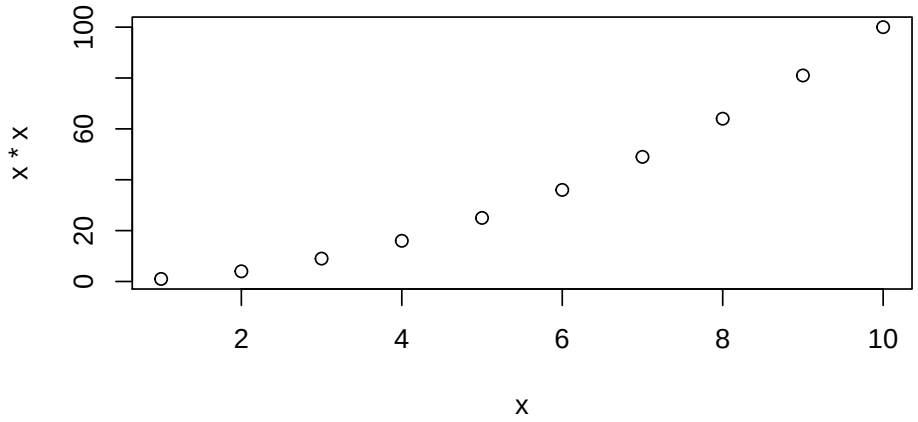
```
## [1] "spam"  "eggs"  "beans" "bacon"
```

Sequences are useful for other things, such as a starting point for calculations:

```
x <- 1:10
x*x
```

```
##  [1]   1   4   9  16  25  36  49  64  81 100
```

```
plot(x, x*x)
```



1.7 Functions

Functions are the things that do all the work for us in R: calculate, manipulate data, read and write to files, produce plots. R has many built in functions and we will also be loading more specialized functions from "packages".

We've already seen several functions: `c( )`, `length( )`, and `plot( )`. Let's now have a look at `sum( )`.

```
sum(myvec)
```

```
## [1] 135
```

We *called* the function `sum` with the *argument* `myvec`, and it *returned* the value 135. We can get help on how to use `sum` with:

```
?sum
```

Some functions take more than one argument. Let's look at the function `rep`, which means "repeat", and which can take a variety of different arguments. In the simplest case, it takes a value and the number of times to repeat that value.

```
rep(42, 10)
```

```
##  [1] 42 42 42 42 42 42 42 42 42 42
```

As with many functions in R—which is obsessed with vectors—the thing to be repeated can be a vector with multiple elements.

```
rep(c(1,2,3), 10)
```

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

So far we have used *positional* arguments, where R determines which argument is which by the order in which they are given. We can also give arguments by *name*. For example, the above is equivalent to

```
rep(c(1,2,3), times=10)
```

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

```
rep(x=c(1,2,3), 10)
```

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

```
rep(times=10, x=c(1,2,3))
```

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

Arguments can have default values, and a function may have many different possible arguments that make it do obscure things. For example, `rep` can also take an argument `each=`. It's typical for a function to be invoked with some number of positional arguments, which are always given, plus some less commonly used arguments, typically given by name.

```
rep(c(1,2,3), each=3)
```

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

```
rep(c(1,2,3), each=3, times=5)
```

```
##  [1] 1 1 1 2 2 2 3 3 3 1 1 1 2 2 2 3 3 3 1 1 1 2 2 2 3 3 3 1 1 1 2 2 2 3 3 3 1 1
## [39] 1 2 2 2 3 3 3
```

Challenge: using functions

1. Use `sum` to sum from 1 to 10,000.
2. Look at the documentation for the `seq` function. What does `seq` do? Give an example of using `seq` with either the `by` or `length.out` argument.

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://monashbioinformaticsplatform.github.io/Proteomics-Intro-R-Workshop-2020/r-intro
  destfile="r-intro-files.zip")
unzip("r-intro-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-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:

```
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
```

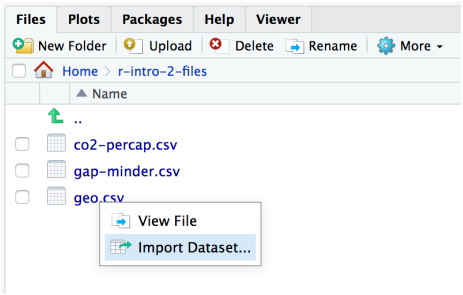
The argument names become column names in the data frame.

Tip

The *path* to the file on our server is `"r-intro-files/geo.csv"`. This says, starting from your working directory, look in the directory `r-intro-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"
```

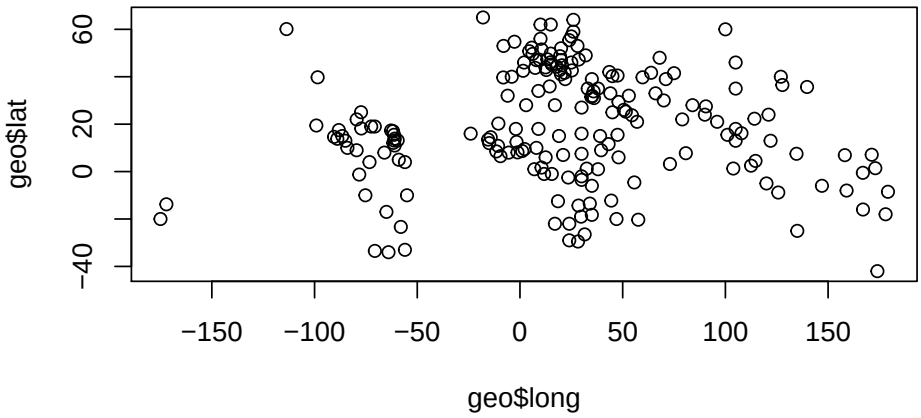
To get the "region" value of the 4th row as above, but unwrapped, we can use:

```
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 americas FALSE TRUE   17.0 -61.8  high       FALSE
##  7 Argentina           americas 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:

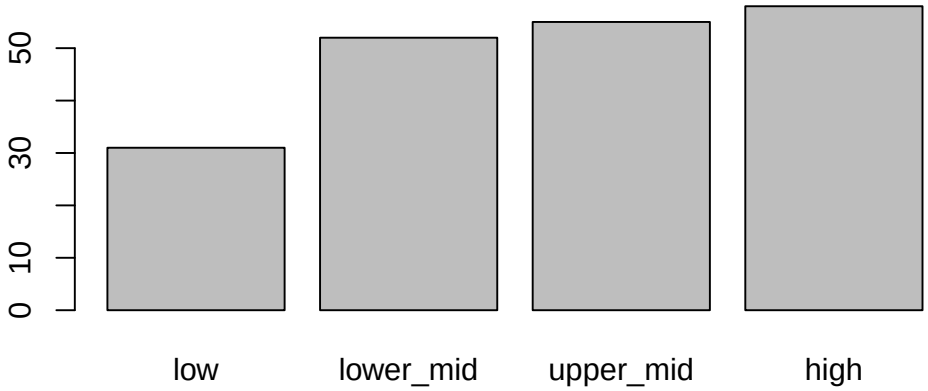
```
count(geo, income2017)
```

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

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

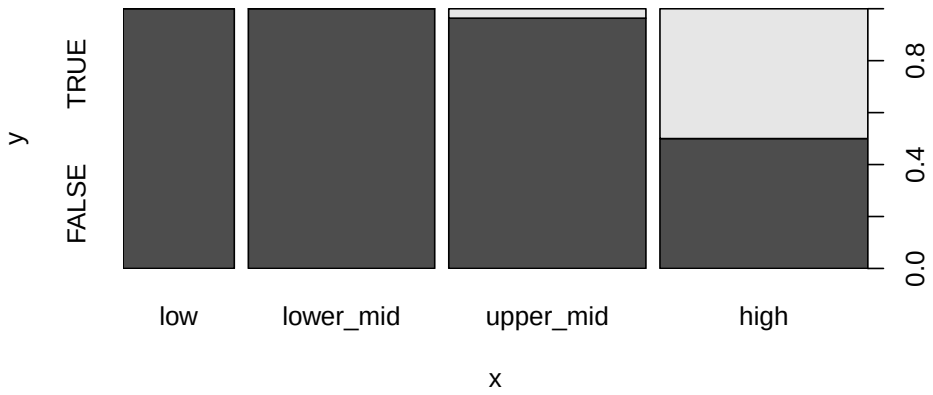
When `plot` is given a factor, it shows a bar plot:

```
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 `pivot_wider` function from `tidyr` to achieve this. (This is also sometimes also called a "cast" or a "spread".)

