Basic Course on R: Introduction and History

Karl Brand* and Elizabeth Ribble†

17-21 December 2018

Contents

*brandk@gmauil.com
†emcclel3@msudenver.edu

1 Course Overview

In the first part of the course we are introduced to the `R` software package, or *environment*, and learn about `R`, how to interact with it and it's basic programming elements. In the second part of the course we'll learn how to use `R` for it's intended function: doing statistics quickly and effectively.

We will cover essential topics including how to use `R` and why we use it; objects, classes and functions; and creating, importing, saving, manipulating, combining, sub-setting and plotting data.

By the end of the first couple of introductory days I expect that you can write your own code and have a reasonable understanding of what's going on when you cut and paste other people's code, which is typical of how people get started with `R` (or any other language for that matter).

Day 1

- Introduction, `R` history, getting familiar with `R` console, RStudio
- Objects, data structures, classes, using functions, arguments
- Containers, names, selection rules, accessing data.frame elements, lists

Day 2

- Entering, importing & manipulating data with `c(), cbind(), rbind(), dim()`; importing from a file; working directory
- Basic plotting of boxplots, bargraphs, scatterplots & line graphs
- Plotting with ggplot2

Day 3

- Hypothesis testing & confidence intervals part 1: summary statistics, t-test, Mann-Whitney U test
- Hypothesis testing & confidence intervals part 2: correlations, ANOVA, chi-squared test
- Distribution-free ANOVA

Day 4

- Basic linear regression, diagnostics and plotting
- Basics of logistic regression
- Programming structures 1: creating your own functions, if/else functions, loops

Day 5

- Programming structures 2: scope, recursion, replacement, search path
- Object-oriented programming and performance enhancement: generic functions and methods, memory optimization

2 Introduction

2.1 History

Before R there was the statistical analysis program, S, developed at Bell Laboratories in 1976 by John Chambers. S was later commercialised as S-PLUS. John Chambers is also a current member of the R core group responsible for developing R.

Ross Ihaka and Robert Gentleman of Dept of Statistics, University of Auckland, New Zealand begin coding (1991) the S clone, R, as an open source alternative for academic use. Together with the 'R core group' they released version 1 under the GNU Public License (GPL) v2 and v3 in 2000. The name, R, probably derives from the first letter of the creator's names. http://cran.r-project.org/doc/FAQ/R-FAQ.html#Why-is-R-named-R_003f

2.2 Strengths and weaknesses

I attribute much of R's success to its GPL v2 license: created by users for users to run well on Linux, Apple and Windows. These licenses give everyone the freedom to use, the freedom to distribute and the freedom to contribute their own intellectual input to the project. The result is that performance and fitness of purpose is and always will be the primary focus. I attribute R's limited general use to its steep learning curve for non-programmers. Base R is also yet to take advantage of the multi-threaded architecture of current computing technology.

2.3 Get it

2.3.1 Install it

To get R on your PC **first** you need to install the core R application. Of course you'll need to download the version for your operating system and architecture first.

For Linux: https://cran.r-project.org/bin/linux/

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

For Windows: https://cran.r-project.org/bin/windows/

Although R for Windows (and Mac?) ships with a basic text editor, and it's possible to execute code from the terminal under linux, a fully featured editor increases convenience

and productivity. Grab an awesome text editor:

RStudio: `https://www.rstudio.com/products/rstudio/`

Emacs/ESS: `http://ess.r-project.org/index.php?Section=download`

Here's a full list available editors: `https://en.wikipedia.org/wiki/R_(programming_language)#Interfaces`

2.4 Not convinced?

r4stats.com has a nice evaluation of R's value 'in the field'. `http://r4stats.com/articles/popularity/`

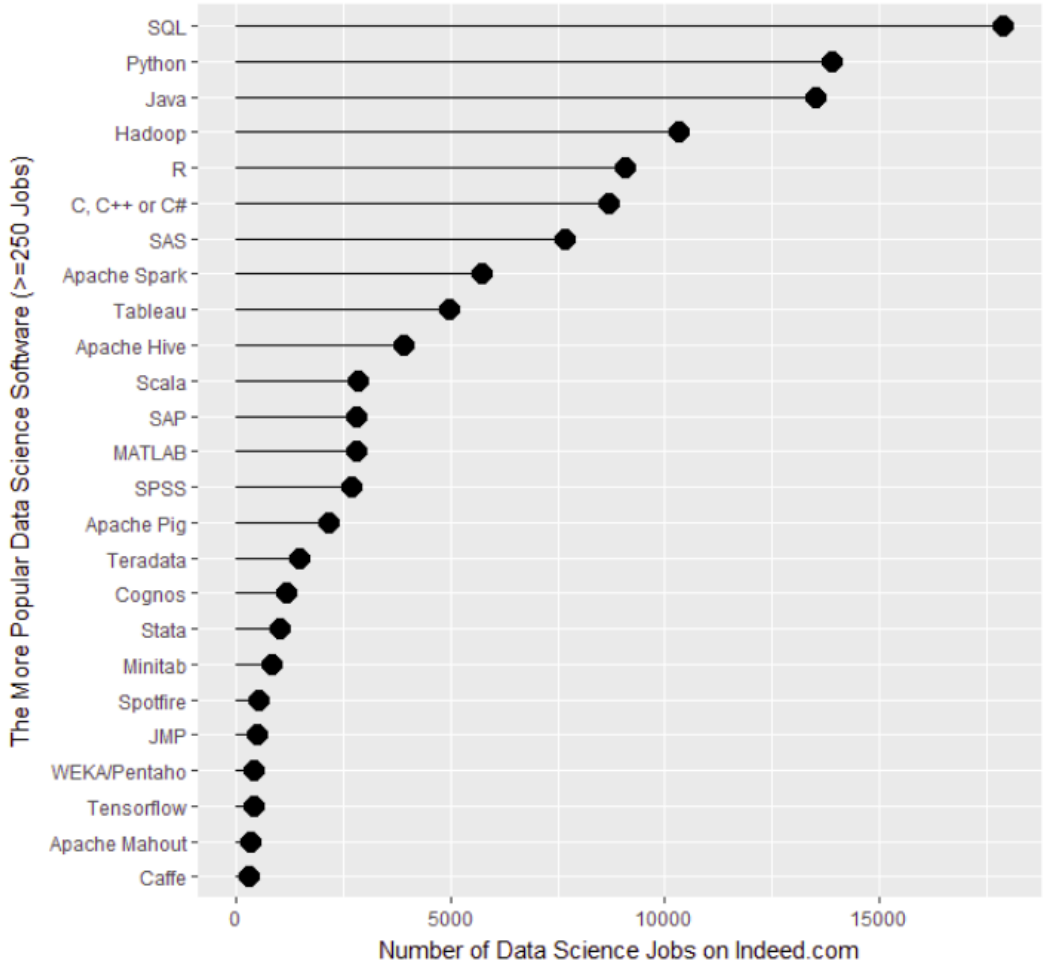
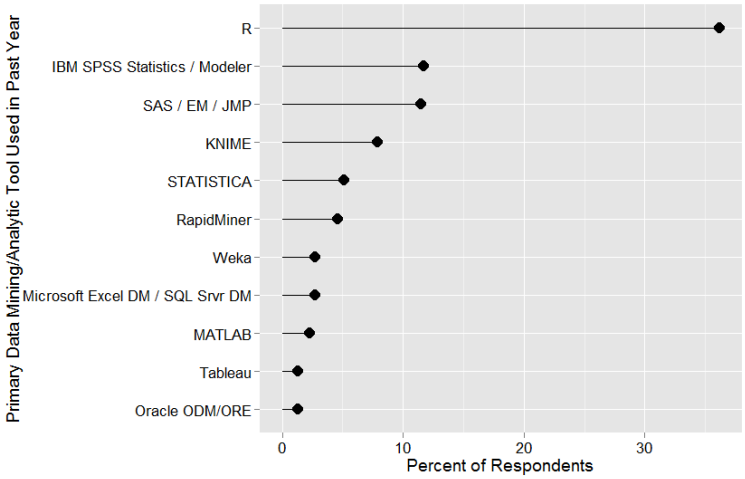
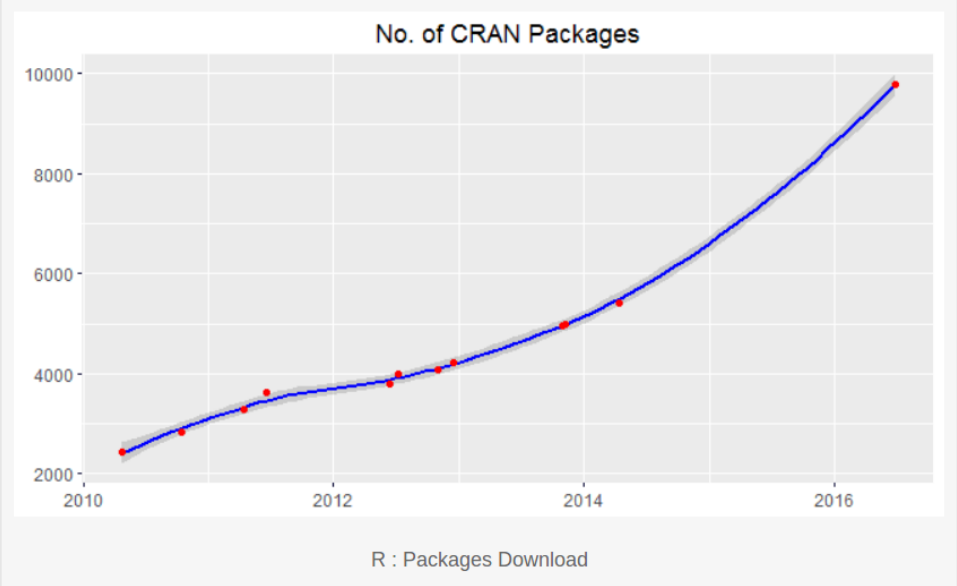


Figure 1a. The number of data science jobs for the more popular software (those with 250 jobs or more, 2/2017).

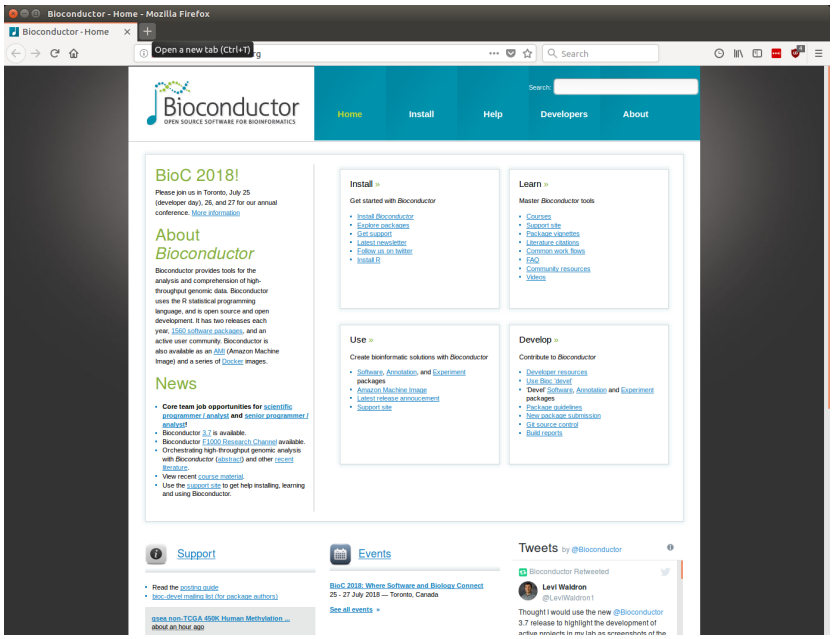
`https://i0.wp.com/r4stats.com/wp-content/uploads/2017/02/Fig-1a-IndeedJobs-2017.png`



