R Recap

Mark Dunning

Last modified: 28 Sep 2016

A short introduction to R

Outline

In this session, we review some of the fundamentals of the R language. It should be a useful refresher prior to the intermediate-level Data Analysis and Visualisation using R course.

Topics covered include:-

- Creating variables
- Using Functions
- Vectors
- Data frame
- Subsetting data, the base R way
- Plotting
- Statistical testing
- How to get help

For a more detailed introduction, we suggest the following ***free*** resources

- Solving Biological Problems with R
- Introduction to Data Science with R
- Coursera course in R
- Beginners Introduction to R Statistical Software
- R programming wiki
- Quick R

R basics

Advantages of R

The R programming language is now recognised beyond the academic community as an effect solution for data analysis and visualisation. Notable users of R include:-

- Facebook,
- google,
- Microsoft (who recently invested in a commerical provider of R)
- The New York Times.
- Buzzfeed use R for some of their serious articles (i.e. not the ones featuring cat pictures), and have made the code publically available
- The New Zealand Tourist Board have R running in the background of their website

The New York Times

Business Computing

WORLD | U.S. | N.Y. / REGION | BUSINESS | TECHNOLOGY | SCIENCE | HEALTH | SPORTS | OPINION

Search Technology

Go

Inside Technology

Internet | Start-Ups | Business Computing | Companies

Bits
Blo

Data Analysts Captivated by R's Power

Stuart Isett for The New York Times

R first appeared in 1996, when the statistics professors Robert Gentleman, left, and Ross Ihaka released the code as a free software package.

Figure 1:

- Airbnb 

Key features

- Open-source
- Cross-platform
- Access to existing visualisation / statistical tools
- Flexibility
- Visualisation and interactivity
- Add-ons for many fields of research
- Facilitating ***Reproducible Research***

Two Biostatiscians (later termed '*Forensic Bioinformaticians*') from M.D. Anderson used R extensively during their re-analysis and investigation of a Clinical Prognostication paper from Duke. The subsequent scandal put Reproducible Research at the forefront of everyone's mind.

Keith Baggerly's talk on the subject is highy-recommended.

Support for R

- Online forums such as Stack Overflow regularly feature R
- Blogs
- Local user groups
- Documentation via `?` or `help.start()`
- Documentation for packages is found via the Packages tab in the bottom-right of RStudio.

- Packages analyse all kinds of Genomic data (>800)
- Compulsory documentation (*vignettes*) for each package
- 6-month release cycle
- Course Materials
- Example data and workflows
- Common, re-usable framework and functionality
- Available Support
  - Often you will be able to interact with the package maintainers / developers and other power-users of the project software
- Annual conferences in U.S and Europe
  - The last European conference was in Cambridge

RESEARCH

75 COMMENTS

How Bright Promise in Cancer Testing Fell Apart

By GINA KOLATA JULY 7, 2011

Email

Share

Tweet

Pin

Save

More

When Juliet Jacobs found out she had lung cancer, she was terrified, but realized that her hope lay in getting the best treatment medicine could offer. So she got a second opinion, then a third. In February of 2010, she ended up at Duke University, where she entered a research study whose promise seemed stunning.

Doctors would assess her tumor cells, looking for gene patterns that would determine which drugs would best

Keith Baggerly, left, and Kevin Coombes, statisticians at M. D. Anderson Cancer Center, found flaws in research on tumors.
Michael Stravato for The New York Times

Figure 2: duke-scandal

Bioconductor
OPEN SOURCE SOFTWARE FOR BIOINFORMATICS

Figure 3:

RStudio

- Rstudio is a free environment for R
- Convenient menus to access scripts, display plots
- Still need to use *command-line* to get things done
- Developed by some of the leading R programmers
- Used by beginners, and experienced users alike

To get started, you will need to install the latest version of R and RStudio Desktop; both of which are ***free***.

Once installed, you should be able to launch RStudio by clicking on its icon:-

Figure 4:

The bottom-left with the blinking cursor is the R console `>` and ready for you to start entering R commands

Getting started

Figure 5:

At a basic level, we can use R as a calculator to compute simple sums with the `+`, `-`, `*` (for multiplication) and `/` (for division) symbols.

```
2 + 2
```

```
## [1] 4
```

```
2 - 2
```

```
## [1] 0
```

```
4 * 3
```

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

```
10 / 2
```

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

The answer is displayed at the console with a `[1]` in front of it. The `1` inside the square brackets is a place-holder to signify how many values were in the answer (in this case only one). We will talk about dealing with lists of numbers shortly…

In the case of expressions involving multiple operations, R respects the BODMAS system to decide the order in which operations should be performed.

```
2 + 2 *3
```

```
## [1] 8
```

```
2 + (2 * 3)
```

```
## [1] 8
```

```
(2 + 2) * 3
```

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

R is capable of more complicated arithmetic such as trigonometry and logarithms; like you would find on a fancy scientific calculator. Of course, R also has a plethora of statistical operations as we will see.

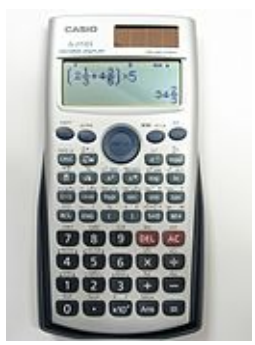


Figure 6:

```
pi
```

```
## [1] 3.141593
```

```
sin (pi/2)
```

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

```
cos(pi)
```

```
## [1] -1
```

```r
tan(2)
```

```
## [1] -2.18504
```

```r
log(1)
```

```
## [1] 0
```

We can only go so far with performing simple calculations like this. Eventually we will need to store our results for later use. For this, we need to make use of *variables*.

Variables

A variable is a letter or word which takes (or contains) a value. We use the assignment 'operator', `<-` to create a variable and store some value in it.

```r
x <- 10
x
```

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

```r
myNumber <- 25
myNumber
```

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

We also can perform arithmetic on variables using functions:

```r
sqrt(myNumber)
```

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

We can add variables together:

```r
x + myNumber
```

```
## [1] 35
```

We can change the value of an existing variable:

```r
x <- 21
x
```

```
## [1] 21
```

- We can set one variable to equal the value of another variable:

```
x <- myNumber
x
```

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

- We can modify the contents of a variable:

```
myNumber <- myNumber + sqrt(16)
myNumber
```

```
## [1] 29
```

When we are feeling lazy we might give our variables short names (`x`, `y`, `i`...etc), but a better practice would be to give them meaningful names. There are some restrictions on creating variable names. They cannot start with a number or contain characters such as `.`, `_`, '-'. Naming variables the same as in-built functions in R, such as `c`, `T`, `mean` should also be avoided.

Naming variables is a matter of taste. Some conventions exist such as a separating words with `-` or using *camelCaps*. Whatever convention you decided, stick with it!

Functions

Functions in R perform operations on **arguments** (the inputs(s) to the function). We have already used:

```
sin(x)
```

this returns the sine of x. In this case the function has one argument: `x`. Arguments are always contained in parentheses – curved brackets, **()** – separated by commas.

Arguments can be named or unnamed, but if they are unnamed they must be ordered (we will see later how to find the right order). The names of the arguments are determined by the author of the function and can be found in the help page for the function. When testing code, it is easier and safer to name the arguments. `seq` is a function for generating a numeric sequence *from* and *to* particular numbers. Type `?seq` to get the help page for this function.

```
seq(from = 2, to = 20, by = 4)
```

```
## [1]  2  6 10 14 18
```

```
seq(2, 20, 4)
```

```
## [1]  2  6 10 14 18
```

Arguments can have *default* values, meaning we do not need to specify values for these in order to run the function.

`rnorm` is a function that will generate a series of values from a *normal distribution*. In order to use the function, we need to tell R how many values we want

```r
rnorm(n=10)
```

```
##  [1] -1.1545987 -0.6001460 -0.3767585  0.6538441  0.6573373  0.9750179
##  [7]  0.8412439  1.8738493 -0.3658604  0.4971670
```

The normal distribution is defined by a *mean* (average) and *standard deviation* (spread). However, in the above example we didn't tell R what mean and standard deviation we wanted. So how does R know what to do? All arguments to a function and their default values are listed in the help page

(*N.B sometimes help pages can describe more than one function*)

```r
?rnorm
```

In this case, we see that the defaults for mean and standard deviation are 0 and 1. We can change the function to generate values from a distribution with a different mean and standard deviation using the `mean` and `sd` *arguments*. It is important that we get the spelling of these arguments exactly right, otherwise R will an error message, or (worse?) do something unexpected.

```r
rnorm(n=10, mean=2,sd=3)
```

```
##  [1] 2.33700980 1.91213659 2.86765013 1.48595756 0.09109697 3.50531707
##  [7] 1.23465247 0.14534529 1.07741759 5.44207601
```

```r
rnorm(10, 2, 3)
```

```
##  [1] -4.0331699  0.5827305 -0.8719965 -1.1680345  6.4138648  0.5763937
##  [7]  2.2514787  4.5439213 -3.0705028 -1.2673496
```

In the examples above, `seq` and `rnorm` were both outputting a series of numbers, which is called a *vector* in R and is the most-fundamental data-type.

