Introduction to R

Version 2

Contents

Chapter 1

Introduction

These are course notes for the "Introduction to R" course given by the Monash Bioinformatics Platform. This is a new version of the course focussing on the modern "Tidyverse" set of packages. We believe this is currently the quickest route to being productive in R.

These workshop notes are online at https://monashdatafluency.github.io/r-intro-2/index.html

- PDF version for printing[1]
- ZIP of files used in this workshop[2]

1.1 Source code

- GitHub page[3]

1.2 Copyright and acknowledgements

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

Data files derived from Gapminder, with a CC BY-4: Creative Common Attribution Licence 4.0. 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[5] for the original form of the data if using it for other uses.

[1] https://github.com/MonashDataFluency/r-intro-2/r-intro-2.pdf
[2] https://github.com/MonashDataFluency/r-intro-2/r-intro-2.zip
[3] https://github.com/MonashDataFluency/r-intro-2
[4] http://creativecommons.org/licenses/by/4.0/
[5] https://www.gapminder.org

Chapter 2

Starting out in R

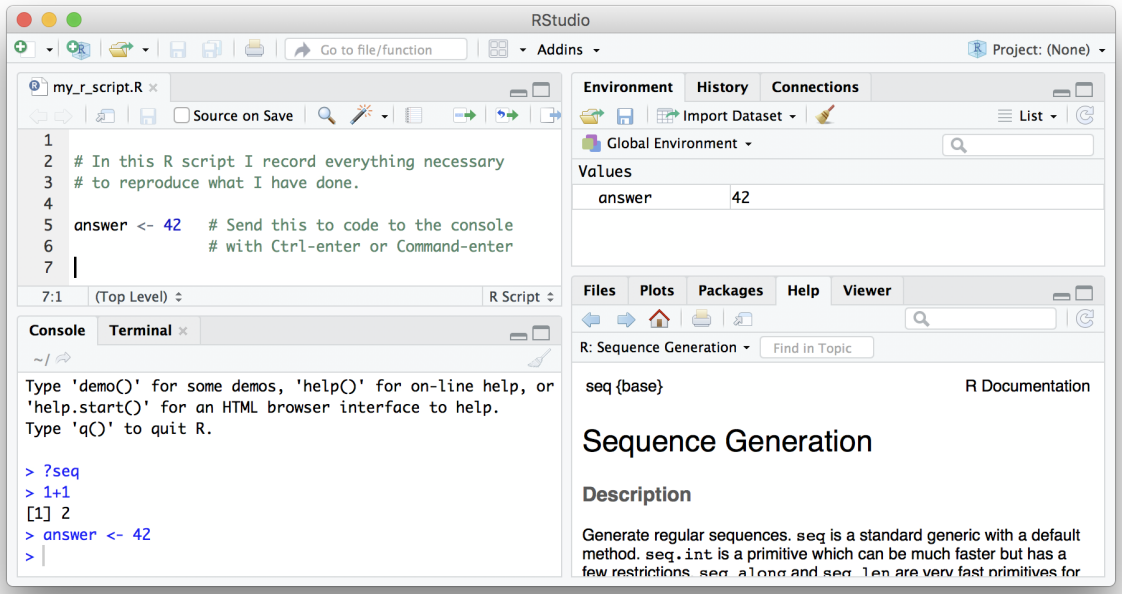
R is both a programming language and an interactive environment for statistics. Today we will be concentrating on R as an *interactive environment*.

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.

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



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+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"`.

2.1 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 isn't ok, with two: `_` 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.

2.2 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 just placing the cursor on the correct line), and then pressing **Control-Enter** (or **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).

2.3 Vectors

A *vector* of numbers is a collection of numbers. "Vector" can mean 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, numbers are just 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.

2.4 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.

2.5 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`.

