Introduction to R

Version 2

Contents

Introduction

These are course notes for the "Introduction to R" course given by the Monash Bioinformatics Platform[1] for the Monash Data Fluency[2] initiative. This is a new version of the course 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 R on a server we run. However R is free, and you can install it on your own computer. There are two things to download and install:

- Download R[6]
- Download RStudio[7]

(R is the language itself. RStudio provides a convenient environment in which to use R.)

Source code

- GitHub page[8]

Authors and copyright

This course is developed for the Monash Bioinformatics Platform by Paul Harrison.

[1]https://www.monash.edu/researchinfrastructure/bioinformatics
[2]https://monashdatafluency.github.io/
[3]https://www.tidyverse.org/
[4]https://monashdatafluency.github.io/r-intro-2/r-intro-2.pdf
[5]https://monashdatafluency.github.io/r-intro-2/r-intro-2-files.zip
[6]https://cran.rstudio.com/
[7]https://www.rstudio.com/products/rstudio/download/
[8]https://github.com/MonashDataFluency/r-intro-2
[9]http://creativecommons.org/licenses/by/4.0/

Data files are derived from Gapminder, which has a CC BY-4 license. The attribution is "Free data from www.gapminder.org". The data is given here in a form designed to teach various points about the R language. Refer to the Gapminder site[10] for the original form of the data if using it for other uses.

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

Chapter 1

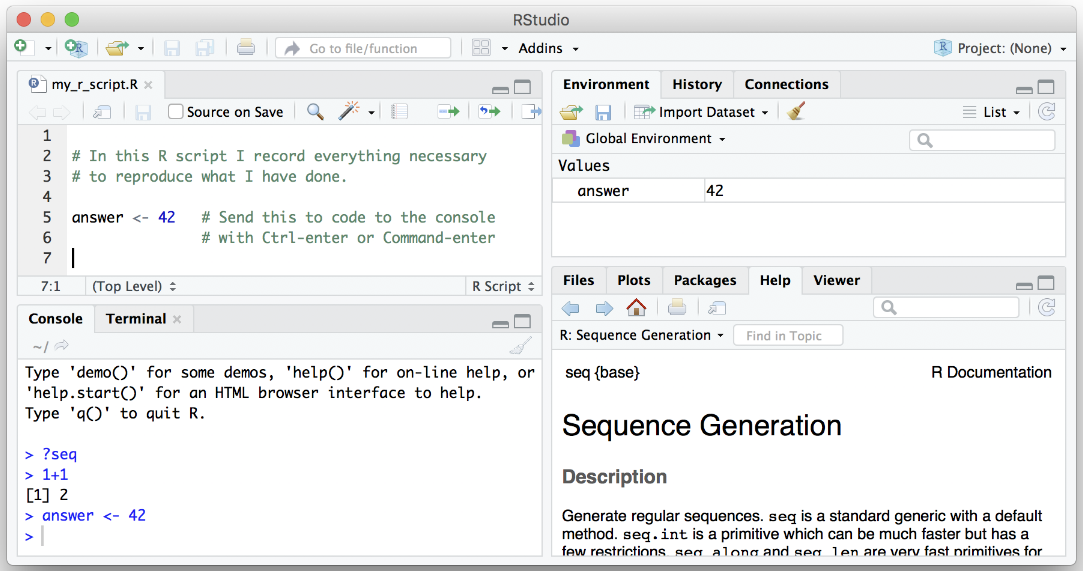
Starting out in R

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]. This surrounds the *console*, where one enters commands and views the results, with various conveniences. In addition to the console, 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.



```

# In this R script I record everything necessary
# to reproduce what I have done.

answer <- 42    # Send this to code to the console
                # with Ctrl-enter or Command-enter

```

Values

answer	42

```
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> ?seq
> 1+1
[1] 2
> answer <- 42
>
```

seq {base} R Documentation

Sequence Generation

Description

Generate regular sequences. `seq` is a standard generic with a default method. `seq.int` is a primitive which can be much faster but has a few restrictions. `seq_along` and `seq_len` are very fast primitives for

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 be showing such an interaction with R as below.

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

```
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 as a single line of R:

```
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]
```

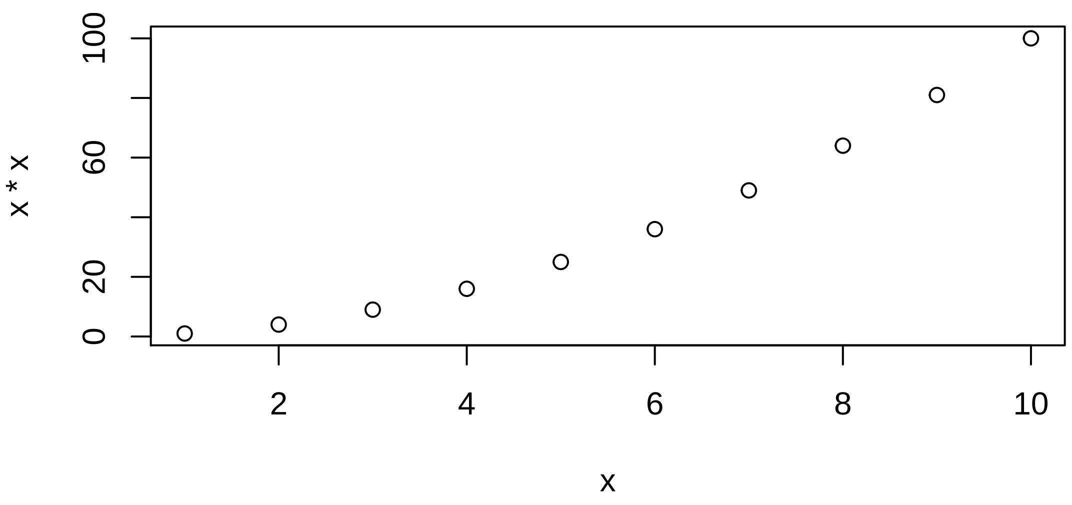
```
    [1] "spam"  "eggs"  "beans" "bacon"
```

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 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
    [36] 3 1 1 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. Sticking to the Tidyverse where possible is generally safer, Tidyverse packages are more willing to generate errors rather than ignore problems.

If the Tidyverse is not already installed, you will need to install it. However on the server we are using today it is already installed.

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

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

2.1 Loading data

We will use the `read_csv` function from `readr` to load a data set. (See also `read.csv` in base R.)

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

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

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

