Visualising Data using R

Craig Alexander, Eilidh Jack & Marnie Low

Contents

Chapter 1

Overview

The R programming language provides researchers with access to a large range of fully customisable data visualisation options, which are typically not available in point-and-click software. These visualisations are not only visually appealing, but can increase transparency about the distribution of the underlying data, rather than relying on commonly used visualisations of aggregations.

In this introductory section of our course, we will provide a practical introduction to using R, particularly in how to visualise data which you will use throughout the course. First, we will explain the rationale behind using R for data visualisation using the `ggplot2` package. This package will allow us to begin with common plotting outputs such as histograms and boxplots, and extend to more complex structures used within spatial data visualisation.

1.1 The `ggplot2` package

There are a host of options to data visualisation in R. In this course, we will mainly use the `ggplot2` package, which forms part of the larger `tidyverse` collection of packages which provide functions for efficient data management in R. We will also use eother packages within `tidyverse` in the course.

A grammar of graphics is a standardised way to describe the components of a graphic. `ggplot2` uses a layered grammar of graphics, in which plots are bulit up in a series of layers. It may be helpful to think about any picture as having multiple elements that sit semi-transparently over each other. Figure 1.1 shows the evolution of a sumple scatterplot using this layered approach. First, the plot space is built (layer 1); the variables are specified (layer 2); the type of visualisation that is desired for these variables is specified (layer 3) - in this case `geom_point()`is called to visualise individual data points; a second `geom` layer is added to include a line of best fit (layer 4); the axis labels are editied

for readability (layer 5) and finally, a theme is applied to change the overall appearance of the plot (layer 6).

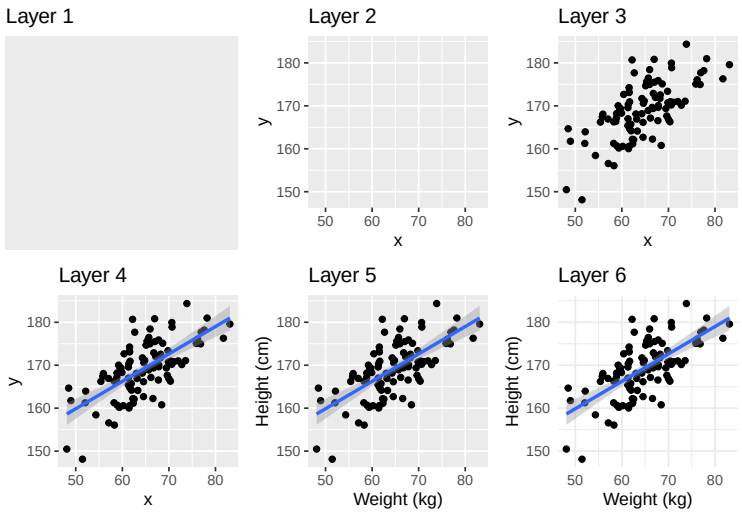


Figure 1.1: Evolution of a layered plot.

Each layer is independent and individually customisable. For example, the size, colour and position of each component can be adjusted. The use of layers makes it easy to build up complex plots step-by-step, and to adapt or extend plots from existing code.

1.2 Data

In this course, we will use some datasets for analysis. You can download these from the following repository:

Link to repository

1.3 Libraries

Throughout this week, we will use some libraries within R for modelling and data visualisation. You will need the following libraries installed on your version of R:

- `tidyverse`
- `lubridate`
- `magrittr`
- `MASS`
- `ggmap`
- `plotly`

- `mgcv`
- `sp`
- `splines`
- `gratia`
- `spdep`
- `sf`
- `CARBayes`
- `CARBayesST`
- `RColorBrewer`
- `gridExtra`
- `leaflet`

Chapter 2

Getting Started

2.1 Setting up R

You need to have access to R for this course. You can download R for free from CRAN.

R is available for Windows, Mac OS and Linux as well as some less common platforms.

You can download the standard version of R from CRAN.

2.1.1 Downloading and installing R for Windows

To download the Windows installer of R, just enter the following URL (or click on the clink).

https://cran.r-project.org//bin/windows/base/release.html

This will download the most installer for the most recent version of R. Alternatively, you can go to the main CRAN page, https://cran.r-project.org/, and then click on "Download R for Windows", click on "base" and then on "Download R x.y.z for Windows" (where x.y.z is the current version number of R).

You can then run the installer, accepting all default settings.

2.1.2 Downloading and installing R for Mac

To download the Windows installer for Mac, just enter the following URL (or click on the clink).

https://cran.r-project.org/bin/macosx/

From here, select the most recent version of R and the .pkg file will automatically download. This file will be in the form "R-x.y.z" (where x.y.z is the current version number of R).

Once the file is opened, the installer will open and you can select the default settings.

2.2 RStudio

It is recommended that you also download and install RStudio Desktop, a powerful integrated development environment (IDE) for R. RStudio contains a much better code editor. It has, for example, syntax highlighting, i.e. it will automatically display your code in different colours to make it easier and quicker to read the code. Even though other IDEs, such as Eclipse, Visual Studio Code, or Emacs can also be used with R, RStudio is by far the most popular among R users.

RStudio is just a front-end for R, so to be able make use of RStudio, you need to also have R installed.

RStudio Desktop Open Source is available for free from RStudio.

2.2.1 Installing RStudio for Windows/Mac

Go to

https://www.rstudio.com/products/rstudio/download/

and scroll down to the section "All installers", then click on "RStudio-x-y-z.exe" in the first row of the table for a Windows install, or click on "RStudio-x-y-z.dmg" for a macOS install. This should start the download of the RStudio installer.

You can then run the installer, accepting all default settings (for macOS, you will need to drag and drop the application into the applications folder once open).

2.3 R packages

R comes with a default selection of packages, which should cover your "basic needs" in terms of data management, data visualisation and modelling. However, there is a large selection of "add-on" R packages available on CRAN, some of which we will use in this course. You can only use these R packages once you have installed them.

Imagine you want to use an R package called `ggplot2` (which we will use later in this course). In order to be able to use it, you first need to install it. You can do so by entering

```
install.packages("ggplot2")
```

into R. This will download and install the package `ggplot2`, as well as any other packages `ggplot` uses.

Alternatively, you can click on the tab "Packages in the bottom-right panel, and then click on"Install".

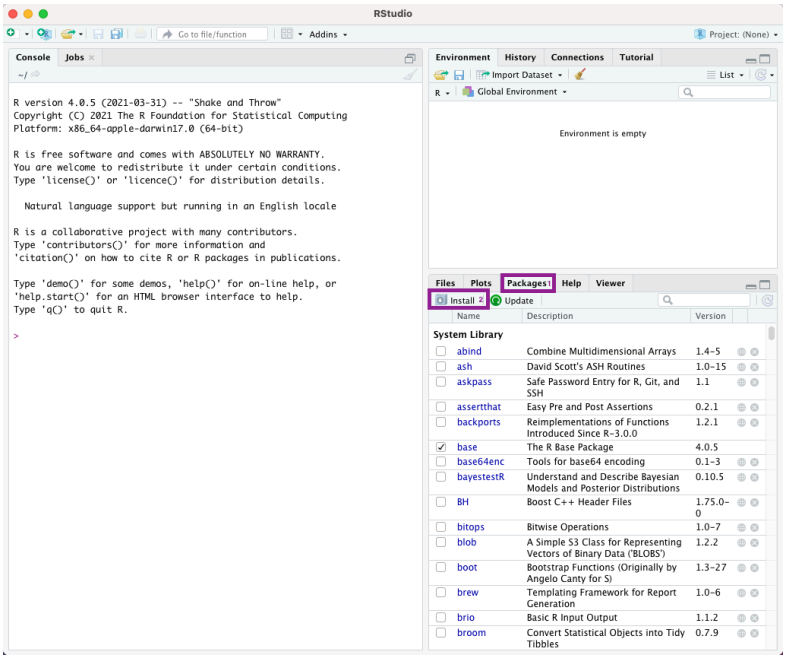


Figure 2.1: Selecting the installation menu for installing packages

You can then enter the name of the package you want to install and click on "Install"

Once you have installed an R package, you can load it using the function `library`

```
library(ggplot2)
```

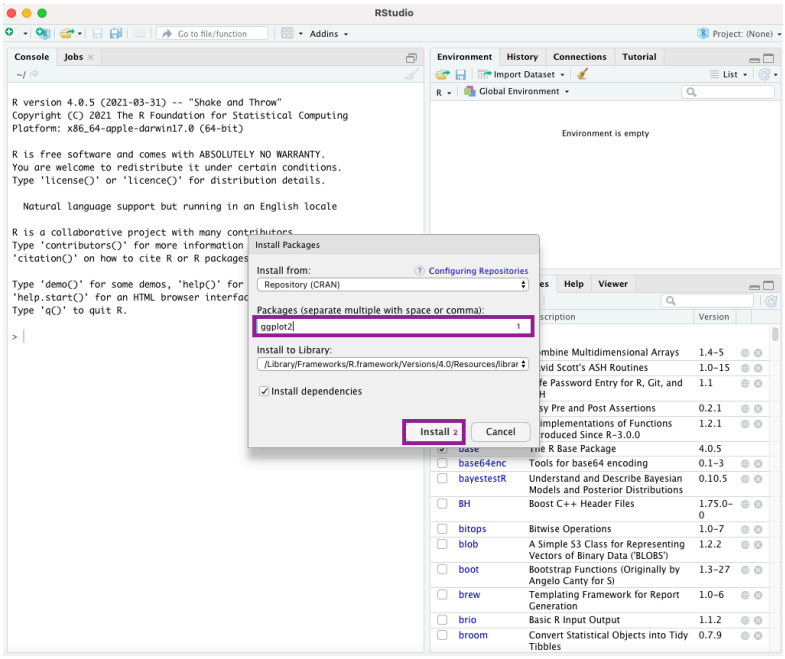


Figure 2.2: Installation menu for packages

Chapter 3

Efficient Data Management in R

3.1 Tidyverse

Tidyverse is a collection of R packages designed to help data scientists to make more efficient use of R. It contains the following packages (and several more, which we will:

- tibble provides a "modern reimagining" of the standard `data.frame`, in R. Tibbles (or `tbl_df`s) are more flexible in terms of what they can store, but (purposefully) less flexible in terms of "sloppy code".
- readr provides alternative functions for reading in text data in tabular form. It provides faster and more consistent alternatives to `read.table` and `read.csv`.
- dplyr provides a powerful suite of functions for data manipulation with a focus on allowing for clean and simple code. We will look at `dplyr` in more detail this week.
- ggplot2 is a very featureful and systematic set of plotting functions, which we will focus on in this tutorial.
- lubridate is a very useful package for handling dates and times in R. Dates and times are often tricky to deal with, and `lubridate` provides many useful functions for efficiently handling these.

3.2 Pipelines

Pipelines are at the centre of all the tidyverse packages. The R package magrittr provides a forward-pipe operator for R.

Suppose we have a function `f` defined in R

```
f <- function(x)
  x^2
```

Then we can apply `f` to an argument `x` using

```
x <- 3
f(x)
```

```
## [1] 9
```

The forward-pipes from magrittr allow us to rewrite this function call as

```
x %>% f
```

```
## [1] 9
```

instead. The advantage of this alternative notation might not become immediately clear, but its advantage becomes more obvious when looking at nested function calls.

Consider the R data set `mtcars`, which contains data from the 1974 edition from the US magazine Motor Trend. Suppose we want to convert the fuel consumption to litres per 100 kilometres and then only retain the cars with a fuel economy better than 10 litres per 100 kilometres.

```
mtcars2 <- transform(mtcars, lp100k=235.21/mpg)
subset(mtcars2, lp100k<=10)
```

```
##                 mpg cyl  disp  hp drat    wt  qsec vs am gear carb   lp100k
## Merc 240D      24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2 9.639754
## Fiat 128       32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1 7.259568
## Honda Civic    30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2 7.737171
## Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1 6.938348
## Fiat X1-9      27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1 8.615751
## Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2 9.046538
## Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2 7.737171
```

(If you are wondering where the number of 235.21 comes from: A US gallon is roughly 3.785 litres and a mile is roughly 1.609 kilometres, and $\frac{100 \times 3.785}{1.609} \approx 235.21$)

If we want to perform both steps in one go, we can nest the two calls within one another and use

```
subset(transform(mtcars, lp100k=235.21/mpg), lp100k<=10)
```

This gives exactly the same results, but is not very easy to read and understand. It is not easy to see that the argument `lp100k<=10` belongs to `subset`. When nesting function calls, the arguments get increasingly far from the function call to which they belong.

The `%>%` operator however allows us to write this much more cleanly:

```
mtcars %>%
  transform(lp100k=235.21/mpg) %>%
  subset(lp100k<=10)
```

3.2.1 Task

The R code below uses pipelines. Convert it to nested function calls.

```
rnorm(1000) %>% sin() %>% max()
```

3.2.2 Solution

The code generates a random sample of size 1000 (from a standard normal distribution), computes the sine of each entry and then takes the maximum.

```
max(sin(rnorm(1000)))
```

```
## [1] 0.9999919
```

In this case the nested function call is easy to read because every function only takes one argument.

3.2.3 Task

Convert the R code below to pipelines.

```
library(MASS)                          # Load package MASS, which contains the data
subset(transform(mammals, ratio=brain/body), ratio==max(ratio))
```

3.2.4 Answer

You can use the following R code using pipelines.

```
library(MASS)
mammals %>%
  transform(ratio=brain/body) %>%
  subset(ratio==max(ratio))
```

```
##                   body brain    ratio
## Ground squirrel 0.101     4 39.60396
```

Oddly enough, ground squirrels have a higher brain-to-body weight ratio than humans.

3.2.5 Additional Resources

Pipelines for Data Analysis in R Hadley Wickham has produced a series of excellent slides about pipelines, which covers much of what we will look at in this tutorial.

Background reading: Chapter 18 of R for Data Science Chapter 18 of *R for Data Science* gives a detailed overview of pipes and some of the underpinning technology (though the latter is rather advanced).

3.3 Tibbles

The package tibble provides `tbl_df`'s (or "tibbles", which is easier to pronounce). They are a modern take on the built-in class `data.frame`.