2.6 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]
```

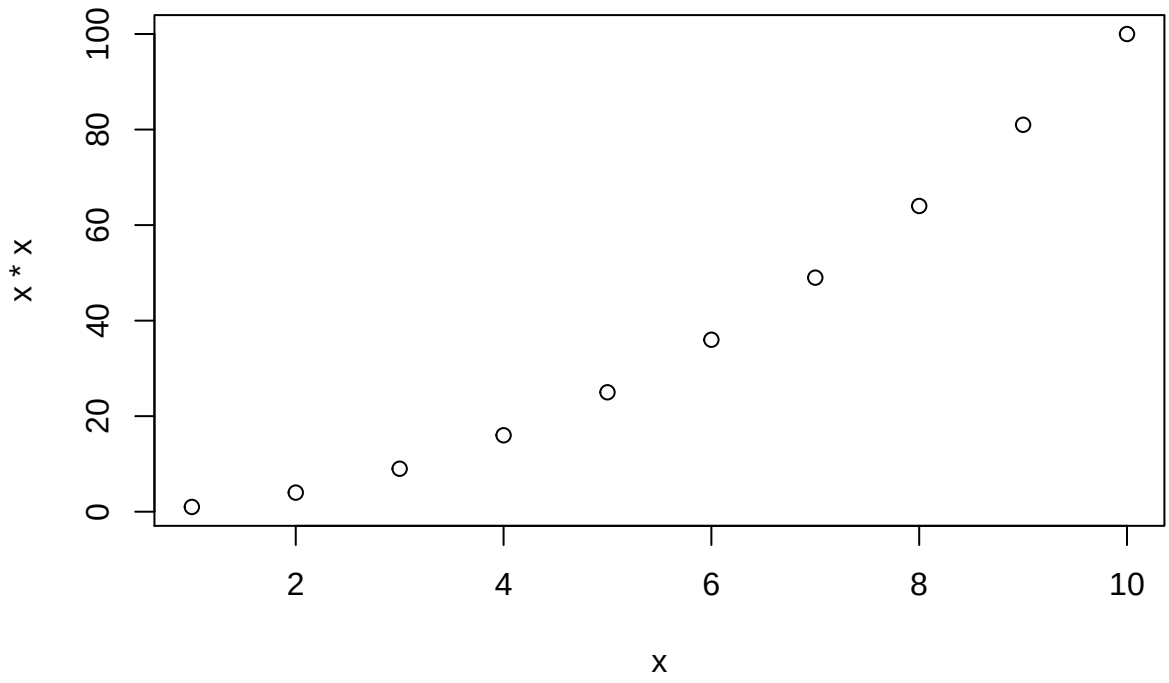
```
## [1] "spam"  "eggs"  "beans" "bacon"
```

Sequences are useful for many 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)
```



2.7 Functions

Functions are the things that do all the work for us in R: calculate, manipulate data, read and write to files, produce plots. Because R is a language for statistics, it has many built in statistics-related functions. We will also be loading more specialized functions from "packages".

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

```
sum(myvec)
```

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

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

```
?sum
```

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

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

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

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

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

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

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

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

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

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

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

```
rep(x=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
```

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 3

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.

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

3.1 Loading data

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

```
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.

3.2 Exploring

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

```
View(geo)
```

However understanding this data frame in R should be less a matter of using a graphical interface, and more about using a variety of R functions to interrogate it.

```
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
```

3.3 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
```