```
geo
```

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

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

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

We will also encounter:

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

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

Tip

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

```
data_frame(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.

2.2 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
```

2.3 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
```

2.4 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"
```

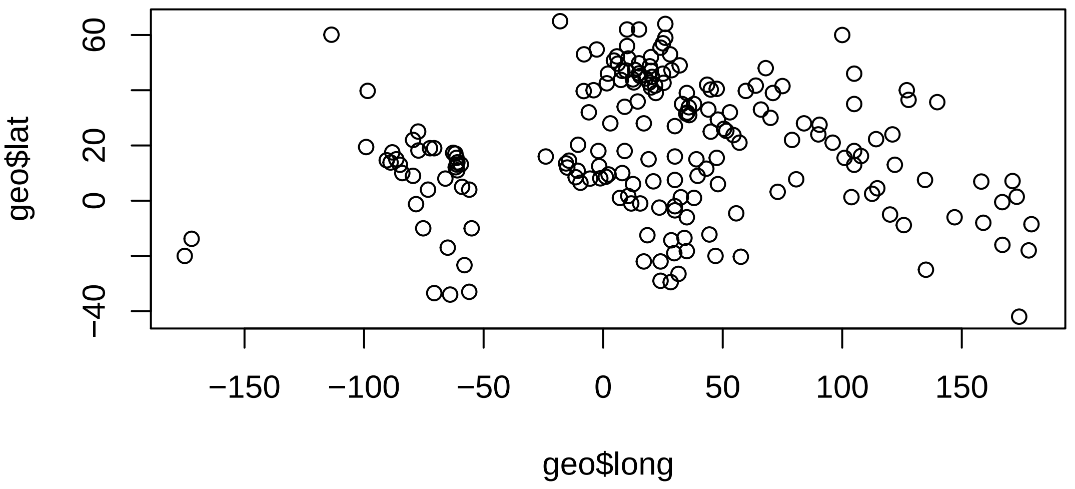
To get the "region" value of the 4th row as above, but unwrapped, we can use:

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

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

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

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



2.5 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 Barb~ americ~ FALSE TRUE    17.0  -61.8  high       FALSE
 7 Argentina          americ~ FALSE TRUE   -34    -64    upper_mid  TRUE
 8 Armenia            europe   FALSE FALSE   40.2   45    lower_mid  FALSE
 9 Australia          asia     TRUE  FALSE -25    135    high       TRUE
10 Austria            europe   TRUE  FALSE  47.3   13.3  high       FALSE
# ... with 186 more rows
```

Challenge: logical indexing

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

2.5.1 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.

2.6 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 to 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"))
```

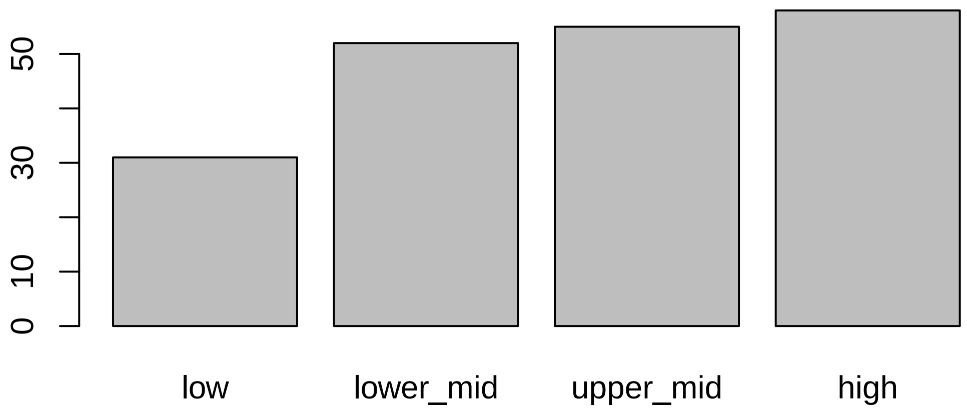
`count` now produces the desired order of output:

```
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
```

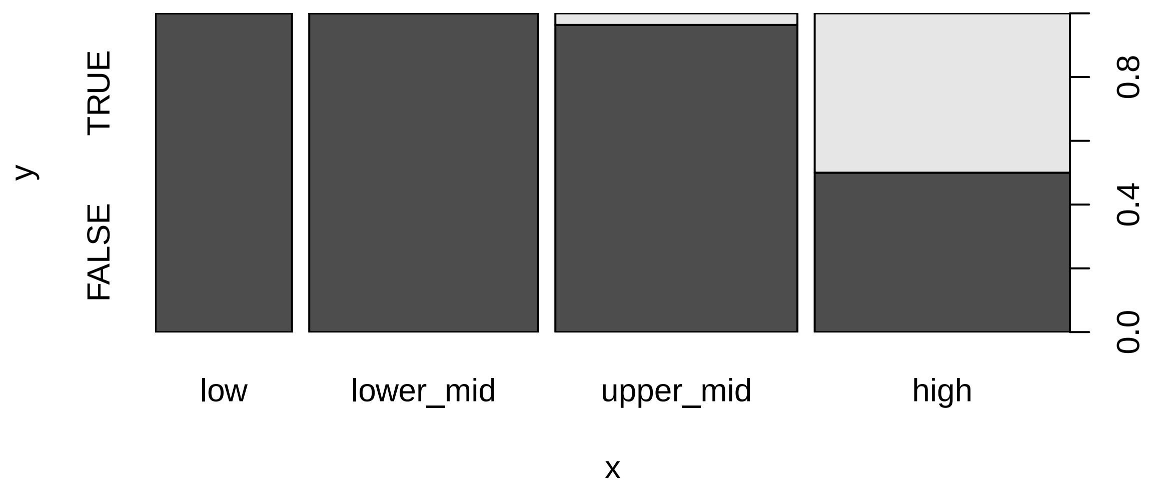
When `plot` is given a factor, it shows a bar plot:

```
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
```

2.7 Readability vs tidyness

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

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

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

Here:

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

Tip

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

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

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
```

2.8 Sorting

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

```
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
```

2.9 Joining data frames

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

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

```
# A tibble: 4,312 x 5
   name                 year population gdp_percap life_exp
   <chr>               <int>      <dbl>      <dbl>    <dbl>
 1 Afghanistan          1800    3280000        603     28.2
 2 Albania              1800     410445        667     35.4
 3 Algeria              1800    2503218        715     28.8
 4 Andorra              1800       2654       1197     NA
 5 Angola               1800    1567028        618     27.0
 6 Antigua and Barbuda  1800      37000        757     33.5
 7 Argentina            1800     534000       1507     33.2
 8 Armenia              1800     413326        514     34
 9 Australia            1800     351014        814     34.0
10 Austria              1800    3205587       1847     34.4
# ... with 4,302 more rows
```

Quiz

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

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

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

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

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

The “left” in “left join” refers to how rows that can’t be paired up are handled. `left_join` keeps all rows from the first data frame but not the second. This is a good default when the intent is to attaching some

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

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.

2.10 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 Bioinformatics Platform "R more" course[6] also covers this.

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

Chapter 3

Plotting with ggplot2

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

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

```
library(tidyverse)
```

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

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

3.1 Elements of a ggplot

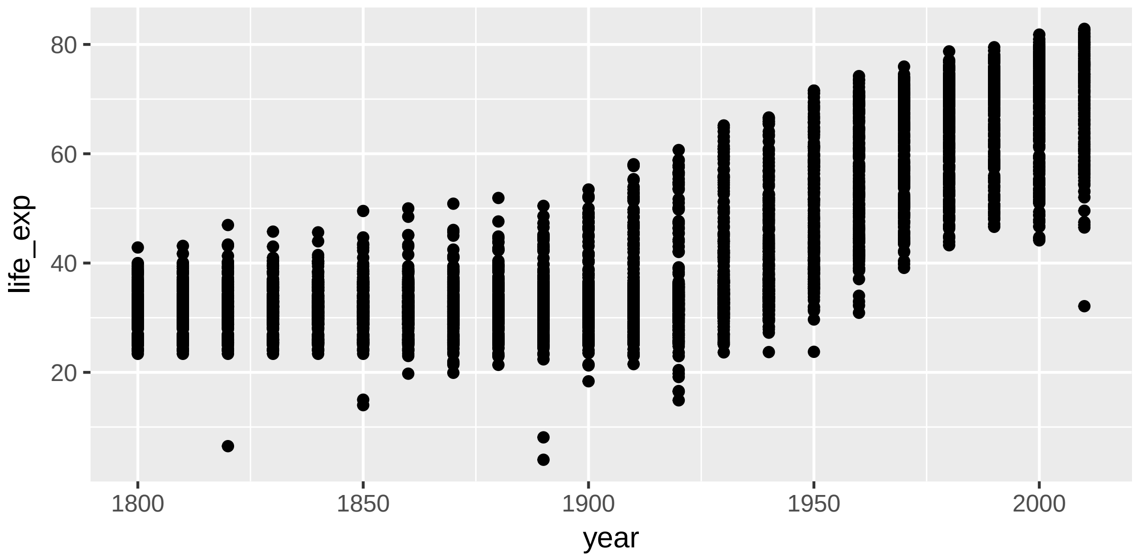
Producing a plot with `ggplot2`, we must give three things:

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

Let's make our first ggplot.