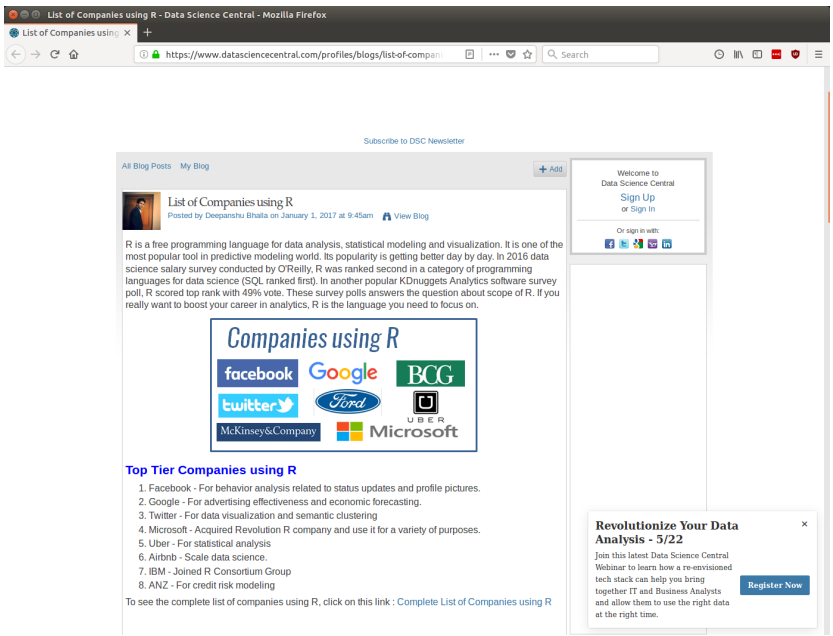
R is the most used software for getting data mining and big data "done". https://i1.wp.com/datasciencepopularity.com/wp-content/uploads/2015/10/fig_6a_rexersurveyprimaryto png



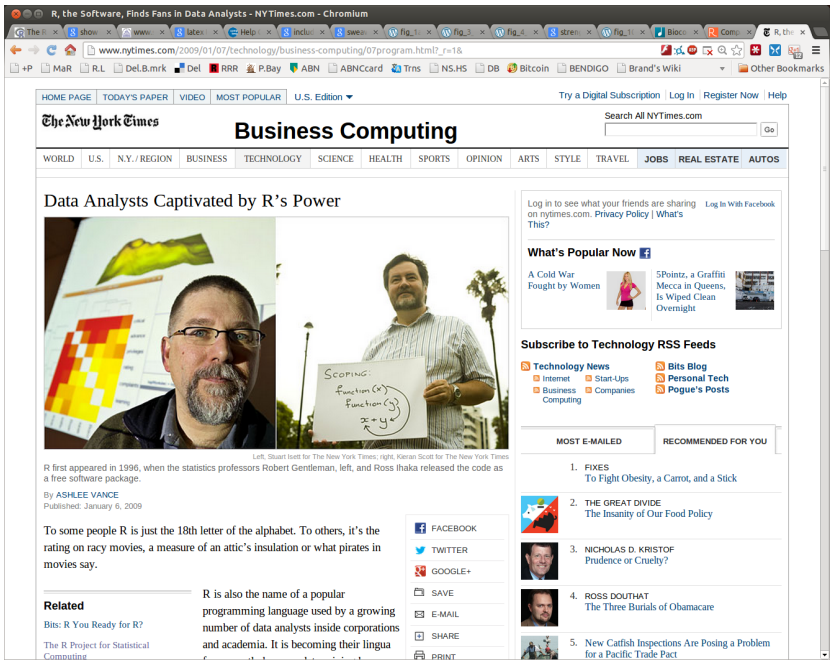
There's an enormous number of packages offering custom functionality. https://www.listendata.com/2016/12/companies-using-r.html



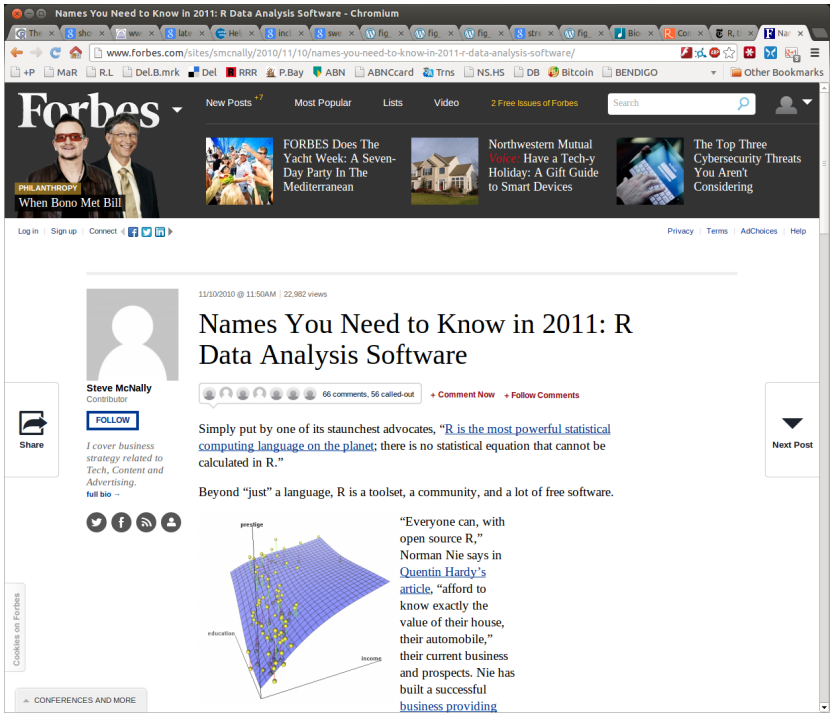
Bioconductor, initiated in 2001 for the analysis of genomics data, has grown from 15 to (currently) 1560 software packages for omics and life science analyses. `http://www.bioconductor.org/`



Used internally by lots of big, profitable companies. `https://www.datasciencecentral.com/profiles/blogs/list-of-companies-using-r`



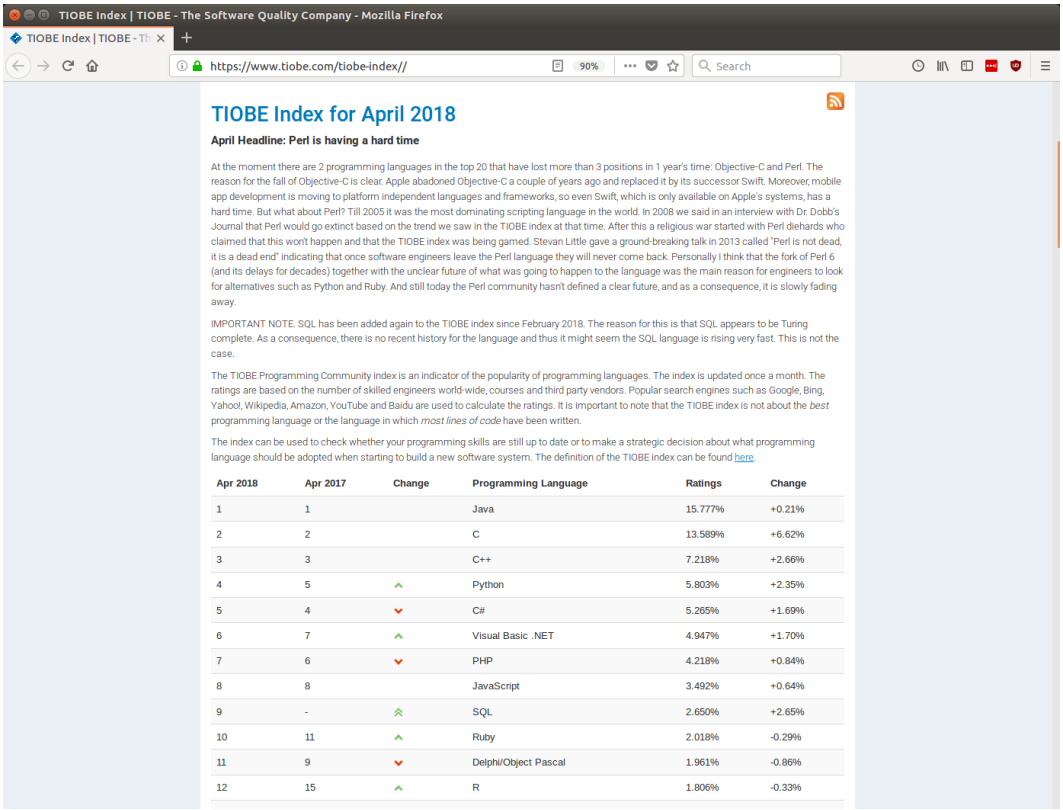
Has gained the popular media's attention. `http://www.nytimes.com/2009/01/07/technology/business-computing/07program.html?_r=0`



Link to reference

Many other statistical and analytical programs have implemented connectors to enable running `R` code within them, including 'competitors' like SPSS, MatLab and SAS. De-

spite this R's popularity continues to grow. In contrast to the 'workhorse' languages like C and Python which are used to build many programs, R is first and foremost a statistics focussed language. Despite this, it's close to being in the top 10 of the most popular computing languages.



Link to current Tiobe index

3 What *is* R and how do we use it

3.1 Definition(s)

- **Wikipedia**: 'R is a programming language and free software environment for statistical computing and graphics that is supported by the R Foundation for Statistical Computing. The R language is widely used among statisticians and data miners for developing statistical software and data analysis.' `http://en.wikipedia.org/wiki/R_(programming_language)`
- **Comprehensive R Archive Network (CRAN):** 'R is "GNU S", a freely available language and environment for statistical computing and graphics which provides a wide variety of statistical and graphical techniques: linear and nonlinear modelling, statistical tests, time series analysis, classification, clustering, etc.' `https://cloud.r-project.org/`.
- **My definition:** A command based, object oriented and functional language; in contrast to excel which is a cell centric, spreadsheet managing graphical user interface (GUI).

```
newObject <- function(oldObject)
```

object(s):	`oldObject`, `newObject`, and `function()`
assignment:	`<-`
expression:	`function(oldObject)`

3.2 How does R work?

You type commands, also known as "expressions," into a text editor or terminal and send them to R for evaluation. R evaluates the command, prints the command (by default), then the result of the evaluated command, if any.

