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
Data Analysts Captivated by R's Power
SCOPING:
function (x)
function (y)
x + y
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

Submit Your

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

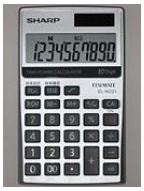


Figure 5:

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.

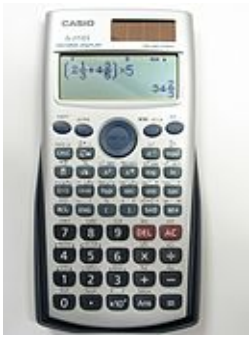


Figure 6:

```
pi
```

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

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

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

```
cos(pi)
```

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

```
tan(2)
```

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

```
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.

```
x <- 10
x
```

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

```
myNumber <- 25
myNumber
```

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

We also can perform arithmetic on variables using functions:

```
sqrt(myNumber)
```

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

We can add variables together:

```
x + myNumber
```

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

We can change the value of an existing variable:

```
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 *c*amel*C*aps. 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

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

```
##  [1]  1.27539694  1.71887371  0.39423972  0.23886899 -0.94258542
##  [6] -0.10256703 -0.69723258  1.01811983 -1.21167694 -0.07007651
```

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*)

```
?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.

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

```
##  [1]  0.9601574  1.5756687  4.1956100  1.6459811  6.7025385  2.3626814
##  [7] -1.0251317  2.3349803  1.3718703  5.8146413
```

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

```
##  [1]  5.5625346 -7.2953643 -1.7354863  0.5143433  5.2175571  3.7066023
##  [7] -0.3289767  7.7021639  5.1199219  2.3097185
```

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
```

```
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.

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

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

```
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.

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

The quote marks are crucial. Why?

```
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`

```
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.

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

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

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

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

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

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

```
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.*

```
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.

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

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

```
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:-

```
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.

```
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
```

```
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

```
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*.

```
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

```
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).

```
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 `,`

```
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
```

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

```
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  66.550  4090500  6424.5191
## 518      Finland    Europe 1957  67.490  4324000  7545.4154
## 519      Finland    Europe 1962  68.750  4491443  9371.8426
## 520      Finland    Europe 1967  69.830  4605744 10921.6363
## 521      Finland    Europe 1972  70.870  4639657 14358.8759
## 522      Finland    Europe 1977  72.520  4738902 15605.4228
## 523      Finland    Europe 1982  74.550  4826933 18533.1576
## 524      Finland    Europe 1987  74.830  4931729 21141.0122
## 525      Finland    Europe 1992  75.700  5041039 20647.1650
## 526      Finland    Europe 1997  77.130  5134406 23723.9502
## 527      Finland    Europe 2002  78.370  5193039 28204.5906
## 528      Finland    Europe 2007  79.313  5238460 33207.0844
## 685      Iceland    Europe 1952  72.490   147962  7267.6884
## 686      Iceland    Europe 1957  73.470   165110  9244.0014
## 687      Iceland    Europe 1962  73.680   182053 10350.1591
## 688      Iceland    Europe 1967  73.730   198676 13319.8957
## 689      Iceland    Europe 1972  74.460   209275 15798.0636
## 690      Iceland    Europe 1977  76.110   221823 19654.9625
## 691      Iceland    Europe 1982  76.990   233997 23269.6075
## 692      Iceland    Europe 1987  77.230   244676 26923.2063
## 693      Iceland    Europe 1992  78.770   259012 25144.3920
## 694      Iceland    Europe 1997  78.950   271192 28061.0997
## 695      Iceland    Europe 2002  80.500   288030 31163.2020
## 696      Iceland    Europe 2007  81.757   301931 36180.7892
## 745      Ireland    Europe 1952  66.910  2952156  5210.2803
## 746      Ireland    Europe 1957  68.900  2878220  5599.0779
## 747      Ireland    Europe 1962  70.290  2830000  6631.5973
## 748      Ireland    Europe 1967  71.080  2900100  7655.5690
## 749      Ireland    Europe 1972  71.280  3024400  9530.7729
## 750      Ireland    Europe 1977  72.030  3271900 11150.9811
## 751      Ireland    Europe 1982  73.100  3480000 12618.3214
## 752      Ireland    Europe 1987  74.360  3539900 13872.8665
## 753      Ireland    Europe 1992  75.467  3557761 17558.8155
## 754      Ireland    Europe 1997  76.122  3667233 24521.9471
## 755      Ireland    Europe 2002  77.783  3879155 34077.0494
## 756      Ireland    Europe 2007  78.885  4109086 40675.9964
## 1081 Netherlands    Europe 1952  72.130 10381988  8941.5719
## 1082 Netherlands    Europe 1957  72.990 11026383 11276.1934
## 1083 Netherlands    Europe 1962  73.230 11805689 12790.8496
## 1084 Netherlands    Europe 1967  73.820 12596822 15363.2514
## 1085 Netherlands    Europe 1972  73.750 13329874 18794.7457
## 1086 Netherlands    Europe 1977  75.240 13852989 21209.0592
## 1087 Netherlands    Europe 1982  76.050 14310401 21399.4605
## 1088 Netherlands    Europe 1987  76.830 14665278 23651.3236
## 1089 Netherlands    Europe 1992  77.420 15174244 26790.9496
## 1090 Netherlands    Europe 1997  78.030 15604464 30246.1306
## 1091 Netherlands    Europe 2002  78.530 16122830 33724.7578
## 1092 Netherlands    Europe 2007  79.762 16570613 36797.9333
## 1093 New Zealand   Oceania 1952  69.390  1994794 10556.5757
## 1094 New Zealand   Oceania 1957  70.260  2229407 12247.3953
```