```
counts <- count(geo, income2017, oecd)
pivot_wider(counts, names_from=income2017, values_from=n)
```

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

We could further specify `values_fill=list(n=0)` to fill in the `NA` values with 0.

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 `pivot_wider` which spreads two columns into multiple columns. The inverse of `pivot_wider` is `pivot_longer`, which gathers multiple columns into two columns: a column of column names, and a column of values. `pivot_longer` is often the first step when tidying a dataset you have received from the wild. (This is sometimes also called a "melt" or a "gather".)

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-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   long
##    <chr> <dbl>      <dbl>      <dbl>    <dbl> <chr>  <lgl> <lgl> <dbl>  <dbl>
##  1 Afgh~  1800    3280000        603     28.2 asia   FALSE TRUE   33    66
##  2 Alba~  1800     410445        667     35.4 europe FALSE FALSE  41    20
##  3 Alge~  1800    2503218        715     28.8 africa FALSE TRUE   28     3
##  4 Ando~  1800       2654       1197     NA   europe FALSE FALSE  42.5   1.52
##  5 Ango~  1800    1567028        618     27.0 africa FALSE TRUE  -12.5  18.5
##  6 Anti~  1800      37000        757     33.5 ameri~ FALSE TRUE   17.0 -61.8
##  7 Arge~  1800     534000       1507     33.2 ameri~ FALSE TRUE  -34   -64
##  8 Arme~  1800     413326        514     34   europe FALSE FALSE  40.2  45
##  9 Aust~  1800     351014        814     34.0 asia   TRUE  FALSE -25   135
## 10 Aust~  1800    3205587       1847     34.4 europe TRUE  FALSE  47.3  13.3
## # ... with 4,302 more rows, and 2 more variables: 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-files/geo.csv")
gap <- read_csv("r-intro-files/gap-minder.csv")
gap_geo <- left_join(gap, geo, by="name")
```

3.1 Elements of a ggplot

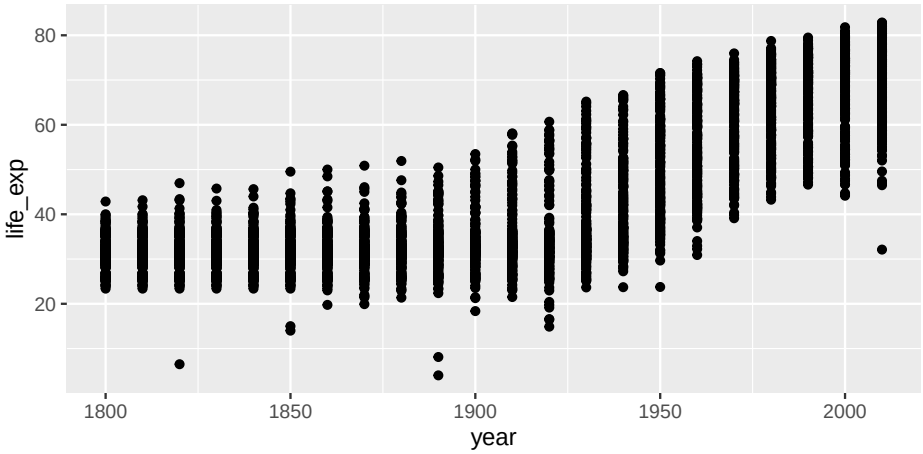
Producing a plot with `ggplot2`, we must give three things:

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()
```

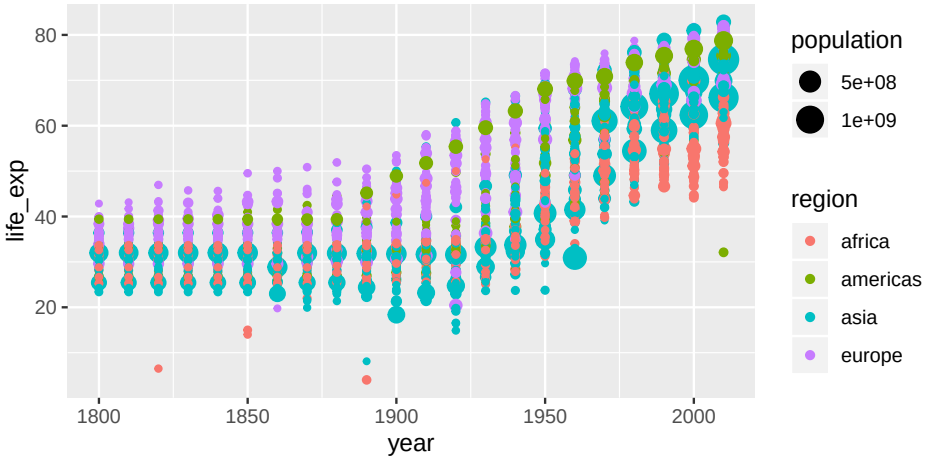
[1] https://www.amazon.com/Grammar-Graphics-Statistics-Computing/dp/0387245448
[2] http://ggplot2.tidyverse.org/reference/



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)
```

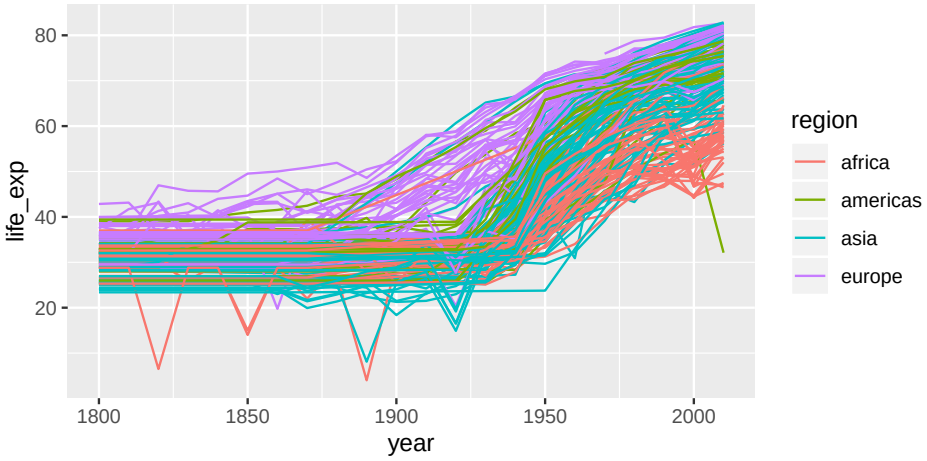
Create a ggplot of this with:

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

3.2 Further geoms

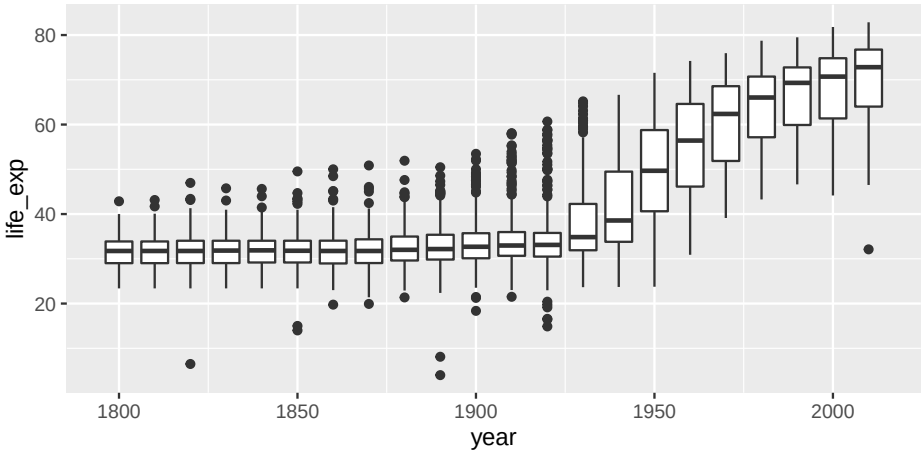
To draw lines, we need to use a "group" aesthetic.

```
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.

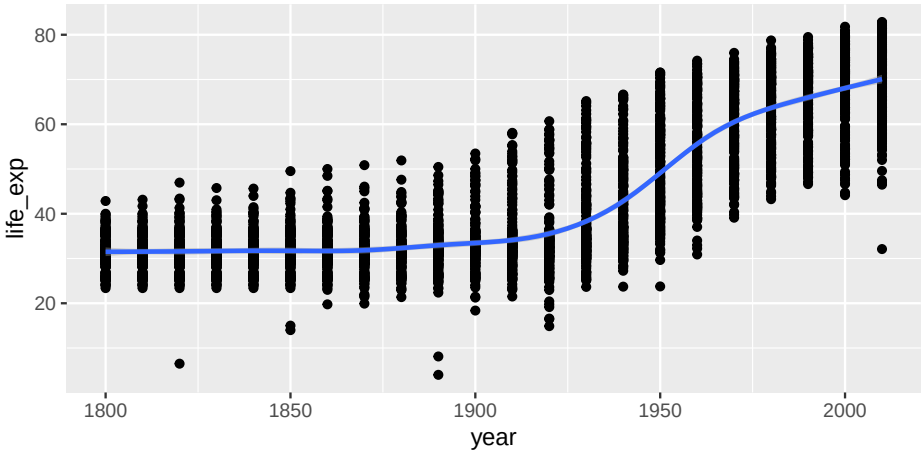
```
ggplot(gap_geo, aes(x=year, y=life_exp, group=year)) +
    geom_boxplot()
```



`geom_smooth` can be used to show trends.

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

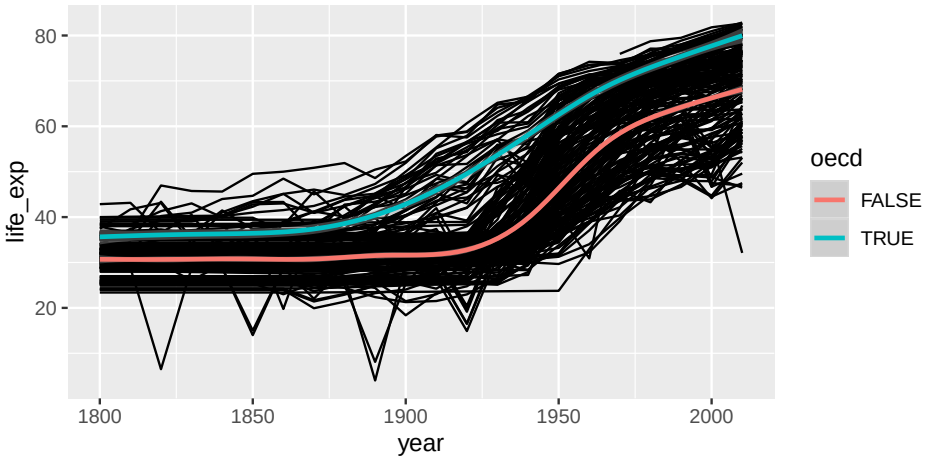
```
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```