In short you apply functions to objects, or "call" a function on an object, e.g.:

```
x <- c(1, 3, 5) #create an object
mean(x)         #calculate the arithmetic mean of the values in x
```

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

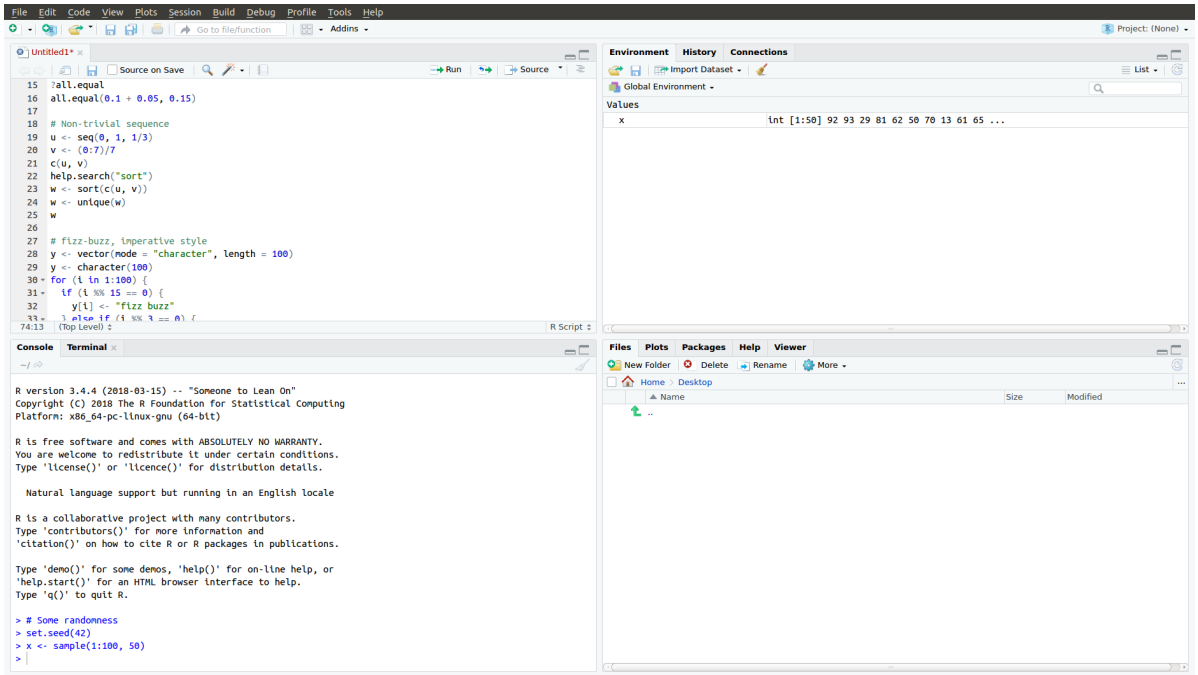
3.3 How we interact with R

Not surprisingly everything about R is built on the principle of KISS, i.e., keep it simple stupid.

- We get things done in `R` by typing commands and asking `R` to execute them with the `enter` key. Assuming you know the commands, it couldn't be any simpler.
- The "front end" of `R`, the part that receives your commands, is the *terminal*. You can type your commands directly into the terminal and hit `enter` to do your work. But if you need thousands of commands for a specific task, or want to share your method, this approach gets tiring fast.
- Since the commands are just text, it's easy to write a series of commands using any program built for working with text, i.e., text editors and word processors, to create *code* or a *script*, to get something done.
- The easiest, most efficient way to write `R` code is using a text editor or *integrated development environment*, like RStudio or Emacs/ESS as mentioned earlier.

3.3.1 RStudio: the easiest way to work with `R`

For most users, especially those new to `R`, RStudio is probably the best IDE to use. Like everything about `R`, you're free to choose whatever editor you like. For Windows platforms, you could just use the basic editor that's bundled with the downloadable installer. Let's have a look at an RStudio "session".



There are four panes.

- The top left is the source pane. **Do your typing in this pane**.
- The bottom left is your console. This is a window looking into what `R` is actually

doing. **Even though it's fine to type your commands in this pane at the `R` '>' prompt, avoid this**: your commands are less conveniently stored and rerun than your source code in the editor pane.

- Top right: environment pane
- Bottom left: files pane

Type your commands in the source pane and send them to `R` *using the keyboard shortcut* `Ctrl + Enter`. This will send the line of code your cursor is on to `R` if no code is selected; or if there is code selected, then only the selected code is sent. Open RStudio on your PC and try it now.

A few other things about RStudio:

- move the focus between panes using the key board shortcuts `Ctrl + 1` and `Ctrl + 2` for the source and console panes respectively
- the little asterisk next your source code file name indicates unsaved code
- **save your source code!**

3.4 Backslashes

One more thing I want to mention is how `R` interprets backslashes and forwardslashes, i.e. `\` and `/` respectively, as relates to defining the locations of files on your PC. James McDonald does a better job of clarifying this than I otherwise would `http://cran.r-project.org/doc/contrib/Lemon-kickstart/kr_scrpt.html`:

"Let's pause to note a little historical bifurcation that has caused almost as much strife as the question of exactly what the `<Delete>` key should do. The convention that the slash (/) character is used as a separator in the notation for a filesystem path is not universal. PC-DOS, for example, used the backslash (\). Like Fords vs Chevrolets, French vs English, and whether you crack the egg at the big or small end, it doesn't really matter a rat's behind which one you use as long as it works. To make it work in R, however, where the *NIX convention of using the backslash as an escape character is respected, you will have to double backslashes in paths for them to be read properly. Thus if the filename above was: `C:\JIM\PSYCH\ADOLDRUG\PARTYUSE1.R`

I would have to refer to it as: `C:\\JIM\\PSYCH\\ADOLDRUG\\PARTYUSE1.R` in an R script."

4 Getting Help

1. Online help. The first place to look. Also works when you're actually *offline*:

```
?help
help(help)
```

Broader search:

```
??help
apropos("help")              # quotes are needed
```

It's fun, try them out!

2. Google. And when you're actually *online*, often the *first* place and *last* place you'll go when you have no idea is Google.
3. Package Vignette's. Where the online help is devoid of the context often needed to grasp the full potential and logical use of functions, vignette's is the best place to find such information, with extended examples, and often more. For example, see the limma `http://www.bioconductor.org/packages/release/bioc/vignettes/limma/inst/doc/usersguide.pdf`
4. Mailing Lists. R-help `http://www.r-project.org/mail.html`. Do check first using Google or the R-help searchable archives `http://tolstoy.newcastle.edu.au/~rking/R/` first if your question has already been asked and has the answer to solve your issue.
5. Other lists: `https://stat.ethz.ch/mailman/listinfo`
6. Stack Overflow `https://stackoverflow.com`. Probably more questions and answers posted here than on the mailing list.

4.1 Trouble shooting

- Clean your environment and start again to be sure your messy environment isn't the cause of your issue. In RStudio, click the broom in the Environment tab; in any R session you can also run:

```
# not run:
rm(list=ls())
ls()
```

- Start `R` without any customisations, i.e., omit loading the `.Rprofile` and `.Renviron` customisation files. From the command line:

  `R --vanilla`

- `trace.back()` function: very helpful pinpointing the source of an error, and thus its cause.

4.2 Further resources

- R focused search engine: http://www.Rseek.org
- Aggregator of all R blogs: http://www.r-bloggers.com
- Quick-R: http://www.statmethods.net/
- R on Youtube: http://www.youtube.com/watch?v=mL27TAJGlWc
- R in two minutes: http://www.twotorials.com/
- *The* list of reference texts: http://www.r-project.org/doc/bib/R-publications.html
- And more..
  http://www.ats.ucla.edu/stat/r/
  http://zoonek2.free.fr/UNIX/48_R/02.html

- *A First Course in Statistical Programming with R.* W. John Braun and Duncan J. Murdoch. Cambridge University Press, 2007
  *I love this book, beautifully explores the programming aspects of R.

5 Document License

Basic Course on R: Objects and Functions

Karl Brand* and Elizabeth Ribble†

14-18 May 2018

Contents

*brandk@gmail.com
†emcclel3@msudenver.edu

1 Things you can have: object classes and modes

Any *thing* in `R` is of a certain type, defined as a `class` and `mode`. As an analogy, think of numbers as *fruits* and text as *vegetables*. They're not the same, although they can be analysed, or 'cooked' like one another.

1.1 Numbers

Fruits.

```
class(1)
```

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

```
mode(1)
```

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

```
class(2.3123)
```

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

```
an_integer <- as.integer(2.3123)
an_integer
```

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

```
class(an_integer)
```

```
## [1] "integer"
```

```
mode(an_integer)
```

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

Change the class to numeric:

```
numeric_again <- as.numeric(an_integer)
class(numeric_again)
```

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

1.2 Text

The *veggies*.

```
class("a")
```

```
## [1] "character"
```

```
mode("a")
```

```
## [1] "character"
```

```
class("1")
```

```
## [1] "character"
```

```
mode("1")
```

```
## [1] "character"
```

```
a_factor <- factor("a")
a_factor
```

```
## [1] a
## Levels: a
```

```
class(a_factor)
```

```
## [1] "factor"
```

```
mode(a_factor)
```

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

1.3 Logical

It can only be a *yes* or a *no*. More specifically, a `TRUE` or a `FALSE`.

```
class(TRUE)
```

```
## [1] "logical"
```

```
class(FALSE)
```

```
## [1] "logical"
```

```
mode(TRUE)
```

```
## [1] "logical"
```

1.4 Functions

An object that does something to the fruits and veggies, a *frying pan!*

2 Containers: more object classes and modes

Besides the basic types of things, there are different *containers* available to hold these things which are also defined by `class` and `mode`. Containers are thus things, or `objects`, that you can put other things, or `objects`, into to conveniently hold them while we work with them.

2.1 Vectors are an ordered series of elements

Think of them like a *string* or a *chain*.

```
a_vector <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)
class(a_vector)
```

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

```
mode(a_vector)
```

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

```
length(a_vector)
```

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

```
ano_vector <- c(1)
length(x = ano_vector)
```

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

Combine vectors to make another vector:

```
combined_vec <- c(a_vector, ano_vector)
combined_vec
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12  1
```

2.2 Matrices are vectors with two dimensions

Matrices are like a spread sheet in that they have rows and columns. In addition the contents have a specific, column on column order or sequence exactly equivalent to a vector. Think of them like a *string on a tray*. Matrices consist of only one type or *class* of data.

```r
a_matrix <- matrix(data = a_vector)
a_matrix
```

```
##       [,1]
##  [1,]    1
##  [2,]    2
##  [3,]    3
##  [4,]    4
##  [5,]    5
##  [6,]    6
##  [7,]    7
##  [8,]    8
##  [9,]    9
## [10,]   10
## [11,]   11
## [12,]   12
```

```r
a_vector
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

```r
dim(a_matrix)
```

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

```r
ano_matrix <- matrix(data = a_vector, nrow = 3)
ano_matrix
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

```r
dim(ano_matrix)
```

```
## [1] 3 4
```

```r
letters_mat <- matrix(data = letters[1:12], nrow = 3)

class(a_matrix)
```

```
## [1] "matrix"
```

```r
mode(a_matrix)
```

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

```r
class(letters_mat)
```

```
## [1] "matrix"
```

```
mode(letters_mat)
```

```
## [1] "character"
```

2.3 Arrays are matrices with multiple dimensions

```
an_array <- array(letters[1:12], c(2, 2, 3))
an_array
```

```
## , , 1
##
##      [,1] [,2]
## [1,] "a"  "c"
## [2,] "b"  "d"
##
## , , 2
##
##      [,1] [,2]
## [1,] "e"  "g"
## [2,] "f"  "h"
##
## , , 3
##
##      [,1] [,2]
## [1,] "i"  "k"
## [2,] "j"  "l"
```

```
class(an_array)
```

```
## [1] "array"
```

```
mode(an_array)
```

```
## [1] "character"
```

2.4 Lists hold just about anything and everything

Similar to vectors, lists have a specific order or sequence. Unlike vectors however, each element of a list can be of any `class` or `mode`. Think of them like a *shopping list* of: a string of fruit, a tray of veggies, another sublist of items

```
a_list <- list(a_vector, a_matrix, letters_mat)
a_list
```

```
## [[1]]
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
##
## [[2]]
##       [,1]
##  [1,]    1
##  [2,]    2
##  [3,]    3
##  [4,]    4
##  [5,]    5
##  [6,]    6
##  [7,]    7
##  [8,]    8
##  [9,]    9
## [10,]   10
## [11,]   11
## [12,]   12
##
## [[3]]
##      [,1] [,2] [,3] [,4]
## [1,] "a"  "d"  "g"  "j"
## [2,] "b"  "e"  "h"  "k"
## [3,] "c"  "f"  "i"  "l"
```

```
class(a_list)
```