```
## 1095 New Zealand   Oceania 1962  71.240  2488550 13175.6780
## 1096 New Zealand   Oceania 1967  71.520  2728150 14463.9189
## 1097 New Zealand   Oceania 1972  71.890  2929100 16046.0373
## 1098 New Zealand   Oceania 1977  72.220  3164900 16233.7177
## 1099 New Zealand   Oceania 1982  73.840  3210650 17632.4104
## 1100 New Zealand   Oceania 1987  74.320  3317166 19007.1913
## 1101 New Zealand   Oceania 1992  76.330  3437674 18363.3249
## 1102 New Zealand   Oceania 1997  77.550  3676187 21050.4138
## 1103 New Zealand   Oceania 2002  79.110  3908037 23189.8014
## 1104 New Zealand   Oceania 2007  80.204  4115771 25185.0091
## 1225      Poland    Europe 1952  61.310 25730551  4029.3297
## 1226      Poland    Europe 1957  65.770 28235346  4734.2530
## 1227      Poland    Europe 1962  67.640 30329617  5338.7521
## 1228      Poland    Europe 1967  69.610 31785378  6557.1528
## 1229      Poland    Europe 1972  70.850 33039545  8006.5070
## 1230      Poland    Europe 1977  70.670 34621254  9508.1415
## 1231      Poland    Europe 1982  71.320 36227381  8451.5310
## 1232      Poland    Europe 1987  70.980 37740710  9082.3512
## 1233      Poland    Europe 1992  70.990 38370697  7738.8812
## 1234      Poland    Europe 1997  72.750 38654957 10159.5837
## 1235      Poland    Europe 2002  74.670 38625976 12002.2391
## 1236      Poland    Europe 2007  75.563 38518241 15389.9247
## 1453   Swaziland    Africa 1952  41.407   290243  1148.3766
## 1454   Swaziland    Africa 1957  43.424   326741  1244.7084
## 1455   Swaziland    Africa 1962  44.992   370006  1856.1821
## 1456   Swaziland    Africa 1967  46.633   420690  2613.1017
## 1457   Swaziland    Africa 1972  49.552   480105  3364.8366
## 1458   Swaziland    Africa 1977  52.537   551425  3781.4106
## 1459   Swaziland    Africa 1982  55.561   649901  3895.3840
## 1460   Swaziland    Africa 1987  57.678   779348  3984.8398
## 1461   Swaziland    Africa 1992  58.474   962344  3553.0224
## 1462   Swaziland    Africa 1997  54.289  1054486  3876.7685
## 1463   Swaziland    Africa 2002  43.869  1130269  4128.1169
## 1464   Swaziland    Africa 2007  39.613  1133066  4513.4806
## 1477 Switzerland    Europe 1952  69.620  4815000 14734.2327
## 1478 Switzerland    Europe 1957  70.560  5126000 17909.4897
## 1479 Switzerland    Europe 1962  71.320  5666000 20431.0927
## 1480 Switzerland    Europe 1967  72.770  6063000 22966.1443
## 1481 Switzerland    Europe 1972  73.780  6401400 27195.1130
## 1482 Switzerland    Europe 1977  75.390  6316424 26982.2905
## 1483 Switzerland    Europe 1982  76.210  6468126 28397.7151
## 1484 Switzerland    Europe 1987  77.410  6649942 30281.7046
## 1485 Switzerland    Europe 1992  78.030  6995447 31871.5303
## 1486 Switzerland    Europe 1997  79.370  7193761 32135.3230
## 1487 Switzerland    Europe 2002  80.620  7361757 34480.9577
## 1488 Switzerland    Europe 2007  81.701  7554661 37506.4191
## 1525    Thailand      Asia 1952  50.848 21289402   757.7974
## 1526    Thailand      Asia 1957  53.630 25041917   793.5774
## 1527    Thailand      Asia 1962  56.061 29263397  1002.1992
## 1528    Thailand      Asia 1967  58.285 34024249  1295.4607
## 1529    Thailand      Asia 1972  60.405 39276153  1524.3589
## 1530    Thailand      Asia 1977  62.494 44148285  1961.2246
## 1531    Thailand      Asia 1982  64.597 48827160  2393.2198
## 1532    Thailand      Asia 1987  66.084 52910342  2982.6538
```

```
## 1533    Thailand      Asia 1992  67.298 56667095  4616.8965
## 1534    Thailand      Asia 1997  67.521 60216677  5852.6255
## 1535    Thailand      Asia 2002  68.564 62806748  5913.1875
## 1536    Thailand      Asia 2007  70.616 65068149  7458.3963
```

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",]
```

```
##       country continent year lifeExp      pop gdpPercap
## 1681   Zambia    Africa 1952  42.038  2672000 1147.3888
## 1682   Zambia    Africa 1957  44.077  3016000 1311.9568
## 1683   Zambia    Africa 1962  46.023  3421000 1452.7258
## 1684   Zambia    Africa 1967  47.768  3900000 1777.0773
## 1685   Zambia    Africa 1972  50.107  4506497 1773.4983
## 1686   Zambia    Africa 1977  51.386  5216550 1588.6883
## 1687   Zambia    Africa 1982  51.821  6100407 1408.6786
## 1688   Zambia    Africa 1987  50.821  7272406 1213.3151
## 1689   Zambia    Africa 1992  46.100  8381163 1210.8846
## 1690   Zambia    Africa 1997  40.238  9417789 1071.3538
## 1691   Zambia    Africa 2002  39.193 10595811 1071.6139
## 1692   Zambia    Africa 2007  42.384 11746035 1271.2116
## 1693 Zimbabwe    Africa 1952  48.451  3080907  406.8841
## 1694 Zimbabwe    Africa 1957  50.469  3646340  518.7643
## 1695 Zimbabwe    Africa 1962  52.358  4277736  527.2722
## 1696 Zimbabwe    Africa 1967  53.995  4995432  569.7951
## 1697 Zimbabwe    Africa 1972  55.635  5861135  799.3622
## 1698 Zimbabwe    Africa 1977  57.674  6642107  685.5877
## 1699 Zimbabwe    Africa 1982  60.363  7636524  788.8550
## 1700 Zimbabwe    Africa 1987  62.351  9216418  706.1573
## 1701 Zimbabwe    Africa 1992  60.377 10704340  693.4208
## 1702 Zimbabwe    Africa 1997  46.809 11404948  792.4500
## 1703 Zimbabwe    Africa 2002  39.989 11926563  672.0386
## 1704 Zimbabwe    Africa 2007  43.487 12311143  469.7093
```

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

```
##       country continent year lifeExp      pop gdpPercap
## 1681   Zambia    Africa 1952  42.038  2672000 1147.3888
## 1682   Zambia    Africa 1957  44.077  3016000 1311.9568
## 1683   Zambia    Africa 1962  46.023  3421000 1452.7258
## 1684   Zambia    Africa 1967  47.768  3900000 1777.0773
## 1685   Zambia    Africa 1972  50.107  4506497 1773.4983
## 1686   Zambia    Africa 1977  51.386  5216550 1588.6883
## 1687   Zambia    Africa 1982  51.821  6100407 1408.6786
## 1688   Zambia    Africa 1987  50.821  7272406 1213.3151
## 1689   Zambia    Africa 1992  46.100  8381163 1210.8846
## 1690   Zambia    Africa 1997  40.238  9417789 1071.3538
## 1691   Zambia    Africa 2002  39.193 10595811 1071.6139
```

```
## 1692   Zambia    Africa 2007  42.384 11746035 1271.2116
## 1693 Zimbabwe    Africa 1952  48.451  3080907  406.8841
## 1694 Zimbabwe    Africa 1957  50.469  3646340  518.7643
## 1695 Zimbabwe    Africa 1962  52.358  4277736  527.2722
## 1696 Zimbabwe    Africa 1967  53.995  4995432  569.7951
## 1697 Zimbabwe    Africa 1972  55.635  5861135  799.3622
## 1698 Zimbabwe    Africa 1977  57.674  6642107  685.5877
## 1699 Zimbabwe    Africa 1982  60.363  7636524  788.8550
## 1700 Zimbabwe    Africa 1987  62.351  9216418  706.1573
## 1701 Zimbabwe    Africa 1992  60.377 10704340  693.4208
## 1702 Zimbabwe    Africa 1997  46.809 11404948  792.4500
## 1703 Zimbabwe    Africa 2002  39.989 11926563  672.0386
## 1704 Zimbabwe    Africa 2007  43.487 12311143  469.7093
```