Exercise

- How can we a create a sequence from 2 to 20 comprised of 5 equally-spaced numbers?
  - check the help page for seq `?seq`
- What is the value of `pi` to 3 decimal places?
  - see the help for `round` `?round`

Vectors

- The basic data structure in R is a **vector** – an ordered collection of values.
- R treats even single values as 1-element vectors.
- The function `c` *combines* its arguments into a vector:
- Remember that as `c` is a function, we specify it's arguments in curved brackets`(...)`

```
x <- c(3,4,5,6)
x
```

```
## [1] 3 4 5 6
```

The `seq` function we saw before was another example of how to create a sequence of values. A useful shortcut is to use the `:` symbol.

```
x <- 3:6
x
```

```
## [1] 3 4 5 6
```

_____

_____

_____

Exercise

- `rep` can be used to replicate values. Construct the following sequences
  - `1 1 1 1 1 2 2 2 2 2`
  - `1 2 1 2 1 2 1 2 1 2`
  - `1 2 1 2 1 2 1 2 1 2 1`
  - (this last sequence has 11 values in it)

_____

_____

_____

The square brackets `[]` indicate the position within the vector (the ***index***). We can extract individual elements by using the `[]` notation:

```
x[1]
```

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

```r
x[4]
```

```
## [1] 6
```

We can even put a vector inside the square brackets: (*vector indexing*)

Exercise

Without using R!

- If `y <- 2:4`, what would `x[y]` give?
  - `[1]  3  5`
  - `[1]  2  4`
  - `[1]  4  5  6`

When applying all standard arithmetic operations to vectors, application is element-wise. Thus, we say that R supports *vectorised* operations.

```r
x <- 1:10
y <- x*2
y
```

```
##  [1]  2  4  6  8 10 12 14 16 18 20
```

```r
z <- x^2

x + y
```

```
##  [1]  3  6  9 12 15 18 21 24 27 30
```

Vectorised operations are extremely powerful. Operations that would require a *for* loop (or similar) in other languages such as **C**, **Python**, can be performed in a single line of R code.

A vector can also contain text; called a character vector. Such a vector can also be constructed using the `c` function.

```r
x <- c("A","B","C","D")
```

The quote marks are crucial. Why?

```r
x <- c(A, B, C, D)
```

```
## Error in try(x <- c(A, B, C, D), silent = TRUE) : object 'A' not found
```

Another useful type of data that we will see is the *logical* or *boolean* which can take either the values of `TRUE` or `FALSE`

```r
x <- c(TRUE,TRUE,FALSE)
```

Logical values are useful when we want to create subsets of our data. We can use *comparison* operators; `==`, `>`, `<`, `!=` to check if values are equal, greater than, less than, or not equal.

```r
x <- c("A","A", "B","B","C")
x == "A"
```

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

```r
x != "A"
```

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

```r
x <- rnorm(10)
x > 0
```

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

However, all items in the vector **must** be the same type. If you attempt anything else, R will convert all values to the same (most basic) type.

```r
x <- c(1, 2, "three")
x
```

```
## [1] "1"     "2"     "three"
```

Packages in R

So far we have used functions that are available with the *base* distribution of R; the functions you get with a clean install of R. The open-source nature of R encourages others to write their own functions for their particular data-type or analyses.

Packages are distributed through *repositories*. The most-common ones are CRAN and Bioconductor. CRAN alone has many thousands of packages.

The **Packages** tab in the bottom-right panel of RStudio lists all packages that you currently have installed. Clicking on a package name will show a list of functions that available once that package has been loaded. The `library` function is used to load a package and make it's functions / data available in your current R session. *You need to do this every time you load a new RStudio session.*

```r
library(beadarray)
```

There are functions for installing packages within R. If your package is part of the main **CRAN** repository, you can use `install.packages`

We will be using the `gapminder` R package in this practical. To install it, we would do.

```r
install.packages("gapminder")
```

Bioconductor packages have their own install script, which you can download from the Bioconductor website

```r
source("http://www.bioconductor.org/biocLite.R")
biocLite("affy")
```

A package may have several *dependancies*; other R packages from which it uses functions or data types (re-using code from other packages is strongly-encouraged). If this is the case, the other R packages will be located and installed too.

So long as you stick with the same version of R, you won't need to repeat this install process.

Dealing with data

We are going to explore some of the basic features of R using data from the gapminder project, which have been bundled into an R package. These data give various indicator variables for different countries around the world (life expectancy, population and Gross Domestic Product). We have saved these data as a `.csv` file to demonstrate how to import data into R.

You can download these data here. Right-click the link and save to somewhere on your computer that you wish to work from.

The working directory

Like other software (Word, Excel, Photoshop. . . .), R has a default location where it will save files to and import data from. This is known as the *working directory* in R. You can query what R currently considers its working directory by doing:-

```r
getwd()
```

N.B. Here, a set of open and closed brackets `()` is used to call the `getwd` function with no arguments.

We can also list the files in this directory with:-

```
list.files()
```

Any `.csv` file in the working directory can be imported into R by supplying the name of the file to the `read.csv` function and creating a new variable to store the result. A useful sanity check is the `file.exists` function which will print `TRUE` is the file can be found in the working directory.

```r
file.exists("gapminder.csv")
```

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

If the file we want to read is not in the current working directory, we will have to write the path to the file; either *relevant* to the current working directory (e.g. the directory "up" from the current working directory, or in a sub-folder), or the full path. In an interactive session, you can do use `file.choose` to open a dialogue box. The path to the the file will then be displayed in R.

```
file.choose()
```

Assuming the file can be found, we can use `read.csv` to import. Other functions can be used to read tab-delimited files (`read.delim`) or a generic `read.table` function. A data frame object is created.

```
gapminder <- read.csv("gapminder.csv")
```

The data frame object in R allows us to work with "tabular" data, like we might be used to dealing with in Excel, where our data can be thought of having rows and columns. The values in each column have to all be of the same type (i.e. all numbers or all text).

Exercise

- What are the dimensions of the data frame?
- What columns are available?
- HINT: see the `dim`, `ncol`, `nrow` and `colnames` functions

In Rstudio , you can view the contents of the data frame we have just created. This is useful for interactive exploration of the data, but not so useful for automation and scripting and analyses.

```
View(gapminder)
```

We should always check the data frame that we have created. Sometimes R will happily read data using an inappropriate function and create an object without raising an error. However, the data might be unsuable. Consider:-

```
test <- read.delim("gapminder.csv")
head(test)
```

```
##        country.continent.year.lifeExp.pop.gdpPercap
## 1  Afghanistan,Asia,1952,28.801,8425333,779.4453145
## 2  Afghanistan,Asia,1957,30.332,9240934,820.8530296
## 3   Afghanistan,Asia,1962,31.997,10267083,853.10071
## 4  Afghanistan,Asia,1967,34.02,11537966,836.1971382
## 5 Afghanistan,Asia,1972,36.088,13079460,739.9811058
## 6   Afghanistan,Asia,1977,38.438,14880372,786.11336
```

```r
dim(test)
```

```
## [1] 1704    1
```

We can access the columns of a data frame by knowing the column name. ***TIP*** Use auto-complete with the ***TAB*** key to get the name of the column correct

```r
gapminder$country
```

A vector (1-dimensional) is returned, the length of which is the same as the number of rows in the data frame. The vector could be stored as a variable and itself be subset or used in further calculations

The `summary` function is a useful way of summarising the data containing in each column. It will give information about the *type* of data (remember, data frames can have a mixture of numeric and character columns) and also an appropriate summary. For numeric columns, it will report some stats about the distribution of the data. For categorical data, it will report the different *levels*.

```r
summary(gapminder)
```

```
##         country        continent        year         lifeExp
##  Afghanistan:  12   Africa  :624   Min.   :1952   Min.   :23.60
##  Albania    :  12   Americas:300   1st Qu.:1966   1st Qu.:48.20
##  Algeria    :  12   Asia    :396   Median :1980   Median :60.71
##  Angola     :  12   Europe  :360   Mean   :1980   Mean   :59.47
##  Argentina  :  12   Oceania : 24   3rd Qu.:1993   3rd Qu.:70.85
##  Australia  :  12                  Max.   :2007   Max.   :82.60
##  (Other)    :1632
##       pop              gdpPercap
##  Min.   :6.001e+04   Min.   :   241.2
##  1st Qu.:2.794e+06   1st Qu.:  1202.1
##  Median :7.024e+06   Median :  3531.8
##  Mean   :2.960e+07   Mean   :  7215.3
##  3rd Qu.:1.959e+07   3rd Qu.:  9325.5
##  Max.   :1.319e+09   Max.   :113523.1
##
```

Exercise

- Save the life expectancy and population as variables
  - what is the maximum life expectancy?
  - what is the smallest population?
  - round the life expectancy and populations to the nearest whole numbers
  - HINT:- `min`, `max`, `round`. . . ..

Subsetting

A data frame can be subset using square brackes `[]` placed after the name of the data frame. As a data frame is a two-dimensional object, you need a *row* and *column* index, or vector indices.

```
gapminder[1,2]
gapminder[2,1]
gapminder[c(1,2,3),1]
gapminder[c(1,2,3),c(1,2,3)]
```

Note that the data frame is not altered we are just seeing what a subset of the data looks like and not changing the underlying data. If we wanted to do this, we would need to create a new variale.

```
gapminder
```

Should we wish to see all rows, or all columns, we can neglect either the row or column index

```
gapminder[1,]
gapminder[,1]
```

Just like subsetting a vector, the indices can be vectors containing multiple values

```
gapminder[1:3,1:2]
gapminder[seq(1,1704,length.out = 10),1:4]
```

A common shortcut is `head` which prints the first six rows of a data frame.

```
head(gapminder)
```

```
##       country continent year lifeExp      pop gdpPercap
## 1 Afghanistan      Asia 1952  28.801  8425333  779.4453
## 2 Afghanistan      Asia 1957  30.332  9240934  820.8530
## 3 Afghanistan      Asia 1962  31.997 10267083  853.1007
## 4 Afghanistan      Asia 1967  34.020 11537966  836.1971
## 5 Afghanistan      Asia 1972  36.088 13079460  739.9811
## 6 Afghanistan      Asia 1977  38.438 14880372  786.1134
```

When subsetting entire rows ***you need to remember the , after the row indices***. If you fail to do so, R may still return a result. However, it probably won't be what you expected. Look what happens tf you wanted to the first three rows but typed the following command

```
gapminder[1:3]
```

Rather than selecting rows based on their *numeric* index (as in the previous example) we can use what we call a *logical test*. This is a test that gives either a `TRUE` or `FALSE` result. When applied to subsetting, only rows with a `TRUE` result get returned.