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")'
```

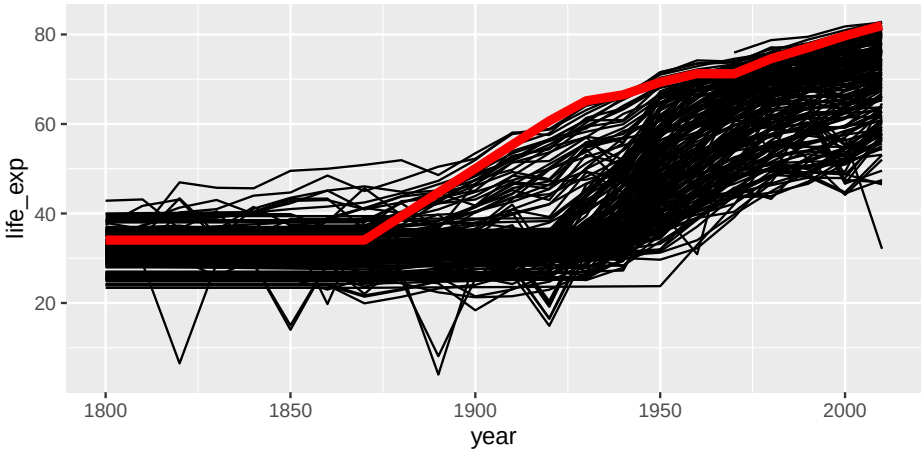


3.3 Highlighting subsets

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)
```

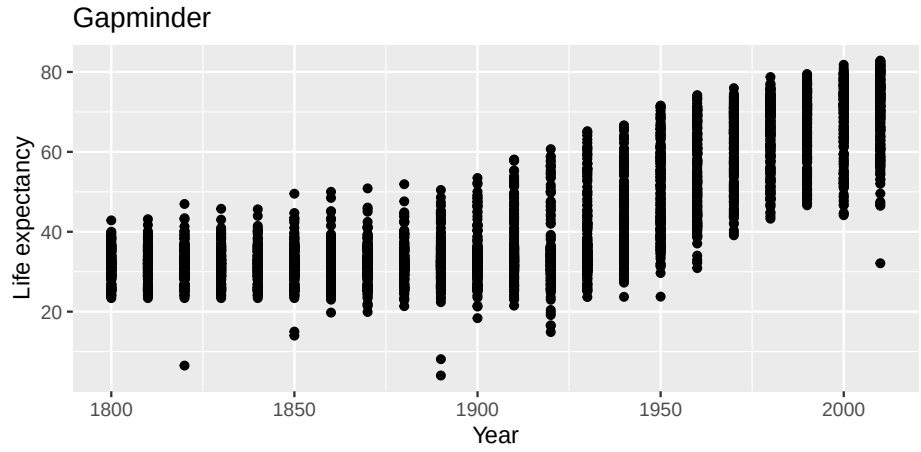


Notice also that the second `geom_line` has some further arguments controlling its appearance. These are **not** aesthetics, they are not a mapping of data to appearance, but rather a direct specification of the appearance. There isn't an associated scale as when color was an aesthetic.

3.4 Fine-tuning a plot

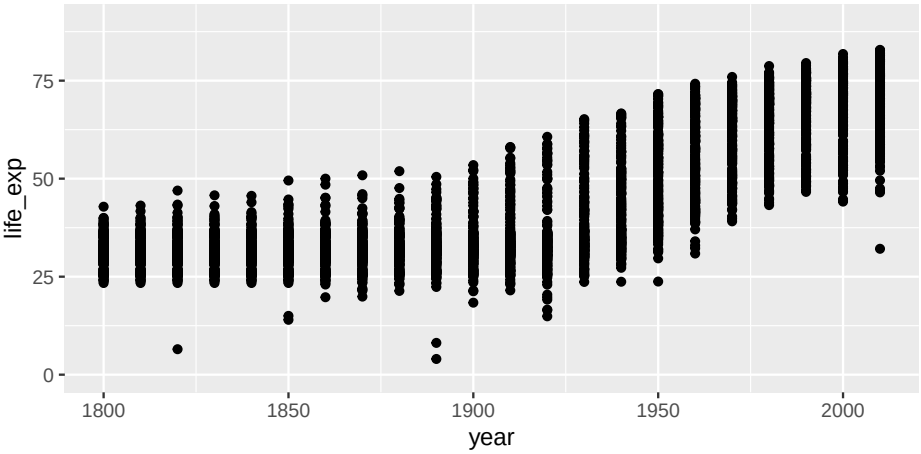
Adding `labs` to a ggplot adjusts the labels given to the axes and legends. A plot title can also be specified.

```
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

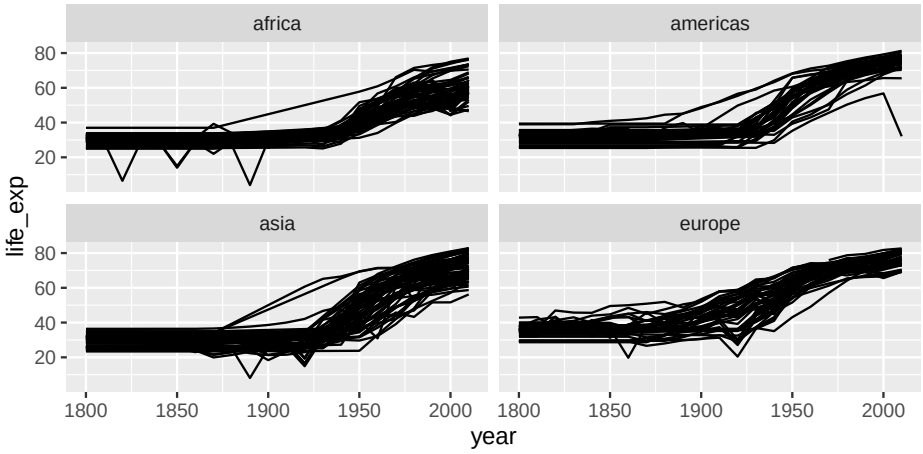
Continuing with your scatter-plot of the 2010 data, add axis labels to your plot.

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 shoum.
p <- ggplot(gap_geo, aes(x=year, y=life_exp)) + geom_point()

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

Proteomics Data Viz

4.1 Visualizing Proteomics data with ggplot2

Last session we worked on Cross-linking Mass Spectrometry Data. The data consisted of interaction between 300 yeast nuclear proteins. We also learned about concepts behind `Grammer of Graphics` and plotting using `ggplot2`.

In this tutorial, we will be working on the data from Yeast Nuclear Protein interaction study using Cross-linking Mass Spectrometry.

```
library(tidyverse)
```

We continue with the Cross-linking Proteomics dataset from Cytoscape tutorial. The data was in excel, therefore we first converted it into `Comma separated file (csv)` format.

4.1.1 Optional: Reading excel file into R

If you want to load the excel files files directly to `R` then you can use another library `readxl`.

```
library(readxl)
nuclear_xl_ms_excel <- readxl::read_excel("r-intro-files/Nuclear_XL_MS.xlsx")
head(nuclear_xl_ms_excel)
```

```
## # A tibble: 6 x 8
##   Protein1 Protein2 NameProtein1 NameProtein2 PPINovelty PPIEvidenceInfo~
##   <chr>    <chr>    <chr>        <chr>        <chr>      <chr>
## 1 P02293   P04911   H2B1         H2A1         Known      Structure
## 2 P02293   P02309   H2B1         H4           Known      Structure
## 3 P02994   P32471   EF1A         EF1B         Known      Structure
## 4 P0CX51   P38011   RS16A        GBLP         Known      Structure
## 5 P02406   P0CX49   RL28         RL18A        Novel      STRING
## 6 P33297   P53549   PRS6A        PRS10        Known      Structure
## # ... with 2 more variables: TotalNumberOfCSMs <dbl>,
```

```
## #   NumberUniqueLysLysContacts <dbl>
```

4.2 Exploring the data

```
library(readxl)
nuclear_xl_ms <- read_csv("r-intro-files/Nuclear_XL_MS.csv")
head(nuclear_xl_ms)
```

```
## # A tibble: 6 x 8
##   Protein1 Protein2 NameProtein1 NameProtein2 PPINovelty PPIEvidenceInfo~
##   <chr>    <chr>    <chr>        <chr>        <chr>      <chr>
## 1 P02293   P04911   H2B1         H2A1         Known      Structure
## 2 P02293   P02309   H2B1         H4           Known      Structure
## 3 P02994   P32471   EF1A         EF1B         Known      Structure
## 4 P0CX51   P38011   RS16A        GBLP         Known      Structure
## 5 P02406   P0CX49   RL28         RL18A        Novel      STRING
## 6 P33297   P53549   PRS6A        PRS10        Known      Structure
## # ... with 2 more variables: TotalNumberOfCSMs <dbl>,
## #   NumberUniqueLysLysContacts <dbl>
```

Now lets examine the dataset with two base R functions `str` and `summary`