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")
```

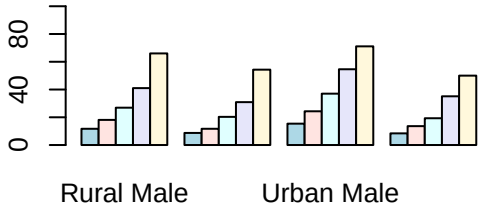
We can even order by more than one condition

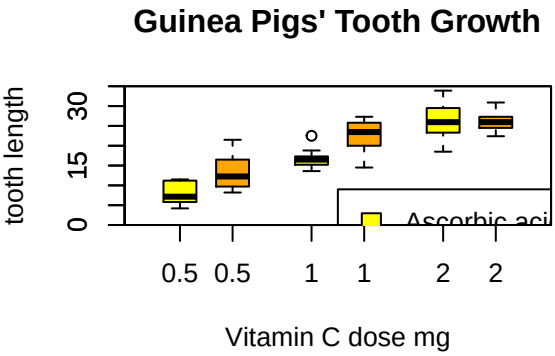
```
gapminder[order(gapminder$year, gapminder$country),]
```

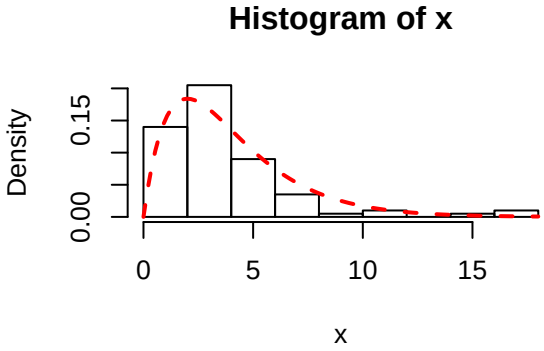
```
##           country continent year lifeExp     pop gdpPercap
## 1     Afghanistan      Asia 1952  28.801 8425333  779.4453
## 553        Gambia    Africa 1952  30.000  284320  485.2307
## 37         Angola    Africa 1952  30.015 4232095 3520.6103
## 1345 Sierra Leone    Africa 1952  30.331 2143249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

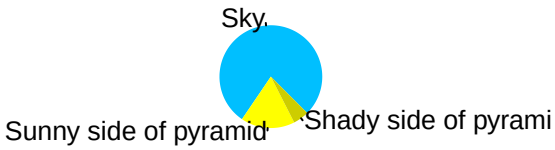
Plotting and stats (in brief!)

All your favourite types of plot can be created in R





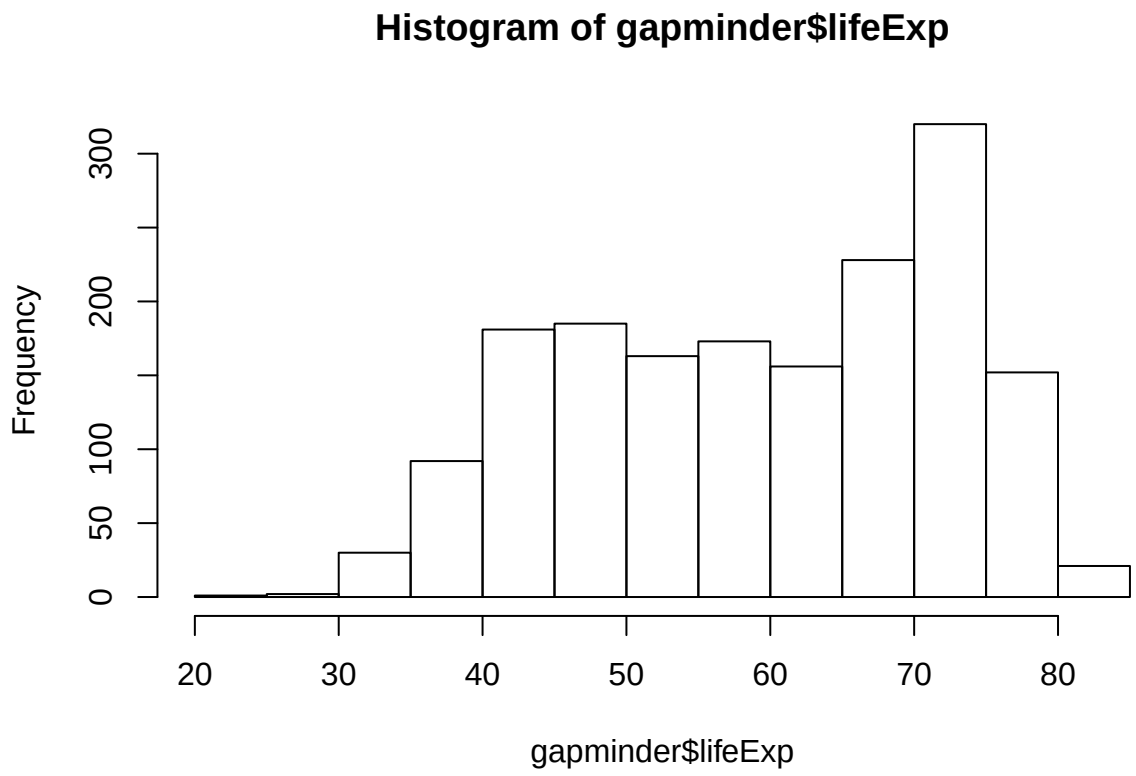




- Simple plots are supported in the *base* distribution of R (what you get automatically when you download R).
  - `boxplot`, `hist`, `barplot`,... all of which are extensions of the basic `plot` function
- Many different customisations are possible
  - colour, overlay points / text, legends, multi-panel 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