One key advantage of tibbles is that they can store anything. A `data.frame` can only store a single value per "cell", for example a number or a character string. However, in a tibble, you can store a list or even another tibble in a cell. An example of this is the tibble `starwars` from the package `dplyr`. The column `starships` contains for each row the list of starships flown by that character (which is a list of different length depending on the character.)

```
library(dplyr)                          # Load library dplyr which contains the dat
starwars[,c("name", "starships")]       # Print columns name and starships
```

```
## # A tibble: 87 x 2
##    name               starships
##    <chr>              <list>
##  1 Luke Skywalker     <chr [2]>
##  2 C-3PO              <chr [0]>
##  3 R2-D2              <chr [0]>
##  4 Darth Vader        <chr [1]>
##  5 Leia Organa        <chr [0]>
##  6 Owen Lars          <chr [0]>
##  7 Beru Whitesun lars <chr [0]>
##  8 R5-D4              <chr [0]>
##  9 Biggs Darklighter  <chr [1]>
## 10 Obi-Wan Kenobi     <chr [5]>
## # ... with 77 more rows
```

```
starwars[10,"starships"][[1]]           # Starships flown by Obi-Wan
```

```
## [[1]]
## [1] "Jedi starfighter"         "Trade Federation cruiser"
## [3] "Naboo star skiff"         "Jedi Interceptor"
## [5] "Belbullab-22 starfighter"
```

We could not have stored this information in a data frame. We would have had to either store the information across several data frames or stored the list of starships as a character string.

3.3.1 Creating tibbles

We can create tibbles using the function `tibble`. We can create the tibble from above using

```
kidstibble <- tibble(name=c("Sarah", "John"), age=c(4,11), weight=c(15,28),
                     height=c(101,132), gender=c("f", "m"))
```

In other words, the function `tibble` assembles a tibble on a column-by-column basis (akin to using `cbind`).

The function `tribble` ("transposed tibble") lets you create a tibble on a row-by-bow basis (akin to using `rbind`), which is typically more legible when creating a matrix in code.

```
kidstibble <- tribble(~name,   ~age, ~weight, ~height, ~gender,
                      "Sarah",    4,      15,     101,     "f",
                      "John",    11,      28,     132,     "m")
```

3.3.2 Working with tibbles

- Variables/Columns can be accessed and added using `tibble$varname` (`varname` needs to be fully spelled out). You can also access a column using `tibble[,"varname"]` or `tibble[["varname"]]`.
- Rows can be selected using `tibble[rowindices,]` (note that you cannot use row names).
- Individual cells can be accessed using `tibble[rowindices, colindices]`.

3.3.3 Subsetting tibbles always results in a tibble

Tibbles are also more consistent. Subsetting tibbles always results in a tibble.

```
kidstibble[,1]                                  # Result is a tibble
```

```
## # A tibble: 2 x 1
##   name
##   <chr>
## 1 Sarah
## 2 John
```

In contrast, subsetting a data frame or matrix is not guaranteed to result in a data frame or matrix (unless you use `drop=FALSE`). If the result is a single column or row, subsetting a data frame or matrix results in a vector.

This "dropping" of the dimension can be very useful when using R interactively, but can be the source of many issues in more complex projects, when programmers incorrectly assume that subsetting a data frame or matrix will always result in another data frame or matrix, rather than possibly just a vector (it is thus a good idea to always use `drop=FALSE` when working with data frames or matrices in complex projects).

Data Import Cheat Sheet RStudio's cheat sheet for data import also covers tibbles.

3.4 Reading in data using readr

The package readr contains alternatives to the functions `read.table` and `read.csv`. The alternative functions from `readr` have four main advantages.

- They read in the data a lot faster and can show a progress bar (though this is only relevant for really big data sets).
- They store the data straight in a tibble, rather than a data frame.
- They allow specifying the intended data type for each column and thus make it easier to identify rows which cause problems.
- They are less intrusive: they don't automatically convert character strings to factors and do not change column names (`read.table` and `read.csv` will for example remove spaces from variable names and replace them by full stops). The functions from `readr` are also guaranteed to give the same result irrespective of the platform or operating system they are run under.

`readr` provides the following functions.

- `read_csv` reads in comma-separated files. `read_csv2` reads in files which are semicolon-separated (common in countries like France or Germany, where a comma is used as decimal separator).
- `read_tsv` reads in tab-separated files.
- `read_delim` is the most general function (like `read.table`). The delimiter has to be specified using the argument `delim`.
- `read_fwd` reads in fixed-width files.

All functions assume that the first row contains the column/variable names. If this is not the case, set the optional argument `col_names` to `FALSE` or to a character vector containing the intended column names.

The strings used to encode missing values can be specified using the optional argument `na`.

For example, we can read in the file chol.txt using

```
library(readr)
read_delim("chol.txt", delim=" ", col_names=c("ldl", "hdl", "trig",
                                              "age", "gender", "smoke"))
```

```
## Rows: 13 Columns: 6
## -- Column specification --------------------------------------------------------
## Delimiter: " "
## chr (2): gender, smoke
## dbl (4): ldl, hdl, trig, age
##
## i Use `spec()` to retrieve the full column specification for this data.
```

```
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
## # A tibble: 13 x 6
##      ldl   hdl  trig   age gender smoke
##    <dbl> <dbl> <dbl> <dbl> <chr>  <chr>
##  1   175    25   148    39 female no
##  2   196    36    92    32 female no
##  3   139    65    NA    42 male   <NA>
##  4   162    37   139    30 female ex-smoker
##  5   140   117    59    42 female ex-smoker
##  6   147    51   126    65 female ex-smoker
##  7    82    81    NA    57 male   no
##  8   165    63   120    48 male   current
##  9   149    49    NA    32 female no
## 10    95    54   157    55 female ex-smoker
## 11   169    59    67    48 female no
## 12   174   117   168    41 female no
## 13    91    52   146    69 female current
```

Note that functions from `readr` show the data type it has used for each column. This makes it easier to spot mistakes like missing values not coded as expected, in which case a numeric column would show up as a character string.

For example, we can read in the file chol.csv using

```
library(readr)
read_csv("chol.csv", na=".")
```

```
## Rows: 13 Columns: 6
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (2): gender, smoke
## dbl (4): ldl, hdl, trig, age
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
## # A tibble: 13 x 6
##      ldl   hdl  trig   age gender smoke
##    <dbl> <dbl> <dbl> <dbl> <chr>  <chr>
##  1   175    25   148    39 female no
##  2   196    36    92    32 female no
##  3   139    65    NA    42 male   NA
##  4   162    37   139    30 female ex-smoker
##  5   140   117    59    42 female ex-smoker
##  6   147    51   126    65 female ex-smoker
##  7    82    81    NA    57 male   no
##  8   165    63   120    48 male   current
```

```
##  9   149    49    NA    32 female no
## 10    95    54   157    55 female ex-smoker
## 11   169    59    67    48 female no
## 12   174   117   168    41 female no
## 13    91    52   146    69 female current
```

3.4.1 Task

Read the data files cars.csv and ships.txt into R using the functions from `readr`.

3.4.2 Answer

The first line of the file `cars.csv` contains the variable names and the fields are separated by commas. Missing values are encoded as asterisks.

```
cars <- read_csv("cars.csv", na="*")
```

```
## Rows: 20 Columns: 5
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (2): Manufacturer, Model
## dbl (3): MPG, Displacement, Horsepower
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
cars
```

```
## # A tibble: 20 x 5
##    Manufacturer Model     MPG Displacement Horsepower
##    <chr>        <chr>   <dbl>        <dbl>      <dbl>
##  1 Chevrolet    Camaro     19          3.4        160
##  2 Oldsmobile   Achieva    NA          2.3        155
##  3 Dodge        Spirit     22          2.5        100
##  4 Chevrolet    Astro      NA          4.3        165
##  5 Chevrolet    Corsica    25          2.2        110
##  6 Volkswagen   Corrado    18          2.8        178
##  7 Dodge        Stealth    18          3          300
##  8 Volkswagen   Fox        25          1.8         81
##  9 Cadillac     DeVille    16          4.9        200
## 10 Hyundai      Excel      29          1.5         81
## 11 Toyota       Tercel     32          1.5         82
## 12 Dodge        Colt       29          1.5         92
## 13 Volkswagen   Passat     21          2          134
## 14 Geo          Storm      30          1.6         90
## 15 Toyota       Previa     18          2.4        138
## 16 Nissan       Sentra     29          1.6        110
## 17 Toyota       Celica     25          2.2        135
```

```
## 18 Honda        Civic      42          1.5        102
## 19 Dodge        Caravan    17          3          142
## 20 Hyundai      Sonata     20          2          128
```

We could have also used the function `read_delim`.

```
read_delim("cars.csv", delim=",", na="*")
```

```
## Rows: 20 Columns: 5
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (2): Manufacturer, Model
## dbl (3): MPG, Displacement, Horsepower
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
## # A tibble: 20 x 5
##    Manufacturer Model     MPG Displacement Horsepower
##    <chr>        <chr>   <dbl>        <dbl>      <dbl>
##  1 Chevrolet    Camaro     19          3.4        160
##  2 Oldsmobile   Achieva    NA          2.3        155
##  3 Dodge        Spirit     22          2.5        100
##  4 Chevrolet    Astro      NA          4.3        165
##  5 Chevrolet    Corsica    25          2.2        110
##  6 Volkswagen   Corrado    18          2.8        178
##  7 Dodge        Stealth    18          3          300
##  8 Volkswagen   Fox        25          1.8         81
##  9 Cadillac     DeVille    16          4.9        200
## 10 Hyundai      Excel      29          1.5         81
## 11 Toyota       Tercel     32          1.5         82
## 12 Dodge        Colt       29          1.5         92
## 13 Volkswagen   Passat     21          2          134
## 14 Geo          Storm      30          1.6         90
## 15 Toyota       Previa     18          2.4        138
## 16 Nissan       Sentra     29          1.6        110
## 17 Toyota       Celica     25          2.2        135
## 18 Honda        Civic      42          1.5        102
## 19 Dodge        Caravan    17          3          142
## 20 Hyundai      Sonata     20          2          128
```

The first line of the file `ships.txt` contains the variable names and the fields are separated by whitespace. Missing values are encoded as ".".

```
ships <- read_delim("ships.txt", delim=' ' , na=".")
```

```
## Rows: 40 Columns: 5
## -- Column specification --------------------------------------------------------
## Delimiter: " "
```

```
## chr (1): type
## dbl (4): year, period, service, incidents
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
ships
```

```
## # A tibble: 40 x 5
##    type   year period service incidents
##    <chr> <dbl>  <dbl>   <dbl>     <dbl>
##  1 A        60     60     127         0
##  2 A        60     75      63         0
##  3 A        65     60      NA         3
##  4 A        65     75    1095         4
##  5 A        70     60    1512         6
##  6 A        70     75    3353        18
##  7 A        75     60       0         0
##  8 A        75     75    2244        11
##  9 B        60     60   44882        39
## 10 B        60     75   17176        29
## # ... with 30 more rows
```

3.4.3 Specifying column types

The functions from `readr` allow specifying the expected column types. This is especially important when writing which will then be run automatically. It provides an easy way of ensuring that the data provided is of the expected format.

The easiest way of specifying expected column types is to provide a character string with each letters standing for a column

Letter	Meaning
`c`	character
`i`	integer
`n`	number
`d`	double
`l`	logical
`D`	date
`T`	date time
`t`	time
`?`	guess the type
`_` or `-`	skip the column

So for the data file chol.csv we would expect the first four columns to be integers

and the latter two to be character strings, so we would use

```
chol <- read_csv("chol.csv", na=".", col_types="iiiicc")
```

Specifying the expected column types can help pinpointing problems when reading in data. Suppose we had forgotten that missing values are coded using "." in this data file. If we use ...

```
chol <- read_csv("chol.csv")
```

```
## Rows: 13 Columns: 6
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (3): trig, gender, smoke
## dbl (3): ldl, hdl, age
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

... we can see from the output that `trig` was read in as a character string, but we do not know why.

However, if we use ...

```
chol <- read_csv("chol.csv", col_types="iiiicc")
```

```
## Warning: One or more parsing issues, see `problems()` for details
```

... we obtain a warning and can print the problematic rows using

```
problems(chol)
```

```
## # A tibble: 3 x 5
##     row   col expected   actual file
##   <int> <int> <chr>      <chr>  <chr>
## 1     4     3 an integer .      /Users/Craig/Library/CloudStorage/OneDrive-Shar~
## 2     8     3 an integer .      /Users/Craig/Library/CloudStorage/OneDrive-Shar~
## 3    10     3 an integer .      /Users/Craig/Library/CloudStorage/OneDrive-Shar~
```

The output from `problems` shows us that for three rows (3, 7 and 9) the data in `chol.csv` was not of the expected format: a value of `.` is not compatible with the column being numeric. This makes it easy to identify the cause of the problem (NAs coded as ".") and rectify the issue.

3.4.4 Additional Resources

Data Import Cheat Sheet RStudio's cheat sheet for data import also covers `readr`.

Background reading: Chapter 11 of R for Data Science Chapter 11 of *R for Data Science* gives a detailed overview of the functions in `readr`. It explains in

some more detail how the functions in `readr` parse files. over pipes and also covers the functions from `readr` that dead with writing files.

3.5 Efficient data manipulation using dplyr

In this section we will work with data from Paris' Vélib' bicycle sharing system available through JCDecaux's API for open cycle data.

The data consists of the number of bikes available and the number of bike stands available at every Vélib' station, recorded every five minutes over six hours on a Tuesday afternoon in October 2017.

The data consists of two tibbles. The first, `bikes` contains data on the number of available bikes and stands at each station.

Variable	Description
`name`	Name of the station
`available_bikes`	Number of available at that time
`available_bike_stands`	Number of available bike stands
`time`	Decimal time for which the number have been recorded

The second, `stations` contains additional information about each station.

Variable	Description
`name`	Unique name of the station
`id`	Internal ID number of the station
`address`	Address of where the station is located
`lng`	GPS coordinate (longitude)
`lat`	GPS coordinate (latitude)
`departement`	Département in which the station is located

You can load the data into R using

```
library(tibble)
load(url("https://github.com/UofGAnalyticsData/R/raw/main/Week%204/velib"))
```

3.5.1 Overview: the key functions ("verbs") for `dplyr`

Function ("verb")	Description	R base equivalent(s)
`filter`	Select observations/rows	`subset`
`slice`	Select observations by row numbers	`[idx,]`
`select`	Select variables/column	`$` or `[,sel]`

Function ("verb")	Description	R base equivalent(s)
`mutate`	Create new variables/column	`transform`
`arrange`	Sort observations/rows	`order`
`group_by`	Group observations by variable	`by` or `aggregate`
`summarise`	Calculate summary statistics	`by` or `aggregate`

The functions in `dplyr` are designed to be used with tibbles, but they also work with data frames. When invoked with a data frame, they will return a data frame as long as this is possible.

3.5.2 Selecting observations (rows) using `filter` and `slice`

3.5.2.1 `filter`

The function `filter` is used to select observations (or rows) in a similar way to the base R function `subset`.

Suppose we want to print all bike stations in Paris (rather than other départements from Île de France)