```
str(nuclear_xl_ms)
```

```
## Classes 'spec_tbl_df', 'tbl_df', 'tbl' and 'data.frame': 228 obs. of  8 variables:
##  $ Protein1                 : chr  "P02293" "P02293" "P02994" "P0CX51" ...
##  $ Protein2                 : chr  "P04911" "P02309" "P32471" "P38011" ...
##  $ NameProtein1             : chr  "H2B1" "H2B1" "EF1A" "RS16A" ...
##  $ NameProtein2             : chr  "H2A1" "H4" "EF1B" "GBLP" ...
##  $ PPINovelty               : chr  "Known" "Known" "Known" "Known" ...
##  $ PPIEvidenceInfoGroup     : chr  "Structure" "Structure" "Structure" "Structure" ...
##  $ TotalNumberOfCSMs        : num  36 21 20 13 13 12 12 12 11 10 ...
##  $ NumberUniqueLysLysContacts: num  12 6 5 1 2 3 2 3 3 1 ...
##  - attr(*, "spec")=
##   .. cols(
##   ..   Protein1 = col_character(),
##   ..   Protein2 = col_character(),
##   ..   NameProtein1 = col_character(),
##   ..   NameProtein2 = col_character(),
##   ..   PPINovelty = col_character(),
##   ..   PPIEvidenceInfoGroup = col_character(),
##   ..   TotalNumberOfCSMs = col_double(),
##   ..   NumberUniqueLysLysContacts = col_double()
##   .. )
```

```
summary(nuclear_xl_ms)
```

```
##    Protein1           Protein2         NameProtein1       NameProtein2
##  Length:228         Length:228         Length:228         Length:228
##  Class :character   Class :character   Class :character   Class :character
```

```
##  Mode  :character   Mode  :character   Mode  :character   Mode  :character
##
##
##
##   PPINovelty        PPIEvidenceInfoGroup TotalNumberOfCSMs
##  Length:228         Length:228           Min.   : 1.000
##  Class :character   Class :character     1st Qu.: 1.000
##  Mode  :character   Mode  :character     Median : 1.000
##                                          Mean   : 2.706
##                                          3rd Qu.: 3.000
##                                          Max.   :36.000
##  NumberUniqueLysLysContacts
##  Min.   : 1.000
##  1st Qu.: 1.000
##  Median : 1.000
##  Mean   : 1.342
##  3rd Qu.: 1.000
##  Max.   :12.000
```

While the data in `PPINovelty` and `PPIEvidenceInfoGroup` are characters/strings, they can also be thought of as categorical.

We will now look into the unique values for the PPINovelty and PPIEvidenceInfoGroup columns

```
nuclear_xl_ms$PPINovelty %>% unique()
```

```
## [1] "Known" "Novel"
```

```
nuclear_xl_ms$PPIEvidenceInfoGroup %>% unique()
```

```
## [1] "Structure"   "STRING"      "APID"        "Unexplained" "Genetic"
```

R has a class for categorical data known as factors. We can convert these columns to factors and provide an order to those categories (levels). By default, R will order the levels of a factor alphabetically but we can override this behaviour by defining the level order. Here we will order the Evidence based on Strength of evidence for the interaction

```
nuclear_xl_ms <- nuclear_xl_ms %>% mutate(
  PPINovelty = factor(PPINovelty),
  PPIEvidenceInfoGroup = factor(PPIEvidenceInfoGroup, levels = c("Structure","APID", "STRI
)
nuclear_xl_ms %>% str()
```

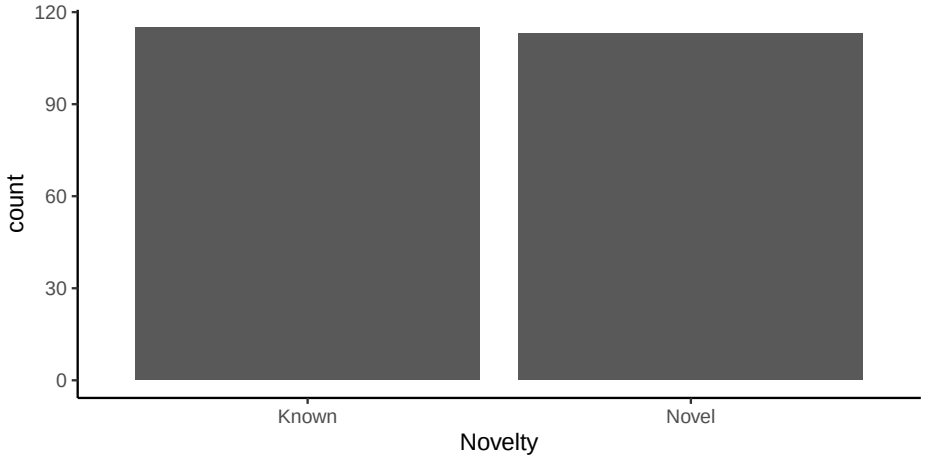
```
## Classes 'spec_tbl_df', 'tbl_df', 'tbl' and 'data.frame': 228 obs. of  8 variables:
##  $ Protein1                 : chr  "P02293" "P02293" "P02994" "P0CX51" ...
##  $ Protein2                 : chr  "P04911" "P02309" "P32471" "P38011" ...
##  $ NameProtein1             : chr  "H2B1" "H2B1" "EF1A" "RS16A" ...
##  $ NameProtein2             : chr  "H2A1" "H4" "EF1B" "GBLP" ...
##  $ PPINovelty               : Factor w/ 2 levels "Known","Novel": 1 1 1 1 2 1 2 1 1 1
##  $ PPIEvidenceInfoGroup     : Factor w/ 5 levels "Structure","APID",..: 1 1 1 1 3 1 3
##  $ TotalNumberOfCSMs        : num  36 21 20 13 13 12 12 12 11 10 ...
```

```
##  $ NumberUniqueLysLysContacts: num  12 6 5 1 2 3 2 3 3 1 ...
```

4.3 Plotting interactions types

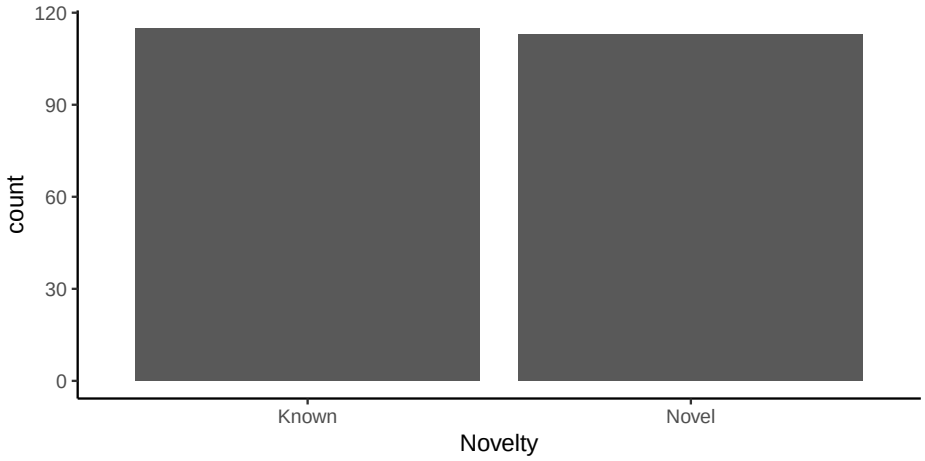
Firstly we will plot the number of Known and novel interactions using with `geom_bar`

```
ggplot(nuclear_xl_ms, aes(x = PPINovelty)) +
geom_bar() +
        xlab("Novelty") +
  theme_classic()
```



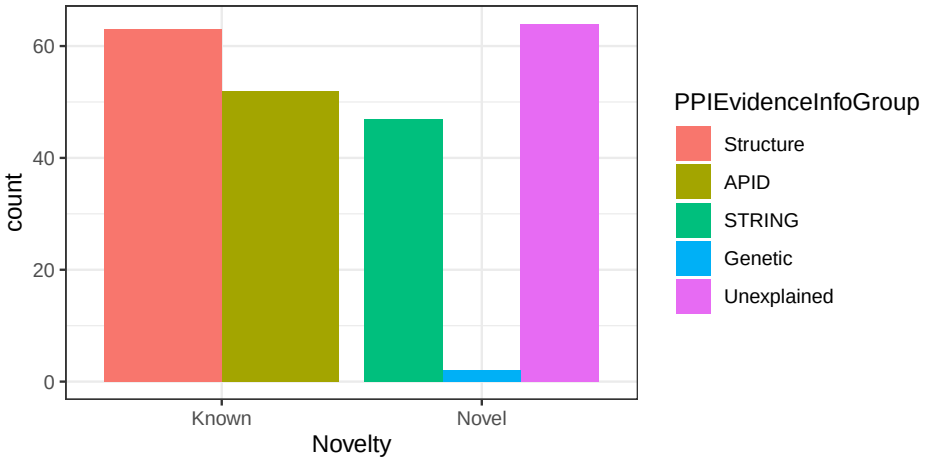
Now, we will rotate the bars to **Y-Axis** using `position="dodge"`

```
ggplot(nuclear_xl_ms, aes(x = PPINovelty)) +
geom_bar(position = "dodge") +
        xlab("Novelty") +
  theme_classic()
```



Next, step would be to create a stacked bar chart by adding **PPIEvidenceInfoGroup** data on top of each bar

```
ggplot(nuclear_xl_ms, aes(PPINovelty)) +
geom_bar(aes(fill=PPIEvidenceInfoGroup),
        position = "dodge") +
        xlab("Novelty") +
  theme_bw()
```



4.4 Individual Proteins

First we will calculate how many times a protein appeared in **NameProtein1** column using `table` function and then sorting by descending order. Next we will use `head()` function to print first five proteins with most observation.

```
table(nuclear_xl_ms$NameProtein1) %>% sort(.,decreasing = TRUE) %>% head()
```

```
##
##  H2B1  EF1A RL27A    H3  EF3A  ODP2
##    10     7     5     4     3     3
```

We could do same thing for **NameProtein2** column as well.