For example we could compare the `lifeExp` variable to 40. The result is a *vector* of `TRUE` or `FALSE`; one for each row in the data frame

```r
gapminder$lifeExp < 40
```

This R code can be put inside the square brackets to select rows of interest (those observations where the life expectancy variable is less than 40).

```r
gapminder[gapminder$lifeExp < 40, ]
```

```
##       country continent year lifeExp      pop gdpPercap
## 1 Afghanistan      Asia 1952  28.801  8425333  779.4453
## 2 Afghanistan      Asia 1957  30.332  9240934  820.8530
## 3 Afghanistan      Asia 1962  31.997 10267083  853.1007
## 4 Afghanistan      Asia 1967  34.020 11537966  836.1971
## 5 Afghanistan      Asia 1972  36.088 13079460  739.9811
## 6 Afghanistan      Asia 1977  38.438 14880372  786.1134
```

The `,` is important as this tells R to display all columns. If we wanted a subset of the columns we would put their indices after the `,`

```r
gapminder[gapminder$lifeExp < 40, 1:4]
```

```
##       country continent year lifeExp
## 1 Afghanistan      Asia 1952  28.801
## 2 Afghanistan      Asia 1957  30.332
## 3 Afghanistan      Asia 1962  31.997
## 4 Afghanistan      Asia 1967  34.020
## 5 Afghanistan      Asia 1972  36.088
## 6 Afghanistan      Asia 1977  38.438
```

Using the column names is also valid

```r
gapminder[gapminder$lifeExp < 40, c("country", "continent","year")]
```

```
##       country continent year
## 1 Afghanistan      Asia 1952
## 2 Afghanistan      Asia 1957
## 3 Afghanistan      Asia 1962
## 4 Afghanistan      Asia 1967
## 5 Afghanistan      Asia 1972
## 6 Afghanistan      Asia 1977
```

Testing for equality can be done using `==`. This will only give `TRUE` for entries that are *exactly* the same as the test string.

```r
gapminder[gapminder$country == "Zambia",]
```

```
##      country continent year lifeExp      pop gdpPercap
## 1681  Zambia    Africa 1952  42.038  2672000  1147.389
## 1682  Zambia    Africa 1957  44.077  3016000  1311.957
## 1683  Zambia    Africa 1962  46.023  3421000  1452.726
## 1684  Zambia    Africa 1967  47.768  3900000  1777.077
```

```
## 1685  Zambia    Africa 1972  50.107  4506497  1773.498
## 1686  Zambia    Africa 1977  51.386  5216550  1588.688
## 1687  Zambia    Africa 1982  51.821  6100407  1408.679
## 1688  Zambia    Africa 1987  50.821  7272406  1213.315
## 1689  Zambia    Africa 1992  46.100  8381163  1210.885
## 1690  Zambia    Africa 1997  40.238  9417789  1071.354
## 1691  Zambia    Africa 2002  39.193 10595811  1071.614
## 1692  Zambia    Africa 2007  42.384 11746035  1271.212
```

N.B. For partial matches, the `grep` function and / or *regular expressions* (if you know them) can be used.

```
gapminder[grep("land", gapminder$country),]
```

```
##     country continent year lifeExp     pop gdpPercap
## 517 Finland    Europe 1952   66.55 4090500  6424.519
## 518 Finland    Europe 1957   67.49 4324000  7545.415
## 519 Finland    Europe 1962   68.75 4491443  9371.843
## 520 Finland    Europe 1967   69.83 4605744 10921.636
## 521 Finland    Europe 1972   70.87 4639657 14358.876
## 522 Finland    Europe 1977   72.52 4738902 15605.423
```

There are a couple of ways of testing for more than one text value. The first uses an *or* `|` statement. i.e. testing if the value of `country` is `Zambia` *or* the value is `Zimbabwe`.

The `%in%` function is a convenient function for testing which items in a vector correspond to a defined set of values.

```
gapminder[gapminder$country == "Zambia" | gapminder$country == "Zimbabwe",]
gapminder[gapminder$country %in% c("Zambia","Zimbabwe"),]
```

```
##      country continent year lifeExp     pop gdpPercap
## 1681  Zambia    Africa 1952  42.038 2672000  1147.389
## 1682  Zambia    Africa 1957  44.077 3016000  1311.957
## 1683  Zambia    Africa 1962  46.023 3421000  1452.726
## 1684  Zambia    Africa 1967  47.768 3900000  1777.077
## 1685  Zambia    Africa 1972  50.107 4506497  1773.498
## 1686  Zambia    Africa 1977  51.386 5216550  1588.688
```

Similar to *or*, we can require that both tests are `TRUE` by using an *and* `&` operation. e.g. which years in Zambia had a life expectancy less than 40

```
gapminder[gapminder$country == "Zambia" & gapminder$lifeExp < 40,]
```

```
##      country continent year lifeExp      pop gdpPercap
## 1691  Zambia    Africa 2002  39.193 10595811  1071.614
```

Exercise

- Can you create a data frame of countries with a population less than a million in the year 2002

```
##                    country continent year lifeExp    pop gdpPercap
## 95                 Bahrain      Asia 2002  74.795 656397 23403.559
## 323                Comoros    Africa 2002  62.974 614382  1075.812
## 431               Djibouti    Africa 2002  53.373 447416  1908.261
## 491      Equatorial Guinea    Africa 2002  49.348 495627  7703.496
## 695                Iceland    Europe 2002  80.500 288030 31163.202
## 1019            Montenegro    Europe 2002  73.981 720230  6557.194
## 1271               Reunion    Africa 2002  75.744 743981  6316.165
## 1307 Sao Tome and Principe    Africa 2002  64.337 170372  1353.092
```

- A data frame of countries with a population less than a million in the year 2002, that are not in Africa?

_____

_____

_____

Ordering and sorting

A vector can be returned in sorted form using the `sort` function.

```
sort(countries)
sort(countries,decreasing = TRUE)
```

However, if we want to sort an entire data frame a different approach is needed. The trick is to use `order`. Rather than giving a sorted set of *values*, it will give sorted *indices*. These indices can then be used for a subset operation.

```
leastPop <- gapminder[order(gapminder$pop),]
head(leastPop)
```

```
##                    country continent year lifeExp   pop gdpPercap
## 1297 Sao Tome and Principe    Africa 1952  46.471 60011  879.5836
## 1298 Sao Tome and Principe    Africa 1957  48.945 61325  860.7369
## 421               Djibouti    Africa 1952  34.812 63149 2669.5295
## 1299 Sao Tome and Principe    Africa 1962  51.893 65345 1071.5511
## 1300 Sao Tome and Principe    Africa 1967  54.425 70787 1384.8406
## 422               Djibouti    Africa 1957  37.328 71851 2864.9691
```

A final point on data frames is that we can export them out of R once we have done our data processing.

```
byWealth <- gapminder[order(gapminder$gdpPercap,decreasing = TRUE),]
head(byWealth)
```

```
##     country continent year lifeExp     pop gdpPercap
## 854  Kuwait      Asia 1957  58.033  212846 113523.13
## 857  Kuwait      Asia 1972  67.712  841934 109347.87
## 853  Kuwait      Asia 1952  55.565  160000 108382.35
## 855  Kuwait      Asia 1962  60.470  358266  95458.11
## 856  Kuwait      Asia 1967  64.624  575003  80894.88
## 858  Kuwait      Asia 1977  69.343 1140357  59265.48
```

```
write.csv(byWealth, file="dataOrderedByWealth.csv")
```

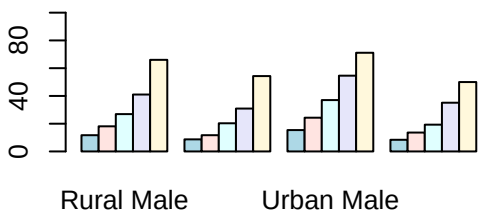
We can even order by more than one condition

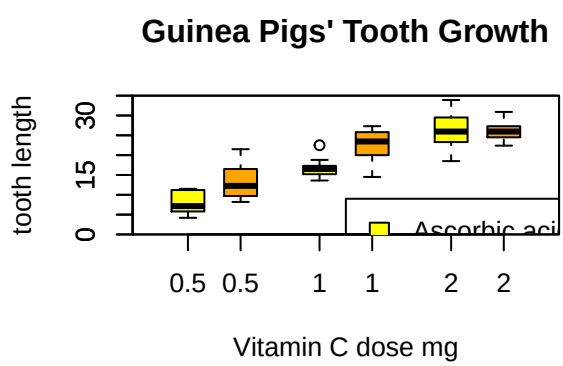
```
gapminder[order(gapminder$year, gapminder$country),]
```

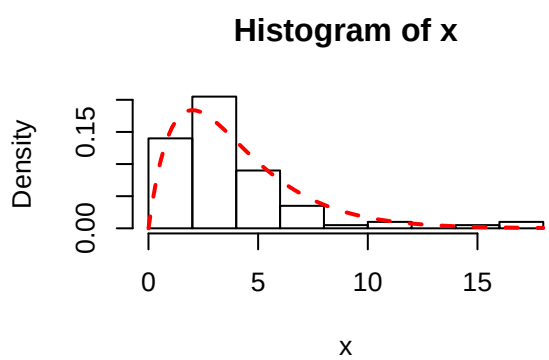
```
##           country continent year lifeExp     pop gdpPercap
## 1     Afghanistan      Asia 1952  28.801 8425333  779.4453
## 553        Gambia    Africa 1952  30.000  284320  485.2307
## 37         Angola    Africa 1952  30.015 4232095 3520.6103
## 1345 Sierra Leone    Africa 1952  30.331 2143249  879.7877
## 1033   Mozambique    Africa 1952  31.286 6446316  468.5260
## 193  Burkina Faso    Africa 1952  31.975 4469979  543.2552
```

Plotting and stats (in brief!)