3.4 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"
```

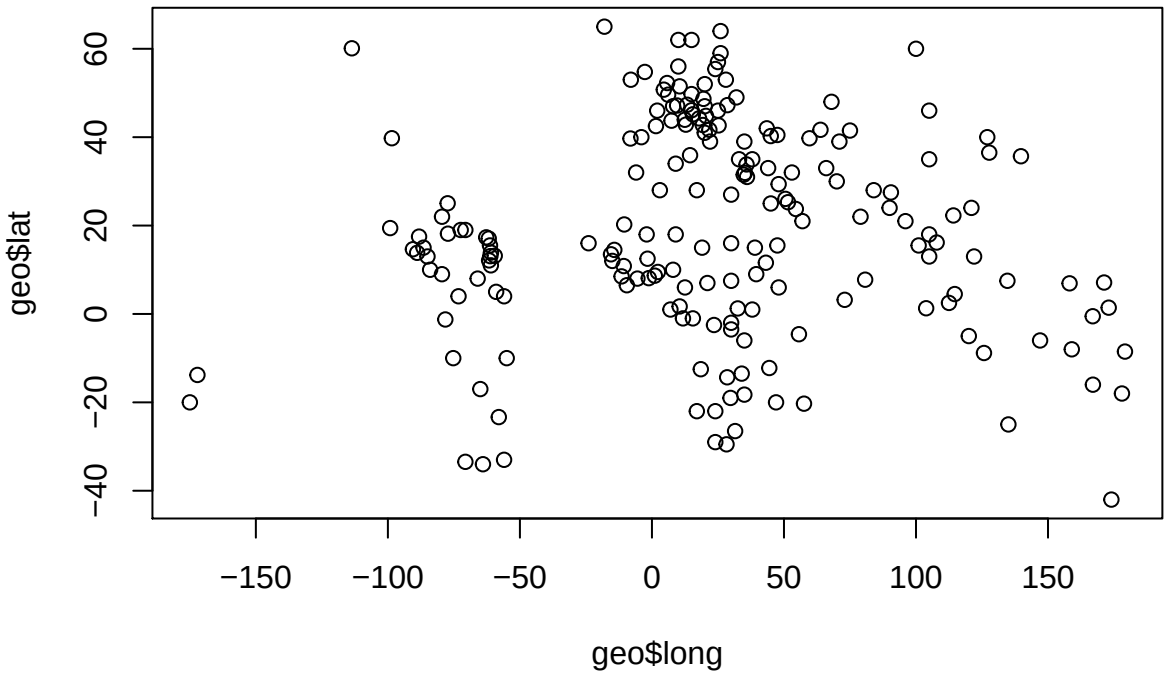
To get the "region" value of the 4th row as above, but unwrapped, we can use:

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

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

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

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



3.5 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 lattidued 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 thhe `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

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.

3.5.1 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)
```

```
## Warning: package 'bindrcpp' was built under R version 3.4.4
```

```
## # 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.

3.6 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"))
```

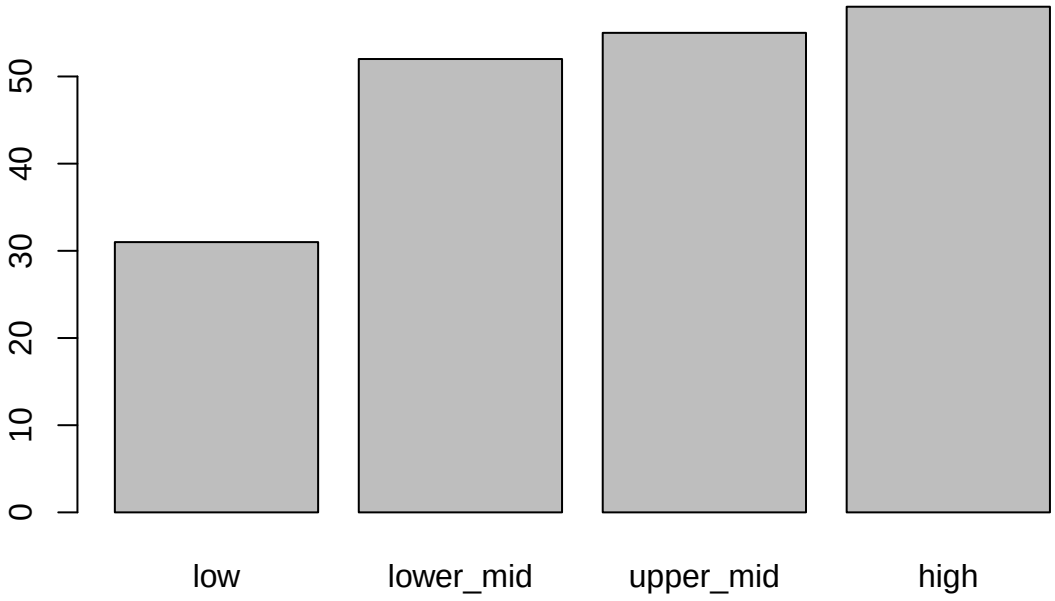
`count` now produces the desired order of output:

```
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
```