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

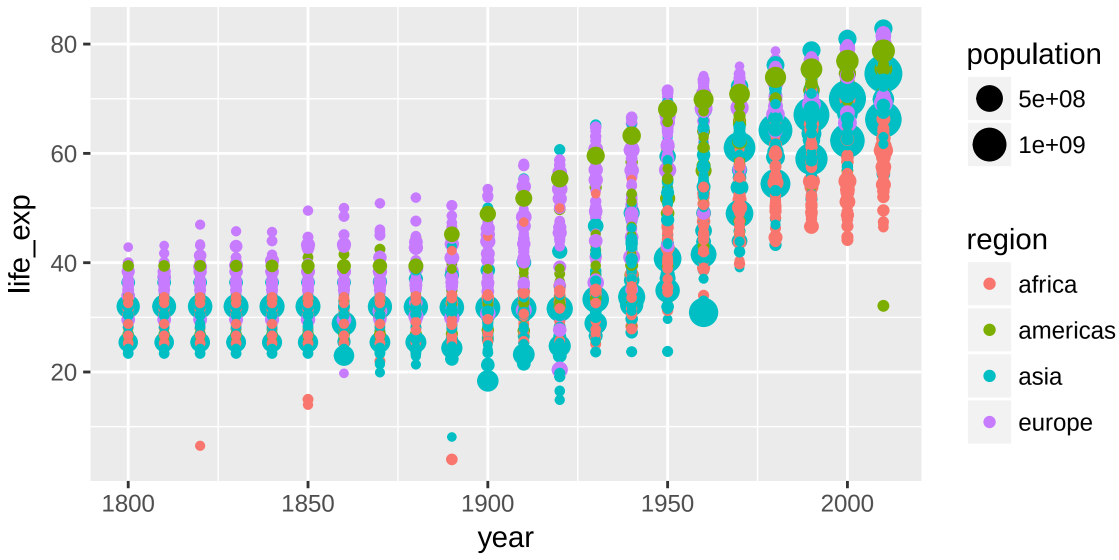
[1] url%20https://www.amazon.com/Grammar-Graphics-Statistics-Computing/dp/0387245448
[2] http://ggplot2.tidyverse.org/reference/



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:

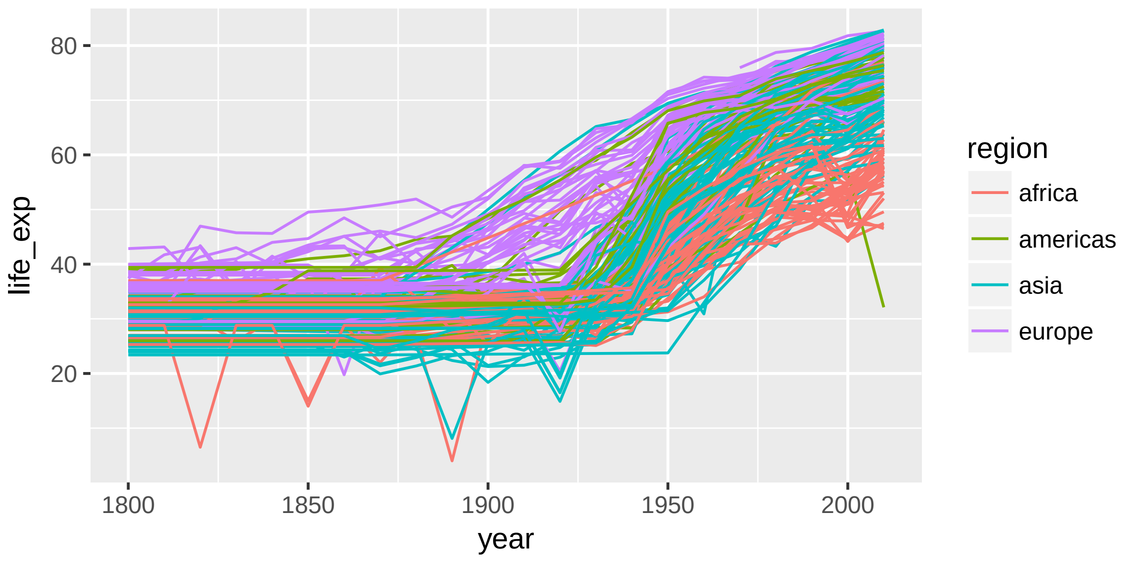
- `gdp_percap` as x.
- `life_exp` as y.
- `population` as the size.

- `region` as the color.