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

```
mode(a_list)
```

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

```
named_list <- list("vec" = a_vector, "mat" = a_matrix,
                   "lets" = letters_mat,
                   "log" = matrix(rep(c(TRUE, FALSE), 5), nrow = 5),
                   "lis" = list(1:5, letters[1:9]))
named_list
```

```
## $vec
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
##
## $mat
##       [,1]
##  [1,]    1
```

```
##  [2,]    2
##  [3,]    3
##  [4,]    4
##  [5,]    5
##  [6,]    6
##  [7,]    7
##  [8,]    8
##  [9,]    9
## [10,]   10
## [11,]   11
## [12,]   12
##
## $lets
##      [,1] [,2] [,3] [,4]
## [1,] "a"  "d"  "g"  "j"
## [2,] "b"  "e"  "h"  "k"
## [3,] "c"  "f"  "i"  "l"
##
## $log
##       [,1]  [,2]
## [1,]  TRUE FALSE
## [2,] FALSE  TRUE
## [3,]  TRUE FALSE
## [4,] FALSE  TRUE
## [5,]  TRUE FALSE
##
## $lis
## $lis[[1]]
## [1] 1 2 3 4 5
##
## $lis[[2]]
## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i"
```

```
class(named_list)
```

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

```
mode(named_list)
```

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

```
named_list[["log"]]
```

```
##       [,1]  [,2]
```

```
## [1,]  TRUE FALSE
## [2,] FALSE  TRUE
## [3,]  TRUE FALSE
## [4,] FALSE  TRUE
## [5,]  TRUE FALSE
```

```
class(named_list[["log"]])
```

```
## [1] "matrix"
```

```
mode(named_list[["log"]])
```

```
## [1] "logical"
```

2.5 Data frames are special lists

Data frames are even more similar to the familiar excel spreadsheets: a series, or `list` of equal length columns, or `vectors`. Notably, the columns (`vectors`) can be of different classes, unlike a matrix or array.

```
aDF <- data.frame("vec" = a_vector, "lets" = letters[1:12])
aDF
```

```
##    vec lets
## 1    1    a
## 2    2    b
## 3    3    c
## 4    4    d
## 5    5    e
## 6    6    f
## 7    7    g
## 8    8    h
## 9    9    i
## 10  10    j
## 11  11    k
## 12  12    l
```

```
dim(aDF)
```

```
## [1] 12  2
```

```
class(aDF)
```

```
## [1] "data.frame"
```

```
mode(aDF)
```

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

```
str(aDF)
```

```
## 'data.frame': 12 obs. of  2 variables:
##  $ vec : num  1 2 3 4 5 6 7 8 9 10 ...
##  $ lets: Factor w/ 12 levels "a","b","c","d",..: 1 2 3 4 5 6 7 8 9 10 ...
```

```
aDF[, 1]
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

```
class(aDF[, 1])
```

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

```
mode(aDF[, 1])
```

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

```
aDF[, 2]
```

```
##  [1] a b c d e f g h i j k l
```

```
## Levels: a b c d e f g h i j k l
class(aDF[, 2])
## [1] "factor"
mode(aDF[, 2])
## [1] "numeric"
str(aDF[, 1])
##  num [1:12] 1 2 3 4 5 6 7 8 9 10 ...
```

It's important to understand that the things we work with in R have a 'mode' and a 'class'. The mode is a mutually exclusive classification of an object's basic structure and the class is a property used to determine how functions operate with object. Pay attention to your modes and classes.

3 Functions are things you can do

R comes with predefined functions which do many things from basic file management to complex statistics. To get started, below are some oft used functions. This default set of functions is easily extended by defining your own functions and adding those defined by others conveniently in CRAN and BioConductor packages:

`http://stat.ethz.ch/R-manual/R-patched/library/base/html/00Index.html`

`http://cran.r-project.org/web/packages/available_packages_by_name.html`

`http://www.bioconductor.org/packages/release/bioc/`

Functions are expressed as:

`function.name()`, e.g., `t.test()`

or,

an operator, e.g., `+`

Note that operators can be expressed in `function.name()` format by enclosing in double quotes, e.g. `"+"()`. This is occasionally required for some expressions.

Easily obtain functions from other R users using `install.packages()`:

```
install.packages("packageName",
                 lib = "/directory/to/my custom R library",
                 repos = "http://cran.xl-mirror.nl")
```

The package name must be quoted when installing. Besides installing the package on your PC, you need to load it into your R session before you can use it:

```
library("packageName")      ## quotes are optional when loading a package
```

3.1 The Arithmetic Operators

That is, R as a *calculator*.

```
x <- 10
y <- 3
x + y
```

```
## [1] 13
```

```
x - y
```

```
## [1] 7
```

```
x * y
```

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

```
x / y
```

```
## [1] 3.333333
```

```
x ^ y           # exponentiation
```

```
## [1] 1000
```

```
x %% y          # modular arithmetic, remainder after division
```

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

```
x %/% y         # integer part of a fraction
```

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

How these functions work on vectors:

```
a_vector
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

3.1.1 By the way, R is Green

By design, R *recycles* data without telling you, at least when the recycling fits neatly.

```
a_vector + x                    # x is recycled without warning
```

```
##  [1] 11 12 13 14 15 16 17 18 19 20 21 22
```

Operators can also be used in the `function.name()` format. Note the quotes:

```
"+"(a_vector, x)
```

```
##  [1] 11 12 13 14 15 16 17 18 19 20 21 22
```

```
a_vector - y
```

```
##  [1] -2 -1  0  1  2  3  4  5  6  7  8  9
```

```
a_vector + a_vector
```

```
##  [1]  2  4  6  8 10 12 14 16 18 20 22 24
```

```
vec_of_thr <- c(2, 4, 6)
a_vector
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

```
a_vector + vec_of_thr           # recycled without warning
```

```
##  [1]  3  6  9  6  9 12  9 12 15 12 15 18
```

```
vec_of_fi <- c(1, 2, 3, 4, 5)
a_vector
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

When it doesn't fit neatly, you'll get a warning:

```
a_vector + vec_of_fi            # warning
```

```
## Warning in a_vector + vec_of_fi: longer object length is not a multiple
of shorter object length
```

```
##  [1]  2  4  6  8 10  7  9 11 13 15 12 14
```

```
## NULL
```

How these functions work on matrices:

```
ano_matrix
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
```

```
## [3,]    3    6    9   12
```

```
ano_matrix + x
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   11   14   17   20
## [2,]   12   15   18   21
## [3,]   13   16   19   22
```

```
ano_matrix - y
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   -2    1    4    7
## [2,]   -1    2    5    8
## [3,]    0    3    6    9
```

```
ano_matrix + ano_matrix
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    2    8   14   20
## [2,]    4   10   16   22
## [3,]    6   12   18   24
```

```
ano_matrix * ano_matrix
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1   16   49  100
## [2,]    4   25   64  121
## [3,]    9   36   81  144
```

3.2 Functions you'll use to interact with data

Relational Operators:

```
x < y
```

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

```
x > y
```

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

```
x <= y
```

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

```
x >= y
```

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

```
x == y
## [1] FALSE
x != y
## [1] TRUE
```

3.3 Functions you need to get work done with R

R has an extensive set of built-in ***functions***, a few of which are listed below:

Print structure of an object:

```
str()
```

Print structure of an object:

```
class()
```

First six elements/rows:

```
head()
```

Last six elements/rows:

```
tail()
```

List all objects and functions held in your global environment:

```
ls()
```

Generate a sequence:

```
seq(from = 1, to = 10, by = 2)
```

Or, use the : operator:

```
1:10
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Run the entire contents of a script:

```
source("myScript.R")
```

Each function accepts one or more values passed to it as ***arguments***, performs computations or operations on those values, and returns a result.

To perform a ***function call***, type the name of the function with the values of its argument(s) in parentheses, then hit 'Enter'. For example:

```
sqrt(2)
```

```
## [1] 1.414214
```

Values passed as arguments can be in the form of variables, such as `x` below:

```
x <- 2
sqrt(x)
```

```
## [1] 1.414214
```

or they can be entire expressions, such as `x^2 + 5` below:

```
sqrt(x^2 + 5)
```

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

3.4 Viewing a Function's Formal Arguments

Most functions take multiple arguments, each of which has a name, and some of which are optional.

One way to see what arguments a function takes and which ones are optional is to use the function:

`args()`

Another way to view a function's arguments is to look at its help file (e.g. `?sqrt`).

For example, to see what arguments `round()` takes (using `args()`), we'd type:

```
args(round)
```

```
## function (x, digits = 0)
## NULL
```

We see that `round()` has two arguments, `x`, a numeric value to be rounded, and `digits`, an integer specifying the number of decimal places to round to. Thus to round 4.679 to 2 decimal places, we type:

```
round(4.679, 2)
```

```
## [1] 4.68
```

3.5 Optional Arguments and Default Values

The specification `digits = 0` in the output from `args(round)` tells us that `digits` has a ***default value*** of 0. This means that it's an ***optional argument*** and if no value

is passed for that argument, rounding is done to 0 decimal places (i.e. to the nearest integer).

3.6 Positional Matching and Named Argument Matching

When we type

```
round(4.679, 2)
```

```
## [1] 4.68
```

R knows, by ***positional matching***, that the first value, 4.679, is the value to be rounded and the second one, 2, is the number of decimal places to round to.

We can also specify values for the arguments by name. For example:

```
round(x = 4.679, digits = 2)
```

```
## [1] 4.68
```

When ***named argument matching*** is used, as above, the order of the arguments is irrelevant. For example, we get the same result by typing:

```
round(digits = 2, x = 4.679)
```

```
## [1] 4.68
```

The two types of argument specification (positional and named argument matching) can be mixed in the same function call.

3.7 Formal Arguments and Actual Arguments

In the function call

```
round(x = 4.679, digits = 2)
```

the values `4.679` and `2` passed to `round()` are referred to as ***actual arguments***.

The ***formal arguments*** for the function `round()` are `x` and `digits`, to which we pass the actual values `4.679` and `2`.

Look at the arguments for the function `signif()`:

```
args(signif)

## function (x, digits = 6)
## NULL
```

`signif()` prints the value passed for `x` to the number of significant digits specified by `digits`.

3.8 Variable Number of Arguments Using "..."

Functions can be written to take a variable number of arguments. The argument name "..." in the function definition will match any number of arguments.

For example, here's a function that returns the mean of all the values in an arbitrary number of vectors:

```
meanOfAll <- function(...) {
  return(mean(c(...)))
}
```

The command

```
meanOfAll(usSales, europeSales, otherSales)
```

would combine the three vectors and take the mean of all the data. The effect of `c(...)` is as if `c()` were called with the same three arguments passed to `meanOfAll()`.

Many of R's built-in functions take a variable number of arguments. For example look at the help files for `list()` and `c()`.

4 Saving and restoring your session

Because the typical way of using `R` involves writing text into a file with the `.r` or `.R` extension it's natural to save your commands written for a specific purpose in this `my-r-file.r`. Hopefully you're doing this right now. However there are other bits and pieces of your `R` session described below you may want or need to save.

Save all objects to the working directory in a file called `.RData`:

```
save.image()
```

Or give the file a name:

```
save.image("mySession.RData")
```

Save all commands entered into the R console during your session to the working directory in a file called .Rhistory:

```
savehistory()
```

Such a file may be hidden by default in some file browsers, including linux.

Note that this typically occurs (console dependent) by default when you quit your R session. If an .Rhistory file already exists from a previous session, the current session is appended to the end of this file and saved. Thus, by design, a complete log of your work is saved together with your script file, if your console session contains less than 512 lines.

Save specific objects:

```
save(object_1, object_2, object_3, file = "rObjects123.RData")
```

Restore or load previous sessions or objects:

```
load("mySession.RData")         # load from the working directory
```

Load .Rhistory file to access the history from the console:

```
loadhistory()
```

5 Miscellaneous Tips

R is case sensitive:

```
c("Hello" == "hello", "goodbye" == "goodbye")
```

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

And is *very* sensitive in general:

```
# c("Hello" == "hello" "goodbye" == "goodbye") # not run
```

Missing data. NA 'Not Available' is how R defines a missing value i.e., an empty cell in excel

```
c(1, NA, 3, 4, NA)
```

```
## [1]  1 NA  3  4 NA
```

R is an environment. What's in yours?