All your favourite types of plot can be created in R



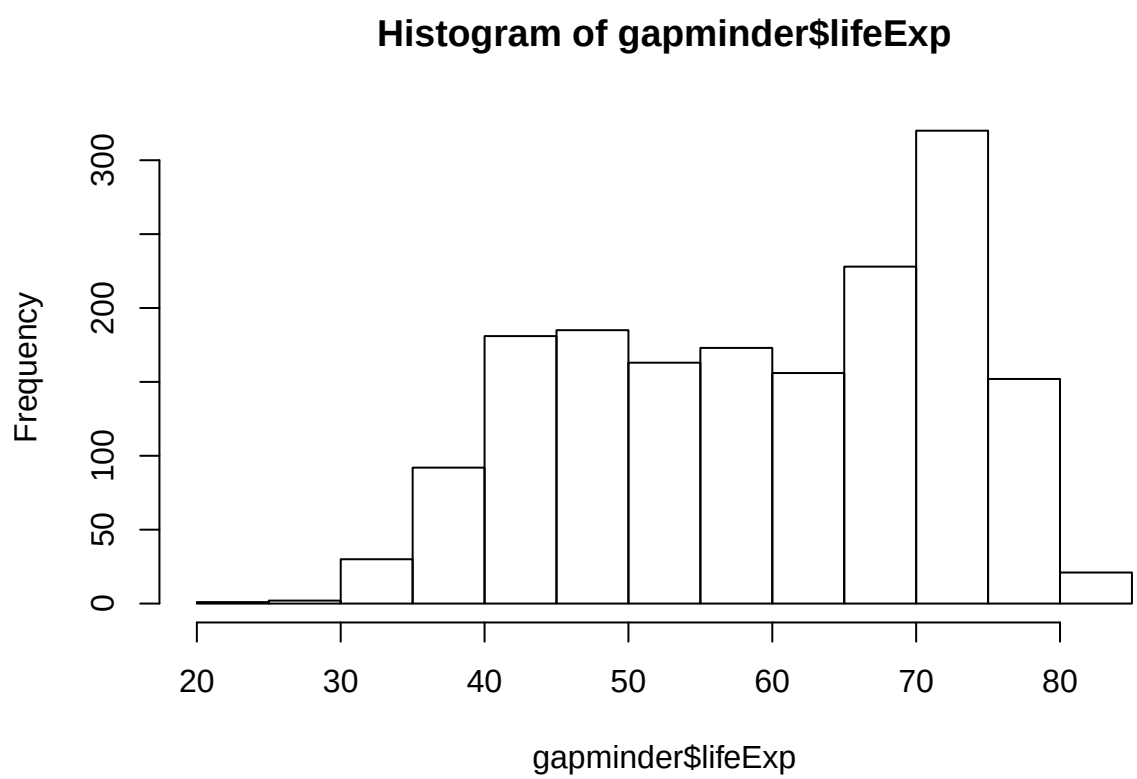




- Simple plots are supported in the *base* distribution of R (what you get automatically when you download R).
  - `boxplot`, `hist`, `barplot`,... all of which are extensions of the basic `plot` function
- Many different customisations are possible
  - colour, overlay points / text, legends, multi-panel figures
- ***You need to think about how best to visualise your data***
  - http://www.bioinformatics.babraham.ac.uk/training.html#figuredesign
- R cannot prevent you from creating a plotting disaster:
  - http://www.businessinsider.com/the-27-worst-charts-of-all-time-2013-6?op=1&IR=T
- References..
  - Introductory R course
  - Quick-R

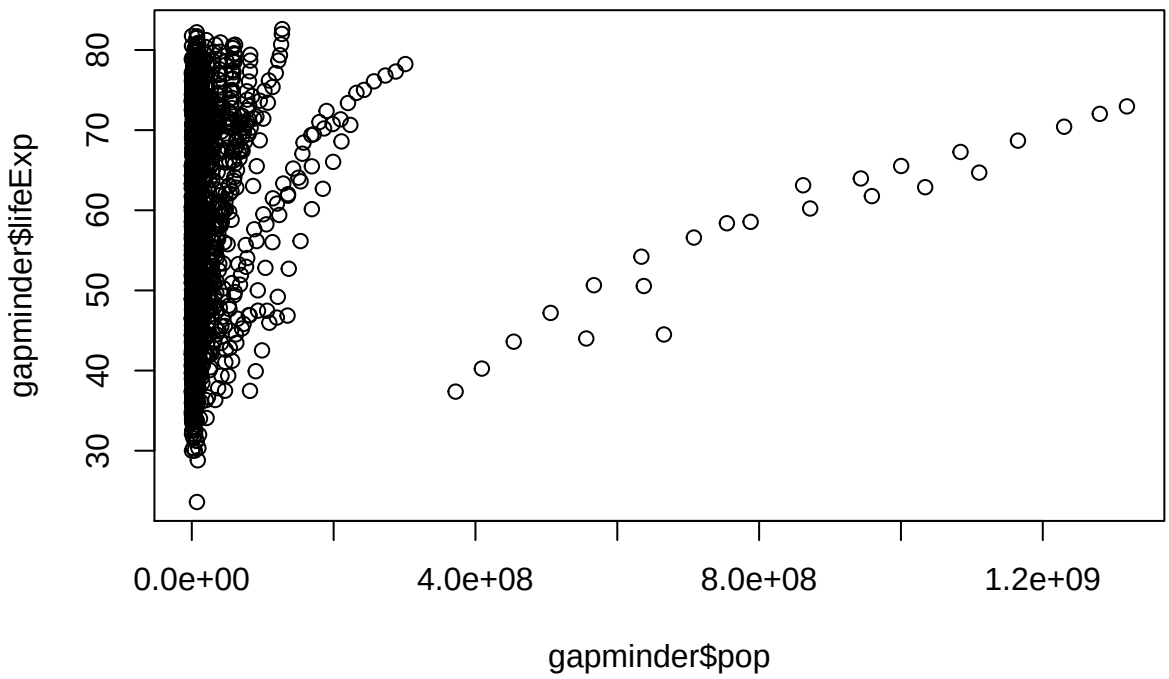
Plots can be constructed from vectors of numeric data, such as the data we get from a particular column in a data frame

```
hist(gapminder$lifeExp)
```



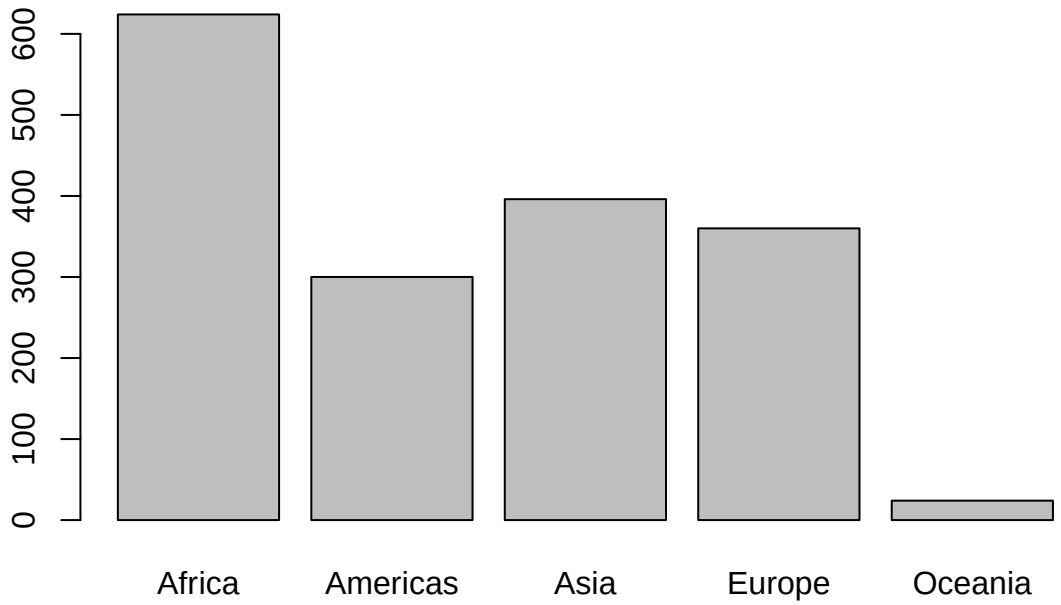
Scatter plots of two variables require two arguments; one for the `x` and one for the `y` axis.

```r
plot(gapminder$pop,gapminder$lifeExp)
```