3.2 Further geoms

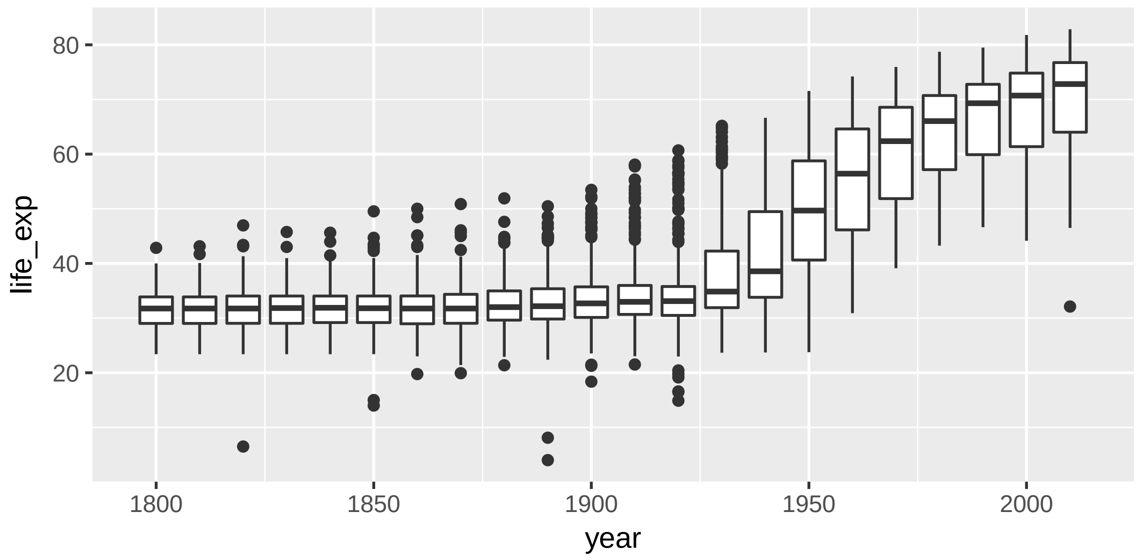
To draw lines, we need to use a "group" aesthetic.

```r
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.

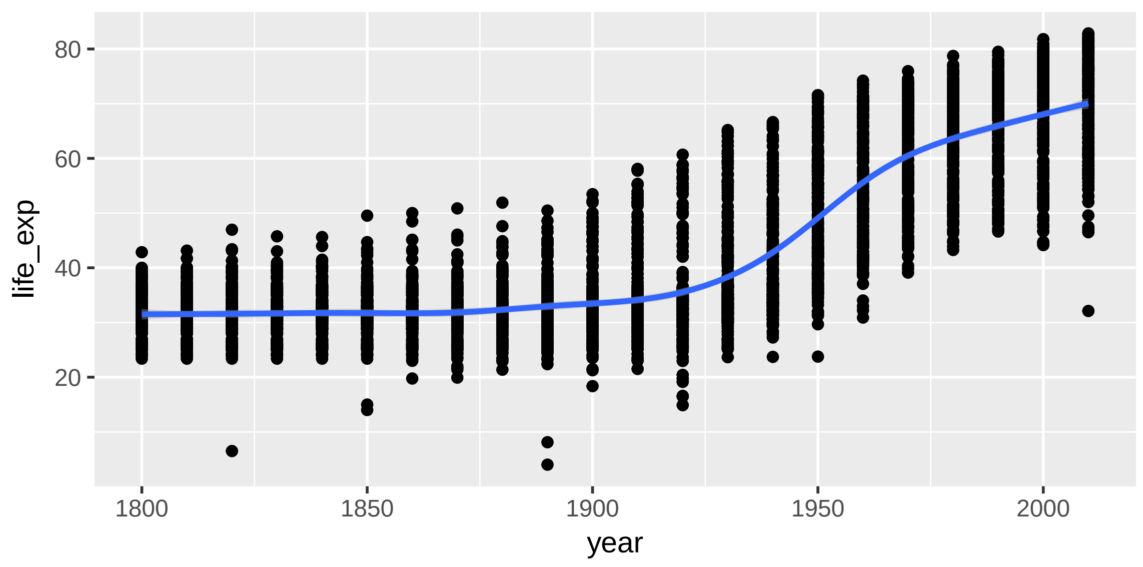
```r
ggplot(gap_geo, aes(x=year, y=life_exp, group=year)) +
    geom_boxplot()
```



`geom_smooth` can be used to show trends.

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

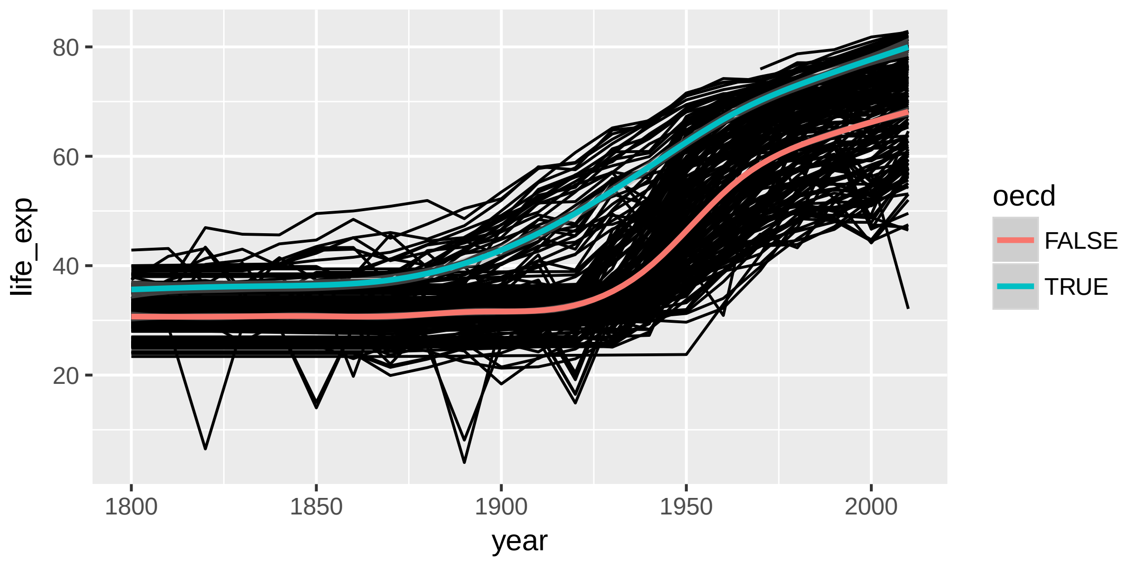
```
`geom_smooth()` using method = 'gam'
```



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:

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