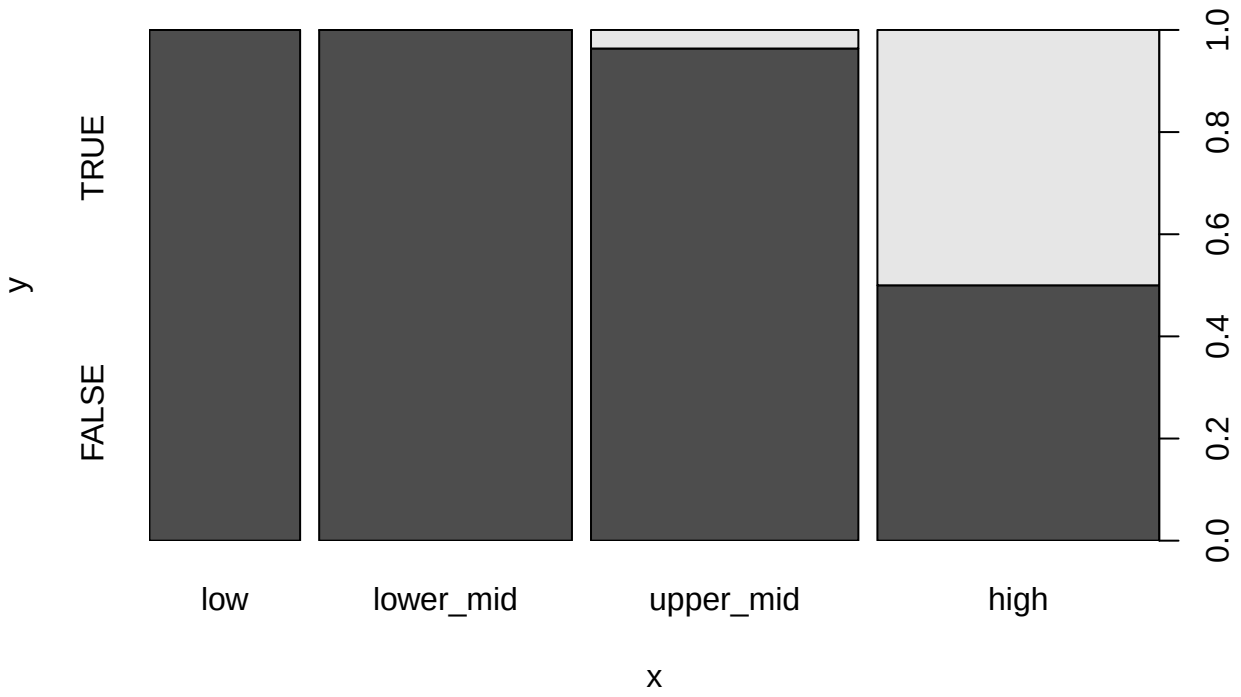
When `plot` is given a factor, it shows a bar plot:

```
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
```

3.7 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 rows and OECD membership as columns. 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 untidy) data. We've just seen `spread` which spreads two columns into multiple columns. The inverse of `spread` is `gather`, which gathers multiple columns into two columns: a column of column names, and a column of values.

Challenge

Investigate which regions of the world OECD members come from by:

1. Counting.
2. Using a mosaic plot.

Remember you may need to convert columns to factors for `plot` to work.

3.8 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
```

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

```
##  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
```

Chapter 4

Plotting with ggplot2

```
library(tidyverse)
```

```
## Warning: package 'tibble' was built under R version 3.4.3
```

```
## Warning: package 'tidyr' was built under R version 3.4.3
```

```
## Warning: package 'forcats' was built under R version 3.4.3
```

```
geo <- read_csv("r-intro-2-files/geo.csv")
geo$income2017 <- factor(geo$income2017, levels=c("low","lower_mid","upper_mid","high"))
```

4.1 A larger data set

Let's move on to a larger data set.

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

```
## Parsed with column specification:
## cols(
##   name = col_character(),
##   year = col_double(),
##   population = col_double(),
##   gdp_percap = col_integer(),
##   life_exp = col_double()
## )
```

```
## Warning in rbind(names(probs), probs_f): number of columns of result is not
## a multiple of vector length (arg 1)
```

```
## Warning: 54 parsing failures.
## row # A tibble: 5 x 5 col     row col        expected               actual file
## ... ................. ... ..........................................................
## See problems(...) for more details.
```

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

Chapter 5

Summarizing data

Having loaded and thoroughly explored a data set, and not before, 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 this is largely beyond the scope of today.

Chapter 6

Thinking in R

6.1 Lists

6.2 Matrices (optional section)

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 sort of ID 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.