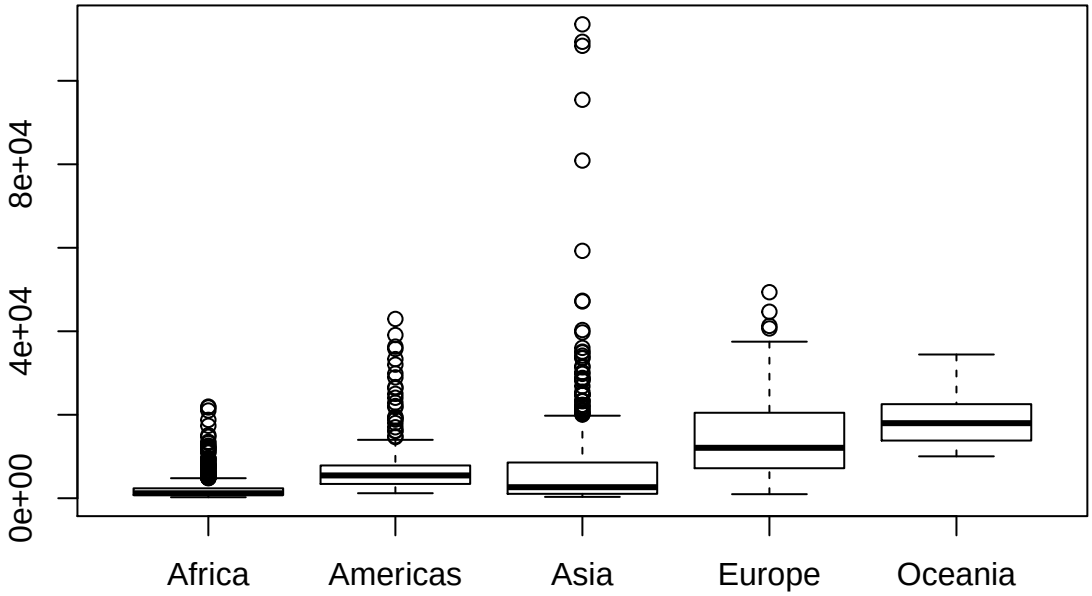
Barplots are commonly-used for counts of categorical data

```r
barplot(table(gapminder$continent))
```



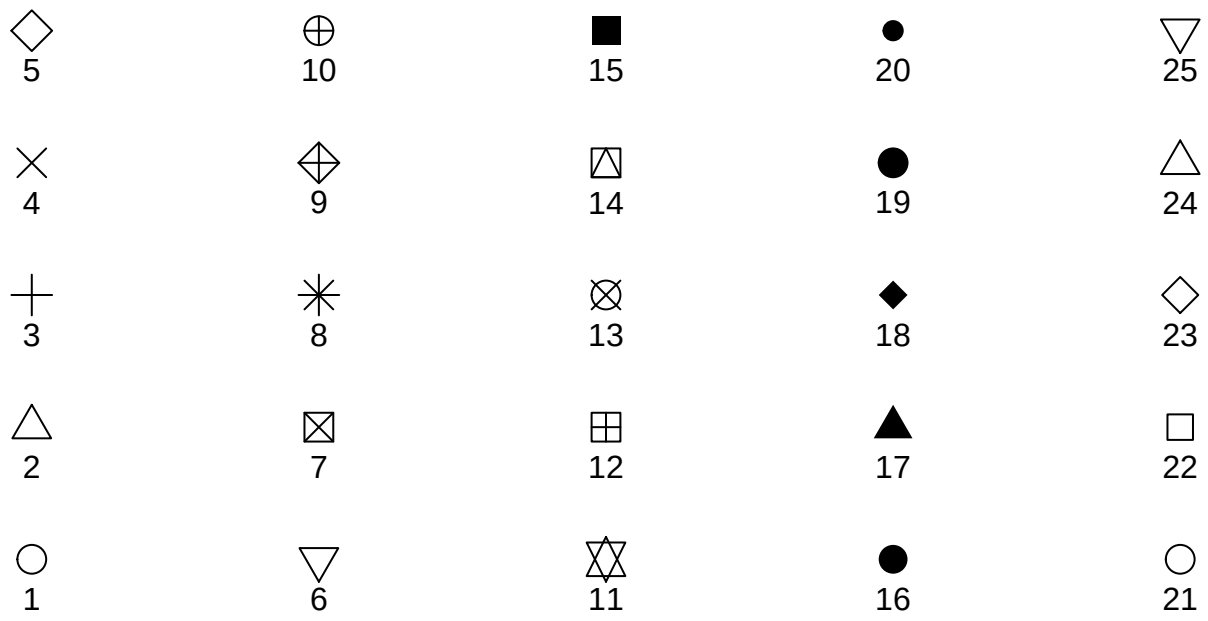
Boxplots are good for visualising and comparing distributions. Here the `~` symbol sets up a formula, the effect of which is to put the categorical variable on the `x` axis and continuous variable on the `y` axis.

```
boxplot(gapminder$gdpPercap ~ gapminder$continent)
```



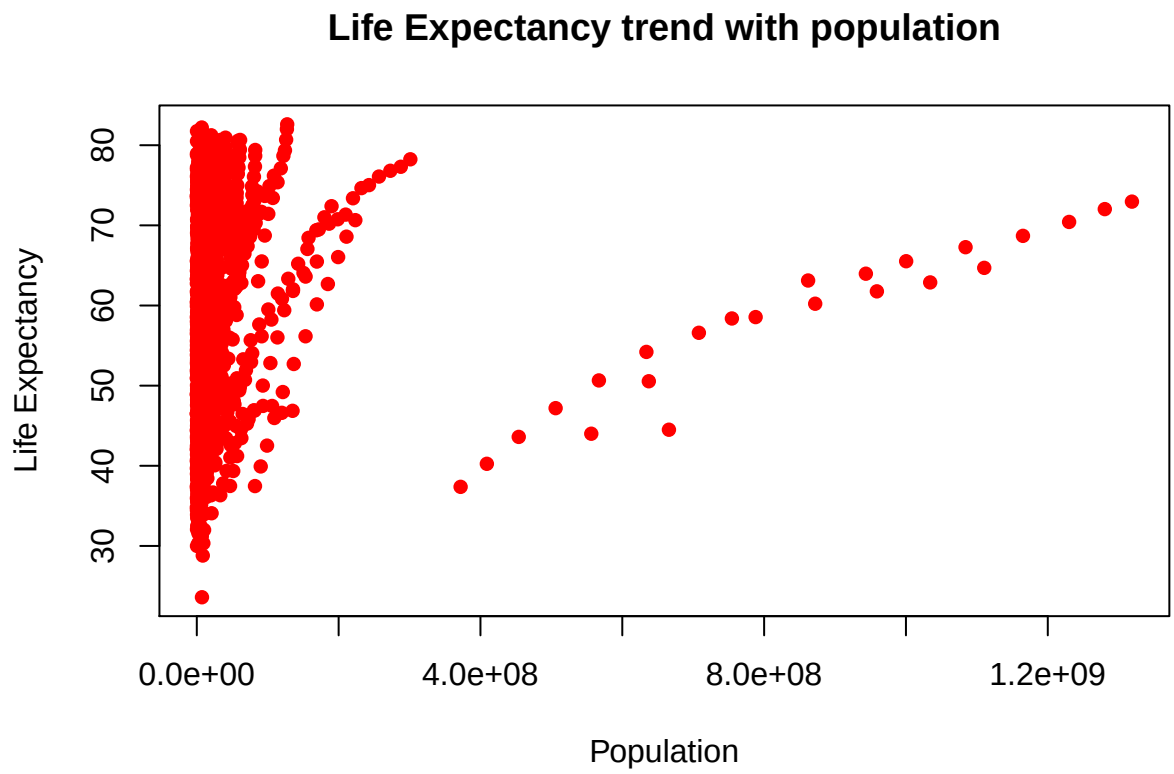
Lots of customisations are possible to enhance the appearance of our plots. Not for the faint-hearted, the help pages `?plot` and `?par` give the full details. In short,

- Axis labels, and titles can be specified as character strings.
- R recognises many preset names as colours. To get a full list use `colours()`, or check this online reference.
- Plotting characters can be specified using a pre-defined number:-



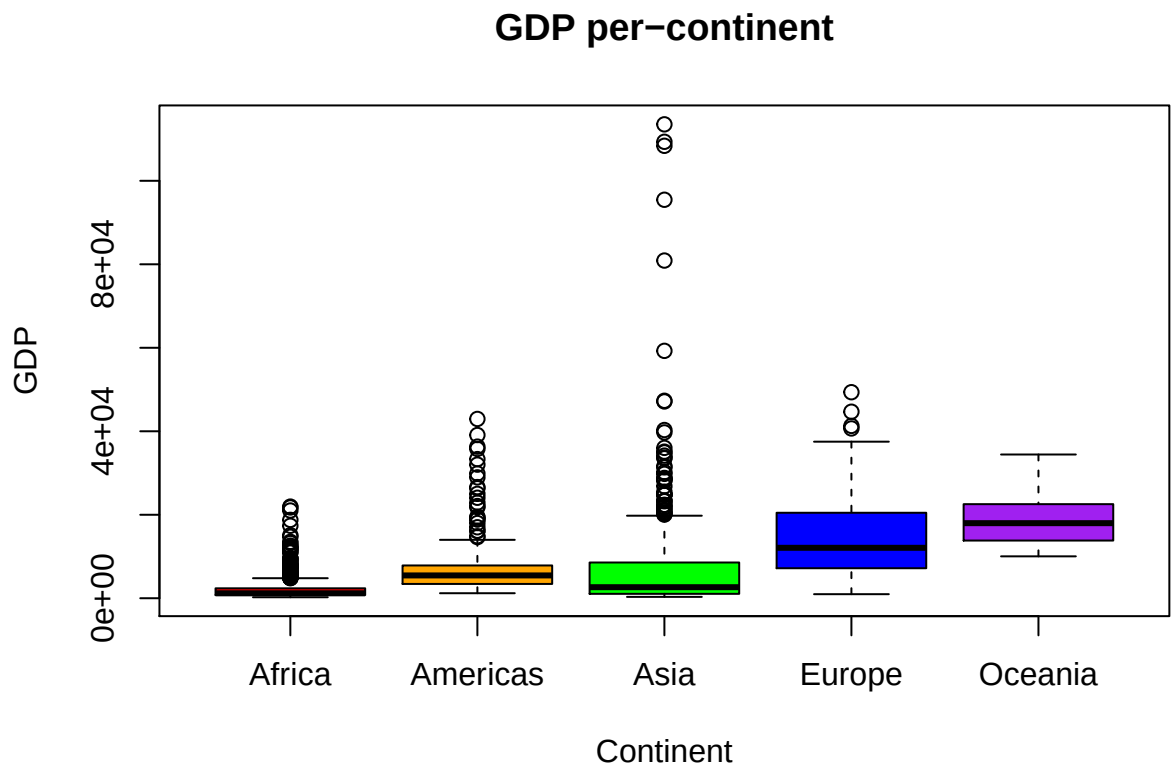
Putting it all together.

```
plot(gapminder$pop,gapminder$lifeExp,pch=16,
     col="red",ylab="Life Expectancy",
     xlab="Population",main="Life Expectancy trend with population")
```



The same customisations can be used for various plots:-

```
boxplot(gapminder$gdpPercap ~ gapminder$continent,col=c("red","orange","green","blue","purple"),
        main="GDP per-continent",
        xlab="Continent",
        ylab="GDP")
```



Plots can be exported by the ***Plots*** tab in RStudio, which is useful in an interactive setting. However, one can also save plots to a file calling the `pdf` or `png` functions before executing the code to create the plot.

```
pdf("myLittlePlot.pdf")
barplot(table(gapminder$continent))
dev.off()
```

```
## pdf
##   2
```

Any plots created in-between the `pdf(..)` and `dev.off()` lines will get saved to the named file. The `dev.off()` line is very important; without it you will not be able to view the plot you have created. `pdf` files are useful because you can create documents with multiple pages. Moreover, they can be imported into tools such as Adobe Illustrator to be incorporated with other graphics.