```
table(nuclear_xl_ms$NameProtein2)  %>% sort(.,decreasing = TRUE) %>% head()
```

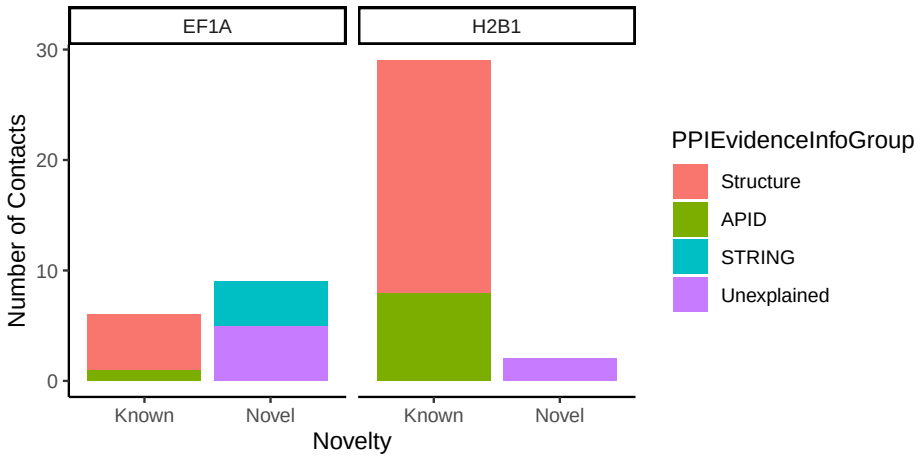
```
##
##    H3  RS15 NOP56  BFR1  PRS4 RL14A
##     6     5     4     3     3     3
```

Now we store the rows containing H2B1 and EF1A proteins in **NameProtein1** column in a datafame.

```
two_protein_df<- nuclear_xl_ms %>% filter(NameProtein1 %in% c("H2B1","EF1A"))

ggplot(two_protein_df, aes(x=PPINovelty, y=NumberUniqueLysLysContacts)) +
```

```
  geom_col(aes(fill=PPIEvidenceInfoGroup)) +
  labs(x= "Novelty",
  y= "Number of Contacts") +
  facet_wrap(~NameProtein1)+
  theme_classic()
```



Chapter 5

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

5.1 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
```

5.2 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 73.79
##  [13] 76.03 70.39 76.68 70.43 79.98 71.38 61.82 72.13 71.64 76.75 57.06 74.19
##  [25] 77.08 73.86 57.89 57.73 66.12 57.25 81.29 72.45 47.48 56.49 79.12 74.59
##  [37] 76.44 65.93 57.53 60.43 80.40 56.34 76.33 78.39 79.88 77.47 79.49 63.69
##  [49] 73.04 74.60 76.72 70.52 74.11 60.93 61.66 76.00 61.30 65.28 80.00 81.42
##  [61] 62.86 65.55 72.82 80.09 62.16 80.41 71.34 71.25 57.99 55.65 65.49 32.11
##  [73] 71.58 82.61 74.52 82.03 66.20 69.90 74.45 67.24 80.38 81.42 81.69 74.66
##  [85] 82.85 75.78 68.37 62.76 60.73 70.10 80.13 78.20 68.45 63.80 73.06 79.85
##  [97] 46.50 60.77 76.10    NA 73.17 81.35 74.01 60.84 53.07 74.46 77.91 59.46
## [109] 80.28 63.72 68.23 73.42 75.47 65.38 69.74    NA 66.18 76.36 73.55 54.48
## [121] 66.84 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 64.59
## [145]    NA 75.48 71.64 71.46    NA 68.91 75.13 64.01 74.65 73.38 55.05 82.69
## [157] 75.52 79.45 61.71 53.13 54.27 81.94 74.42 66.29 70.32 46.98 81.52 82.21
## [169] 76.15 79.19 69.61 59.30 76.57 71.10 58.74 69.86 72.56 76.89 78.21 67.94
## [181]    NA 56.81 70.41 76.51 80.34 78.74 76.36 68.77 63.02 75.41 72.27 73.07
## [193] 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
```

5.3 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.

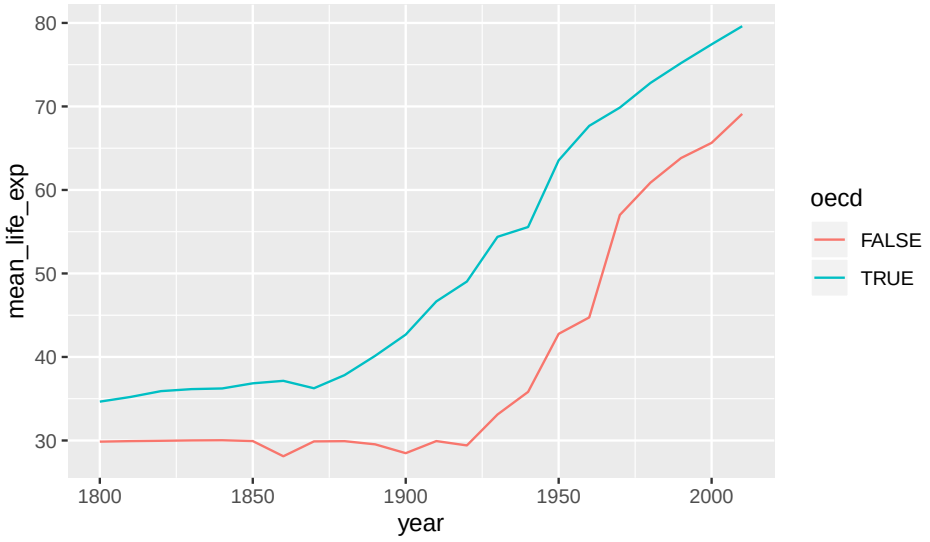
`group_by` can be used to group by multiple columns, much like `count`. We can use this to see how the rest of the world is catching up to OECD nations in terms of life expectancy.

```
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.

5.4 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
```

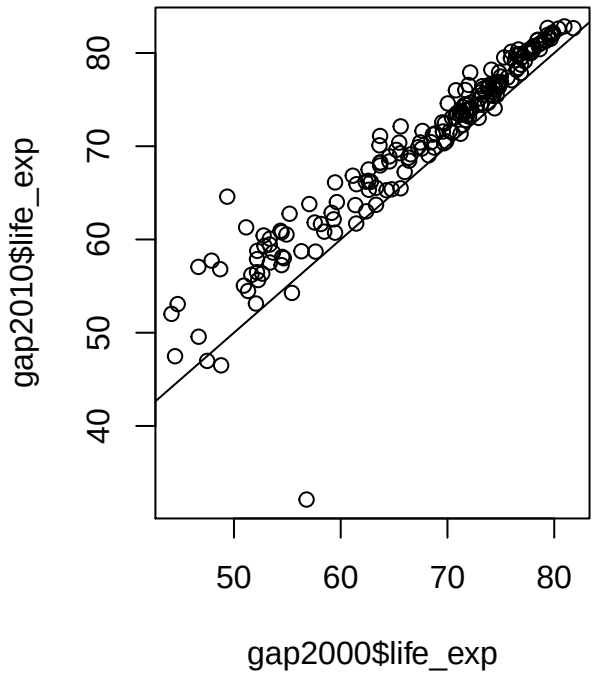
```
t.test(gap2010$life_exp, gap2000$life_exp, paired=TRUE)
```

```
##
##  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.

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"  "stderr"      "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.

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
```

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.

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

R Markdown

6.1 Introduction to markdown

Markdown is a powerful "language" for writing different kinds of documents, such as **PDF** or **HTML** in an efficient way, but markdown documents can also be published as is. The underlying idea for then markdown is that it is easy-to-write and easy-to-read.

You can use any text editor[1] to write your markdown. RStudio[2] already has an inbuilt text editor and because it also has a few additional things that make markdown writing much easier, we are going to use it's text editor.

There are a few different flavours of markdown around. We're going to mention a few but only focus on one, R Markdown[3]

- CommonMark[4]
- GitHub Flavored Markdown (GFM)[5]
- R Markdown[6]

R markdown[7] like most other flavours builds on top of standard markdown. It has some R language[8] specific features as well as bunch of general enhancers to markdown. When R Markdown[9] is coupled with Rstudio[10] it creates a powerfull means of documenting your work while you are doing it, which you can then share with colleagues and the public in rapid and clean way.

Let's get right into it. Open **R Markdown file** using these drop down menu steps: **File** -> **New File** -> **R Markdown**. You can put any `title` and any

[1] https://en.wikipedia.org/wiki/Text_editor
[2] https://rstudio.com
[3] https://rmarkdown.rstudio.com/
[4] http://commonmark.org/
[5] https://guides.github.com/features/mastering-markdown/
[6] https://rmarkdown.rstudio.com/
[7] http://R%20Markdown.rstudio.com/
[8] https://www.r-project.org/
[9] http://R%20Markdown.rstudio.com/
[10] https://rstudio.com

`author` name. For now select **Document** and Document type **HTML**. Once you have opened your `.Rmd` file, click on the `Knit HTML` button at the top of your pane.

`Knitr` is an R package that does all the magic of converting and running your R markdown and R code respectively.

These are three main parts to any R markdown document

1. YAML header section (will talk about it at the very end)

```
---
title: "Hello world"
author: "Kirill"
date: "13 July 2016"
output: html_document
---
```

2. The R code blocks section

```
 ```{r}
 plot(pressure)
 ```
```

3. Everything else is plain old markdown

```
# Have I been Marked Down
```

6.2 Document types

There are numerous document types that you can turn your markdown into. This all depends on the tool, markdown compiler, but for Rstudio[11] at least these a few that a supported.

6.2.0.1 Documents

- `html_notebook` - Interactive R Notebooks
- `html_document` - HTML document w/ Bootstrap CSS
- `pdf_document` - PDF document (via LaTeX template)
- `word_document` - Microsoft Word document (docx)

6.2.0.2 Presentations (slides)

- `ioslides_presentation` - HTML presentation with ioslides

[11] https://rstudio.com

6.2.0.3 More

- `tufte::tufte_handout` - PDF handouts in the style of Edward Tufte
- `tufte::tufte_html` - HTML handouts in the style of Edward Tufte
- `tufte::tufte_book` - PDF books in the style of Edward Tufte

More here[12]

We are not going to cover all of them, we are mainly going to be working with either `html_document` or `html_notebook` both produce very similar results though behave slightly differently. We'll try to touch a little on `ioslides_presentation` towards the end.

6.3 Vanilla Markdown

There's actually not that much to core (vanilla) markdown. Essentially all of it can be summarised below:

```
# Header1
## Header2
### Header3

Paragraphs are separated
by a blank line.