```
ls()                                # global environment
objects()                           # alias for ls()
```

Too many objects cluttering up your environment and computer memory? Remove the ones you don't need:

```
rm(object_name)
```

What is the current *working directory* where files are saved and loaded from by default?

```
getwd()
```

Need to set or change the current working directory?

```
setwd(dir = "c:\\Windows\\Users\\karl\\My Documents\\R stuff")
setwd(dir = "c:/Windows/Users/karl/My Documents/R stuff")
```

Note the double backslashes required by `R` running under Windows. Single forward slashes, the Unix convention works equivalently.

Also be aware that if you started your `R` session by double clicking a `file.name.r` file, by design the directory where this file resides is set as the working directory. This directory is also a good place, and the default place, to store the respective `.RData` and `.Rhistory` files if you have any. When these files are present in the same directory they will be loaded automatically, unless you explicitly tell `R` not to load them by using the `---no-restore-data` argument when starting `R`.

Quit your `R` session:

```
q()                                # not run
```

Object names can not begin with numbers. For example:

```
## 3vector <- c(2, 4, 6))     # not run
```

But can end with numbers if necessary:

```
vector3 <- c(2, 4, 6)
```

Try to code with style, for example:

`https://google.github.io/styleguide/Rguide.xml`

`http://adv-r.had.co.nz/Style.html`

6 Document License

GNU General Public License v2.0 or higher (GPL>=v2)

Basic Course on R: Objects and Functions Practical

Karl Brand* and Elizabeth Ribble†

17-21 December 2018

Contents

*brandk@gmail.com
†emcclel3@msudenver.edu

1 Objects and Functions

1.1 Function Arguments

(a) Look at the help file for the function `mean()`. How many arguments does function have? What types of vectors are accepted?

(b) Use the function `mean()` to calculate the mean of the following values (note the NA and use named arugment matching):

1 2 NA 6

(c) Do (b) again but rearrange the arguments.

(d) Do (b) again using positional matching.

(e) Determine the class of `mean()` using `class()`.

(f) Determine the class of `mean()` using `str()`.

(g) Determine the class of the value output in part (d) using `class()`.

(h) Determine the class of the value output in part (d) using `str()`.

1.2 Use R as a calculator to calculate the following values:

$$17^4, \quad 45 - 2 \cdot 3, \quad (45 - 2) \cdot 3$$

1.3 Use the operators %% and %/% to do the following:

(a) Calculate the remainder after dividing 29,079 into 184,277,809.

(b) How many times does 29,079 go into 184,277,809 (i.e. what's the "integer divide" value)?

1.4 Do the last calculation from Q1.4 in another way, like this:

```
a <- 45
b <- 2
c <- 3
d <- (a - b) * c
```

Now check what `a`, `b`, `c`, and `d` are. You can just type the variable name (e.g. `a`) and hit 'Control' then 'Return' or use the command `print(a)`.

1.5 Do the following to practice saving and opening files in R.

(a) Look at the variables (or other objects) that are stored in your Workspace by typing either `objects()` or `ls()`.

(b) Check your working directory by typing `getwd()`. Now change it to a different directory - preferably your own flash drive - by using the function `setwd()`, for example:

```
setwd("C:/Users/Elizabeth/My Documents/R Course")
```

(c) Use the function "save.image()" to save your R session to a file called `YourLastName_practical1.RData` (replace YourLastName with your last name). Note that this will save a .RData file that contains only those objects you see when you run `ls()`. It does not save any code you typed into the console or into the source pane.

(d) Use the RStudio "File" drop-down menu to save your R source code to a file called `YourLastName_practical1.R` (replace YourLastName with your last name). Note that this will only save the text you've typed into the source pane. It does not save any objects or anything typed into or ran through the console.

(e) Use the function "save()" to save only the objects `myFirstList` and `pets` to a file called `YourLastName_objects.RData` (replace YourLastName with your last name).

(f) Now close out RStudio entirely, select "Save" or "Yes" in any dialog boxes that pop up, and then reopen RStudio. Is your source code still there?

(g) Run `ls()`; are your objects still there?

(h) You can change these kinds of options by going to "Tools - Global Options". Go there and deselect "Restore .RData into workspace at startup". Then close RStudio and choose to save the .RData file.

(i) Reopen RStudio; your environment should be empty. Load your objects back in using `load()` (e.g. `load("Ribble_practical1.RData")`) and then run ls() again. Do you see your objects now?

(j) Check what the working directory is by again running `getwd()` - has it been reset?

2 Document License

Basic Course on R: Objects and Functions Practical Answers

Karl Brand* and Elizabeth Ribble†

17-21 December 2018

Contents

*brandk@gmail.com
†emcclel3@msudenver.edu

1 Objects and Functions

1.1 Function Arguments

(a) Look at the help file for the function `mean()`. How many arguments does function have? What types of vectors are accepted? What is the default setting for dealing with NA values?

Three arguments (plus further arguments). Numerical and logical vectors are accepted. The default setting is to NOT remove NA (missing) values.

(b) Use the function `mean()` to calculate the mean of the following values (note the NA and use named arugment matching):

1 2 NA 6

```
mean(x = c(1, 2, NA, 6), na.rm = TRUE)
```

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

(c) Do (b) again but rearrange the arguments.

```
mean(na.rm = TRUE, x = c(1, 2, NA, 6))
```

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

(d) Do (b) again using positional matching.

```
mean(c(1, 2, NA, 6), 0, TRUE)
```

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

(e) Determine the class of `mean()` using `class()`.

```
class(mean)
```

```
## [1] "function"
```

The class is "function".

(f) Determine the class of `mean()` using `str()`.

```
str(mean)
```

```
## function (x, ...)
```

The class is "function".

(g) Determine the class of the value output in part (d) using `class()`.

```
class(mean(c(1, 2, NA, 6), 0, TRUE))
```

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

The class is "numeric".

(h) Determine the class of the value output in part (d) using `str()`.

```
str(mean(c(1, 2, NA, 6), 0, TRUE))
```

```
##  num 3
```

The "num" means the class is "numeric".

1.2 Use R as a calculator to calculate the following values:

$$17^4, \quad 45 - 2 \cdot 3, \quad (45 - 2) \cdot 3$$

```
17^4
```

```
## [1] 83521
```

```
45 - 2 * 3
```

```
## [1] 39
```

```
(45 - 2) * 3
```

```
## [1] 129
```

1.3 Use the operators %% and %/% to do the following:

(a) Calculate the remainder after dividing 29,079 into 184,277,809.

```
184277809 %% 29079
```

```
## [1] 4186
```

(b) How many times does 29,079 go into 184,277,809 (i.e. what's the "integer divide" value)?

```
184277809 %/% 29079
```

```
## [1] 6337
```

1.4 Do the last calculation from Q1.4 in another way, like this:

```
a <- 45
b <- 2
c <- 3
d <- (a - b) * c
```

Now check what `a`, `b`, `c`, and `d` are. You can just type the variable name (e.g. `a`) and hit 'Control' then 'Return' or use the command `print(a)`.

```
a <- 45
b <- 2
c <- 3
d <- (a - b) * c
a
```

```
## [1] 45
```

```
b
```

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

```
c
```

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

```
d
```

```
## [1] 129
```

1.5 Do the following to practice saving and opening files in R.

(a) Look at the variables (or other objects) that are stored in your Workspace by typing either `objects()` or `ls()`.

Answers will vary.

(b) Check your working directory by typing `getwd()`. Now change it to a different directory - preferably your own flash drive - by using the function `setwd()`, for example:

```
setwd("C:/Users/Elizabeth/My Documents/R Course")
```

Answers will vary.

(c) Use the function "save.image()" to save your R session to a file called `YourLastName_practical1.RData` (replace YourLastName with your last name). Note that this will save a .RData file that contains only those objects you see when

you run `ls()`. It does not save any code you typed into the console or into the source pane.

```
save.image(file = "Ribble_practical1.RData")
```

(d) Use the RStudio "File" drop-down menu to save your R source code to a file called `YourLastName_practical1.R` (replace YourLastName with your last name). Note that this will only save the text you've typed into the source pane. It does not save any objects or anything typed into or ran through the console.

```
savehistory(file = "Ribble_practical1.Rhistory")
```

You may get this error: `'Error in savehistory(file) : no history available to save'` This means no commands have been directly in entered into your *console*, but only from your text editor, which is of course it's own permanent record, assuming you save it!

(e) Use the function "save()" to save only the objects `myFirstList` and `pets` to a file called `YourLastName_objects.RData` (replace YourLastName with your last name).

```
save(myFirstList, pets, file = "Ribble_objects.RData")
```

(f) Now close out RStudio entirely, select "Save" or "Yes" in any dialog boxes that pop up, and then reopen RStudio. Is your source code still there?

It should be; the default in RStudio is to keep source code open!

(g) Run `ls()`; are your objects still there?

The objects available in your previous session are still available and RStudio reloads them for the new session.

(h) You can change these kinds of options by going to "Tools - Global Options". Go there and deselect "Restore .RData into workspace at startup". Then close RStudio and choose to save the .RData file.

(i) Reopen RStudio; your environment should be empty. Load your objects back in using `load()` (e.g. `load("Ribble_practical1.RData")`) and then run ls() again. Do you see your objects now?

They should appear after you've loaded in the file that contained your previously created objects.

(j) Check what the working directory is by again running `getwd()` - has it been reset?

It should have been reset to the default RStudio working directory. This means everytime you open RStudio you should specify your desired working directory!

2 Document License

GNU General Public License v2.0 or higher (GPL>=v2)

Basic Course on R: Manipulating / Selecting Data

Karl Brand* and Elizabeth Ribble†

17-21 December 2018

Contents

*brandk@gmail.com
†emcclel3@msudenver.edu

1 Manipulating / Selecting Data

1.1 Indexing

Select data from vectors, arrays, lists, rows/columns in matrices and data.frames, using

- Explicit numeric selection
- Named element selection
- Logical and relational filtering

1.1.1 Explicit numeric selection

We can specify positions or *indices* i in numbers of the data we wish to select using square brackets which are the 'extract' function, i.e., `object[i]`.