The canvas model

It is important to realise that base graphics in R uses a *"canvas model"* to create graphics. We can only overlay extra information on-top of an exising plot and cannot "undo" what is already drawn.

Let's suppose we want to visualise and life expectancy and population of countries in Europe and Africa. First, create two datasets to represent European and African countries in the year 2002

```
euroData <- gapminder[gapminder$continent == "Europe" & gapminder$year == 2002,]
dim(euroData)
```

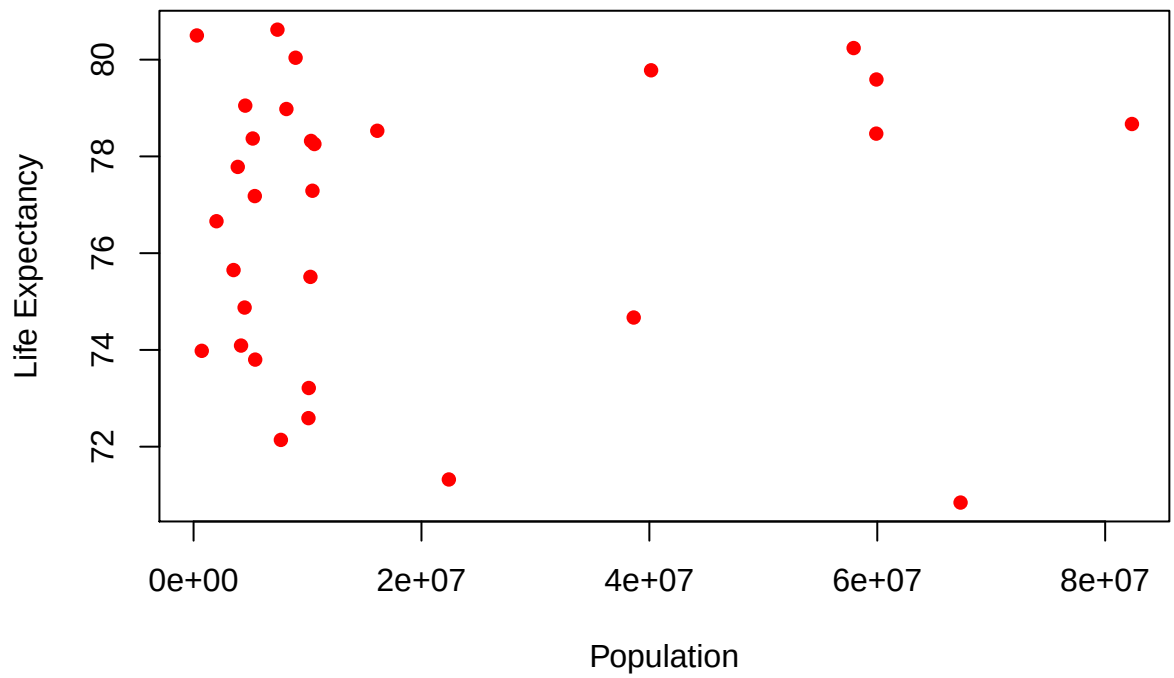
```
## [1] 30  6
```

```r
afrData <- gapminder[gapminder$continent == "Africa" & gapminder$year == 2002,]
dim(afrData)
```

```
## [1] 52  6
```

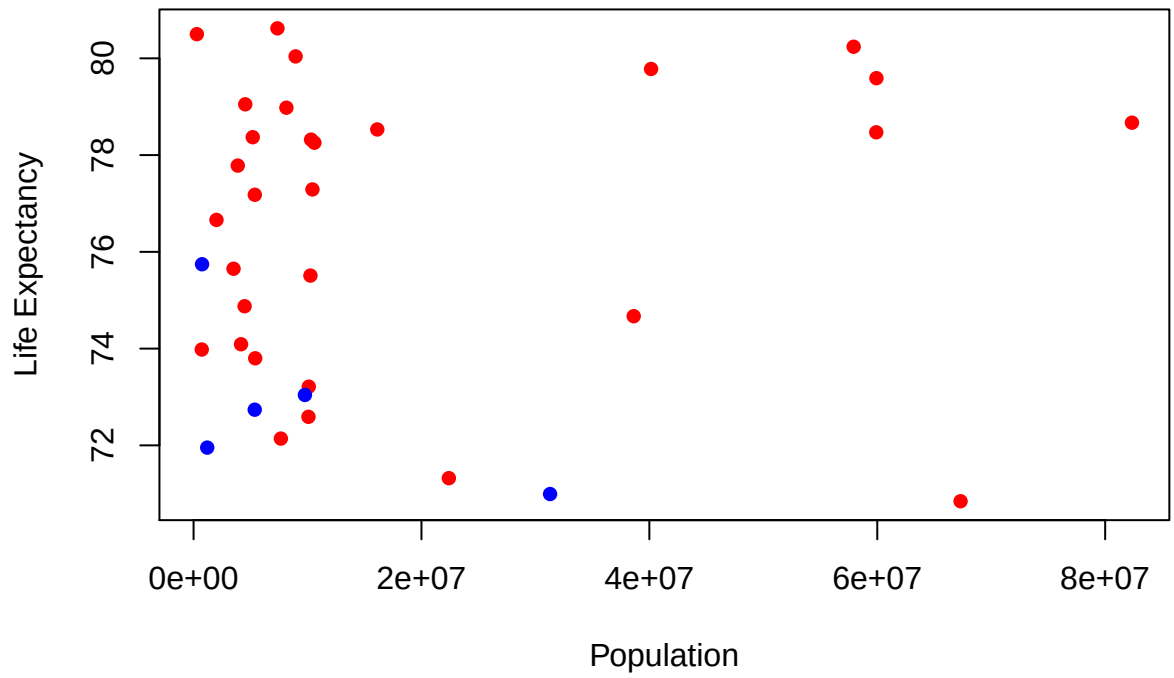
We can start by plotting the life expectancy of the European countries as red dots.

```r
plot(euroData$pop, euroData$lifeExp,col="red",
     pch=16,
     xlab="Population",
     ylab="Life Expectancy")
```



The `points` function can be used put extra points corresponding to African countries on the existing plot.

```r
points(afrData$pop, afrData$lifeExp,col="blue",pch=16)
```



Wait, how many African countries did we have?

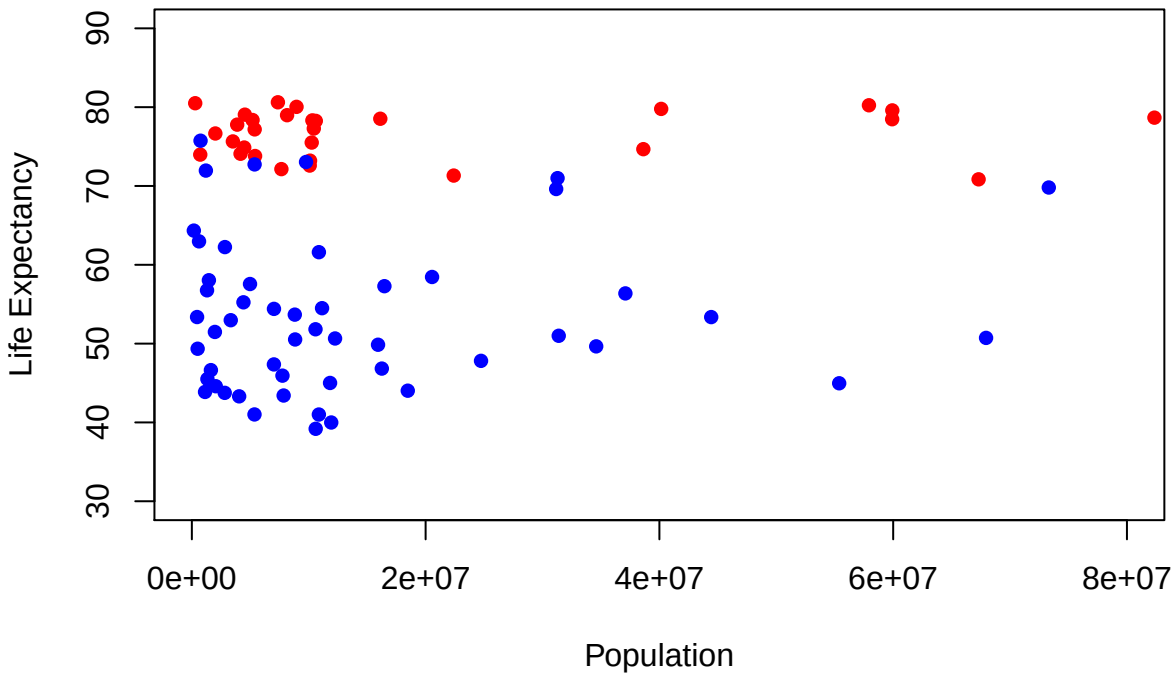
```
nrow(afrData)
```

```
## [1] 52
```

The problem here is that the initial limits of the y axis were defined using the life expectancy range of the European data. We can only add points to the existing plotting window, so any African countries with life expectancy outside this range will not get displayed.