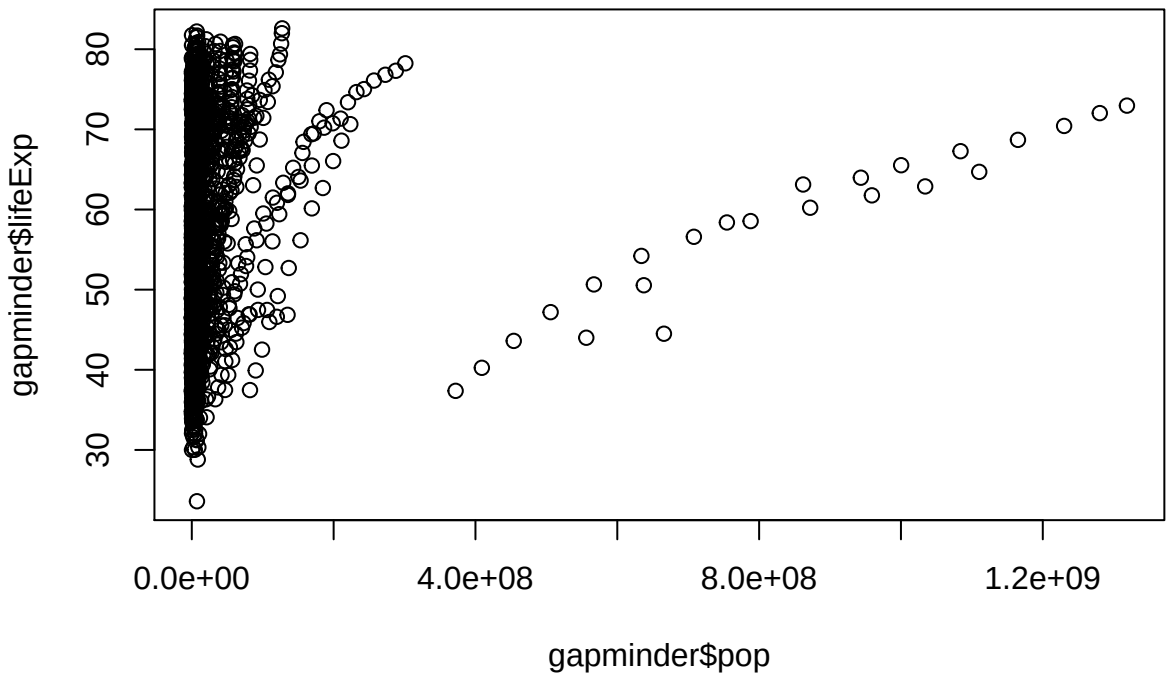
Plots can be constructed from vectors of numeric data, such as the data we get from a particular column in a data frame

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



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

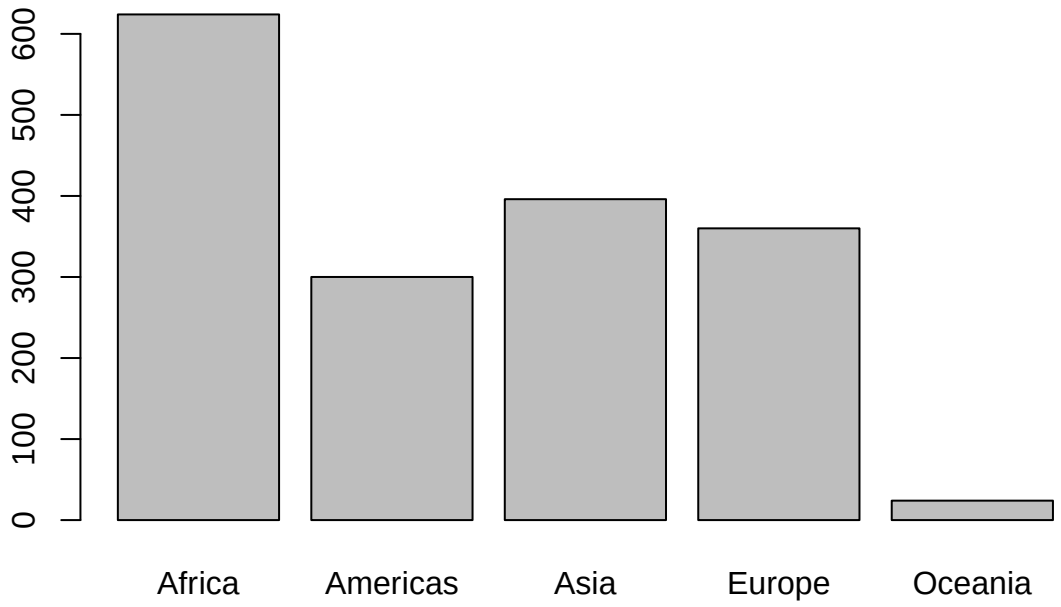
```
plot(gapminder$pop,gapminder$lifeExp)
```



What are those points on the right? We'll find out soon...

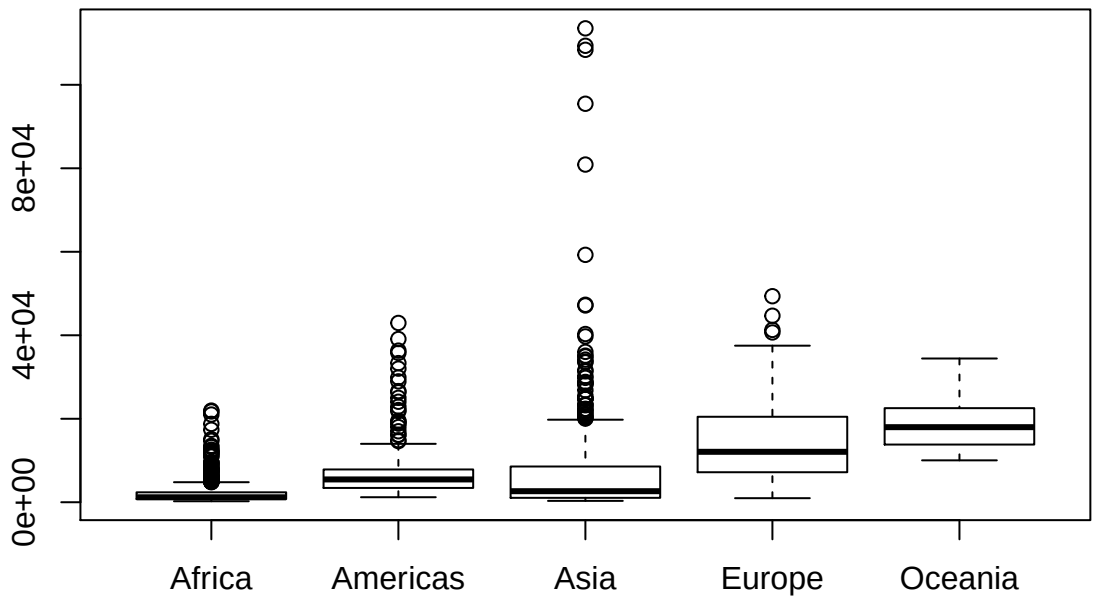
Barplots are commonly-used for counts of categorical data