```
library(dplyr)
stations75 <- stations %>%
                filter(departement=="Paris")
stations75
```

```
## # A tibble: 743 x 6
##    name                                    id address   lng   lat departement
##    <chr>                                <dbl> <chr>   <dbl> <dbl> <chr>
##  1 PORT SOLFERINO (STATION MOBILE)        901 BERGES~  2.32  48.9 Paris
##  2 QUAI MAURIAC  / PONT DE BERCY          903 FETE D~  2.37  48.8 Paris
##  3 17/19 PLACE JOFFRE / ECOLE MILITAIRE   904 ECOLE ~  2.30  48.9 Paris
##  4 CONCORDE/BERGES DE SEINE (STATION MOBI~  905 BERGES~  2.32  48.9 Paris
##  5 PORT DU GROS CAILLOU (STATION MOBILE)  908 BERGES~  2.31  48.9 Paris
##  6 PONT D'ARCOLE (STATION MOBILE)         909 Voie G~  2.35  48.9 Paris
##  7 ILE DE LA CITE PONT NEUF              1001 41 QUA~  2.34  48.9 Paris
##  8 PLACE DU CHATELET                     1002 14 AVE~  2.35  48.9 Paris
##  9 RIVOLI SAINT DENIS                    1003 7 RUE ~  2.35  48.9 Paris
## 10 MARGUERITE DE NAVARRE                 1004 12 RUE~  2.35  48.9 Paris
## # ... with 733 more rows
```

Note the use of a double `==` to test whether the département is equal to "Paris".

We can create more complex expressions using the standard logical operators `&` ("and"), `|` ("or") and `!` ("not"). Note that you *cannot* use `&&` and `||` in this context, as they only work with scalar arguments.

For example, if we want to extract the stations which are in Paris or Hauts-de-Seine we can use

```
stations7592 <- stations %>%
                  filter(departement=="Paris" | departement=="Hauts-de-Seine")
```

Rather than using a logical or we could have used `%in%`:

```
stations7592 <- stations %>%
                  filter(departement %in% c("Paris" , "Hauts-de-Seine"))
```

Even though the functions from `dplyr` are designed to be used with pipelines, you can also provide the data set as first argument:

```
stations7592 <- filter(stations, departement %in% c("Paris" , "Hauts-de-Seine"))
```

3.5.2.2 `slice`

You can use the function `slice` to select observations based on their row numbers.

```
stations %>%
  slice(5:7)
```

```
## # A tibble: 3 x 6
##   name                                    id address      lng   lat departement
##   <chr>                                <dbl> <chr>      <dbl> <dbl> <chr>
## 1 PORT DU GROS CAILLOU (STATION MOBILE)   908 BERGES DE~  2.31  48.9 Paris
## 2 PONT D'ARCOLE (STATION MOBILE)          909 Voie Geor~  2.35  48.9 Paris
## 3 ILE DE LA CITE PONT NEUF               1001 41 QUAI D~  2.34  48.9 Paris
```

selects the observations in rows 5 to 7 and is equivalent to

```
stations[5:7,]
```

```
## # A tibble: 3 x 6
##   name                                    id address      lng   lat departement
##   <chr>                                <dbl> <chr>      <dbl> <dbl> <chr>
## 1 PORT DU GROS CAILLOU (STATION MOBILE)   908 BERGES DE~  2.31  48.9 Paris
## 2 PONT D'ARCOLE (STATION MOBILE)          909 Voie Geor~  2.35  48.9 Paris
## 3 ILE DE LA CITE PONT NEUF               1001 41 QUAI D~  2.34  48.9 Paris
```

3.5.3 Task

Identify the stations which had more than 60 bikes available at 3pm (i.e. `time` taking the value 15).

3.5.4 Answer

You can use the following R code:

```
bikes %>%
  filter(time==15 & available_bikes>60)
```

```
## # A tibble: 6 x 4
##   name                available_bikes available_bike_stands  time
##   <chr>                         <int>                 <int> <dbl>
## 1 MUSÉE D'ORSAY                    63                     2    15
## 2 DUPLEIX                          65                     3    15
## 3 ASSEMBLEE NATIONALE              62                     0    15
## 4 SAINT EMILION                    65                     1    15
## 5 METZ                             63                     1    15
## 6 PRIMO LEVI                       61                     1    15
```

3.5.5 Selecting variables (columns) using `select`

The function `select` can be used to subset the variables (columns) of a data set.

You can either specify the columns to retain or (with a minus) those you do not want to retain.

We can only retain the name and département of each station using either

```
stations.small <- stations %>%
                    select(name, departement)
stations.small
```

```
## # A tibble: 928 x 2
##    name                                    departement
##    <chr>                                   <chr>
##  1 PORT SOLFERINO (STATION MOBILE)         Paris
##  2 QUAI MAURIAC  / PONT DE BERCY           Paris
##  3 17/19 PLACE JOFFRE / ECOLE MILITAIRE    Paris
##  4 CONCORDE/BERGES DE SEINE (STATION MOBILE) Paris
##  5 PORT DU GROS CAILLOU (STATION MOBILE)   Paris
##  6 PONT D'ARCOLE (STATION MOBILE)          Paris
##  7 ILE DE LA CITE PONT NEUF                Paris
##  8 PLACE DU CHATELET                       Paris
##  9 RIVOLI SAINT DENIS                      Paris
## 10 MARGUERITE DE NAVARRE                   Paris
## # ... with 918 more rows
```

or

```
stations.small <- stations %>% select(-id, -address, -lng, -lat)
```

You can also use `select` to change the order of the columns of a data set.

3.5.6 Adding new variables using `mutate`

The function `mutate` can be used to create new variables (columns) in a data set. `mutate` is similar in functionality to the base R function `transform`.

We can add the total number of stands to the data set `bikes` using

```
bikes <- bikes %>%
           mutate(total_stands = available_bikes+available_bike_stands)
```

More than one new variable can be defined by adding further arguments to `mutate`.

`transmute` is a sibling of `mutate`. Just like `mutate` it creates new columns. It however also removes all existing columns so that only the new columns remain.

3.5.7 Task

The time is currently encoded as decimal (e.g. `13.5` for 13:30). Create two columns `time_hours`, which contains the hour (13 in our example), and `time_minutes`, which contains the minutes, (30 in our example).

You can calculate `time_hours` as the floor of `time` (R function `floor`) and `time_minutes` as the remainder after integer division of 60 times `time` by 60 (R operator `%%`).

3.5.8 Answer

We can create both columns in one call to `mutate`.

```
bikes %>%
  mutate(time_hour=floor(time), time_minutes=(60*time)%%60)
```

```
## # A tibble: 67,354 x 7
##    name            available_bikes available_bike_~  time total_stands time_hour
##    <chr>                     <int>            <int> <dbl>        <int>     <dbl>
##  1 CHAMPEAUX (BAG~               9               41    13           50        13
##  2 POISSONNIÈRE --~             33                0    13           33        13
##  3 METRO ROME                    6               38    13           44        13
##  4 DE GAULLE (PAN~               2               16    13           18        13
##  5 PARC DE BELLEV~               4               22    13           26        13
##  6 SOLJENITSYNE (~              56                4    13           60        13
##  7 SERRES                        5               18    13           23        13
##  8 PYRAMIDE ARTIL~              14               40    13           54        13
##  9 SAINT GEORGES                12               10    13           22        13
## 10 MUSÉE D'ORSAY                65                0    13           65        13
## # ... with 67,344 more rows, and 1 more variable: time_minutes <dbl>
```

The output does not show the new columns (as they would take the output of a single row to more than one line). We can show them all, for example, if we

remove the station name.

```
bikes %>%
  mutate(time_hour=floor(time), time_minutes=(60*time)%%60) %>%
  select(-name)
```

```
## # A tibble: 67,354 x 6
##    available_bikes available_bike_sta~  time total_stands time_hour time_minutes
##              <int>               <int> <dbl>        <int>     <dbl>        <dbl>
##  1               9                  41    13           50        13            0
##  2              33                   0    13           33        13            0
##  3               6                  38    13           44        13            0
##  4               2                  16    13           18        13            0
##  5               4                  22    13           26        13            0
##  6              56                   4    13           60        13            0
##  7               5                  18    13           23        13            0
##  8              14                  40    13           54        13            0
##  9              12                  10    13           22        13            0
## 10              65                   0    13           65        13            0
## # ... with 67,344 more rows
```

Alternatively, we can explicitly invoke the print method of the tibble and ask it to print everything.

```
bikes %>%
  mutate(time_hour=floor(time), time_minutes=(60*time)%%60) %>%
  print(width=Inf)
```

```
## # A tibble: 67,354 x 7
##    name                        available_bikes available_bike_stands  time
##    <chr>                                 <int>                 <int> <dbl>
##  1 CHAMPEAUX (BAGNOLET)                      9                    41    13
##  2 POISSONNIÈRE - ENGHIEN                   33                     0    13
##  3 METRO ROME                                6                    38    13
##  4 DE GAULLE (PANTIN)                        2                    16    13
##  5 PARC DE BELLEVILLE (20040)                4                    22    13
##  6 SOLJENITSYNE (PUTEAUX)                   56                     4    13
##  7 SERRES                                    5                    18    13
##  8 PYRAMIDE ARTILLERIE                      14                    40    13
##  9 SAINT GEORGES                            12                    10    13
## 10 MUSÉE D'ORSAY                            65                     0    13
##    total_stands time_hour time_minutes
##           <int>     <dbl>        <dbl>
##  1           50        13            0
##  2           33        13            0
##  3           44        13            0
##  4           18        13            0
##  5           26        13            0
```

```
##  6           60        13            0
##  7           23        13            0
##  8           54        13            0
##  9           22        13            0
## 10           65        13            0
## # ... with 67,344 more rows
```

3.5.9 Sorting data sets using `arrange`

The function `arrange` can be used to sort a data set by one or more variables. We can sort the data set `bikes` by the number of available bikes suing

```
bikes %>%
  arrange(available_bikes)
```

```
## # A tibble: 67,354 x 5
##    name                       available_bikes available_bike_~  time total_stands
##    <chr>                                <int>            <int> <dbl>        <int>
##  1 KARMAN (AUBERVILLIERS)                   0                0    13            0
##  2 PIGALLE GERMAIN PILLON                   0               20    13           20
##  3 ROND POINT DES CHAMPS EL~                0                0    13            0
##  4 MONTCALM                                 0               47    13           47
##  5 PLACE HENOCQUE VERSION 2                 0               34    13           34
##  6 PLACE DES FETES                          0               19    13           19
##  7 MANIN SECRETAN                           0               20    13           20
##  8 MARTINIE (VANVES)                        0               24    13           24
##  9 HORTENSIAS (LES LILAS)                   0               22    13           22
## 10 HAIES REUNION                            0               22    13           22
## # ... with 67,344 more rows
```

You can use the function `desc` to sort in descending order

```
bikes %>%
  arrange(desc(available_bikes))
```

```
## # A tibble: 67,354 x 5
##    name    available_bikes available_bike_stands  time total_stands
##    <chr>             <int>                <int> <dbl>        <int>
##  1 DUPLEIX              68                    0  16.2           68
##  2 DUPLEIX              68                    0  16.2           68
##  3 DUPLEIX              67                    1  15.4           68
##  4 DUPLEIX              67                    1  15.5           68
##  5 DUPLEIX              67                    1  15.8           68
##  6 DUPLEIX              67                    1  16.1           68
##  7 DUPLEIX              67                    1  16.3           68
##  8 SAHEL                67                    0  17.6           67
##  9 SAHEL                67                    0  18             67
## 10 SAHEL                67                    0  18.1           67
```

```
## # ... with 67,344 more rows
```

3.5.10 Task

Identify the three bike stations that are furthest to the West (i.e. the ones with the smallest longitude `lng`).

3.5.11 Answer

We first sort the stations by the longitude and the select to top three observations.

```
stations %>%
  arrange(lng) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 6
##   name                           id address               lng   lat departement
##   <chr>                       <dbl> <chr>               <dbl> <dbl> <chr>
## 1 GARE ROUTIERE ( SAINT CLOUD) 22101 GARE ROUTIERE - AR~  2.22  48.8 Hauts-de-S~
## 2 SELLIER (SURESNES)           21501 RUE DE SAINT CLOUD~  2.23  48.9 Hauts-de-S~
## 3 VERDUN (SURESNES)            21502 18 BIS RUE DE VERD~  2.23  48.9 Hauts-de-S~
```

We could have also used the function `filter` and the ranking function `min_rank`:

```
stations %>%
  filter(min_rank(lng)<=3)
```

```
## # A tibble: 3 x 6
##   name                           id address               lng   lat departement
##   <chr>                       <dbl> <chr>               <dbl> <dbl> <chr>
## 1 SELLIER (SURESNES)           21501 RUE DE SAINT CLOUD~  2.23  48.9 Hauts-de-S~
## 2 VERDUN (SURESNES)            21502 18 BIS RUE DE VERD~  2.23  48.9 Hauts-de-S~
## 3 GARE ROUTIERE ( SAINT CLOUD) 22101 GARE ROUTIERE - AR~  2.22  48.8 Hauts-de-S~
```

`min_rank` returns the rank of the observation when considering the variable given as argument (there are many different ways of computing ranks, see `?min_rank` for details.)

However, the latter answer does not show the stations in increasing order of longitude.

3.5.12 Grouping data and calculating group-wise summary statistics: `group_by` and `summarise`

Suppose we want to identify the busiest stations in the system in the sense of having, on average, the most bikes taken out (and thus the highest number of available bike stands – this is assuming JCDecaux replenish all bike stations in

the same way, which is not quite what is happening in reality; there are better, but more complex, ways of defining "busy").

To calculate the average number of available bike stands per station we need to first group the data by bike station and then compute the average number of bike stands available

```
bikes %>% group_by(name) %>%                                  # Group by station name
  summarise(avg_stands=mean(available_bike_stands)) %>%  # Calculate averages
  arrange(desc(avg_stands))                                   # Sort in descending order
```

```
## # A tibble: 928 x 2
##    name                                avg_stands
##    <chr>                                    <dbl>
##  1 PANTIN                                    70.3
##  2 BELLEVILLE (20041)                        65.1
##  3 PLACE ADOLPHE CHERIOUX                    60
##  4 HIPPODROME D AUTEUIL                      60.0
##  5 RUE DES BOULETS ( COMPLEMENTAIRE )        55
##  6 PLACE DE LA PORTE DE CHATILLON            54.9
##  7 PORTE DE LA CHAPELLE                      54.1
##  8 CHARMES (FONTENAY SOUS BOIS)              53.5
##  9 PORTE DE MONTROUGE                        53
## 10 ALLENDE (PANTIN)                          52.9
## # ... with 918 more rows
```

3.5.13 Task

Find the number of bike stations in each département.

You might find the function `n()` helpful, which returns the number of cases and is the `dplyr` equivalent of `COUNT(*)` in SQL (type `?n` to get help).

3.5.14 Answer

We can use the following R code:

```
stations %>% group_by(departement) %>%          # Group by department
  summarise(n_stations=n()) %>%                 # Count cases
  arrange(desc(n_stations))                     # Sort in descending order
```

```
## # A tibble: 4 x 2
##   departement       n_stations
##   <chr>                  <int>
## 1 Paris                    743
## 2 Hauts-de-Seine            75
## 3 Seine-Saint-Denis         60
## 4 Val-de-Marne              50
```

`group_by` can be also used to limit the scope of subsequent calls to other functions such as `filter`, `arrange` or `slice`. To make this more concrete, suppose we want to find for each time point the station which the most available bikes. We first have group the data by `time` and then find the station with the most available bikes.