We can define the axes when we create the plot using `xlim` and `ylim`.

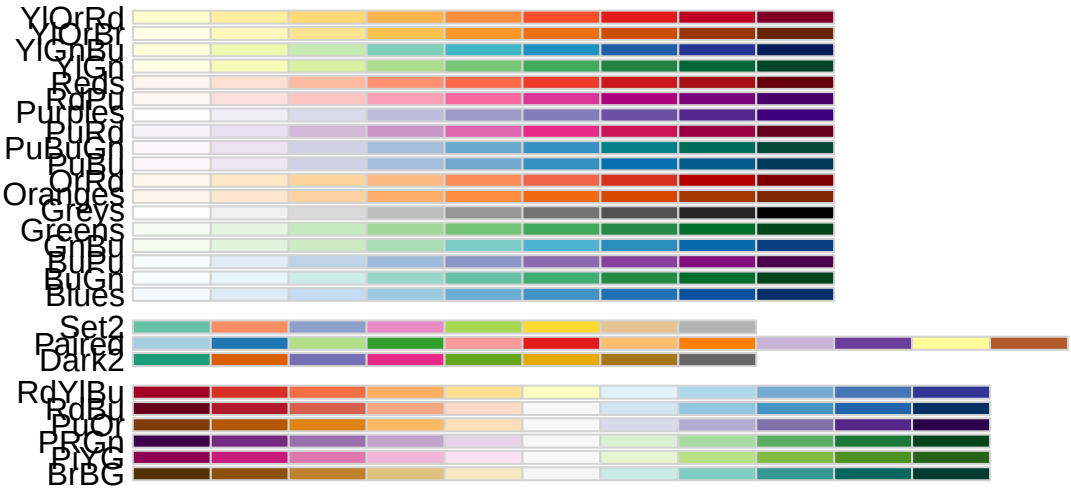
```
plot(euroData$pop, euroData$lifeExp,col="red",
     pch=16,
     xlab="Population",
     ylab="Life Expectancy",
     xlim=c(0,8e7),ylim=c(30,90))
points(afrData$pop, afrData$lifeExp,col="blue",pch=16)
```



Other useful functions for adding features to an existing plot include `text`, `abline`, `grid`, `legend` among others

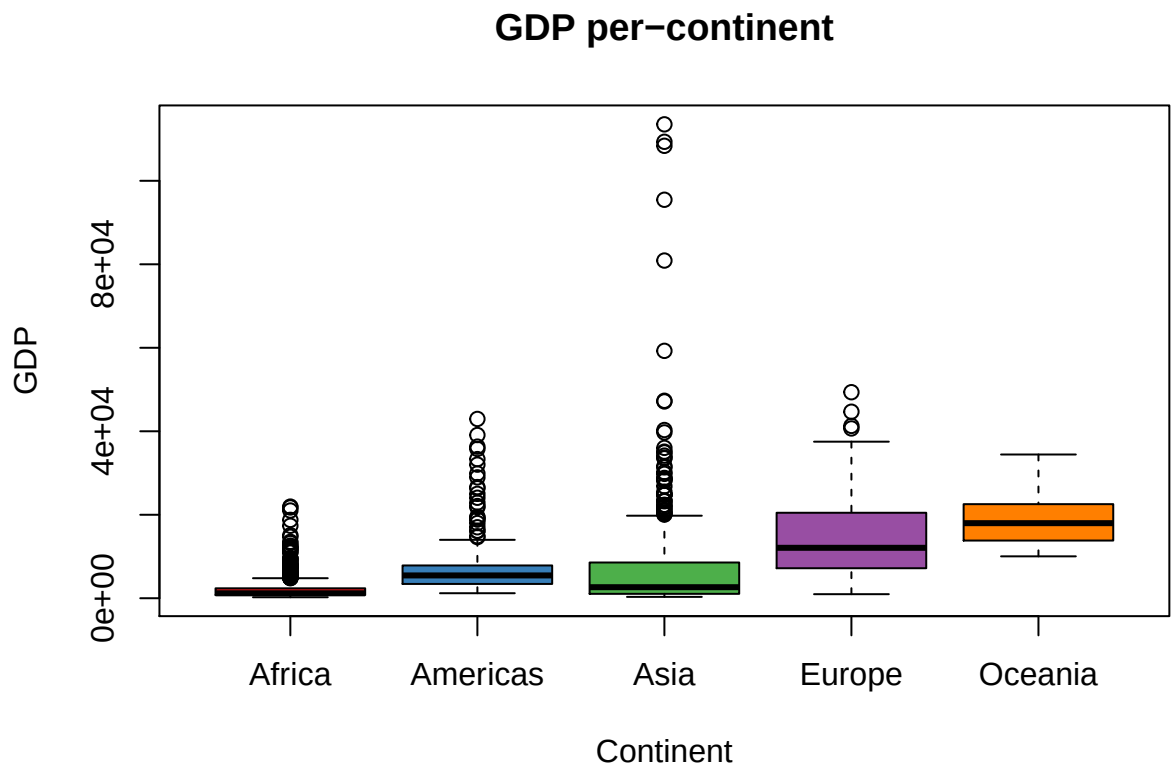
Another useful trick for plotting is to take advantage of pre-existing colour palettes in R. The `RColorBrewer` package is a useful package for such palettes; many of which are friendly to those with visual impairments.

```
library(RColorBrewer)
display.brewer.all(colorblindFriendly = TRUE)
```



The `brewer.pal` function can return the names of `n` colours from one of the pre-defined palettes to be used as a `col` argument to a plotting function.

```
boxplot(gapminder$gdpPercap ~ gapminder$continent,col=brewer.pal(5,"Set1"),
        main="GDP per-continent",
        xlab="Continent",
        ylab="GDP")
```



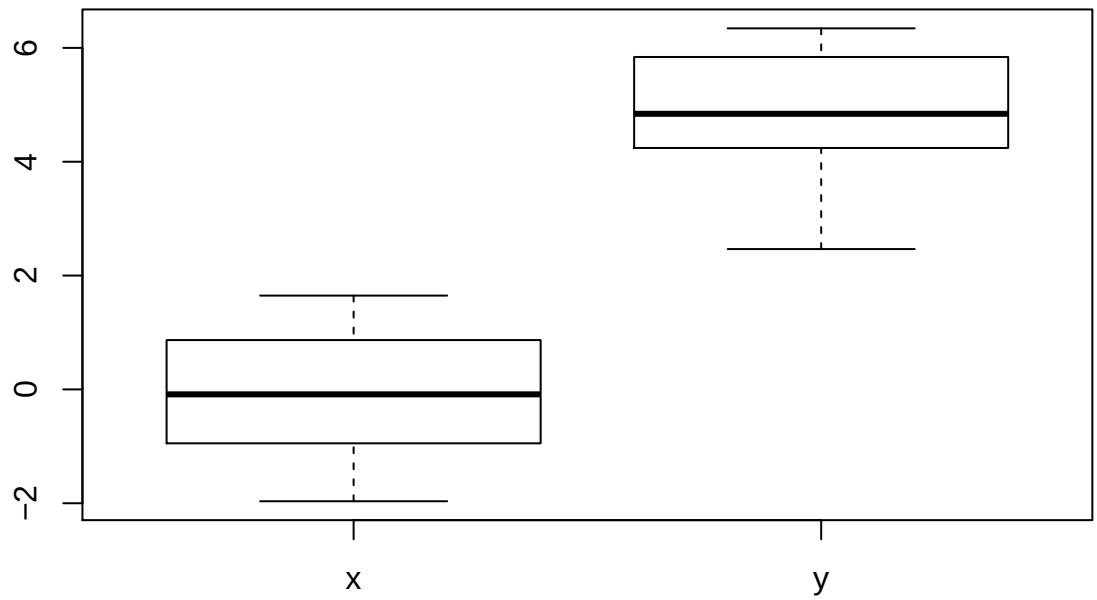
Statistical Testing

We can't really have a run-through of the R language without at least *mentioning* statistics! However, like plotting it is a vast field. The main challenges are putting your data in the correct format (which we have covered here), and deciding which test to use (**which R will not advise you on!**)

- If you have some background in statistics you can see this course from the Babraham Institute Bioinformatics Core about how to perform statistical testing in R.
- If you need a more basic grounding in which statistical test to use, you can see this course from CRUK Cambridge Institute

The `t.test` function is probably the most fundamental statistical testing function in R, and can be adapted to many different situations. Full details are given in the help page `?t.test`. Lets consider we have two vectors of normally-distributed data that we can visualise using a boxplot.

```
x <- rnorm(20)
y <- rnorm(20, 5,1)
df <- data.frame(x,y)
boxplot(df)
```



The output from `t.test` can be used to judge if there is a statistically-significant difference in means:-

```
t.test(x,y)
```

```
##
##  Welch Two Sample t-test
##
## data:  x and y
## t = -14.644, df = 37.916, p-value < 2.2e-16
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -5.648556 -4.276400
## sample estimates:
##  mean of x  mean of y
## -0.1040468  4.8584312
```

If our data were paired we could set the argument `paired=TRUE` to use a different flavour of the test