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



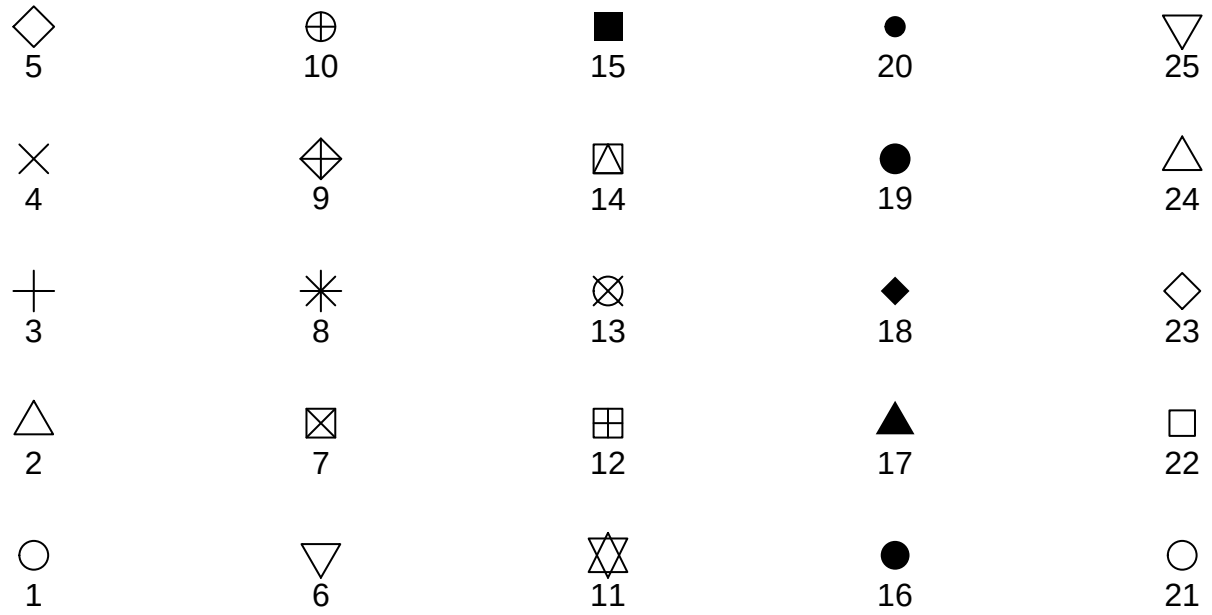
Boxplots are good for visualising and comparing distributions. Here the `~` symbol sets up a formula, the effect of which is to put the categorical variable on the `x` axis and continuous variable on the `y` axis.

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



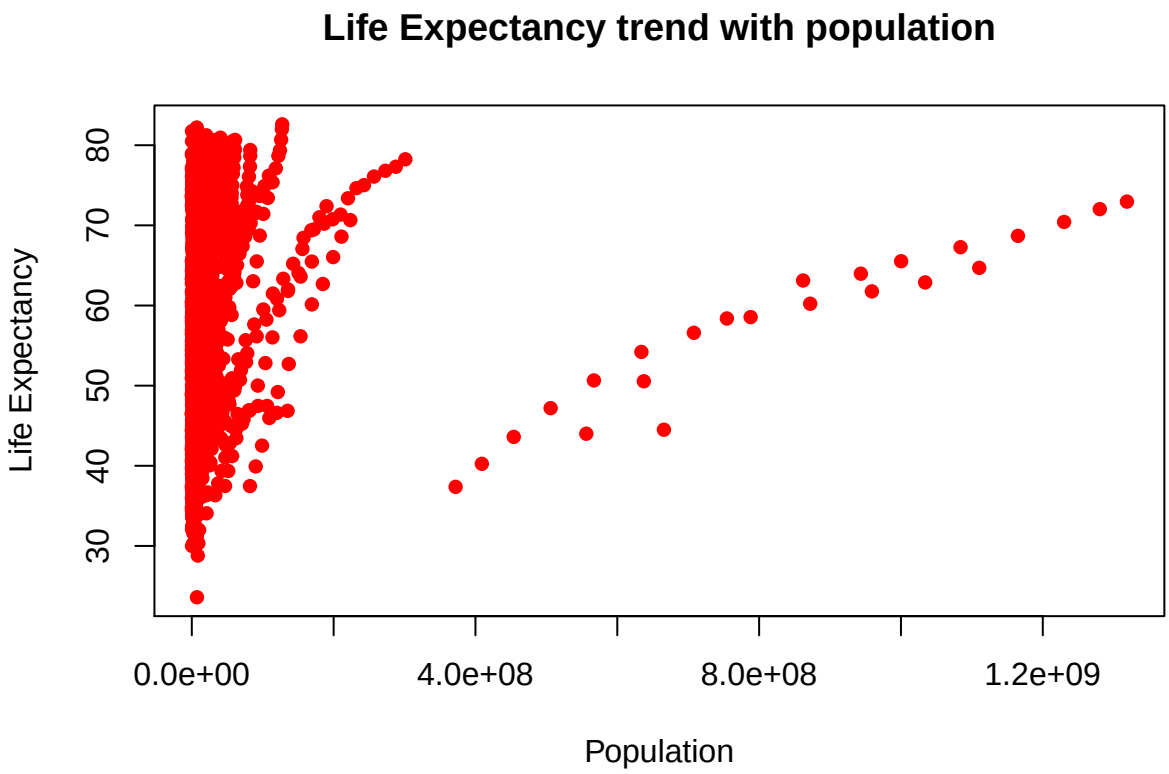
Lots of customisations are possible to enhance the appaerance of our plots. Not for the faint-hearted, the help pages `?plot` and `?par` give the full details. In short,