```
    `geom_smooth()` using method = 'gam'
```

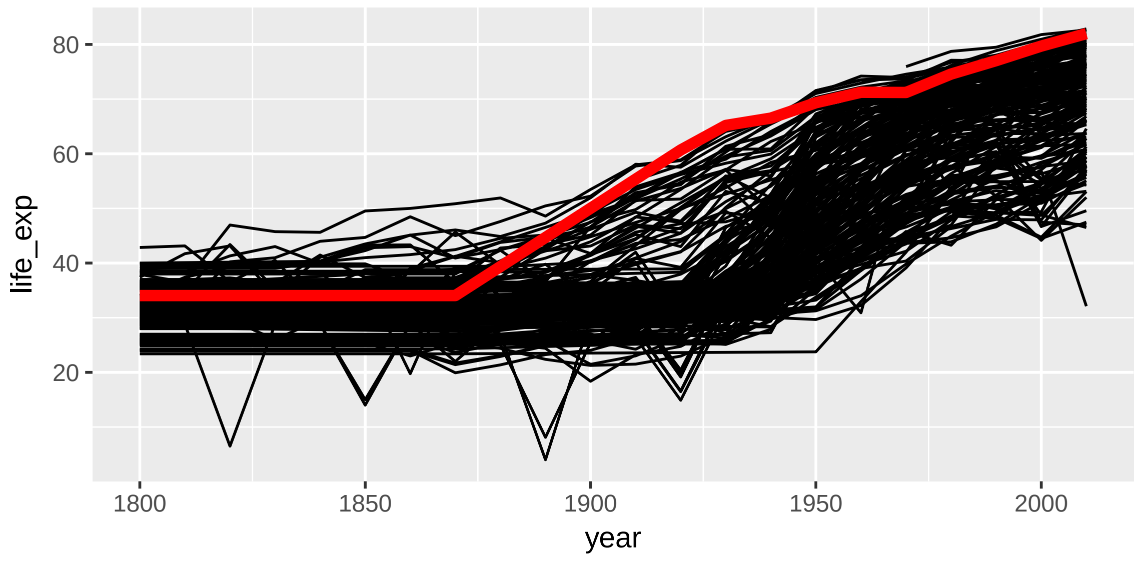


3.3 Highlighting subsets

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

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

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

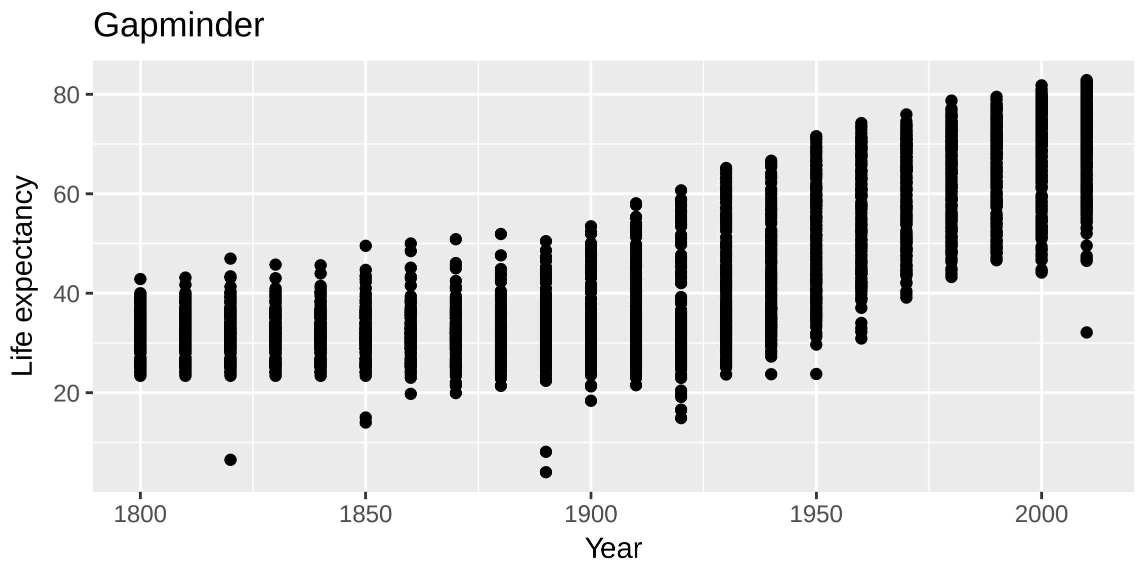


Notice also that the second `geom_line` has some further arguments controlling its appearance. These are **not** aesthetics, they are not a mapping of data to appearance, but rather a direct specification of the appearance. There isn't an associated scale as when color was an aesthetic.

3.4 Fine-tuning a plot

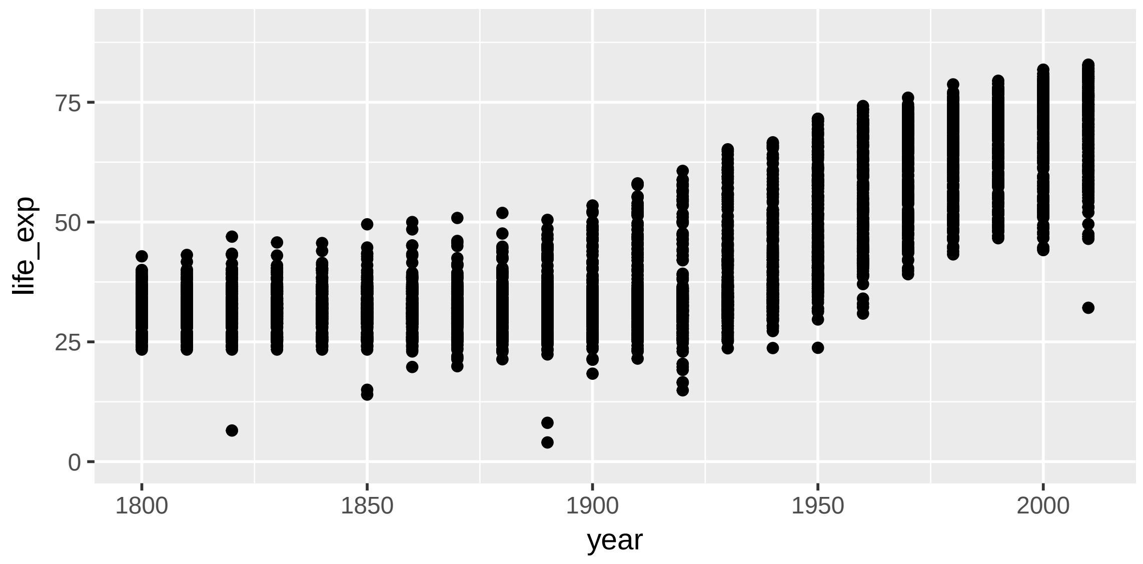
Adding `labs` to a ggplot adjusts the labels given to the axes and legends. A plot title can also be specified.

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



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

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



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

3.4.1 Challenge: refine your ggplot

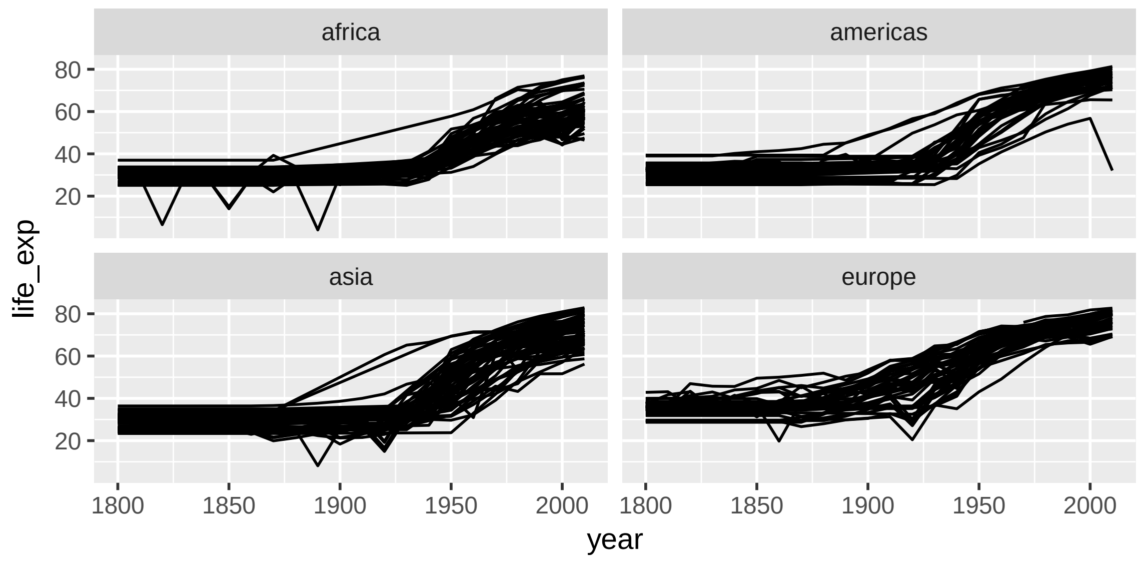
Continuing with your scatter-plot of the 2010 data, add axis labels to your plot.

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

3.5 Faceting

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

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



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

3.5.1 Challenge: facet your ggplot

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

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

Advanced: Highlight Australia in your plot.

3.6 Saving ggplots

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

Ggplots can be saved using `ggsave`.

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

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

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

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


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

Chapter 4

Summarizing data

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

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