```
x <- c(11, 10, 12, 15)
x[2]

## [1] 10

y <- c(13, 17, 21, 18)
y[c(2, 3)]

## [1] 17 21

m <- matrix(c(x, y), ncol = 2)
m

##      [,1] [,2]
## [1,]   11   13
## [2,]   10   17
## [3,]   12   21
## [4,]   15   18

m[, 1]

## [1] 11 10 12 15

m[c(2, 4), 1:2]

##      [,1] [,2]
## [1,]   10   17
## [2,]   15   18
```

We can also use negative indices to remove/exclude elements we do not want:

```
x[-2]

## [1] 11 12 15

m[, -c(1, 3)]

## [1] 13 17 21 18
```

1.1.2 Named element selection

Here we make use of the `names` attribute:

```
names(x) <- c("a", "b", "c", "d")
x

##  a  b  c  d
## 11 10 12 15

str(x)

##  Named num [1:4] 11 10 12 15
##  - attr(*, "names")= chr [1:4] "a" "b" "c" "d"

x["b"]

##  b
## 10

row.names(m) <- LETTERS[1:4]
m

##   [,1] [,2]
## A   11   13
## B   10   17
## C   12   21
## D   15   18

m["A", ]

## [1] 11 13
```

Note that this is case sensitive and that `LETTERS` is different from `letters`.

We can even combine the use of numbers and names:

```
m["A", 2]

##  A
## 13
```

Let's look at an example where row names are gene symbols and column names are sample IDs:

```
mygenes <- data.frame(samp1 = c(33, 22, 12),
                      samp2 = c(44, 111, 13),
                      samp3 = c(33, 53, 65))
row.names(mygenes) <- c("CRP", "BRCA1", "HOXA")
mygenes

##       samp1 samp2 samp3
## CRP      33    44    33
## BRCA1    22   111    53
## HOXA     12    13    65

mygenes["CRP", ]

##     samp1 samp2 samp3
## CRP    33    44    33

mygenes[, "samp1"]

## [1] 33 22 12

mygenes[, c("samp1", "samp3")]

##       samp1 samp3
## CRP      33    33
## BRCA1    22    53
## HOXA     12    65

mygenes["HOXA", "samp2"]

## [1] 13
```

Note that we can also change the names of a `data.frame` in the same way we would do so for a `list` using the `names` attribute:

```
names(mygenes) <- c("samp10", "samp20", "samp30")
mygenes

##       samp10 samp20 samp30
## CRP       33     44     33
## BRCA1     22    111     53
## HOXA      12     13     65

## but let's change it back...
names(mygenes) <- c("samp1", "samp2", "samp3")
```

Note that the `colnames` function also performs the same job for data frames.

1.1.3 Logical and relational filtering

Expressions like `<`, `<=`, `|`, and `!=` can also be used to select data:

```
x

##  a  b  c  d
## 11 10 12 15

y

## [1] 13 17 21 18

keep <- c(TRUE, TRUE, FALSE, FALSE, TRUE)
x[keep]

##    a    b <NA>
##   11   10   NA

x[y >= 18]

##  c  d
## 12 15
```

Why?

```
y>=18

## [1] FALSE FALSE  TRUE  TRUE

which(y >= 18)

## [1] 3 4

x[x <= 11 | x == 15]

##  a  b  d
## 11 10 15

x[x != 10]

##  a  c  d
## 11 12 15
```

This works for factors as well:

```
gender <- factor(c("M", "M", "F", "F"))
gender

## [1] M M F F
## Levels: F M

males <- gender[gender == "M"]
levels(males)

## [1] "F" "M"
```

1.2 Other useful functions

Besides square brackets (`[`, `[[`), other useful functions exist for selecting data: `duplicated`, `match()`, `%in%`, `grep`, `is.na` and `$`.

To select e.g. rows that are not duplicated:

```
mm <- matrix(c(x, x, y, y), nrow = 4, byrow = T)
```

```
mm
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   11   10   12   15
## [2,]   11   10   12   15
## [3,]   13   17   21   18
## [4,]   13   17   21   18
```

```
mm[!duplicated(mm), ]
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   11   10   12   15
## [2,]   13   17   21   18
```

The above can also be done with `unique`, but the use of `duplicated` might be more appropriate in more complex situations:

```
unique(mm)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   11   10   12   15
## [2,]   13   17   21   18
```

Calling `match` returns the position of the first match of its first argument in the second argument:

```
match(c("a", "b"), c("a", "c", "a", "b", "a", "b"))
```

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

whereas `%in%` tells you whether the elements of the first argument appear in the second argument:

```
c("a", "b", "d") %in% c("a", "c", "a", "b", "a", "b")
```

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

Recall our data frame mygenes:

```
mygenes
```

```
##       samp1 samp2 samp3
## CRP      33    44    33
## BRCA1    22   111    53
## HOXA     12    13    65
```

```
is.data.frame(mygenes)

## [1] TRUE
```

Note that since `mygenes` is a data frame, it is therefore also an array, which means we can select by the name of the elements in the array:

```
mygenes[match(c("samp1", "samp3"), colnames(mygenes))]

##       samp1 samp3
## CRP      33    33
## BRCA1    22    53
## HOXA     12    65

mygenes[colnames(mygenes) %in% c("samp1", "samp4")]

##       samp1
## CRP      33
## BRCA1    22
## HOXA     12
```

However, in this case we could just use the names...

```
mygenes[c("samp1", "samp3")]

##       samp1 samp3
## CRP      33    33
## BRCA1    22    53
## HOXA     12    65
```

But this gives an error:

```
mygenes[c("samp1", "samp30")] ## not run
```

where this does not:

```
mygenes[colnames(mygenes) %in% c("samp1", "samp30")]

##       samp1
## CRP      33
## BRCA1    22
## HOXA     12
```

We can also use functions like `grep` to search for the names we are interested in:

```
mygenes[grep(2, names(mygenes))]

##       samp2
## CRP      44
## BRCA1   111
## HOXA     13

mygenes[grep("A", row.names(mygenes)), ]

##       samp1 samp2 samp3
## BRCA1    22   111    53
## HOXA     12    13    65
```

If we want to find or exclude data with missing values, we can use `is.na`:

```
z <- c(1:4, NA, 5:10)
z

##  [1]  1  2  3  4 NA  5  6  7  8  9 10

is.na(z)

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

which(is.na(z))

## [1] 5

z[!is.na(z)]

##  [1]  1  2  3  4  5  6  7  8  9 10
```

Double brackets `[[` and `$` extract *single* elements of objects, whereas `[` can extract multiple elements. Recall that a data frame is a special list, where each column is an element of a list:

```
## retrieve element
mygenes[[3]]

## [1] 33 53 65
```

```
## retrieve element
mygenes$samp3

## [1] 33 53 65

## subset
mygenes[3]

##       samp3
## CRP      33
## BRCA1    53
## HOXA     65
```

These work for lists as well:

```
mygenelist <- list(mygenes = mygenes, mygenes2 = mygenes * 2)
mygenelist

## $mygenes
##       samp1 samp2 samp3
## CRP      33    44    33
## BRCA1    22   111    53
## HOXA     12    13    65
##
## $mygenes2
##       samp1 samp2 samp3
## CRP      66    88    66
## BRCA1    44   222   106
## HOXA     24    26   130

## retrieve list element
mygenelist[[1]]

##       samp1 samp2 samp3
## CRP      33    44    33
## BRCA1    22   111    53
## HOXA     12    13    65

mygenelist$mygenes

##       samp1 samp2 samp3
## CRP      33    44    33
## BRCA1    22   111    53
## HOXA     12    13    65
```

```
## subset list
mygenelist[1]

## $mygenes
##       samp1 samp2 samp3
## CRP      33    44    33
## BRCA1    22   111    53
## HOXA     12    13    65
```

And we can even combine `$` with `[` :

```
mygenelist$mygenes[2]

##       samp2
## CRP      44
## BRCA1   111
## HOXA     13
```

The `$` notation is useful for accessing elements of objects output by functions, e.g. a t-test:

```
tt <- t.test(x, y)
names(tt)

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

tt$p.value

## [1] 0.04342819

tt$estimate

## mean of x mean of y
##     12.00     17.25
```

Basic Course on R: Manipulating Data Practical

Karl Brand* and Elizabeth Ribble†

17-21 December 2018

Contents

*brandk@gmail.com
†emcclel3@msudenver.edu

1 Manipulating / Selecting Data

1.1 Answer the following without typing the commands into R. Use ? if you're not sure what the object is or what the function does.

1.1.1 What is

```
length(letters)
```

1.1.2 What is

```
length(letters == LETTERS)
```

1.1.3 What is

```
which(letters %in% c("a", "d"))
```

1.1.4 What is

```
which(c("a", 7, "d") %in% letters)
```

1.1.5 What is

```
letters[LETTERS > "W"]
```

1.1.6 What is

```
letters[!LETTERS > "C"]
```

1.1.7 What is

```
seq(from = 1, to = 20, by = 3)
```

1.1.8 Why is `x` filled in the way it is? Hint: read about the arguments for `matrix`!

```
x <- matrix(8:11, nrow = 6, ncol = 4)
x
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    8   10    8   10
## [2,]    9   11    9   11
## [3,]   10    8   10    8
## [4,]   11    9   11    9
## [5,]    8   10    8   10
## [6,]    9   11    9   11
```

1.1.9 What are

```
x + 4
x + x
2 * x
x / c(2, 3, 4, 5)
x[, 3] + 2 * x[, 2]
nrow(x)
x[x[, 3] > 10, ]
```

1.2 Use R to answer the following.

1.2.1 Create a vector (using `c()`) called `a` (i.e. assign it to an object called `a`) with four elements which are the integers 5 to 8 (inclusive).

1.2.2 Display element 2 of `a`.

1.2.3 Display element 4 of `a`.

1.2.4 Calculate the product of elements 2 and 4 of `a`.

1.2.5 Assign the integers 3 and 4 to object `b` and use `b` to select elements 3 and 4 of object `a`.

1.2.6 Display every element of `a` except element 2.

1.2.7 Display every element of `a` except elements 3 and 4.

1.2.8 Display only those elements of `a` that are greater than or equal to 6.

1.2.9 Display only those elements of `a` that are not equal to 7.

1.2.10 Use the `list` function to create an object `ab` which is a list of the two objects `a` and `b`.

1.2.11 Display `ab`.

1.2.12 Change the names of the elements in `ab` to "a" and "b".

1.2.13 Display `ab` again. What has changed?

1.2.14 Create this matrix `m`:

```
m <- matrix(1:9, nrow = 3, byrow = T)
m
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
```

Why are the numbers 1, 2, and 3 in the first row and not the first column?

1.2.15 Display the element on the second row and second column of `m`.

1.2.16 Display only the 2nd row of `m`.

1.2.17 Display only the 3rd column of `m`.

1.2.18 Display only the 2nd and 3rd columns of `m`. Do so in two different ways.

If you want to save your work: save your R session and/or source code!

Basic Course on R: Manipulating Data Practical Answers

Karl Brand* and Elizabeth Ribble†

17-21 December 2018

Contents

*brandk@gmail.com
†emcclel3@msudenver.edu

1 Manipulating / Selecting Data

1.1 Answer the following without typing the commands into R. Use ? if you're not sure what the object is or what the function does.

1.1.1 What is

```
length(letters)
```

26

1.1.2 What is

```
length(letters == LETTERS)
```

26

1.1.3 What is

```
which(letters %in% c("a", "d"))
```

1, 4

1.1.4 What is

```
which(c("a", 7, "d") %in% letters)
```

1, 3

1.1.5 What is

```
letters[LETTERS > "W"]
```

x, y, z

1.1.6 What is

```
letters[!LETTERS > "C"]
```

a, b, c

1.1.7 What is

```
seq(from = 1, to = 20, by = 3)
```

1, 4, 7, 10, 13, 16, 19

1.1.8 Why is `x` filled in the way it is? Hint: read about the arguments for `matrix`!

```
x <- matrix(8:11, nrow = 6, ncol = 4)
x
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    8   10    8   10
## [2,]    9   11    9   11
## [3,]   10    8   10    8
## [4,]   11    9   11    9
## [5,]    8   10    8   10
## [6,]    9   11    9   11
```

`x` looks this way because we defined the number of rows to be 6 and the number of columns to be 4, the default `byrow` is FALSE so the array is filled columnwise, and the numbers `8:11` are repeated until the array is filled.

1.1.9 What are

```
x + 4
x + x
2 * x
x / c(2, 3, 4, 5)
x[, 3] + 2 * x[, 2]
nrow(x)
x[x[, 3] > 10, ]
```

- 4 added to each value of `x`
- 2 multiplied by each value of `x`
- 2 multiplied by each value of `x`
- columns of x are divided by the vector c(2, 3, 4, 5); the vector is recycled so that the first and third columns are still the same and the second and fourth columns are still the same
- a single vector with values from the third column of x added to two times the second column of x

- 6
- the 4th row of x, since it's the only row where the third column is greater than 10

1.2 Use R to answer the following.

1.2.1 Create a vector (using `c()`) called `a` (i.e. assign it to an object called `a`) with four elements which are the integers 5 to 8 (inclusive).

```
a <- 5:8
```

1.2.2 Display element 2 of `a`.