- 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:-



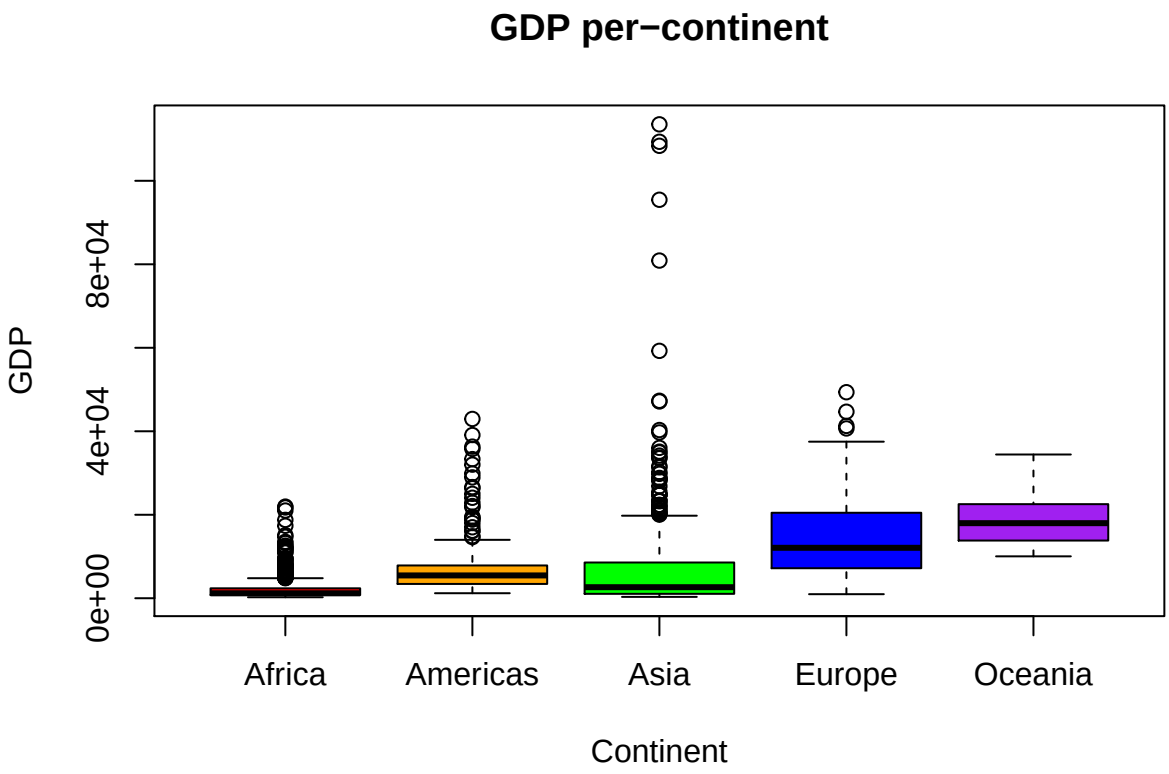
Putting it all together.

```
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)
```