```
bikes %>%
  group_by(time) %>%                              # Group by time
  arrange(desc(available_bikes)) %>%              # Sort by bikes within each group
  slice (1)                                       # Return only top one per group
```

```
## # A tibble: 73 x 5
## # Groups:   time [73]
##    name           available_bikes available_bike_stands  time total_stands
##    <chr>                    <int>                 <int> <dbl>        <int>
##  1 MUSÉE D'ORSAY               65                     0  13             65
##  2 MUSÉE D'ORSAY               65                     0  13.1           65
##  3 MUSÉE D'ORSAY               65                     0  13.2           65
##  4 MUSÉE D'ORSAY               62                     3  13.2           65
##  5 METZ                        64                     0  13.3           64
##  6 DUPLEIX                     64                     4  13.4           68
##  7 METZ                        64                     0  13.5           64
##  8 MUSÉE D'ORSAY               63                     2  13.6           65
##  9 SAINT EMILION               63                     3  13.7           66
## 10 MUSÉE D'ORSAY               65                     0  13.8           65
## # ... with 63 more rows
```

Alternatively, we can use `filter` and `min_rank`:

```
bikes %>%
  group_by(time) %>%                              # Group by time
  filter(min_rank(desc(available_bikes))==1)      # Find largest in each group
```

```
## # A tibble: 92 x 5
## # Groups:   time [73]
##    name                       available_bikes available_bike_~  time total_stands
##    <chr>                                <int>            <int> <dbl>        <int>
##  1 MUSÉE D'ORSAY                           65                0  13             65
##  2 MUSÉE D'ORSAY                           65                0  13.1           65
##  3 MUSÉE D'ORSAY                           65                0  13.2           65
##  4 MUSÉE D'ORSAY                           62                3  13.2           65
##  5 MOUFFETARD EPEE DE BOIS                 62                1  13.2           63
##  6 SAINT PLACIDE CHERCHE MI~               62                0  13.2           62
##  7 METZ                                    64                0  13.3           64
##  8 DUPLEIX                                 64                4  13.4           68
##  9 METZ                                    64                0  13.4           64
## 10 METZ                                    64                0  13.5           64
```

```
## # ... with 82 more rows
```

You might have noticed that the answers differ a little. The reason for this are ties: for example, at 1.15pm the stations at Mussée d'Orsay, Mouffetard Epée de Bois and Sainte Placide Cherche-Midi all had 62 bikes available. The former commands extracts just one of them, whereas the bottom command extracts all three. (You would obtain the same results if you replaced `min_rank` by `row_number`, which breaks ties by using in doubt the order in the data set).

3.5.15 Merging (joining) data sets using the `join`-type functions

Suppose we want to extract the data from `bikes` relating to bike stations in Hauts-de-Seine only. The table `bikes` does not however contain any information about the département in which the stations are located. We need to merge the information from the `stations` and `bikes`. This can be done using one of the `join` functions of `dplyr`. We will use `inner_join`, which only retains cases if there are corresponding entries in both data sets: this corresponds to the default behaviour of the R function `merge`.

The `join` functions will be default use the columns with common names across the two data sets ("natural join").

```
bikes %>% inner_join(stations) %>%              # Merge data (using common variable: n
  filter(departement=="Hauts-de-Seine")
```

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

```
## # A tibble: 5,333 x 10
##    name  available_bikes available_bike_~  time total_stands    id address   lng
##    <chr>           <int>            <int> <dbl>        <int> <dbl> <chr>   <dbl>
##  1 SOLJ~              56                4    13           60 28002 BOULEV~  2.25
##  2 DE G~               3               19    13           22 22005 195 AV~  2.26
##  3 NATI~              20                3    13           23 21015 39 RUE~  2.24
##  4 MONT~              20                5    13           25 22011 7 RUE ~  2.28
##  5 PETI~              22                0    13           22 21113 2 RUE ~  2.30
##  6 GREN~               9               12    13           21 21013 4 AVEN~  2.25
##  7 MART~               0               24    13           24 21703 5-7 AV~  2.29
##  8 MORI~              22                3    13           25 21106 2-4 RU~  2.31
##  9 SELL~              34               17    13           51 21501 RUE DE~  2.23
## 10 VALI~              22                2    13           24 21101 4 RUE ~  2.30
## # ... with 5,323 more rows, and 2 more variables: lat <dbl>, departement <chr>
```

We could have specified the column to used to join the data sets manually by adding the argument `by="name"` (or `by=c("name"="name")`, which allows using columns with different names in the two data set).

As a side note, in this example, we could have avoided joining the two tables. We could have first extracted the names of the stations in Hauts-de-Seine and

then used those to subset the data from `bikes` (essentially the equivalent of a subquery in SQL):

```
names92 <- stations %>% filter(departement=="Hauts-de-Seine") %>%
               select(name)
bikes %>% filter(name %in% names92[[1]])
```

```
## # A tibble: 5,333 x 5
##    name                       available_bikes available_bike_~  time total_stands
##    <chr>                                <int>             <int> <dbl>        <int>
##  1 SOLJENITSYNE (PUTEAUX)                  56                 4    13           60
##  2 DE GAULLE 3 (NEUILLY)                    3                19    13           22
##  3 NATIONALE (BOULOGNE-BILL~               20                 3    13           23
##  4 MONTROSIER (NEUILLY)                    20                 5    13           25
##  5 PETIT (CLICHY)                          22                 0    13           22
##  6 GRENIER (BOULOGNE-BILLAN~                9                12    13           21
##  7 MARTINIE (VANVES)                        0                24    13           24
##  8 MORICE 2 (CLICHY)                       22                 3    13           25
##  9 SELLIER (SURESNES)                      34                17    13           51
## 10 VALITON (CLICHY)                        22                 2    13           24
## # ... with 5,323 more rows
```

We had to use `names92[[1]]` to extract the entries of the tibble `names92` as a character vector (we could have also used `unlist(names92)`).

You might notice a small difference in the results returned by the two approaches. The former retains the columns from `stations` which we have inserted, whereas the latter only contains the columns which `bikes` contained to start with.

3.5.16 Additional Resources

Data Transformation Cheat Sheet RStudio have put together a very handy and compact cheat sheet for dplyr.

Background reading: Chapter 13 of R for Data Science Chapter 13 of *R for Data Science* gives a detailed overview of the functions in `dplyr`.

Chapter 4

Creating Graphics with ggplot2

The package ggplot2 provides an abstract and declarative environment for creating graphics.

The graphics system built into R is already quite powerful and flexible, but creating sophisticated graphics can be time-consuming and many steps that could be performed automatically, like adding a legend, have to be performed manually. Code producing more complex visualisations tends be "procedural": rather than describing how the visualisation should look like, the code describes the detailed control flow of how the plot is constructed.

In this section, we will use the `health` dataset. You can download this data to your R console by running the following command:

```
load(url("https://github.com/UofGAnalyticsData/R/raw/main/Week%205/w5.RData"))
```

4.1 ggplot terms

This section gives an overview of key terms in the `ggplot2` world. `ggplot2` is based on the philosophy of a "layered grammar of graphics": plots in `ggplot2` are made up of at least one layer of geometric objects.

4.1.1 Geometric objects

A geometric object (or `geom_<type>(...)` in `ggplot2` commands) controls what type of plot a layer contains. The are many different geometric objects: the most important ones are …

Geometry name	Description	Basic R equivalent	Common aesthetics
`geom_point`	Points (scatter plot)	`plot` / `points`	`x`, `y`, `alpha`, `colour`, `shape`,`size`
`geom_line`	Lines (drawn left to right)	`lines` (after ordering)	`x`, `y`, `alpha`, `colour`,`linetype`, `size`
`geom_path`	Lines (drawn in original order)	`lines`	`x`, `y`, `alpha`, `colour`, `group`,`linetype`, `size`
`geom_abline`	Line (one line)	`abline`	`intersept`, `slope`, `alpha`,`colour`, `linetype`, `size`
`geom_hline`	Horizontal line	`abline`	`yintercept`, `alpha`,`colour`, `linetype`, `size`
`geom_vline`	Vertical line	`abline`	`xintercept`, `alpha`, `colour`,`linetype`, `size`
`geom_text`	Text	`text`	`x`, `y`, `label`, `alpha`, `angle`, `colour`, `size`, `family`, `hjust`,`vjust`, `check_overlap`
`geom_label`	Text (styled as label)	`text`	`x`, `y`, `label`, `alpha`, `angle`, `colour`, `size`, `family`, `hjust`, `vjust`, `check_overlap`
`geom_rect`	Rectangle	`rect`	`xmin`, `xmax`, `ymin`, `ymax`,`alpha`, `colour`, `fill`,`linetype`, `size`
`geom_polygon`	Polygon	`polygon`	`x`, `y`, `alpha`, `colour`, `fill`,`group`, `linetype`, `size`

Geometry name	Description	Basic R equivalent	Common aesthetics
`geom_ribbon`	Ribbon (for confidence bands)	-	`x`, `ymin`, `ymax`, `alpha`, `colour`,`fill`, `group`, `linetype`,`size`
`geom_bar`	Bar plot	`barplot`	`x`, `alpha`, `colour`, `fill`,`linetype`, `size`
`geom_boxplot`	Boxplot	`boxplot`	`x`, `y`, `alpha`, `colour`, `fill`,`group`, `linetype`,`shape`, `size`
`geom_histogram`	Histogram	`hist`	`x`, `y`, `alpha`, `colour`, `fill`,`linetype`, `size`
`geom_raster` /`geom_tile`	Image plot	`image`	`x`, `y`, `alpha`, `fill` (both) and `linetype`, `size`, `width` (`geom_tiles` only)
`geom_counter`	Contour lines	`contour`	`x`, `y`, `z`, `alpha`, `colour`, `group`,`linetype`,`size`

There is a cheat sheet providing a detailed overview of the different geometries and data.

4.1.2 Aesthetics

An aesthetic (or `aes(...)` in `ggplot2` commands) controls which variables are mapped to which properties of the geometric objects (like x-coordinates, y-coordinates, colours, etc.). The aesthetics available depend on the geometric object. Aesthetics commonly available are …

Aesthetic	Description
`x`	x-coordinate
`y`	y-coordinate
`color` or `colour`	Colour (outline)

Aesthetic	Description
`fill`	Fill colour
`alpha`	Transparency (transparent $0 \leq \alpha \leq 1$ opaque)
`linetype`	Line type ("`lty`")
`symbol`	Plotting symbol ("`pch`")
`size`	Size of plotting symbol / font or line thickness

The help file for each geometry lists the available aesthetics.

4.2 Using the more general ggplot interface

4.2.1 A typical ggplot call

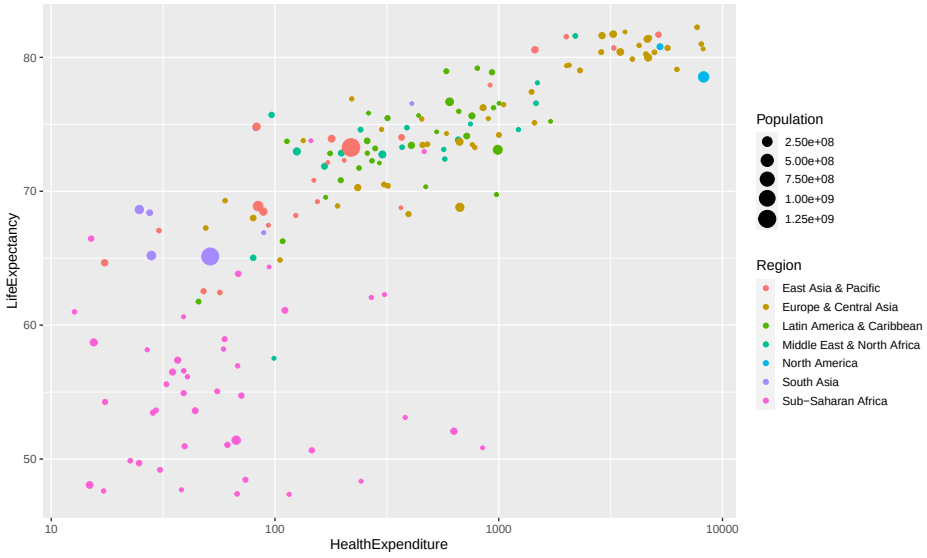
A plotting command for `ggplot` consists of a sequence of function calls added together using the standard sum operator `+`:

```
ggplot(data=...) +                  # Specify data source
    aes(...) +                      # Generic aesthetics applying to all layers
    geom_<type>(aes(...), ...)  + # Geometry for one layer with layers-specific aesthe
    geom_<type>(aes(...), ...)  +
    ...                             # Further arguments for fine-tuning (themes, scales,
```

`geom_<type>` objects do not necessarily have to use the same data as specified in the call to `ggplot`. If the optional argument `data` is specified, then the data source provided is used for this layer.

For the following example, we will use data on health expenditure by country, reported on an annual basis. We can produce a plot like the following using `ggplot` commands.