Two spaces at the end of a line
produces a line break.

Text attributes _italic_,
**bold**, `monospace`.

Horizontal rule:

---

Bullet list:

  * apples
  * oranges
  * pears

Numbered list:

  1. wash
  2. rinse
  3. repeat

A [link](http://example.com).
```

[12] https://rmarkdown.rstudio.com/lesson-9.html

```
![Image](link_to_image)

> Markdown uses email-style
> characters for blockquoting.
```

Which produces:

Chapter 7

Header1

7.1 Header2

7.1.1 Header3

Paragraphs are separated by a blank line.

Two spaces at the end of a line
produces a line break.

Text attributes *italic*, **bold**, `monospace`.

Horizontal rule:

Bullet list:

- apples
- oranges
- pears

Numbered list:

1. wash
2. rinse
3. repeat

A link[1].

> Markdown uses email-style > characters for blockquoting.

Whenever I need a refresher on markdown basics, I use this cheatsheet[2].

[1] http://example.com
[2] https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet

7.1.2 Practice vanilla markdown

Now it's just a matter of learning some of the markdown syntax. Let's delete all current text from the opened document except the YAML header and type this new text in `Hello world, I'm learning R markdown!` and pressing the `Knit HTML` button.

```
Hello world, I'm learning R markdown!
```

Not much happened. This is because we didn't mark our text in any way. You can put as much text as you want and it will appear as is, unless "specially" marked to look differently.

Now add the `#` symbol at the start of the line and press the `Knit HTML` button again. We'll be pressing this button a lot! For those who like keyboard short cuts use `ctrl+shift+k` instead.

```
# Hello world, I'm learning R markdown!
```

How about now? A single hash symbol made it whole lot bigger didn't it? We've marked this whole line to be the header line.

Now make three new lines with the same text, but different numbers of `#` symbols, one, two and three respectively and keep pressing the `Knit HTML` button

```
### Hello world, I'm learning R markdown!
## Hello world, I'm learning R markdown!
# Hello world, I'm learning R markdown!
```

This is how you can specify different headers type using markdown.

Let's now practice making very short document in markdown with a main topic section and two subsections. We will add short sentences in each section. We will start with main section header and a quote. Let's type the following text and `knit` our document.

```
# Learning Markdown

> I'm still learning
```

Now let's add three bullet points summarising what we are going to learn next and then `knit` the document again.

```
# Learning Markdown

> I'm still learning

Here I'll be learning:

- markdown
- R Markdown
- ggplot2
```

Now let's add each one of those bullet items as a subsection to the main "Learning Markdown" section.We are going to use `##` to mark subsections and don't forget to `knit` again.

```
# Learning Markdown

> I'm still learning

Here I'll be learning:

- markdown

- R Markdown

- git and github

## Markdown
## R Markdown
## ggplot2
```

Now let's add a sentence to each section, briefly describing what they are.

```
# Learning Markdown

> I'm still learning

Here I'll be learning:

- markdown
- R Markdown
- git and github

## Markdown

Here I'll learng vanilla markdown

## R Markdown

Whereas here I'll be learning R Markdown

## ggplot2

And this section is about plotting
```

Let's add a emphasis to some of the words in our document. We are going to add *italic* emphasis to the word "vanilla" and we are going to add **bold** emphasis to the capital letter "R" in the word R Markdown. You'll need to `knit` your document still.

```
# Learning Markdown

> I'm still learning

Here I'll be learning:

- markdown
```

```
- R Markdown
- git and github

## Markdown

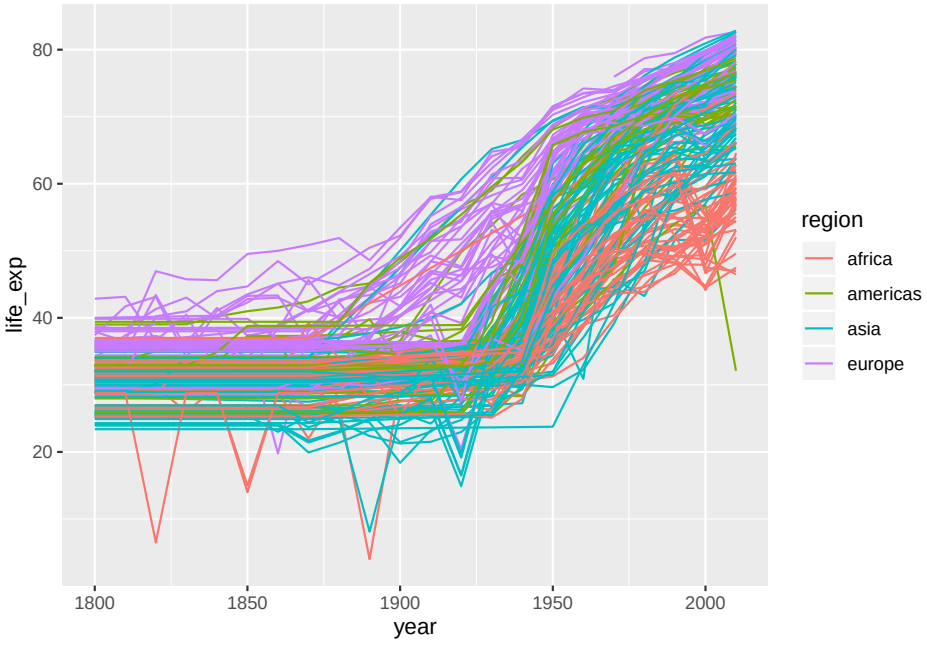
Here I'll learning _vanilla_ markdown

## R Markdown

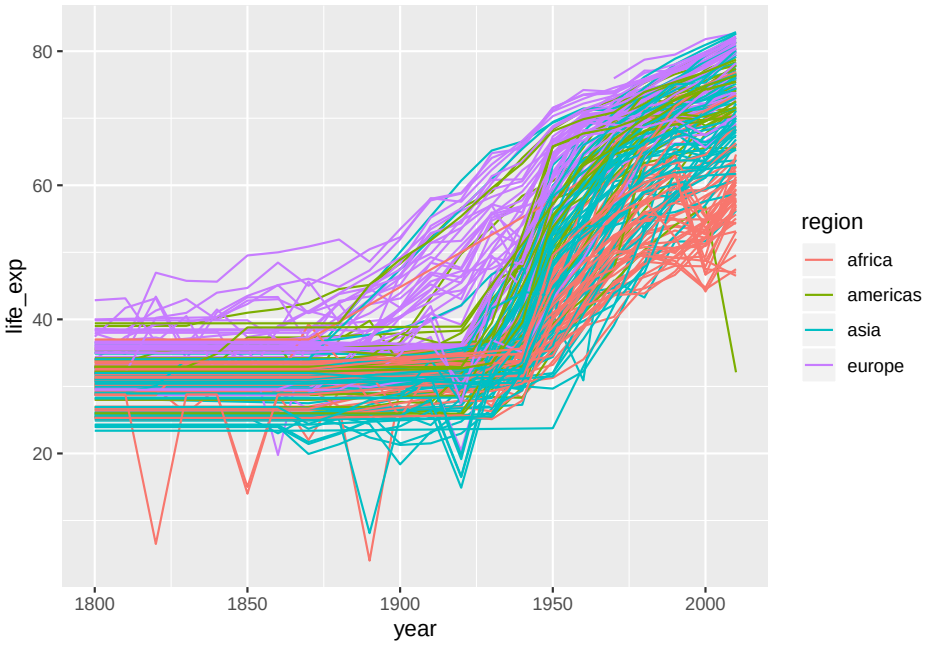
Whereas here I'll be learning R Markdown

## ggplot2

And this section is about plotting
```

Remember that vanilla markdown[3] is comprised entirely of punctuation characters.

7.2 R Markdown

The reason that we are learning R Markdown[4] is because it gives us a very straightforward way of writing plain text documents with inline R code that will become a very sophisticated document types. The bonus points also come from the fact that R Markdown files are easy to version control (git) and see the difference between versions.

This approach of interleaving analysis code, commentary and description is very explicit, which has direct implication in reproducibility, shareability and collaboration.

7.2.1 Embedding R code

An R chunk is a "special" block within the document that will be read and evaluated by `knitr`, ultimately converting everything into plain markdown. But for us it means that we can focus on our analysis and embed R code without having to worry about it. Additionally there are large number of chunk options that helps with different aspects of the document including code decoration and evaluation, results and plots rendering and display.

This is how an R code chunk looks like. If you want to include code into your documents it has to be via R chunks. You can further customise the appearance of your code in the final document with chunk options.