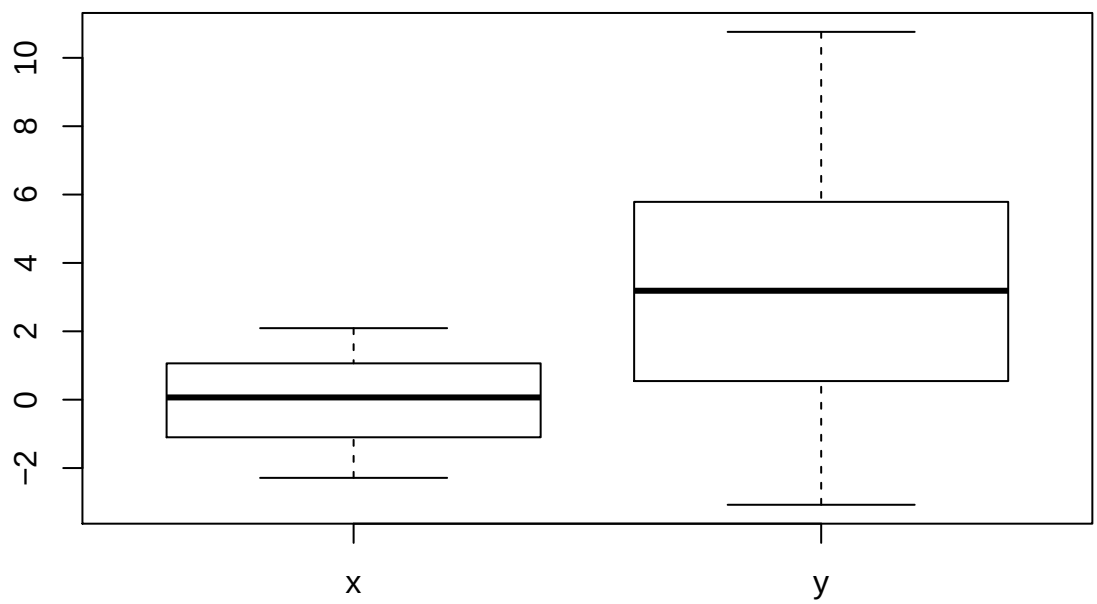
```
t.test(x,y,paired = TRUE)
```

```
##
##  Paired t-test
##
## data:  x and y
## t = -14.505, df = 19, p-value = 9.913e-12
```

```
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -5.678529 -4.246427
## sample estimates:
## mean of the differences
##               -4.962478
```

Similarly, if our data have different variances we can adjust the test accordingly:-

```r
x <- rnorm(20)
y <- rnorm(20, 5,4)
df <- data.frame(x,y)
boxplot(df)
```




```r
t.test(x,y,var.equal = FALSE)
```

```
##
##  Welch Two Sample t-test
##
## data:  x and y
## t = -3.7609, df = 23.609, p-value = 0.0009824
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -5.067749 -1.474437
## sample estimates:
```

```
##  mean of x  mean of y
## 0.04085385 3.31194675
```

Were our data not normally-distributed we could use `wilcox.test`, for example. Fortunately, most statistical tests can be accessed in a similar manner, so it is easy to switch between using different tests provided your data are in the correct format. To re-iterate, the skill is in choosing which test is appropriate.

Towards Reproducibility

Let's say we want an analysis comprising the following steps:-

- read the gapminder data into R
- select countries in Europe
- show the relationship between gdp and life expectancy
- compute the correlation coefficient
  - (the `cor` function can be used)

Having an R script to do the analysis is fine, but what we really want is a document with the code we used *and* the results it generates. This is where a reporting framework such as ***markdown*** is critical

Let's create a new Markdown file with **File -> New File -> R markdown** and look at the contents.

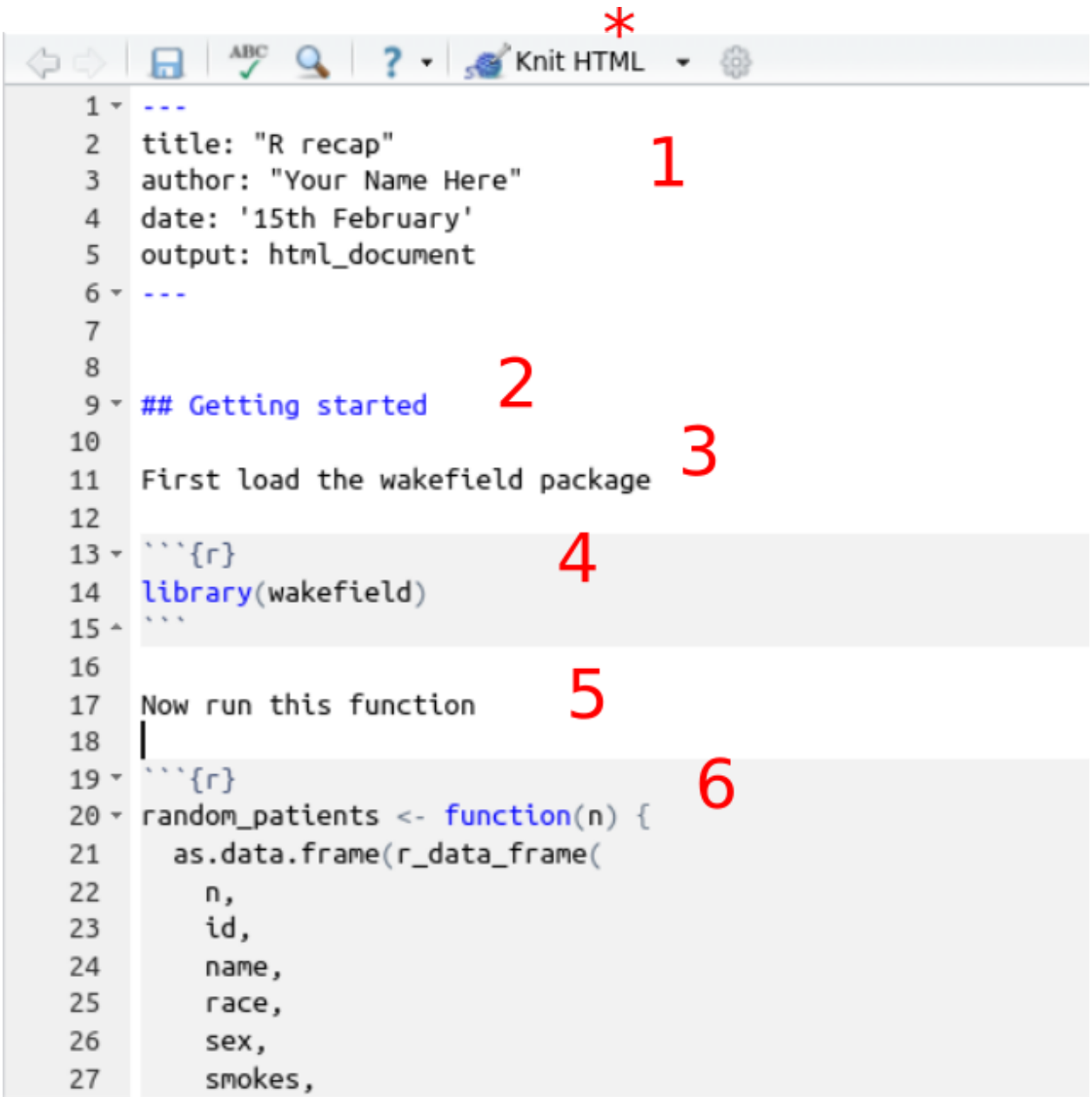
The markdown file can be used as a template to generated PDF, HTML, or even Word documents. The clever bit is that all R code in the template can be execute and the results displayed (tables, graphics etc) along with the code. The compiled document can be passed to your collaborators and they should be able to generate the same results. Alternatively, you can choose to hide the code if your PI just wants to see the results, and not neccesarily what packages, parameters you used. Long-term R users may have heard of *Sweave*. Markdown is the same concept, but an easier to write (and read) syntax

Markdown can also generate presentations and courses. Indeed, all the materials for this course were written in markdown.

1. Header information
2. Section heading
3. Plain text
4. R code to be run
5. Plain text
6. R code to be run

Each line of R code can be executed in the R console by placing the cursor on the line and pressing `CTRL + ENTER`. You can also highlight multiple lines of code. NB. You do not need to highlight to the backtick ("`) symbols.

Hitting the ***Knit*** button (*) will run all R code in order and (providing there are no errors!) you will get a PDF or HTML document. The resultant document will contain all the plain text you wrote, the R code, and any outputs (including graphs, tables etc) that R produced. You can then distribute this document to have a reproducible account of your analysis.


*
Knit HTML
1
2
3
4
5
6
---
title: "R recap"
author: "Your Name Here"
date: '15th February'
output: html_document

Getting started
First load the wakefield package
```{r}
library(wakefield)
```
Now run this function
```{r}
random_patients <- function(n) {
as.data.frame(r_data_frame(
n,
id,
name,
race,
sex,
smokes,

Figure 7:
```

Exercise

- Open the file `gapminder-analysis.Rmd`
- Fill the gaps to complete the analysis described above
- Make a note of the correlation coefficient in the report
- Knit into a pdf or HTML document

_____

_____

_____

You should notice that there is a final code chunk that runs the command `sessionInfo()`. This will print details of the package names and version numbers that were used in the analysis. This is useful to have for housekeeping purposes, but some people you share the report with might not necesarily care about this information. In markdown, we can stop particular *code chunks* being run or printed to the final report by putting the arguments `eval=FALSE` or `results='hide'` at the start of the chunk.

Another useful feature of a markdown report is to embed R code within the plain text section. For example, rather than hard-coding the correlation coefficient, we could have the R code print this result.