```
ggplot(data=health) +
  aes(x=HealthExpenditure,  y=LifeExpectancy) +
  geom_point(aes(colour=Region, size=Population)) +
  scale_x_log10()
```

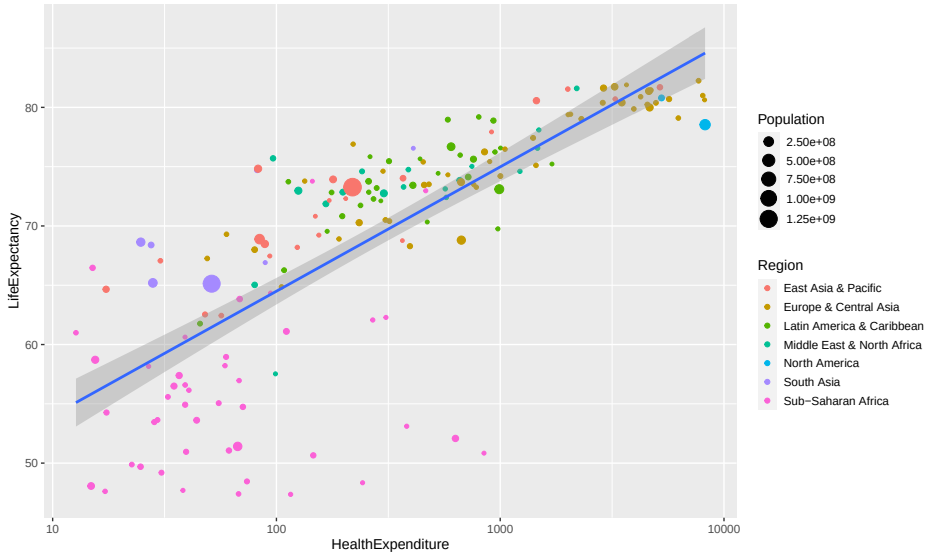


4.2.2 Adding additional layers

Additional layers can simply be added to the plot. For example, we can add an overall regression line with confidence bands using

```
ggplot(data=health) +
    aes(x=HealthExpenditure,  y=LifeExpectancy) +
    geom_point(aes(colour=Region, size=Population)) +
    geom_smooth(method="lm")  +
    scale_x_log10()
```

```
## `geom_smooth()` using formula 'y ~ x'
```



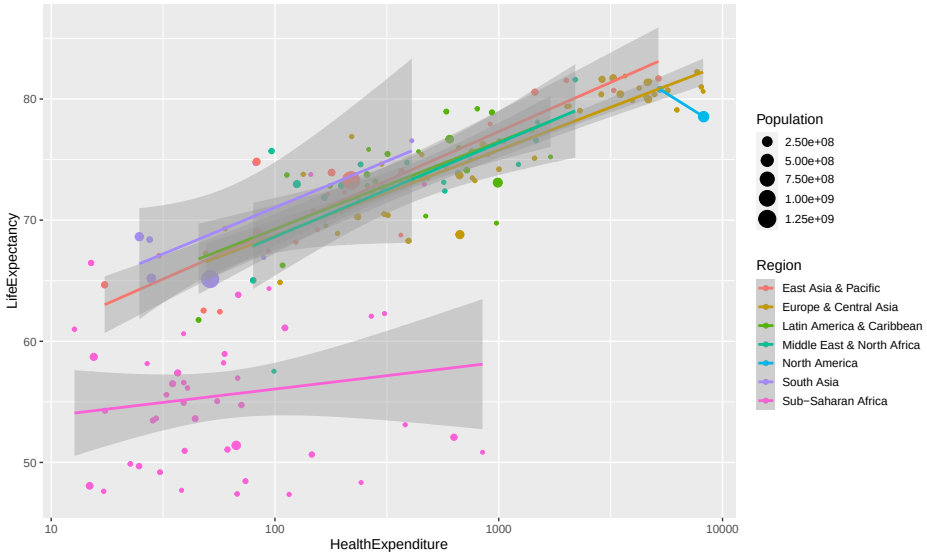
If we want to add a different regression line for each country we have to make sure that a `group` or `colour` aesthetic is passed to `geom_smooth`. We could pass `aes(colour=Region)` to `geom_smooth`. Alternatively, we can move `colour=Region` from the aesthetics specific to `geom_point` to the generic aesthetics, so that `colour=Region` now applies to both `geom_point` and `geom_smooth`.

```
ggplot(data=health) +
    aes(x=HealthExpenditure,  y=LifeExpectancy, colour=Region) +
    geom_point(aes(size=Population)) +
    geom_smooth(method="lm")  +
    scale_x_log10()
```

```
## `geom_smooth()` using formula 'y ~ x'
```

```
## Warning in qt((1 - level)/2, df): NaNs produced
```

```
## Warning in max(ids, na.rm = TRUE): no non-missing arguments to max; returning
## -Inf
```

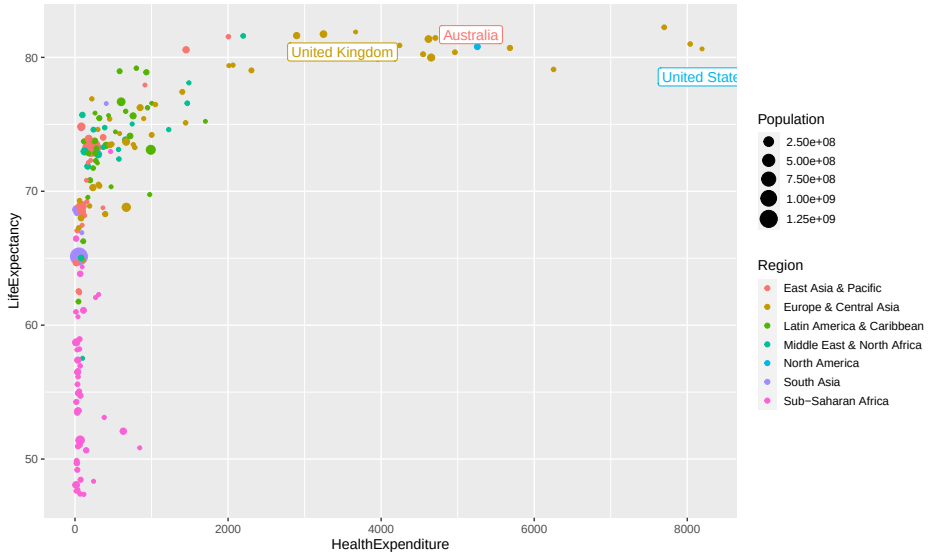


The warning comes from the fact that there are only two North American countries, so we can fit a line through them with no error, which means we cannot draw confidence bands.

The plot looks slightly messy, we will use `facet_wrap` later on to split it into separate panels.

Suppose we want to annotate the observations belonging to Australia, the UK, the US.

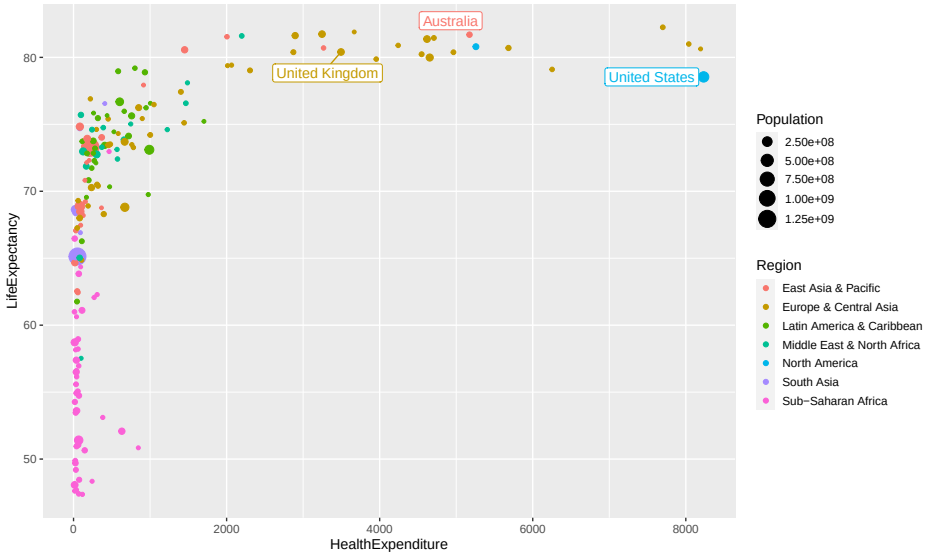
```
health2 <- health %>%
    filter(Country %in% c("Australia", "United Kingdom", "United States"))
ggplot(data=health) +
    aes(x=HealthExpenditure,  y=LifeExpectancy, colour=Region) +
    geom_point(aes(size=Population)) +
    geom_label(data=health2,
               aes(x=HealthExpenditure,  y=LifeExpectancy, label=Country),
               show.legend=FALSE)
```



The labels however cover the observations and might not be fully visible. This can be avoided by using the function `geom_label_repel` from ggrepel.

```
health <- health %>%
    mutate(CountryLabel=ifelse(Country%in%c("Australia", "United Kingdom", "United Stat
                               as.character(Country),""))

library(ggrepel)
ggplot(data=health) +
    aes(x=HealthExpenditure,  y=LifeExpectancy, colour=Region) +
    geom_point(aes(size=Population)) +
    geom_label_repel(aes(label=CountryLabel), show.legend=FALSE)
```



This time, we have used a different approach. Rather than subsetting the data and creating a separate data frame only containing the data for the three countries, we have created a new column in the data frame `health`, which is blank except for the three countries. This is required because `ggrepel` layers are only aware of data drawn in their own layer: this way we can avoid the labels covering observations we have not labelled.

4.2.3 Explicit drawing

The standard R plotting functions draw a plot as soon as the plot function is invoked.

Plotting commands in `ggplot2` (including `qplot`) return objects (otherwise the `+` notation would not work) and only draw the plot when their `print` or `plot` methods are invoked. In the console this is the case when they are used without an assignment.

```
a <- ggplot(data=health) +              # Does not draw anything
        aes(x=HealthExpenditure, y=LifeExpectancy) +
        geom_point()

b <- a + scale_x_log10()                # Does not draw anything either

a                                       # Now the plot stored in a gets drawn
print(a)                                # Draw a again (explicit invocation)

b                                       # Now the plot stored in b gets drawn
```

Inside loops and functions the `print` or `plot` methods need to be invoked ex-

plicitly by using the methods `print` or `plot`.

4.2.4 Task

Consider two vectors `x` and `y` created using

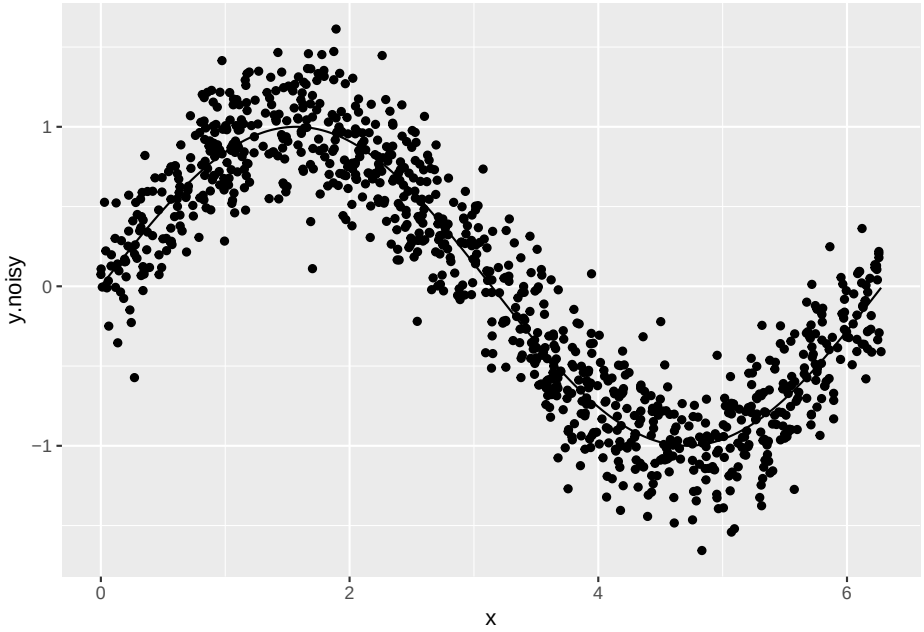
```
n <- 1e3
x <- runif(n, 0, 2*pi)                  # x is random uniform from (0,2*pi)
# x <- sort(x)                          # Sorting of x _not_ needed for ggplot
y <- sin(x)                             # Set y to the sine of x
y.noisy <- y + .25 * rnorm(n)           # Create noisy version of y
```

Use `ggplot2` to create a scatterplot of `y.noisy` against `x`, which also shows the noise-free sine curve in `y`.

4.2.5 Answer

We can use the following R code:

```
ggplot() +                              # No need to use data=... as x, y and y.noisy
                                        # are variables in the workspace and not colu
                                        # in a dataset
    geom_point(aes(x, y.noisy)) +
    geom_line(aes(x, y))
```



It does not matter whether `geom_point` or `geom_line` comes first. `ggplot2` adapts the axes so that all objects drawn fit (and not just the first one as is the

case when using standard R plotting functions `plot` and `points`).

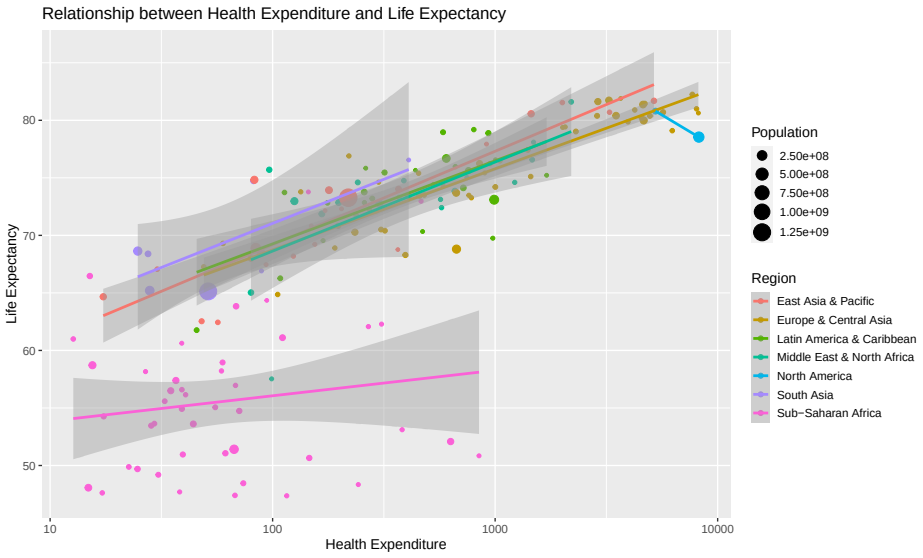
4.3 Modifying Plots

4.3.1 Labels and titles

We can set the plot title using `ggtitle(title)` and the axis labels using `xlab(label)` and `ylab(label)`.