```
```{r}

```
```

[3] https://daringfireball.net/projects/markdown/syntax
[4] https://rmarkdown.rstudio.com/

The little `r` there specifies the "engine", basically telling R Markdown how to evaluate the code inside the chunk. Here we are saying use `R` engine (language) to evaluate the code. The list of languages[5] is rather long, R Markdown can span a much greater area then one might think. In this workshop we are only going to focus on R language.

Let's write our first bit of R code inside the R Markdown document. First we need to start a new R chunk, which we can be done in these ways:

- simply type it out
- press insert button at the top of the window
- `ctrl+alt+i`

Let's return to our document with the main header section `Learning R Markdown`. Now let's add simple R code to our chunk, type the following code `a <- "Hello world, I'm learning R Markdown !"` and press `knitr` button to build html document. Note that as mentioned above we need to use `print()` statement to get the content of the variable to the scree/final document.

```
```{r}
a <- 'Hello world, I'm learning R Markdown !'
print(a)
```
```

Tip: each chunk can be run independently in the console by pressing `ctrl^enter` or little green arrow.

7.2.2 Chunk Options

We can tweak many things about your output using different options that we can include inside curly brackets e.g

```
```{r chunk_options, more_chunk_options...}

```
```

The two rather common options are `echo=TRUE` and `eval=TRUE`, both by default are set to true and this is why we didn't have to pass them in previously.

- `echo` means show what has been typed in i.e show the code in the final document
- `eval` means evaluate or execute that code

Sometimes we might want to show the code, but not execute it and other times we might just want to execute it and get the results without actually bore audience with the code.

Let's try both of these options one at a time. We start with passing `echo=FALSE` options first

```
```{r, echo = FALSE}
print("Hello world, I'm learning R Markdown")
```
```

[5] https://bookdown.org/yihui/R%20Markdown/language-engines.html

Okay, we shouldn't see our original `print()` statement in the output document. And now let's pass `eval=FALSE` options instead

```
 ```{r eval = FALSE}
  print("Hello world, I'm learning R markdown !")
 ```
```

And now we should only see the result of the `print()` statement and no output.

Here is a nice reference[6] that has comprehensive cover of all the options you can pass in.

Let's now try a different example and use the `gapminder` dataset from the ggplot2 segment for this next example.

```
```{r}
library(tidyverse)
geo <- readr::read_csv('r-intro-files/geo.csv')
gap <- read_csv('r-intro-files/gap-minder.csv')
gap_geo <- left_join(gap, geo, by='name')
head(gap_geo)
```
```

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

head(gap_geo)
```

```
## # A tibble: 6 x 11
##   name   year population gdp_percap life_exp region oecd  g77     lat   long
##   <chr> <dbl>      <dbl>      <dbl>    <dbl> <chr>  <lgl> <lgl> <dbl>  <dbl>
## 1 Afgh~  1800    3280000        603     28.2 asia   FALSE TRUE   33    66
## 2 Alba~  1800     410445        667     35.4 europe FALSE FALSE  41    20
## 3 Alge~  1800    2503218        715     28.8 africa FALSE TRUE   28     3
## 4 Ando~  1800       2654       1197     NA   europe FALSE FALSE  42.5   1.52
## 5 Ango~  1800    1567028        618     27.0 africa FALSE TRUE  -12.5  18.5
## 6 Anti~  1800      37000        757     33.5 ameri~ FALSE TRUE   17.0 -61.8
## # ... with 1 more variable: income2017 <chr>
```

This table is a bit ugly to look at in your final document. We'll come back to data-frame printing later in the YAML section.

Note: you might be thinking that you've already run `library(tidyverse)` previously in the session and that you already have tidyverse packages loaded, you shouldn't need to run the command again. When you knit a document, it ignores the state of your RStudio session and runs through the code from start to finish. If your code points to missing files or uses packages that haven't been explicitly loaded somewhere in the document, it will fail to render your document.

[6] https://bookdown.org/yihui/rmarkdown/r-code.html

Remember: You can go between R Markdown and *console*, to check your code, at any time. You should see your code block is highlighted differently and you should see a green arrow at the right hand site of that block. Press the green arrow to get an output in the *console*. You can also use `ctrl+enter` to do the same with the keyboard short cut.

How about little data exploration? From the ggplot2 segment, we can include some plots into our html document, rather than saving the plots to file, moving the plots off the server onto our local computer, then manually including them into a document.

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

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

```
## Warning: Removed 205 rows containing missing values (geom_path).
```



Here is a perfect example where we can hide our code from the viewer, since it isn't most interesting bit about this data. Let's turn `echo=FALSE` options for all our plots below. We also don't want to get that warning message included in the final document, so we can also include `warning=FALSE`.

Figure alignment can be done with `fig.align` options e.g `{r fig.align=default}` . default means what ever your style sheet has. The other options are, "left", "center" and "right". let's try one out.

````
```{r echo = FALSE, fig.align = 'right', warning=FALSE}
ggplot(gap_geo, aes(x=year, y=life_exp, group=name, color=region)) +
```
````

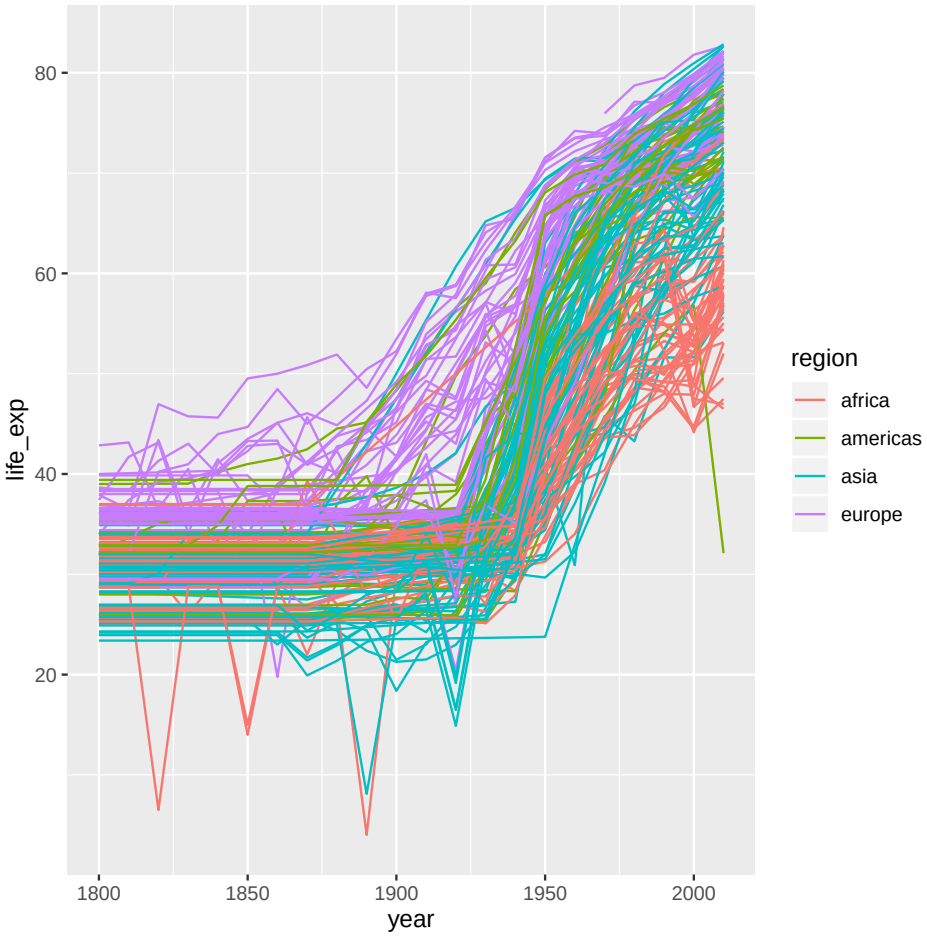
```
geom_line()
```
```




**Remember** that you can you always execute code in the *console* by pressing "green arrow" or using keyborad short cut `ctrl+enter`

We now know how to align figure to where we want, how about changing the size of it? We can do that with `fig.height` and `fig.width`, the units are inches. Let's make 4 X 4 inches figure e.g `{r fig.height=6, fig.width=6}` . and also align the figure to the center

```
```{r echo = FALSE, fig.align = 'center', warning=FALSE, fig.height=6, fig.width=6}
ggplot(gap_geo, aes(x=year, y=life_exp, group=name, color=region)) +
geom_line()
```
```
```



One last thing we'd like to share with you is how to add a figure legend or a caption - with `fig.cap` of course e.g `{r, fig.cap="This is my legend"}` . Go ahead and add a figure description.

```
```{r echo = FALSE, fig.align = 'center', warning=FALSE, fig.height=6, fig.width=6, fig.ca
ggplot(gap_geo, aes(x=year, y=life_exp, group=name, color=region)) +
geom_line()
```
```

Note that the figure legend follows the same alignment as the figure itself.

There are more chunk options which we encourage you to explore in greater depth in the R Markdown documentation[7]. We have only examined a handful of figure specific options but there are many more options that allow fine control over the behavior of the code and cosmetics of the document.

Lastly, we'll mention that the `engine` option can be used to specficy different language types. So you can embed python, BASH, Javascript and a heap of other languages[8] all within the same document.