```
library(tidyverse)

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

4.1 Summary functions

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

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

```
[1] 2.5
```

We can use these on the Gapminder data.

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

```
[1] 6949495061
```

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

```
[1] NA
```

4.2 Missing values

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

```
gap2010$life_exp
```

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

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

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

```
[1] 70.34005
```

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

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

```
[1] 70.96192
```

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

```
NA + 1
```

```
[1] NA
```

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

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

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

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

```
[1] 70.96192
```

4.3 Grouped summaries

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

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

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

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

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

```
# A tibble: 22 x 2
   year mean_life_exp
```

```
     <int>          <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.

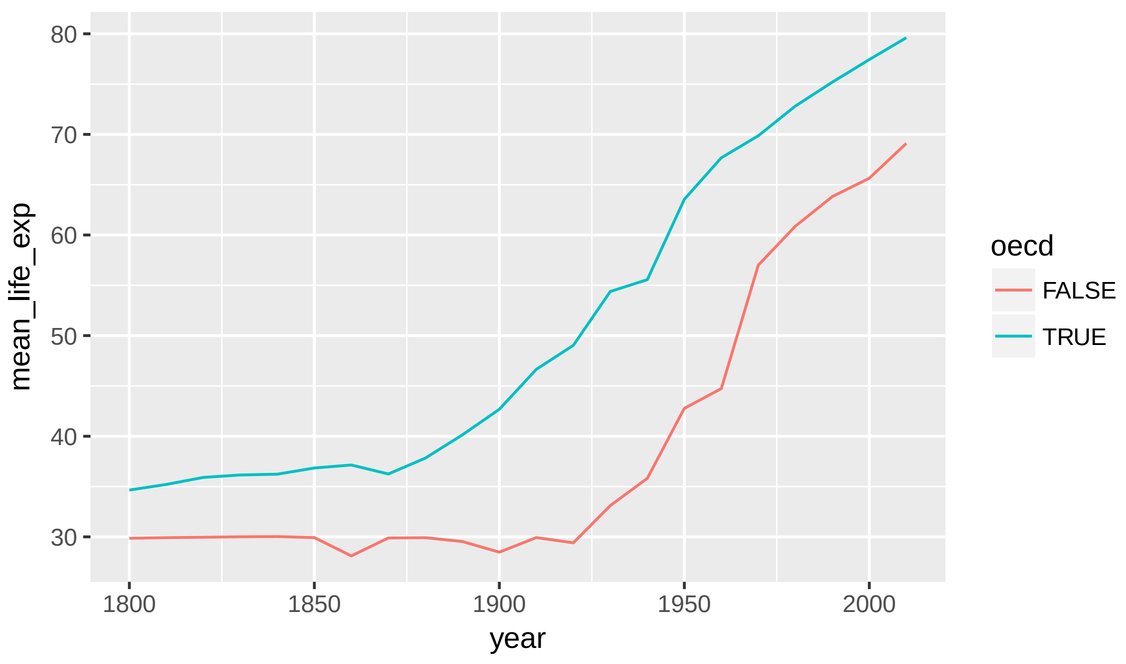
Advanced: What is the total GDP for each year? For this you will first need to calculate GDP per capita times the population of each country.

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

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



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

4.4 t-test

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

Has life expectancy increased from 2000 to 2010?

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

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