```
ggplot(data=health) +
    aes(x=HealthExpenditure,  y=LifeExpectancy, colour=Region) +
    geom_point(aes(size=Population)) +
    geom_smooth(method="lm")  +
    scale_x_log10() +
    ggtitle("Relationship between Health Expenditure and Life Expectancy") +
    xlab("Health Expenditure") +
    ylab("Life Expectancy")
```

```
## `geom_smooth()` using formula 'y ~ x'
```



Changing the text shown in legends (like in our case the names of the regions) is more complicated. It is almost always easier to simply change the levels of the categorical variable in the dataset itself before invoking `ggplot2` commands.

4.3.2 Scales

Aesthetics control *which* variables are mapped to *which* property of the geometric object. However, aesthetics do not specify *how* this mapping is performed. This is where scales come into play. Scales control *how* any value from the

variable is translated into a property of a geometric object: scales control for example how a variable is translated into coordinates (say through a log transform) or into colours (say though a discrete colour palette).

`ggplot2` automatically chooses (what it thinks is) a suitable scale. This is usually reasonable, but on occasions it might be necessary to override this.

There is a family of scale functions for each aesthetic. The template for the function name for scales is `scale_<aesthetic>_<type>`.

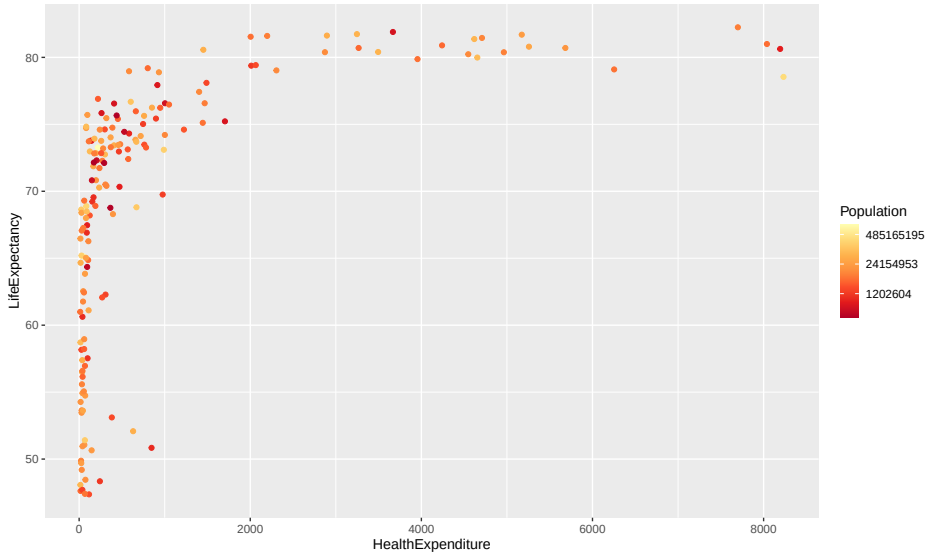
4.3.2.1 Scales for continuous data

We have already seen that we can log-transform the axes using `scale_x_log10` and `scale_x_log10`. The more general functions for coordinate transforms are `scale_<x or y>_continous(...)`. We can. amongst others, set the axis label (argument `name`, the ticks and tickmarks (arguments `breaks` and `labels`) the limits (argument `limit`) and the transform to be used (argument `trans`).

The axes might use scientific notation (e.g. "4e5"). If you want to avoid using scientific notation and use fixed notation, change the `scipen` option in R, which controls when scientific notation is used (for example run `options(scipen=1e3)`).

There are functions for mapping continuous data to other aesthetics, too. For example, `scale_colour_gradient` converts a continuous variable to a colour using a gradient of colours. The arguments `low` and `high` specify the colours used at the two ends. `scale_colour_gradient2` allows for also specifying a mid-point colour (argument `mid`). `scale_colour_gradientn` is the most general function it allows specifying a vector of colours and corresponding vector of colours. The function `scale_colour_distiller` uses the colour brewer available at http://colorbrewer2.org/ and allows for constructing colours scales which are photocopier-safe and/or work for colour-blind readers.

```
a <- ggplot(data=health) +
        aes(x=HealthExpenditure, y=LifeExpectancy) +
        geom_point(aes(colour=Population)) +
        scale_colour_distiller(palette="YlOrRd" , trans="log")
a
```



We have used `trans="log"` to use the log-transformed values of the population sizes (due to its skewness). The values given in the legend seem slightly odd choices: this is due to the log-transform (they are roughly exp(14), exp(17) and exp(20), so "nice" numbers on the log scale).

We have stored the plot in a variable `a` so that we can redraw it later on with different themes.

4.3.2.2 Scales for discrete data

There are also various scaling functions for discrete data, such as `scale_colour_brewer`.

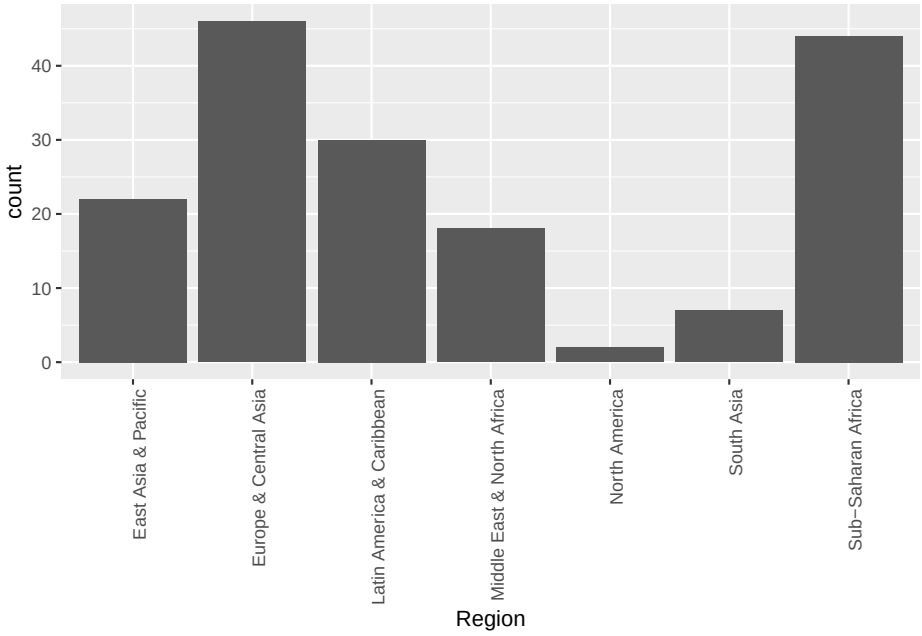
Note that there are separate scales for colour (outline colour – example: `scale_colour_brewer`) and fill (fill colour – example: `scale_fill_brewer`).

4.3.3 Statistics

Sometimes data has to be aggregated before it can be used in a plot. For example, when creating a bar plot illustrating the distribution of a categorical variable we have to count how many observations there are in each category. This will then determine the height of the bars. `ggplot2` automatically chooses (what it thinks is) a suitable statistic.

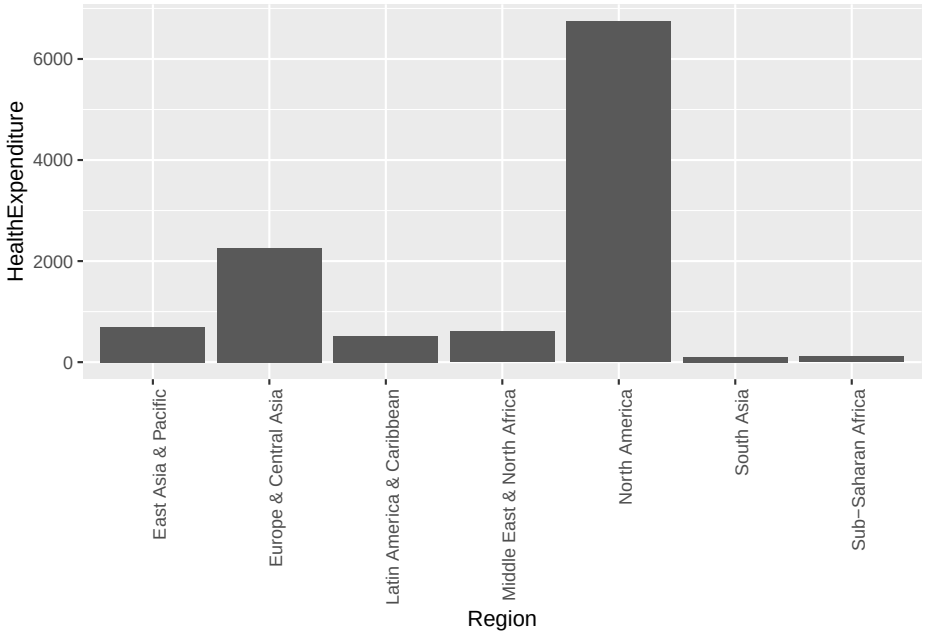
For example, when we draw a bar plot using `geom_bar`, it uses by default the statistic `count`, which first produces a tally. We don't need to worry about this, `ggplot2` does all the work for us.

```
ggplot(data=health) +
    geom_bar(aes(x=Region)) +
    theme(axis.text.x = element_text(angle = 90, hjust = 1)) # Rotate x axis labels
```



Suppose we now want to a draw bar chart visualising the mean health expenditure in each region. Now we don't want `ggplot2` to produce a tally of how often which value occurs, we want it to simply draw the bars to the heights specified in the data. Because we now want no aggregation, we have to use the statistic `identity`.

```
library(dplyr)
HESummary <- health %>%                                    # Get avg health exp
                group_by(Region) %>%
                summarise(HealthExpenditure=mean(HealthExpenditure))
ggplot(data=HESummary) +
    geom_bar(aes(x=Region, y=HealthExpenditure), stat="identity") +
    theme(axis.text.x = element_text(angle = 90, hjust = 1)) # Rotate x axis labels
```

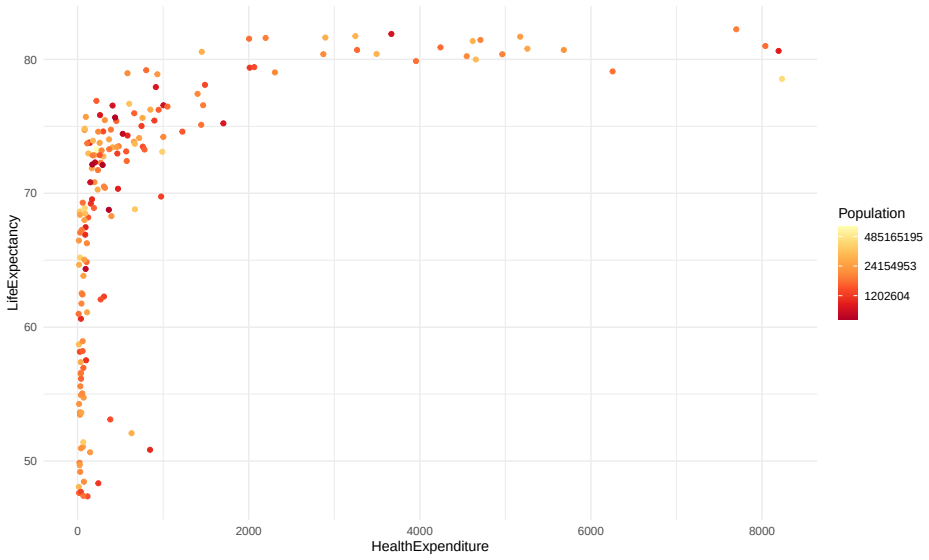


4.3.4 Theming

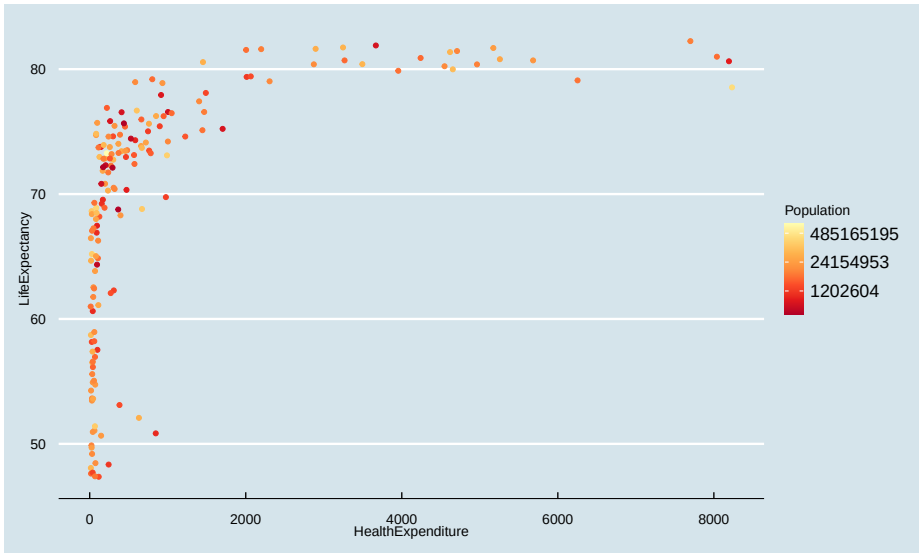
Themes can be used to customise how `ggplot2` graphics look like. We have already used `theme` to change how the horizontal axis is typeset.

`ggplot2` has several themes built-in. The default theme is `theme_gray`. Other themes available are `theme_bw` (monochrome), `theme_light`, `theme_lindedraw` and `theme_minimal`. Further themes are available in extension packages such ggthemes.