[7] https://bookdown.org/yihui/rmarkdown/r-code.html
[8] https://bookdown.org/yihui/rmarkdown/language-engines.html

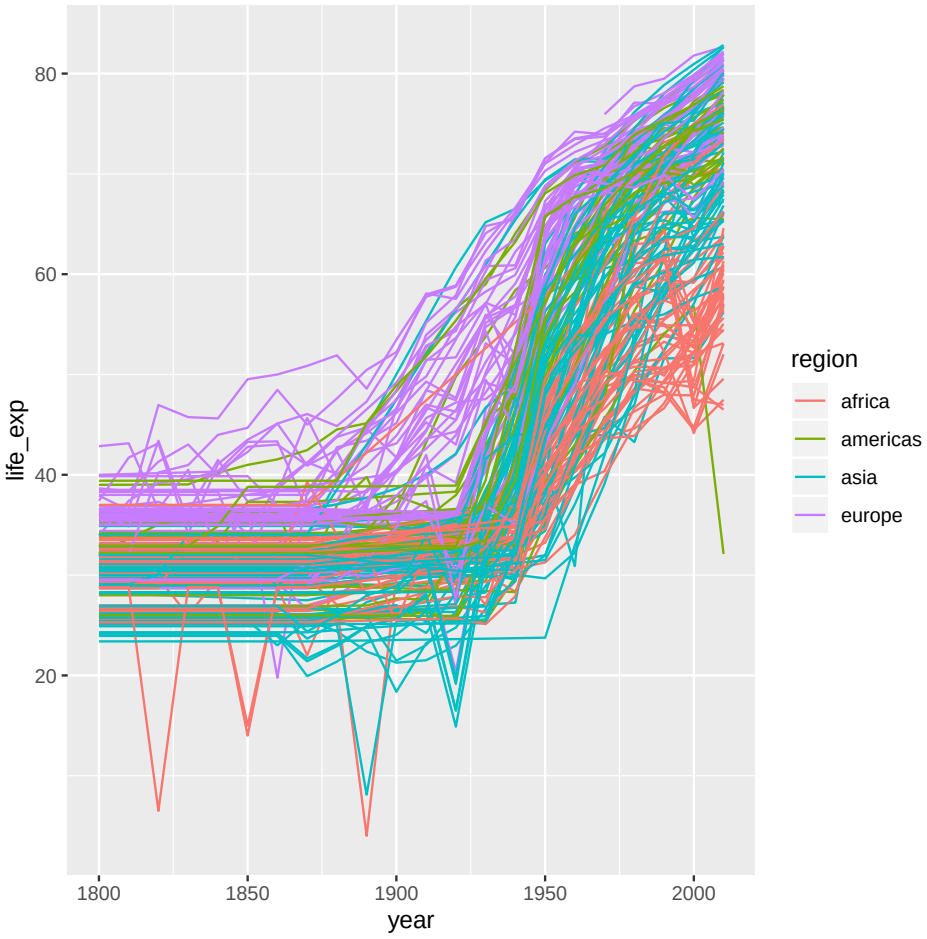


Figure 7.1: This figure illustrates the life expectancy from 1800 to 2000 by region.

7.3 YAML header

At the very top of your `.Rmd` file you can, optionaly, include a YAML block. In this block you can fine turn your output document, add some metadata and change the document's font and theme. You can also pass additional files such as stylesheet file `.css` and bibliography file `.bib` for text citation. We're only going to show you a few possible options and will let you explore the rest on your own.

Navigate to the top of your `.Rmd` document and find the YAML section there. Just like with the options we passed in to manipulate R code block, YAML block also has **key = value** pairs, but instead they are separated by colon (`:`). Now let's add table of content to our document, this will make it easier to navigate your page as well as give nice over view of the content our **key** is `toc` with value `true` or `yes` which one you prefer better.

```
---
title: "Hello world"
author: "Kirill"
date: "13 July 2016"
output:
  html_document:
    toc: true
---
```

Note that you need to bring `html_document` onto new line and indent it with two spaces. `html_document` is a value of `output` key. `output` can have other values e.g `pdf_document`, `word_document`. However `html_document` also becomes a key for `toc` value and `toc` becames a key for its own value.

Now that we have sort out the initial YAML layout, we can continue adding more options to style our HTML document. The other two useful options that I like to pass in are `toc_depth` and `number_sections`

```
---
title: "Hello world"
author: "Kirill"
date: "13 July 2016"
output:
  html_document:
    toc: true
    toc_depth: 4
    number_sections: yes
---
```

Most of those options are self explanatory. The best way to learn what each does, is to pass them in. Note that you can comment lines out inside YAML section with `#` symbol.

Another two options that can change your document apperance are `theme` and `highlight`[9]. There are number of different themes and highlight options. I

[9] https://bookdown.org/yihui/rmarkdown/html-document.html#appearance-and-style

suggest you find the one you like in your own time.

Remember when we printed the `gap_geo`, the table rendered wasn't particularly nice to look at? We can control the behavior of tables in html documents in the YAML header[10]. There are a number of options to select one from and pass to `df_print`.

Try out `paged`.

```
---
output:
  html_document:
    toc: true
    toc_depth: 4
    df_print: paged
---
```

7.4 Alternate Formats

As mentioned in previous section, `output` has many options, one of which is `ioslides_presentation`. You can simple add:

```
---
title: "Hello world"
author: "Kirill"
date: "13 July 2016"
output: ioslides_presentation
---
```

at the top of your document and your `.Rmd` files will be compiled to a slide presentation instead.

Another way to start with an ioslides_presentation is select **presentation** options when you were opening R markdown file. Either way you'll notice YAML header reflects your selected output type.

Let's open new R markdown document and let's select presentation instead and let's select HTML (ioslides) option there. You can still save your files as `.Rmd`, and then press the the `Knit HTML` button.

The syntax for the document is more or less the same, except `##` is now used to mark new slide.

ioslides are fairly basic in terms of slideshow presentations. If you find yourself frustrated with the limitations of ioslides, there are a number of official format options[11], we haven't had much experience with using them. I've been using the xaringan package[12], which isn't an official R Markdown format but I've found it to be quite powerful, though requires a fair degree of familiarity with R Markdown/HTMl/CSS.

[10] https://bookdown.org/yihui/rmarkdown/html-document.html#data-frame-printing
[11] https://rmarkdown.rstudio.com/formats.html
[12] https://github.com/yihui/xaringan

Alternatively, if you want to produce a PDF document:

```
---
title: "Hello world"
author: "Kirill"
date: "13 July 2016"
output: pdf_document
---
```

R Markdown documents that render to PDF are compatible with raw LaTeX. The `df_print` option is not compatible with `paged` but will take `kable` and `tibble`. If changing document type, it's always important to check which YAML options will carry to the new format and which won't.

The more format specific syntax used in a document, the harder it is to swap a document from one format to another. For example, you can generate a very detailed and customised PDFs from an R Markdown document with heavy usage of LaTeX. However, LaTeX will not be rendered in a HTML document. So it's important to have an idea of how the final document will be used as you work through it.

7.5 Extras

- R Markdown cheatsheet[13]

For a more in-depth R-Markdown tutorial, we recommend:

- R Markdown: The Definitive Guide[14]
- bookdown: Authoring Books and Technical Documents with R Markdown[15] - the bookdown package greatly extends R Markdown
- R for Reproducible Research[16] - a workshop delivered by the Monash Bioinformatics Platform focused on R Markdown and reproducible workflows with git.

7.5.1 Code chunk names

You can name code chunks! In your R-Studio session this even adds a little table of contents in the bottom left of your source panel that lets you navigate your R Markdown document via headers and code chunk names.

```
```{r chunk_name, chunk_opts...}

```
```

[13] https://rstudio.com/wp-content/uploads/2015/02/rmarkdown-cheatsheet.pdf
[14] https://bookdown.org/yihui/rmarkdown/
[15] https://bookdown.org/yihui/bookdown/
[16] https://monashdatafluency.github.io/r-rep-res/

7.5.2 html_document or html_notebook

We've been using the `html_document` format for most of this tutorial. A very similar looking document is the `html_notebook`. There's a large overlap in terms of YAML options for both.

e.g:

```
---
output:
  html_notebook:
    toc: true
    toc_depth: 4
    df_print: paged
---
```

One difference is that a html_notebook will include a button on the rendered webapge to download the `.Rmd` file that generated the html file. This is one way to easily share the document, in that someone can view your report and then download it and run it on their own machine.

The other difference is that when rendering a html_document, it will ignore the state of the RStudio session and re-run every code block. Variables that have been exist within the environment but have not been defined inside the html_document will cause the render step to fail and must be included for the document to render successfully.

A html_notebook however uses the state of the session. It will only include the output of code blocks that have been run. It's less 'safe' using a notebook, because it doesn't double-check that the document from start to finish is self-contained. It can include variables and functions that were created seperately in the environment but then the document doesn't include instructions on how that variable was generated or what the function is doing.

7.5.3 Cross-referencing

Let's learn how to add external and internal links to your document, remember the syntax for adding links is `[DESCRIPTION](link-address)`. The external link that we are going to add is going to be this https://rmarkdown.rstudio.com/. Each one of the bullet points above going to become a link to it section. The way you reference internal section is by starting your address with a `#` symbol then simply using all lower case letters for the section name and all spaces need to be converted to a dash symbol `-`. Let's add those things in and re-build our document.

```
# Learning Markdown

> I'm still learning

[External resource](https://rmarkdown.rstudio.com/)

Here I'll be learning:
```

```
- [markdown](#markdown)
- [R Markdown](#R Markdown)
- [git and github](#git-and-github)

## Markdown
Here I'll learng _vanilla_ markdown

## R Markdown

Whereas here I'll be learning **R**markdown

## ggplot2

And this section is about plotting
```

A bonus exercise is to add logos to each sections. Search internet for:

- Markdown logo, and add the image using `![](link_address)` syntax
- R Markdown logo, and add the image using `![](link_address)` syntax
- Git logo, and add the image using `![](link_address)` syntax
- ggplot2 logo, and add the image using `![](link_address)` syntax

Note for the external resource that is on internet the address must start with `www` or `https` otherwise address will be interpreted as file path.

Chapter 8

Next steps

8.1 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/

8.2 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.

8.3 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/