```

	Welch Two Sample t-test

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

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

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

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

```
[1] TRUE
```

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

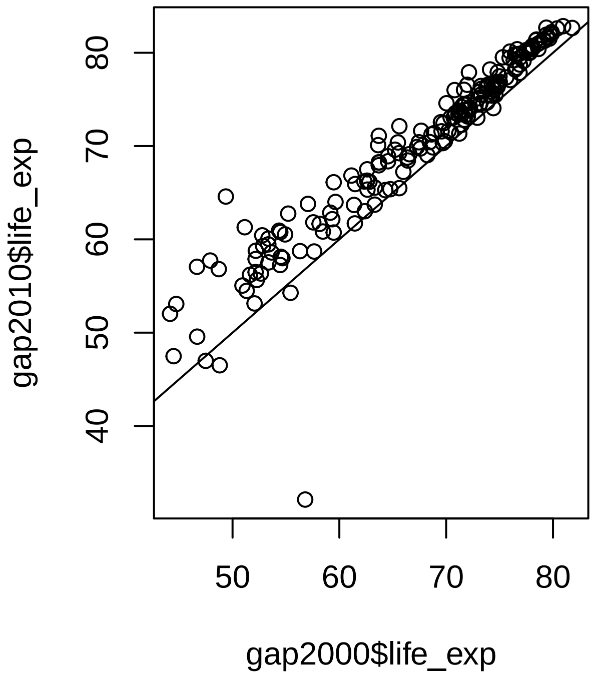
```

	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.

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



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

Chapter 5

Thinking in R

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

```
class(42)
```

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

```
typeof(42)
```

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

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

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

class(result)
```

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

```
typeof(result)
```

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

```
names(result)
```

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

```
result$p.value
```

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

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

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

5.1 Lists

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

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

```
$hello
[1] "Hello" "world"

$numbers
[1] 1 2 3 4
```

```
class(mylist)
```

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

```
typeof(mylist)
```

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

```
names(mylist)
```

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

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

```
mylist$hello
```

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

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

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

5.2 Other types not covered here

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

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

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

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

5.3 Programming

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

- Writing your own functions[2].
- For-loops[3] to do things multiple times.

[1] http://bioconductor.org/
[2] http://r4ds.had.co.nz/functions.html
[3] http://r4ds.had.co.nz/iteration.html

- If-statements[4] to make decisions.

The "R for Data Science" book[5] is an excellent source to learn more. The Monash Bioinformatics Platform "R more" course[6] also covers this.

[4]http://r4ds.had.co.nz/functions.html#conditional-execution
[5]http://r4ds.had.co.nz/
[6]https://monashbioinformaticsplatform.github.io/r-more/

Chapter 6

Next steps

6.1 Deepen your understanding

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

Also, statistical tasks such as model fitting, hypothesis testing, confidence interval calculation, and prediction are a large part of R, and one we haven't demonstrated fully today. "Modern Applied Statistics with S" by W.N. Venable and B.D. Ripley is a well respected reference covering R and its predecessor S. "Linear Models with R" and "Extending the Linear Model with R" by Julian J. Faraway cover linear models, with many practical examples. 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` for differential gene expression, `glm` for logistic regression (etc), survival analysis with `coxph`, and mixed models to characterize variation within populations.

6.2 Expand your vocabulary

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

- RStudio's collection of cheat sheets[2] cover newer packages in R.
- An old-school cheat sheet[3] for dinosaurs and people wishing to go deeper.
- Bioconductor cheat sheet[4] for biological data.

6.3 Join the community

Join the Data Fluency community at Monash[5].

- Mailing list for workshop and event announcements.
- Slack for discussion.
- Drop-in sessions on Friday afternoon.

Meetups in Melbourne:

- MelbURN[6]
- R-Ladies[7]

[1]http://r4ds.had.co.nz/
[2]https://www.rstudio.com/resources/cheatsheets/
[3]https://cran.r-project.org/doc/contrib/Short-refcard.pdf
[4]https://github.com/mikelove/bioc-refcard/blob/master/README.Rmd
[5]https://monashdatafluency.github.io/
[6]https://www.meetup.com/en-AU/MelbURN-Melbourne-Users-of-R-Network/
[7]https://www.meetup.com/en-AU/R-Ladies-Melbourne/

For bioinformatics, COMBINE[8] is a student and early career researcher organization, and runs Software Carpentry workshops.

[8]https://combine.org.au/