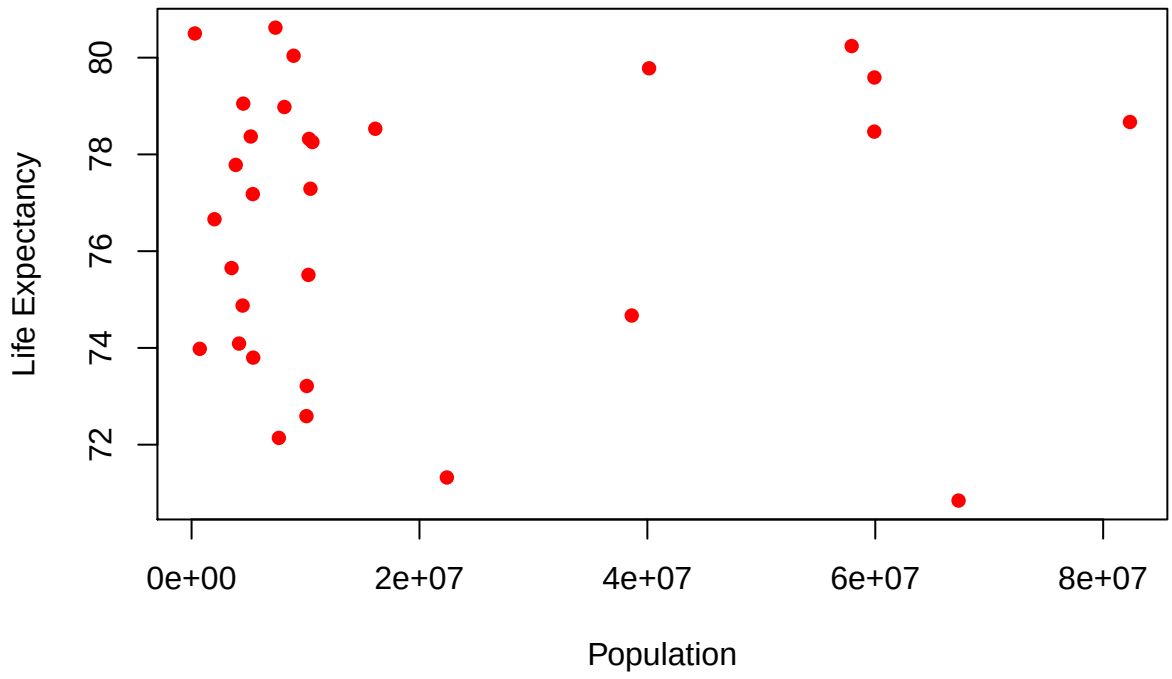
```
## [1] 30  6
```

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

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

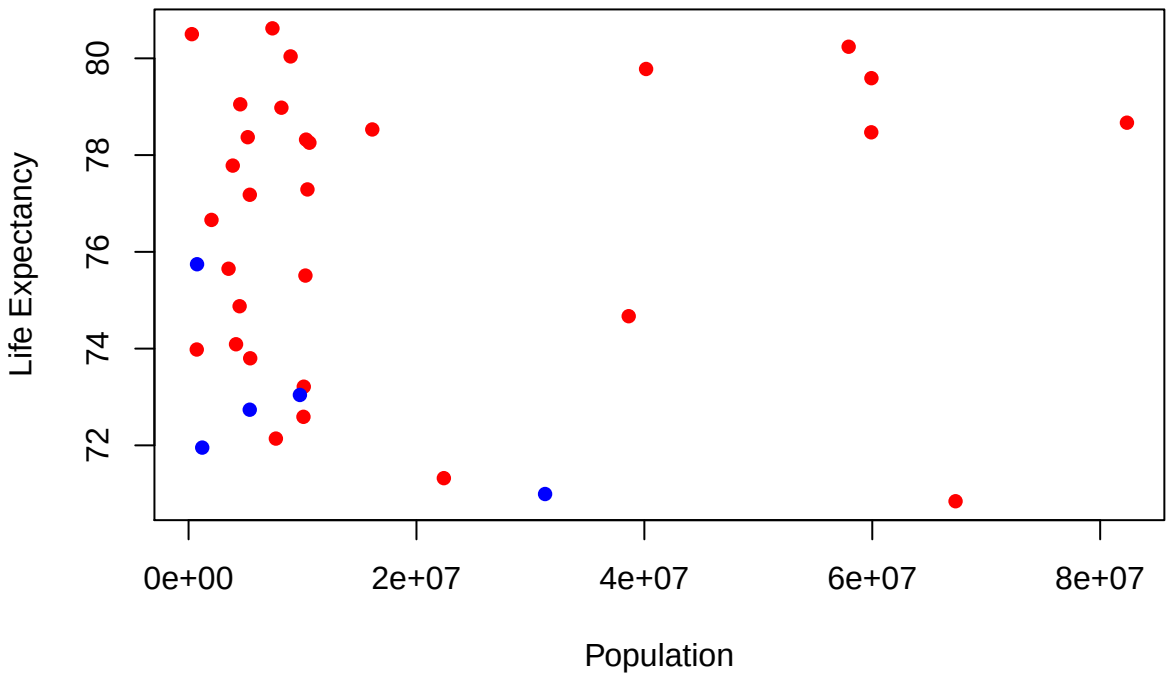
We can start by plotting the life expectancy of the European countries as red dots.

```
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.

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



Wait, how many African countries did we have?

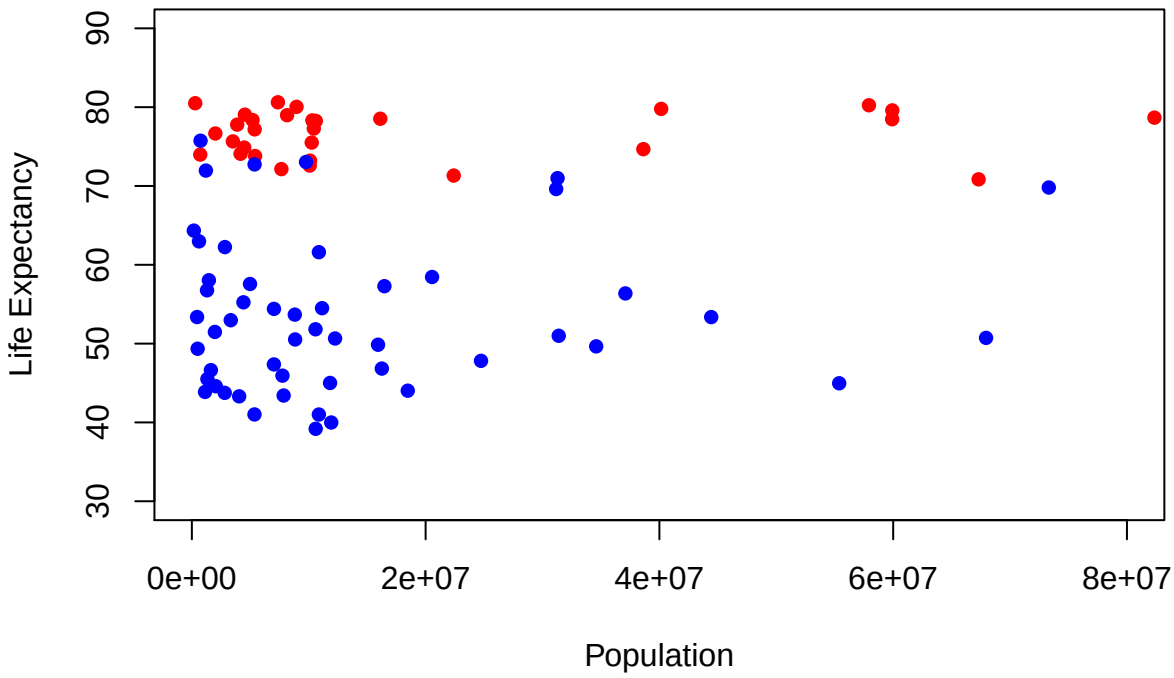
```
nrow(afrData)
```

```
## [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`.

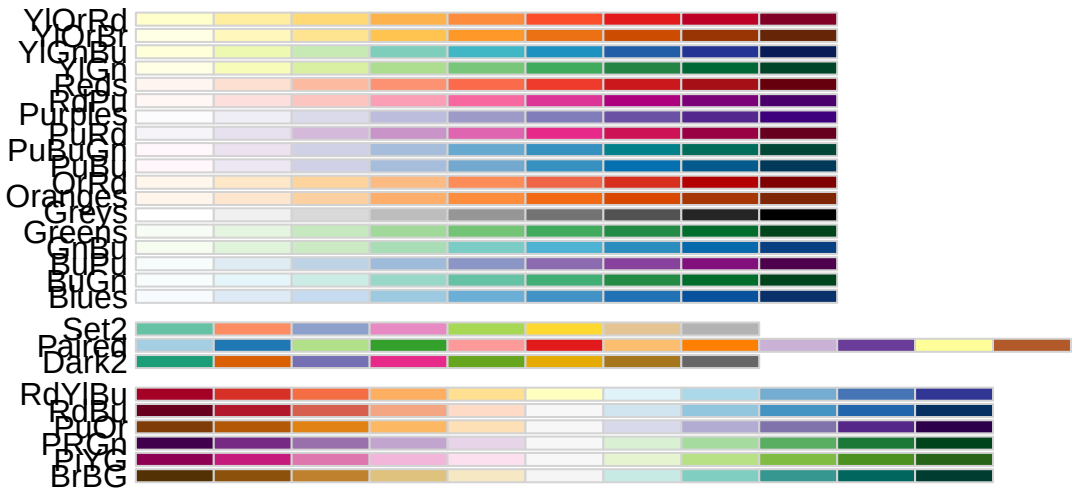
```
plot(euroData$pop, euroData$lifeExp,col="red",
     pch=16,
     xlab="Population",
     ylab="Life Expectancy",
     xlim=c(0,8e7),ylim=c(30,90))
points(afrData$pop, afrData$lifeExp,col="blue",pch=16)
```