```
a + theme_minimal()
```




```
library(ggthemes)
a + theme_economist() + theme(legend.position="right")
```



4.3.5 Arranging plots (faceting)

The function `facet_grid(rvar~cvar)` creates separate plots based on the values `rvar` (rows) and `cvar` (columns) takes. The function `facet_wrap(~var1+var2)` arranges the plots in several rows and columns without rigidly associating one variable with rows and one with columns. Continuous variables need to be discretised (for example using `cut`) before they

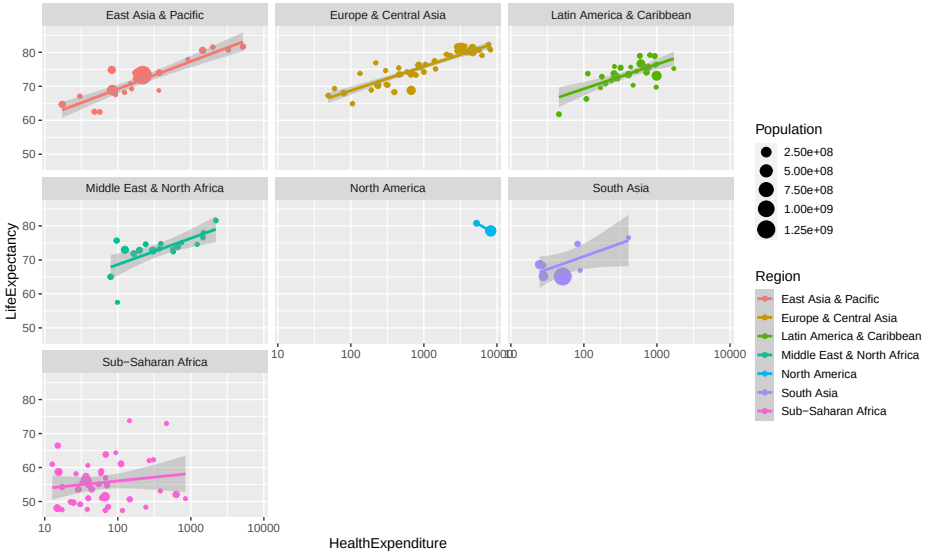
can be used for defining facets.

```
ggplot(data=health) +
    aes(x=HealthExpenditure,  y=LifeExpectancy, colour=Region) +
    geom_point(aes(size=Population)) +
    geom_smooth(method="lm")  +
    scale_x_log10() +
    facet_wrap(~Region)
```

```
## `geom_smooth()` using formula 'y ~ x'
```

```
## Warning in qt((1 - level)/2, df): NaNs produced
```

```
## Warning in max(ids, na.rm = TRUE): no non-missing arguments to max; returning
## -Inf
```



Arranging plots in more general ways (like in `par(mfrow=c(...))` or `layout`) is not directly possible with `ggplot2`. The package gridExtra however provides a function `grid.arrange`, which allows for arranging `ggplot2` plots side by side.

Chapter 5

Handling date-time data with `lubridate`

Date-time data can be complex to handle in R. Classic R commands for date-times are generally unintuitive and vary depending on the date-time object being used. the `lubridate` package makes it easier to handle date-times in R and handle many tasks classic R functions cannot handle.

We will cover some useful commands within the package but do consult the `lubridate` cheat sheet below for a series of useful commands.

Lubridate cheat sheet

5.1 Creating date/times

There are three types of date/time data that refer to an instant in time:

- A *date*. Tibbles print this as `<date>`
- A *time* within a day. Tibbles print this as `<time>`
- A *date-time* is a date plus a time: it uniquely identifies an instant in time. Tibbles print this as `<dttm>`.

You should always use the simplest possible data type for what you need.

There are three ways you are likely to create a date/time:

- From a string
- From date/time components
- From an existing date/time object

These can be created as follows.

5.1.1 From strings

We can convert a string to a date/time object using functions within `lubridate`. These functions automatically work out the format once you specify the order of the component. To use them, identify the order in which year, month and day appear in your dates, then arrange "y", "m", and "d" in the same order. This gives you the name of the function you need to call. For example:

```
ymd("2022-08-02")
```

```
## [1] "2022-08-02"
```

```
mdy("August 2nd, 2022")
```

```
## [1] "2022-08-02"
```

```
dmy("02-Aug-2022")
```

```
## [1] "2022-08-02"
```

These functions also take unquoted numbers. This is the most concise way to create a single date/time object.

```
ymd(20220802)
```

```
## [1] "2022-08-02"
```

To create a date-time, add an underscore to the previous functions and one or more of "h", "m" and "s" to the name of the function

```
ymd_hms("2022-08-02 13:05:02")
```

```
## [1] "2022-08-02 13:05:02 UTC"
```

```
mdy_hm("08/02/2022 13:05")
```

```
## [1] "2022-08-02 13:05:00 UTC"
```

You can also force the creation of a date-time from a date by supplying a time-zone

```
ymd(20220802,tz="UTC")
```

```
## [1] "2022-08-02 UTC"
```

5.1.2 From individual components

Instead of a string, you sometimes may have individual components of the date-time spread across multiple columns. Looking at data provided within the `nycflights13` library, we see

```
library(nycflights13)
library(tidyverse)
```

```
flights %>%
  select(year,month,day,hour,minute)
```

```
## # A tibble: 336,776 x 5
##     year month   day  hour minute
##    <int> <int> <int> <dbl>  <dbl>
##  1  2013     1     1     5     15
##  2  2013     1     1     5     29
##  3  2013     1     1     5     40
##  4  2013     1     1     5     45
##  5  2013     1     1     6      0
##  6  2013     1     1     5     58
##  7  2013     1     1     6      0
##  8  2013     1     1     6      0
##  9  2013     1     1     6      0
## 10  2013     1     1     6      0
## # ... with 336,766 more rows
```

To create a date/time for this data, we can use `make_date()` for dates, or `make_datetime()` for date-times:

```
flights <- flights %>%
  select(year,month,day,hour,minute) %>%
  mutate(departure_time=make_datetime(year,month,day,hour,minute))
```

5.1.3 From existing types

You may wish to switch between a date-time and a date. This can be done using `as_datetime()` and `as_date()`

```
as_datetime(today())
```

```
## [1] "2022-08-15 UTC"
```

```
as_date(now())
```

```
## [1] "2022-08-15"
```

5.2 Date-time Components

Here, we will look at functions which can let us access certain components of a date-time object.

You can obtain certain parts of a date with functions like `year()`, `month()`, `mday()` (day of month), `yday()` (day of year), `hour()`, `minute()` and `second()`.

```
date_time <- ymd_hms("2021-12-25,09:10:25")

year(date_time)
```

```
## [1] 2021
```

```
month(date_time)
```

```
## [1] 12
```

```
mday(date_time)
```

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

```
yday(date_time)
```

```
## [1] 359
```

For `month()` and `wday()`, you can set `label=TRUE` to return the abbreviated name of the month or day of the week. Set `abbr=FALSE` to return the full name.

```
month(date_time,label=TRUE)
```

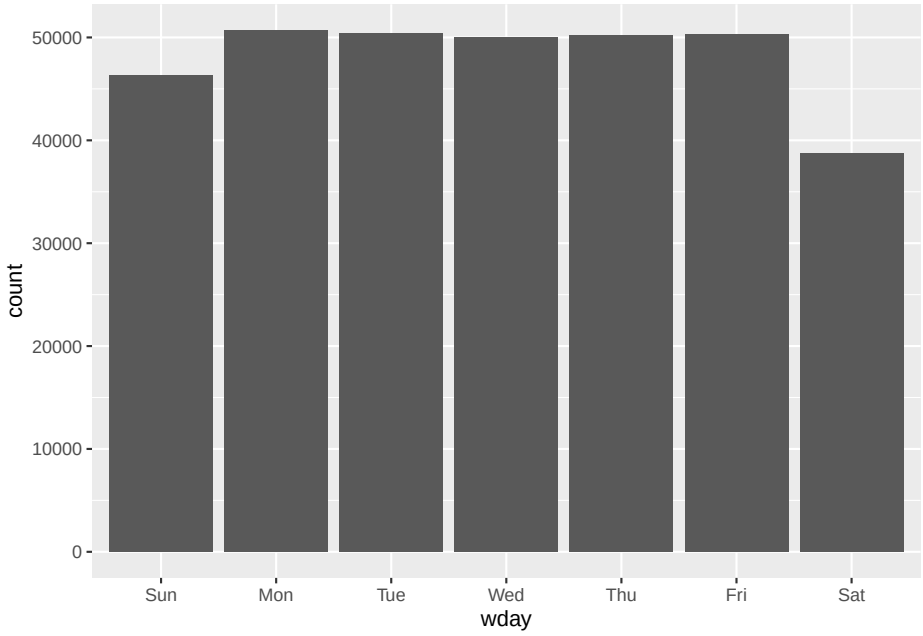
```
## [1] Dec
## 12 Levels: Jan < Feb < Mar < Apr < May < Jun < Jul < Aug < Sep < ... < Dec
```

```
wday(date_time,label=TRUE,abbr=FALSE)
```

```
## [1] Saturday
## 7 Levels: Sunday < Monday < Tuesday < Wednesday < Thursday < ... < Saturday
```

We can use `wday()` to see that more flights depart during the week than on the weekend

```
flights %>%
  mutate(wday = wday(departure_time, label = TRUE)) %>%
  ggplot(aes(x = wday)) +
    geom_bar()
```



5.2.1 Time Spans

Now we will look at how arithmetic with dates works, including subtraction, addition and division. We will explore the following three classes:

- *durations*, which represent an exact number of seconds
- *periods*, which represent units like weeks and months
- *intervals*, which represent a start and end point

5.2.2 Durations

We can find out the duration (given in seconds) using the `as.duration()` function as shown below

```
# How old is Craig?
c_age <- today() - ymd(19910502)
as.duration(c_age)
```

```
## [1] "987379200s (~31.29 years)"
```

Durations also come with some useful additional functions shown below

```
dseconds(10)
```

```
## [1] "10s"
```

```
dminutes(60)
```

```
## [1] "3600s (~1 hours)"
```

```
dhours(30)
```

```
## [1] "108000s (~1.25 days)"
```

```
ddays(c(4,15))
```

```
## [1] "345600s (~4 days)"      "1296000s (~2.14 weeks)"
```

```
dweeks(1:4)
```

```
## [1] "604800s (~1 weeks)"  "1209600s (~2 weeks)" "1814400s (~3 weeks)"
## [4] "2419200s (~4 weeks)"
```

```
dyears(2)
```

```
## [1] "63115200s (~2 years)"
```

We can also add and multiply durations

```
3*dmonths(5)
```

```
## [1] "39447000s (~1.25 years)"
```

```
dyears(3) + dweeks(14) +dhours(6)
```

```
## [1] "103161600s (~3.27 years)"
```

5.2.3 Periods

Periods are time spans, but don't have a fixed length in seconds, and work more as "human" times, like days and weeks.

```
today() + days(1)
```

```
## [1] "2022-08-16"
```

Like durations, periods can be constructed using well named constructor functions.

```
seconds(15)
```

```
## [1] "15S"
```

```
minutes(10)
```

```
## [1] "10M 0S"
```

```
hours(30)
```

```
## [1] "30H 0M 0S"
```

```
days(c(2,5))
```

```
## [1] "2d 0H 0M 0S" "5d 0H 0M 0S"
```

```
months(2:5)
```

```
## [1] "2m 0d 0H 0M 0S" "3m 0d 0H 0M 0S" "4m 0d 0H 0M 0S" "5m 0d 0H 0M 0S"
```

```
weeks(3)
```

```
## [1] "21d 0H 0M 0S"
```

```
years(1)
```

```
## [1] "1y 0m 0d 0H 0M 0S"
```

Like durations, we can add and multiply periods

```
3*(months(4) + days(12))
```

```
## [1] "12m 36d 0H 0M 0S"
```

```
days(25) + hours(16) + minutes(12)
```

```
## [1] "25d 16H 12M 0S"
```

5.2.4 Intervals

An interval is a duration with a starting point, this makes it precise so you can determine exactly how long it is:

```
next_year <- today() + years(1)
(today() %--% next_year)/ddays(1)
```

```
## [1] 365
```

To find out how many periods fall into an interval, you need to use integer division `%/%`

Chapter 6

Producing maps for plotting

6.1 Producing maps using ggmap

The R package `ggmap` can download maps from Google maps (or OpenStreetMap) which can then be used as a background layer in a `ggplot2` plot.

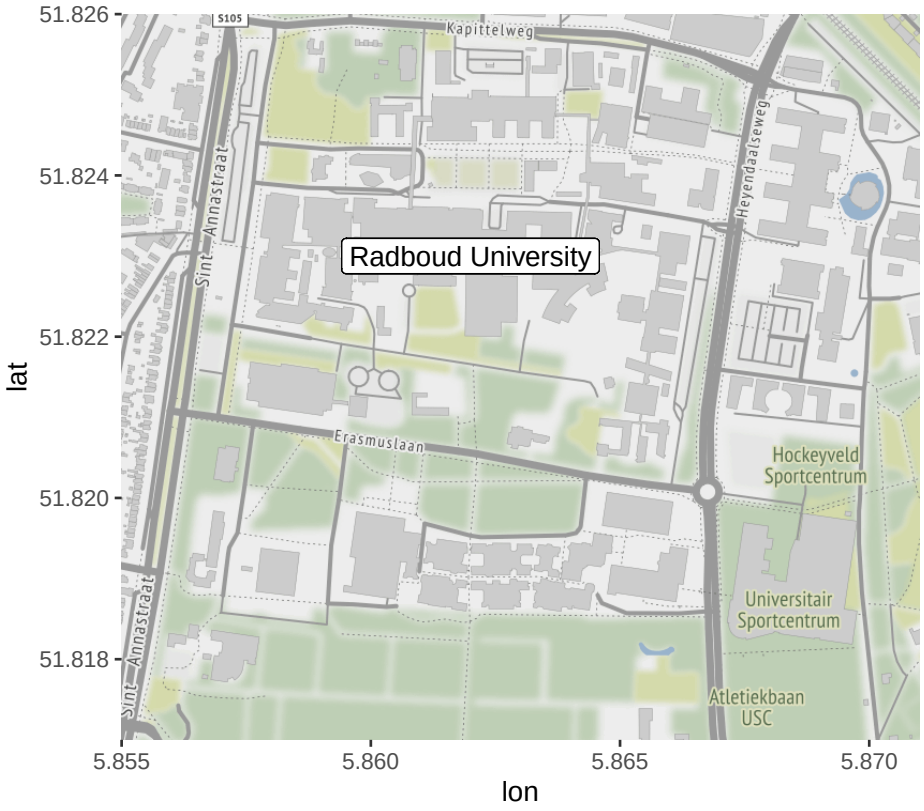
The function `get_map(location, zoom)` downloads a map. `location` can be a pair of longitude and latitude, a character string describing the location, or a bounding box. `zoom` controls the zoom level (from 3 (continent) to 21 (highest resolution)). The optional argument `maptype` can be used to select the type of map downloaded (for example `"roadmap"`, `"hybrid"` or `"satellite"` when using Google maps)

Note that access to the Google API (for Google map tiles and for geolocation (translation of location description to GPS coordinates) requires a Google API key. When using a bounding box and `"stamen"` as `source`, no API key is required.

```
library(ggmap)
boundingbox <- c(left = 5.855, bottom = 51.817, right = 5.871, top = 51.826)
map <- get_map(boundingbox, zoom=16, source="stamen")
```

The map can the be plotted using `ggmap(map)`. Layers can be added to the map using the usual `ggplot2` commands.

```
ggmap(map) +
    geom_label(x=5.862, y=51.823, label="Radboud University")
```



6.1.1 Task

In this task, we will use data from two tibbles, `stations` and `trips`. These contain information on the list of bike stations of the Bay Area Bike Share system in the San Francisco Bay Area.

You can download the data using the following command

```
load(url("https://github.com/UofGAnalyticsData/R/raw/main/Week%206/t3.RData"))
```

It has the following columns.

Column name	Description
`station_id`	Numeric identifier of the station
`name`	Name of the station
`lat`	Latitude of the station
`long`	Longitude of the station
`dockcount`	Number of docks at the station
`city`	City in which the station is located

The tibble `trips` contains all trips made during August 2015. It has the following columns.

Column name	Description
`trip_id`	Numeric identifier of the trip
`trip_duration`	Duration of the trip in seconds
`day`	Day of the month the trip was started
`hour`	Decimal hour when the strip was started
`start_station_id`	Numeric identifier of the station where the trip started
`end_station_id`	Numeric identifier of the station where the trip ended
`bike_id`	Numeric identifier of the bike used
`end_date`	Date and time the trip ended
`subscriber_type`	User type ("Subscriber" or "Customer")

a) Plot the locations of each of the bike stations. You can use the following bounding box below for your map

```
boundingbox <- c(left = -122.5, bottom = 37.25, right = -121.75, top = 38)
```

b) (Harder) For trips within the city of San Francisco, use the code below to create an origin-destination matrix. The (i, j)th entry contains the number of trips from station i to station j.

Create a plot representing the number of trips between the stations. Use the line thickness or transparency to indicate the number of trips. You can use the following bounding box below for your map

```
library(tidyverse)

sf_stations <- stations %>%
    filter(city=="San Francisco")

od <- trips %>%
         filter(start_station_id%in%sf_stations$station_id,
                end_station_id%in%sf_stations$station_id) %>%
         group_by(start_station_id, end_station_id) %>%
         summarise(ntrips=n())
```

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

```
odm <- od %>%
         spread(end_station_id, ntrips, fill=0)

odm <- as.matrix(od[,-1])
rownames(odm) <- od$start_station_id
```

```
od2 <- od %>%
    full_join(od, by=c("start_station_id"="end_station_id", "end_station_id"=
"start_station_id")) %>%
    replace_na(list(ntrips.x=0, ntrips.y=0)) %>%
    mutate(ntrips=ntrips.x+ntrips.y) %>%
    select(-ntrips.x, -ntrips.y) %>%
    filter(start_station_id<end_station_id)

odall <-
    od2 %>%
    inner_join(sf_stations, by=c("start_station_id"="station_id")) %>%
    inner_join(sf_stations, by=c("end_station_id"="station_id"), suffix=c("","_end"))


boundingbox <- c(left = -122.43, bottom = 37.76, right = -122.38, top = 37.81)
```

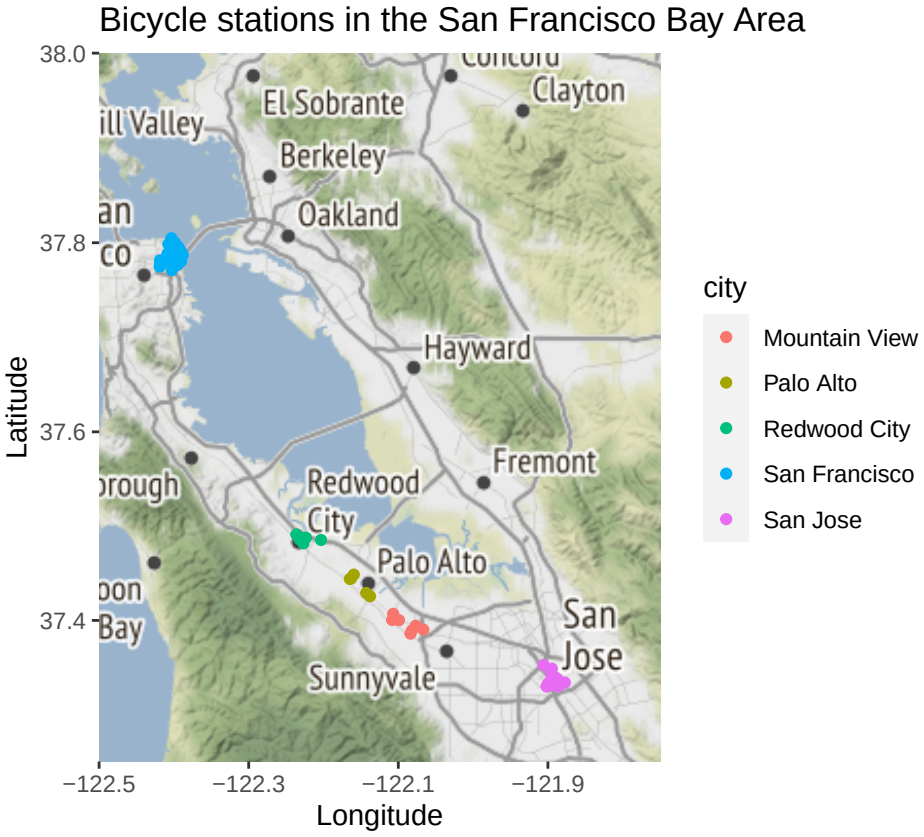
6.1.2 Answer

For part (a) we can use the following code.

```
boundingbox <- c(left = -122.5, bottom = 37.25, right = -121.75, top = 38)

map <- get_map(boundingbox, zoom=9, source="stamen")

ggmap(map) +
    geom_point(data=stations, aes(x=long, y=lat,  colour=city)) +
    xlab("Longitude") + ylab("Latitude") +
    ggtitle("Bicycle stations in the San Francisco Bay Area")
```

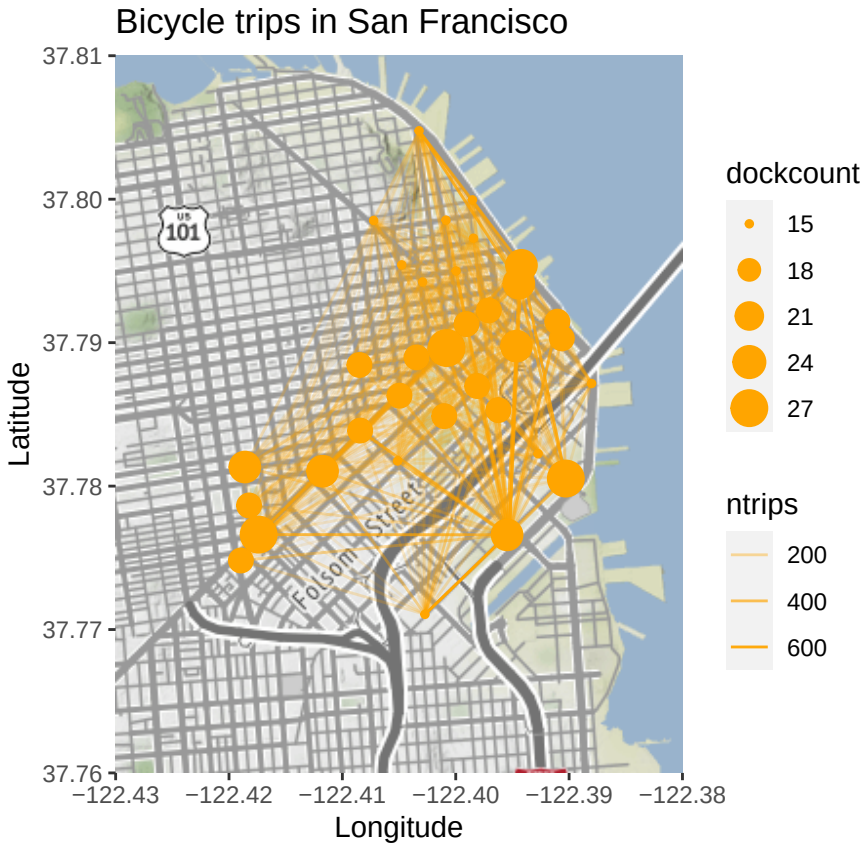


For part (b) we can use the following code.

```
library(magrittr)
boundingbox <- c(left = -122.43, bottom = 37.76, right = -122.38, top = 37.81)
map <- get_map(boundingbox, zoom=13, source="stamen")



ggmap(map) +
    geom_point(data=sf_stations, aes(long, lat, size=dockcount), col="orange") +
    geom_segment(data=odall, aes(long, lat, xend=long_end, yend=lat_end, alpha=ntrips), col="oran
    xlab("Longitude") + ylab("Latitude") +
    ggtitle("Bicycle trips in San Francisco")
```



6.1.3 Producing maps using leaflet

Maps plotted using `ggmap` cannot be panned and zoomed in and out like maps on Google Maps or OpenStreetMap. The package `leaflet` allows for this. It works somewhat the other way round than `ggmap`: rather than downloading the map and integrating it into an R plot it overlays the data over the map interface.

The following command puts a marker where Radboud University is located.

```
library(leaflet)
leaflet() %>%
   addTiles(urlTemplate = "http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png") %>%
   addMarkers(lng=5.862, lat=51.823, popup="Radboud University")
```

The argument `urlTemplate` is only required when opening the file locally.

Lines can be added to the map using the function `addPolylines`.

The data frame `subway` contains the GPS coordinates of all subway stations in Glasgow. You can produce a map of the Glasgow subway network using the

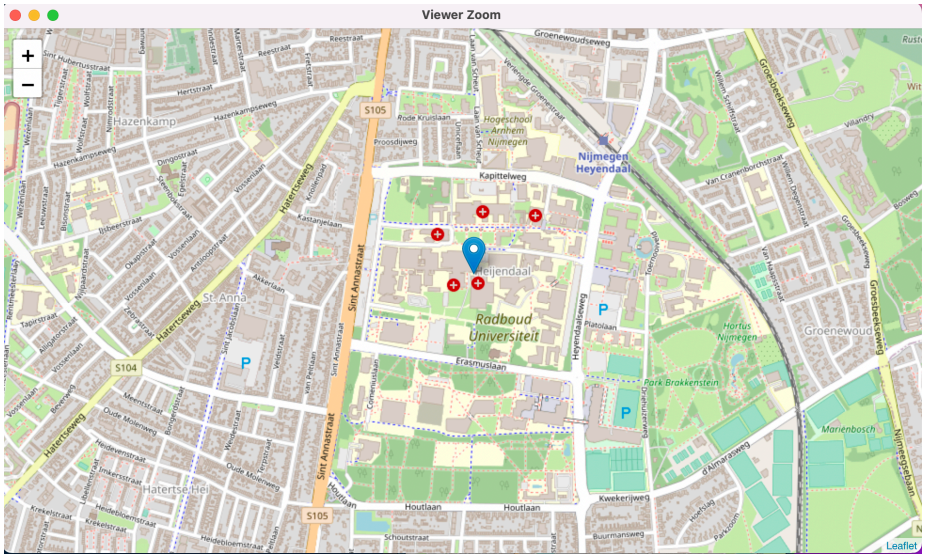


Figure 6.1: Leaflet map showing Radboud University

following code. (You should have access to this data frame from the previous download).

```
subway2 <- rbind(subway, subway[1,])      # Make sure line goes back to Hillhead
leaflet() %>%
   addTiles(urlTemplate = "http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png") %>%
   addMarkers(lng=-4.2885, lat=55.8715, popup="The University of Glasgow") %>%
   addPolylines(subway2$long, subway2$lat, color="#ff6200", opacity=0.5, weight=10) %>%
   addCircleMarkers(subway$long, subway$lat, popup=subway$station, color="#ff6200",
                    opacity=1, fillColor="#4d4f53", fillOpacity=1)
```

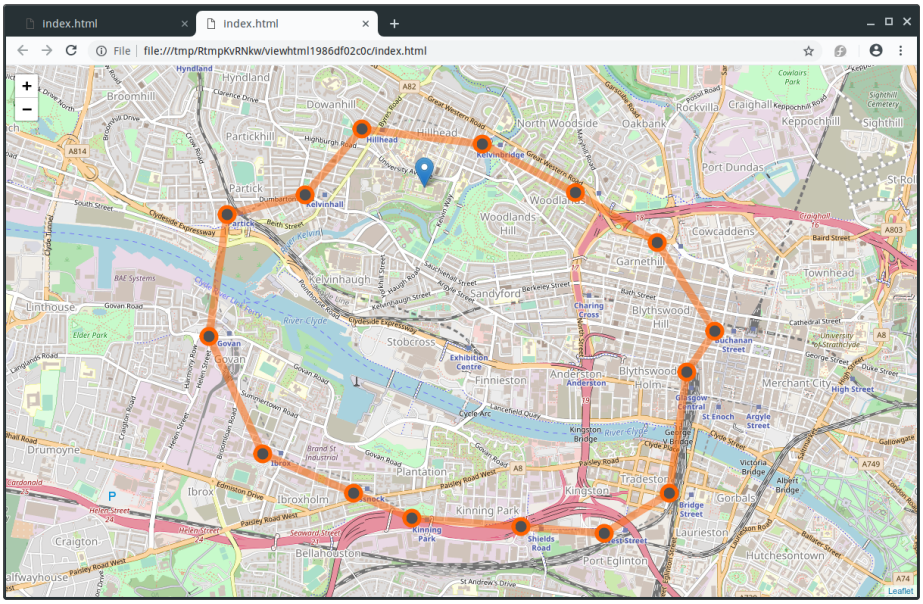


Figure 6.2: Leaflet map showing the Glasgow Subway