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

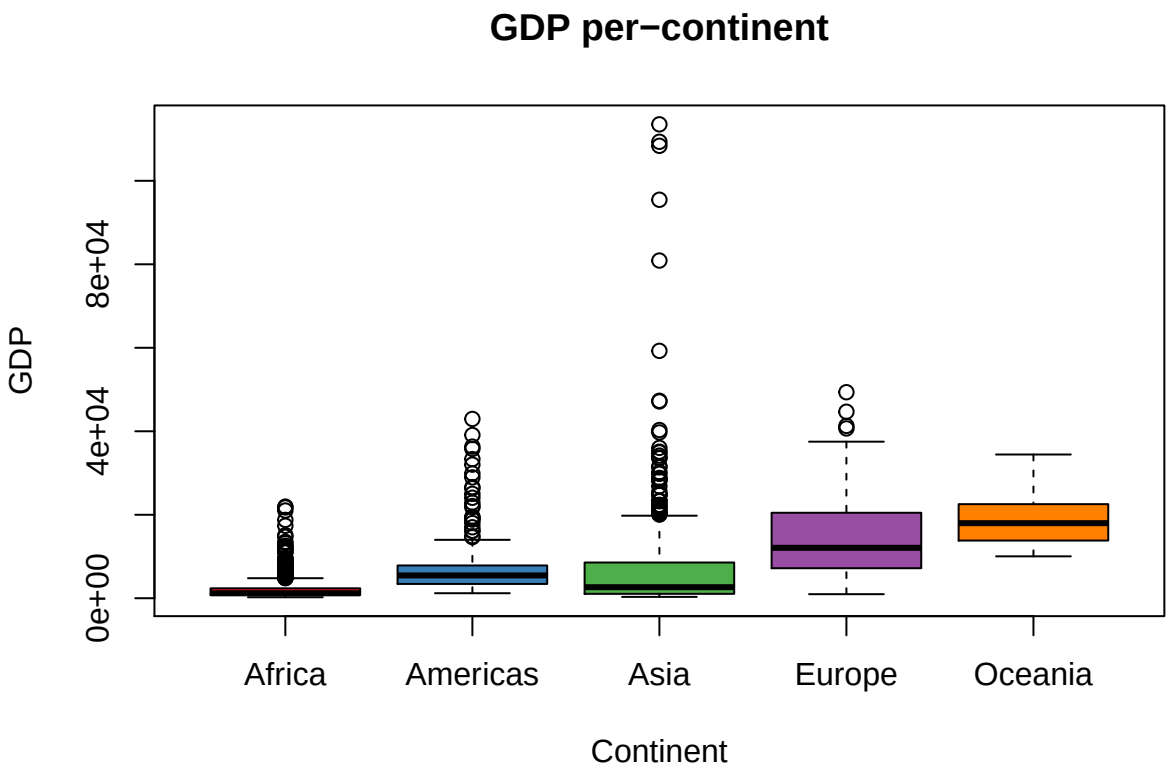
Another useful trick for plotting is to take advantage of pre-existing colour palettes in R. The `RColorBrewer` package is a useful package for such palettes; many of which are friendly to those with visual impairments.

```
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")
```



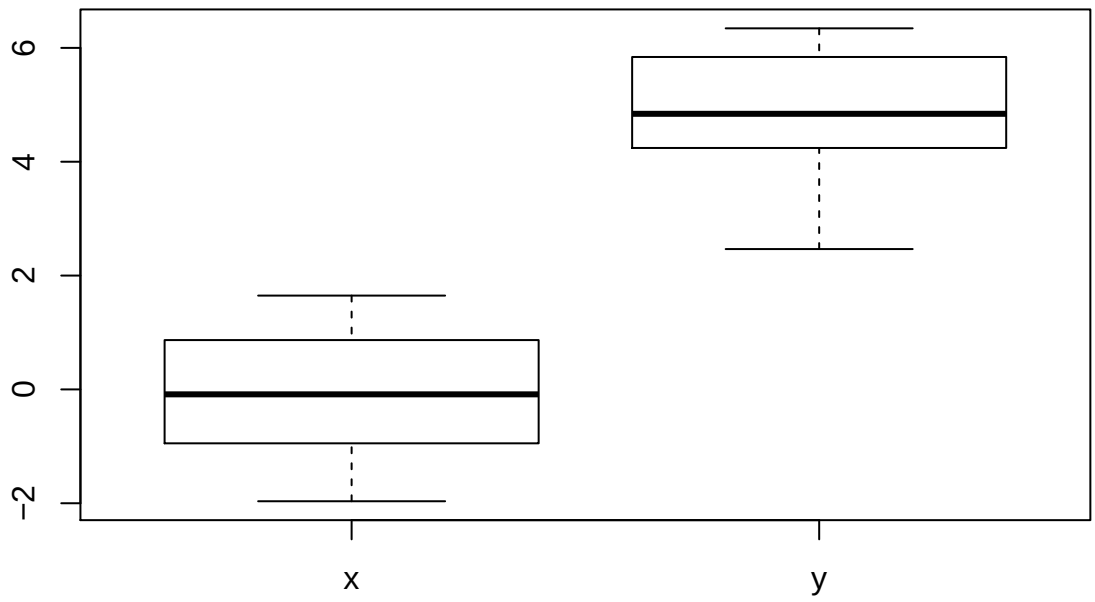
Statistical Testing

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

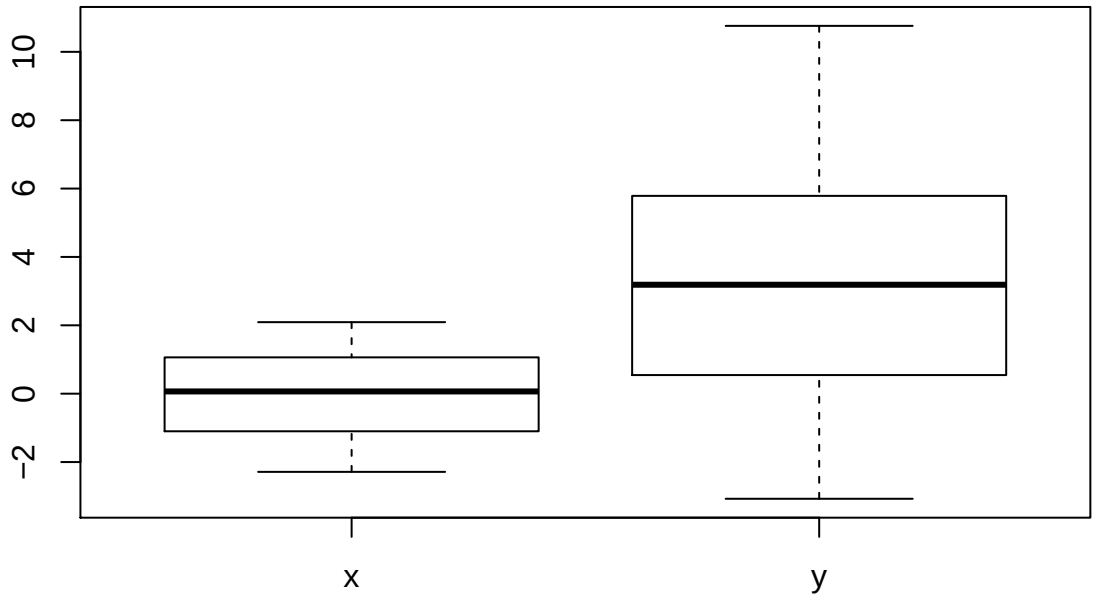
```
t.test(x,y,paired = TRUE)
```

```
##
##  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:-

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




```
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.