```
a[2]

## [1] 6
```

1.2.3 Display element 4 of `a`.

```
a[4]

## [1] 8
```

1.2.4 Calculate the product of elements 2 and 4 of `a`.

```
a[2] * a[4]

## [1] 48
```

1.2.5 Assign the integers 3 and 4 to object `b` and use `b` to select elements 3 and 4 of object `a`.

```
b <- c(3, 4)
# or
b <- 3:4
a[b]

## [1] 7 8
```

1.2.6 Display every element of `a` except element 2.

```
a[-2]

## [1] 5 7 8
```

1.2.7 Display every element of `a` except elements 3 and 4.

```
a[-c(3:4)]

## [1] 5 6
```

1.2.8 Display only those elements of `a` that are greater than or equal to 6.

```
a[a >= 6]

## [1] 6 7 8
```

1.2.9 Display only those elements of `a` that are not equal to 7.

```
a[a != 7]

## [1] 5 6 8
```

1.2.10 Use the `list` function to create an object `ab` which is a list of the two objects `a` and `b`.

```
ab <- list(a, b)
```

1.2.11 Display `ab`.

```
ab

## [[1]]
## [1] 5 6 7 8
##
## [[2]]
## [1] 3 4
```

1.2.12 Change the names of the elements in `ab` to "a" and "b".

```
names(ab) <- c("a", "b")
```

1.2.13 Display `ab` again. What has changed?

```
ab

## $a
## [1] 5 6 7 8
##
## $b
## [1] 3 4
```

Displaying `ab` now shows `$a` and `$b` in place of the original `[[1]]` and `[[2]]`. This indicates how to select and subset the list when we have names versus having no names.

1.2.14 Create this matrix `m`:

```
m <- matrix(1:9, nrow=3, byrow=T)
m

##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
```

Why are the numbers 1, 2, and 3 in the first row and not the first column?

Because we set the argument `byrow` equal to TRUE.

1.2.15 Display the element on the second row and second column of `m`.

```
m[2, 2]

## [1] 5
```

1.2.16 Display only the 2nd row of `m`.

```
m[2, ]

## [1] 4 5 6
```

1.2.17 Display only the 3rd column of `m`.

```
m[, 3]

## [1] 3 6 9
```

1.2.18 Display only the 2nd and 3rd columns of `m`. Do so in two different ways.

```
m[, 2:3]

##      [,1] [,2]
## [1,]    2    3
## [2,]    5    6
## [3,]    8    9

m[, -1]

##      [,1] [,2]
## [1,]    2    3
## [2,]    5    6
## [3,]    8    9
```

If you want to save your work: save your R session and/or source code!

Basic Course on R: Entering and Importing Data

Karl Brand* and Elizabeth Ribble†

17-21 December 2018

Contents

*brandk@gmail.com
†emcclel3@msudenver.edu

1 Entering and Importing Data

1.1 Entering Data

In the previous section we created a couple of short vectors, a small matrix, a data frame, and a list. There are many ways to create such objects, especially by utilizing a few convenient functions, and many combinations of these data types are also possible.

1.1.1 A few functions to get started

The most basic function presented first is one we've seen before: the combine function: `c`. This combines into a vector the entries passed to it, so long as they are of the same class:

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

## [1] 1 2 3 4

a <- c(1:10, 100)
a

##  [1]   1   2   3   4   5   6   7   8   9  10 100
```

If we want to replicate numbers in the vector, we can type them out (e.g. `1, 1, 1, ...`) or use the function `rep`, with the number to be replicated followed by the number of times to replicate it:

```
b <- rep(x = 1, times = 11)
b

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

The function `seq` generates a sequence of numbers based on the specified start, end and increment of the sequence:

```
d <- seq(from = 1, to = 110, by = 10)
d

##  [1]   1  11  21  31  41  51  61  71  81  91 101
```

To bind two or more vectors (or data frames) together, we can use `cbind` (to combine columns) or `rbind` (to combine rows):

```
abdc <- cbind(a, b, d)
abdc
```

```
##         a b   d
##  [1,]   1 1   1
##  [2,]   2 1  11
##  [3,]   3 1  21
##  [4,]   4 1  31
##  [5,]   5 1  41
##  [6,]   6 1  51
##  [7,]   7 1  61
##  [8,]   8 1  71
##  [9,]   9 1  81
## [10,]  10 1  91
## [11,] 100 1 101
```

```
abdr <- rbind(a, b, d)
abdr
```

```
##   [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11]
## a    1    2    3    4    5    6    7    8    9    10   100
## b    1    1    1    1    1    1    1    1    1     1     1
## d    1   11   21   31   41   51   61   71   81    91   101
```

We can use `View` to look at the data - it displays the data values along with the names of the columns:

```
View(abdr)
```

or, to just get the dimension of the resulting object, use `dim`, `nrow` or `ncol`:

```
dim(abdr)
```

```
## [1]  3 11
```

```
nrow(abdr)
```

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

```
ncol(abdr)
```

```
## [1] 11
```

And remember `str()` will display the data structure and `class()` the class:

```
str(abdr)

##  num [1:3, 1:11] 1 1 1 2 1 11 3 1 21 4 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : chr [1:3] "a" "b" "d"
##   ..$ : NULL

class(abdr)

## [1] "matrix"
```

1.1.2 Merging data frames

Suppose we want to add some new samples to `mygenes`:

```
mygenes <- data.frame(samp1 = c(33, 22, 12),
                      samp2 = c(44, 111, 13),
                      samp3 = c(33, 53, 65))
row.names(mygenes) <- c("CRP", "BRCA1", "HOXA")
```

but we happen to have the rows (here, genes) in a different order than before, and in fact we have an extra gene as well:

```
newsamp <- data.frame(samp4 = c(56, 13, 106, 10),
                      samp5 = c(45, 15, 99, 13))
row.names(newsamp) <- c("CRP", "HOXA", "BRCA1", "GAPDH")
newsamp

##       samp4 samp5
## CRP      56    45
## HOXA     13    15
## BRCA1   106    99
## GAPDH    10    13
```

We can't use e.g. `cbind` because not only are the dimensions different, but the row order differs. The `merge` function solves this problem: it concatenates based on matching attributes. Use the argument `by` to specify what columns to merge by, i.e., the name(s) of a column(s), or the row names:

```
mygenes2 <- merge(x = mygenes, y = newsamp, by = "row.names")
mygenes2

##   Row.names samp1 samp2 samp3 samp4 samp5
## 1     BRCA1    22   111    53   106    99
## 2       CRP    33    44    33    56    45
## 3      HOXA    12    13    65    13    15
```

The default setting however is to not include anything that doesn't match, so above we lost our new gene, GAPDH. We can force the function to keep all rows of the first and/or second argument with the arguments `all`, `all.x`, or `all.y`:

```
mygenes2 <- merge(x = mygenes, y = newsamp, by = "row.names", all.y = TRUE)
mygenes2

##   Row.names samp1 samp2 samp3 samp4 samp5
## 1     BRCA1    22   111    53   106    99
## 2       CRP    33    44    33    56    45
## 3     GAPDH    NA    NA    NA    10    13
## 4      HOXA    12    13    65    13    15
```

To be conservative and keep everything, use `all = TRUE`. Note that for every column of the first argument that didn't have a value for GAPDH we now have an `NA`. Also note that the row names are no longer the row names, but instead have been put in a new column. We can leave it as it is, and perhaps change the name (as shown in the first line of code below), or use our indexing and naming skills to create another data frame with the row names back where they were:

```
names(mygenes2)[1] <- "gene"
## or
mygenes3 <- mygenes2[, -1]
row.names(mygenes3) <- mygenes2[, 1]
mygenes3

##       samp1 samp2 samp3 samp4 samp5
## BRCA1    22   111    53   106    99
## CRP      33    44    33    56    45
## GAPDH    NA    NA    NA    10    13
## HOXA     12    13    65    13    15
```

1.1.3 Concatenating strings

The function `paste` is great for (re)naming, writing, and creating data. For example, in the last section we created the data frame `mygenes` by entering the name of each data

frame element as follows:

```
mygenes <- data.frame(samp1 = c(33, 22, 12),
                      samp2 = c(44, 111, 13),
                      samp3 = c(33, 53, 65))
```

but with a naming convention where the same text is repeated several times ("samp") it is handy to be able to type it in only once:

```
mygenes <- data.frame(c(33, 22, 12),
                      c(44, 111, 13),
                      c(33, 53, 65))
names(mygenes) <- paste("samp", 1:3, sep = "")
row.names(mygenes) <- c("CRP", "BRCA1", "HOXA")
mygenes

##       samp1 samp2 samp3
## CRP      33    44    33
## BRCA1    22   111    53
## HOXA     12    13    65
```

Note the `sep` argument specifies how to separate the strings we are pasting together. We can use any character we want for this parameter:

```
paste("The", "Club", sep = " Mickey Mouse ")

## [1] "The Mickey Mouse Club"
```

By default `sep = " "`. That is, omitting `sep` in the `paste` function will insert spaces:

```
paste("The", "Mickey", "Mouse", "Club")

## [1] "The Mickey Mouse Club"
```

In the next subsection we'll see how paste can be helpful for reading and writing files.

1.2 Importing from a Dataset or File

1.2.1 Importing from a built-in dataset

Many `R` packages come with built-in datasets. Extracting these data are straightforward using `data()`. Invoking this will output a list of all available datasets in a new window:

```
data()
```

If we introduce the name of the dataset, then the function loads that dataset into the workspace:

```
HairEyeColor
```

```
## , , Sex = Male
##
##        Eye
## Hair    Brown Blue Hazel Green
##   Black    32   11    10     3
##   Brown    53   50    25    15
##   Red      10   10     7     7
##   Blond     3   30     5     8
##
## , , Sex = Female
##
##        Eye
## Hair    Brown Blue Hazel Green
##   Black    36    9     5     2
##   Brown    66   34    29    14
##   Red      16    7     7     7
##   Blond     4   64     5     8
```

1.2.2 Importing from a file

Suppose we have collected some data that we want to analyze in `R` and the data are in matrix format in e.g. a .txt or .csv file. Then we can read this in with a function like `read.table` and assign the output to an object:

```
data <- read.table(file = "Rcourse_data.txt",
                   header = TRUE, row.names = 1)
data
```

```
##       cohort age
## 1001 disease  34
## 1002 control  38
## 1003 disease  22
## 1004 disease  50
## 1005 control  46
## 1006 control  27
```

```
## 1007 disease  44
## 1008 control  49
## 1009 control  33
## 1010 disease  30
```

In the arguments we can specify things like:

- the separator e.g., `sep = "\t"` for tab separated (.txt) or `sep = ","` for comma separated (.csv) files;
- the header (as above);
- which column contains the row names (as above);
- whether to skip any columns or fill in blank lines with `NA`;

and so on. The function `read.csv` is the same as `read.table` but uses the default `sep = ","` for reading in .csv files (note that `read.csv2` has default `sep = ";"` and `dec = ","`).

Also be aware of R's default behaviour of converting columns of text strings in a dataframe into the class `factor`. You may not want your column of text strings to be factors, if so, set the argument `stringsAsFactors = FALSE` in the `read.table()` function.

Now suppose the file we want isn't in our working directory (displayed by calling `getwd()`). Then we can type in the path in the name of the file:

```
data <- read.table(file = "C:/temp/Rcourse_data.txt",
                   header = TRUE,
                   row.names = 1)
data
```

But let's say we have a lot of files to read in from places other than our working directory and don't want to keep typing in the same parts of a path every time. Then we can `paste`:

```
mydir <- "C:/temp"
newfile <- paste(mydir, "Rcourse_data.txt", sep = "/")
data <- read.table(file = newfile,
                   header = TRUE,
                   row.names = 1)
```

See also `?file.path` which is built specifically for this purpose, whereas `paste` has more general utility.

1.3 Writing to a File

Now suppose we've done some analysis and want to store our results outside of R. The functions `write.table` and `write.csv` do this:

```
write.table(x = data, file = "C:/temp/results.txt", sep = "\t")
## or
write.csv(x = data, file = "C:/temp/results.csv")
```

We specified the separator in `write.table` (default is a space), but the default in `write.csv` is a comma. If you do not want to write the row names to the file, set the argument `row.names = FALSE`.

Note we can change the location of our working directory using `setwd()` so that we don't have to worry about specifying a path over and over when reading/writing files, e.g.:

```
# not run
setwd("C:/Users/Documents/RSTUFF")
```

2 Document License

Basic Course on R: Entering and Importing Data Practical

Karl Brand* and Elizabeth Ribble†

17-21 December 2018

Contents

*brandk@gmail.com
†emcclel3@msudenver.edu

1 Entering and Importing Data

1.1 Use R to do the following exercises.

1.1.1 Enter the following into a data structure with the name `color`:

- purple
- yellow
- red
- brown
- green
- purple
- red
- purple

1.1.2 Display the 2nd element of `color`.

1.1.3 Enter the following into a data structure with the name `weight`:

- 23
- 21
- 18
- 26
- 25
- 22
- 26
- 19

1.1.4 What are the lengths of `color` and `weight`? Use a function to answer this.

1.1.5 Join `color` and `weight` together using `cbind()` and assign it to the object `micecbind`.

1.1.6 What is the data structure of `micecbind`? What are the dimensions? Are the weights still numbers (`num`) or were they converted to characters (`chr`)?

1.1.7 Now join `color` and `weight` together using `data.frame()` with argument `stringsAsFactors = FALSE` and assign it to the object `micedf`.

1.1.8 What is the data structure of `micedf`? What are the dimensions? Are the weights still numbers (`num`) or were they converted to characters (`chr`)?

1.1.9 Display only the 3rd row of `micedf`.

1.1.10 Display only the 2nd column ("weight") of `micedf`. Do so in two different ways.

1.1.11 Display the dimensions of `micedf`.

1.1.12 Assign the following strings to the row names of `micedf`:

- mouse1
- mouse2
- mouse3
- mouse4
- mouse5
- mouse6
- mouse7
- mouse8

Hint: try using `paste`.

1.1.13 Create a list containing three elements and assign it to `mylist`:

- `micedf`
- A data frame of `micedf` with a new column called `double` that is 2 times the second column of `micedf` (`weight`).
- The names of `micedf`.

1.1.14 Assign these names to `mylist`: `first`, `second`, `third`.

1.1.15 Display `mylist`. What does it look like?

1.1.16 Display only the second element of `mylist`. Do so in two different ways.

1.1.17 Write `micedf` to a file called "micedf1.csv".

1.1.18 Open "micedf1.csv" in Excel and describe what you see. Repeat the step above but do not include row names and call the file "micedf2.csv". How is the output different from "micedf1.csv"?

1.1.19 Now read in "micedf1.csv" and "micedf2.csv" into R in two new objects (`newmice1` and `newmice2`, respectively). Describe any differences between the two objects. What are the dimensions of each object?

1.1.20 Read in "micedf1.csv" into R (assign to object `newmice3`). Use the argument `row.names` to indicate that the first column should be row names and do not allow strings to be turned into factors. Display the object and the structure of the object and describe how it is different from `newmice1` and `micedf`. What are the dimensions of `newmice3`?

1.2 Use R to do the following exercises on the `Puromycin` data.

1.2.1 Load the `Puromycin` data using the `data()` function.

1.2.2 What is the data structure of `Puromycin`? What are the dimensions (use a function other than `str`)?

1.2.3 What are the names of `Puromycin`? Use a function other than `str`.

1.2.4 What are the levels of the `state` variable? Use a function other than `str`.

1.2.5 Display the rate for all concentrations less than 0.10 in the treated group.

1.2.6 What are the row indices for the concentrations of 0.22?

If you want to save your work: save your R session and/or source code!

Basic Course on R: Entering and Importing Data Practical Answers

Karl Brand* and Elizabeth Ribble†

17-21 December 2018

Contents

*brandk@gmail.com
†emcclel3@msudenver.edu

1 Entering and Importing Data

1.1 Use R to do the following exercises.

1.1.1 Enter the following into a data structure with the name `color`:

- purple
- yellow
- red
- brown
- green
- purple
- red
- purple

```
color <- c("purple", "yellow", "red", "brown", "green",
           "purple", "red", "purple")
```

1.1.2 Display the 2nd element of `color`.

```
color[2]

## [1] "yellow"
```

1.1.3 Enter the following into a data structure with the name `weight`:

- 23
- 21
- 18
- 26
- 25
- 22
- 26
- 19

```
weight <- c(23, 21, 18, 26, 25, 22, 26, 19)
```

1.1.4 What are the lengths of `color` and `weight`? Use a function to answer this.

```
length(color)

## [1] 8

length(weight)

## [1] 8
```

1.1.5 Join `color` and `weight` together using `cbind()` and assign it to the object `micecbind`.

```
micecbind <- cbind(color = color, weight = weight)
```

1.1.6 What is the data structure of `micecbind`? What are the dimensions? Are the weights still numbers (`num`) or were they converted to characters (`chr`)?

```
micecbind

##      color    weight
## [1,] "purple" "23"
## [2,] "yellow" "21"
## [3,] "red"    "18"
## [4,] "brown"  "26"
## [5,] "green"  "25"
## [6,] "purple" "22"
## [7,] "red"    "26"
## [8,] "purple" "19"

str(micecbind)

##  chr [1:8, 1:2] "purple" "yellow" "red" "brown" "green" "purple" "red" ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : NULL
##   ..$ : chr [1:2] "color" "weight"

dim(micecbind)

## [1] 8 2
```

Using `str` we see that `micecbind` is a matrix with dimensions 8 x 2 and the weights were converted to characters, which we don't want! The function `dim` can also be used to extract only the dimensions.

1.1.7 Now join `color` and `weight` together using `data.frame()` with argument `stringsAsFactors = FALSE` and assign it to the object `micedf`.

```
micedf <- data.frame(color = color, weight = weight, stringsAsFactors = FALSE)
```

1.1.8 What is the data structure of `micedf`? What are the dimensions? Are the weights still numbers (`num`) or were they converted to characters (`chr`)?

```
micedf
```

```
##    color weight
## 1 purple     23
## 2 yellow     21
## 3    red     18
## 4  brown     26
## 5  green     25
## 6 purple     22
## 7    red     26
## 8 purple     19
```

```
str(micedf)
```

```
## 'data.frame': 8 obs. of  2 variables:
##  $ color : chr  "purple" "yellow" "red" "brown" ...
##  $ weight: num  23 21 18 26 25 22 26 19
```

```
dim(micedf)
```

```
## [1] 8 2
```

Using `str` we see that `micedf` is a data frame with dimensions 8 x 2, with variable `color` a character vector and `weight` a numeric vector. The function `dim` can also be used to extract only the dimensions. Thankfully the weights were not converted to characters!

1.1.9 Display only the 3rd row of `micedf`.

```
micedf[3, ]

##   color weight
## 3   red     18
```

1.1.10 Display only the 2nd column ("weight") of `micedf`. Do so in two different ways. Any two of the following are satisfactory:

```
micedf[, 2]
micedf[, -1]
micedf[, "weight"]
micedf[[2]]
micedf$weight
micedf["weight"]
```

1.1.11 Display the dimensions of `micedf`.

```
dim(micedf)

## [1] 8 2
```

1.1.12 Assign the following strings to the row names of `micedf`:

- mouse1
- mouse2
- mouse3
- mouse4
- mouse5
- mouse6
- mouse7
- mouse8

Hint: try using `paste`.

```
row.names(micedf) <- paste("mouse", 1:8, sep = "")
```

1.1.13 Create a list containing three elements and assign it to `mylist`:

- `micedf`
- A data frame of `micedf` with a new column called `double` that is 2 times the second column of `micedf` (`weight`). (Did you get an error? Make sure that the second column is numeric and if it isn't, change it!)
- The names of `micedf`.

```
mylist <- list(micedf,
               data.frame(micedf, double = 2 * micedf$weight),
               names(micedf))
```

1.1.14 Assign these names to `mylist`: `first`, `second`, `third`.

```
names(mylist) <- c("first", "second", "third")
```

1.1.15 Display `mylist`. What does it look like?

```
mylist
```

```
## $first
##         color weight
## mouse1 purple     23
## mouse2 yellow     21
## mouse3    red     18
## mouse4  brown     26
## mouse5  green     25
## mouse6 purple     22
## mouse7    red     26
## mouse8 purple     19
##
## $second
##         color weight double
## mouse1 purple     23     46
## mouse2 yellow     21     42
## mouse3    red     18     36
## mouse4  brown     26     52
```

```
## mouse5  green     25     50
## mouse6 purple     22     44
## mouse7    red     26     52
## mouse8 purple     19     38
##
## $third
## [1] "color"  "weight"
```

A list with three items with the given names preceded by `$`. The first two entries are data frames and have row names, but the third does not (it is a character vector).

1.1.16 Display only the second element of `mylist`. Do so in two different ways.

Any two of the following are satisfactory:

```
mylist[[2]] ## extract
mylist$second ## extract
mylist[2] ## subset
mylist[-c(1, 3)] ## subset
mylist["second"] ## subset
```

1.1.17 Write `micedf` to a file called "micedf1.csv" in the course directory.

```
write.csv(micedf, "micedf1.csv")
```

1.1.18 Open "micedf1.csv" in Excel and describe what you see. Repeat the step above but do not include row names and call the file "micedf2.csv". How is the output different from "micedf1.csv"?

The file "micedf1.csv" looks exactly like the data frame when displayed in R.

```
write.csv(micedf, "micedf2.csv", row.names = FALSE)
```

The file "micedf2.csv" looks just like the data frame above, but it is missing the row names.

1.1.19 Now read in "micedf1.csv" and "micedf2.csv" into R in two new objects (`newmice1` and `newmice2`, respectively). Describe any differences between the two objects. What are the dimensions of each object?

```
newmice1 <- read.csv("micedf1.csv")
newmice2 <- read.csv("micedf2.csv")
str(newmice1)

## 'data.frame': 8 obs. of  3 variables:
##  $ X     : Factor w/ 8 levels "mouse1","mouse2",..: 1 2 3 4 5 6 7 8
##  $ color : Factor w/ 5 levels "brown","green",..: 3 5 4 1 2 3 4 3
##  $ weight: int  23 21 18 26 25 22 26 19

str(newmice2)

## 'data.frame': 8 obs. of  2 variables:
##  $ color : Factor w/ 5 levels "brown","green",..: 3 5 4 1 2 3 4 3
##  $ weight: int  23 21 18 26 25 22 26 19

newmice1

##        X  color weight
## 1 mouse1 purple     23
## 2 mouse2 yellow     21
## 3 mouse3    red     18
## 4 mouse4  brown     26
## 5 mouse5  green     25
## 6 mouse6 purple     22
## 7 mouse7    red     26
## 8 mouse8 purple     19

newmice2

##    color weight
## 1 purple     23
## 2 yellow     21
## 3    red     18
## 4  brown     26
## 5  green     25
## 6 purple     22
## 7    red     26
## 8 purple     19
```

The first file was written with row names, but when we read it into R, the row

names are now just a column in the data frame. Because there was an empty cell in the space where a column name should have been, R names it `X`. The second file had no row names, and, just like the first file, is given the default row names of 1 to the number of rows. The dimensions are 8 x 3 and 8 x 2, respectively (seen in the output of the `str` call).

1.1.20 Read in "micedf1.csv" into R (assign to object `newmice3`). Use the argument `row.names` to indicate that the first column should be row names and do not allow strings to be turned into factors. Display the object and the structure of the object and describe how it is different from `newmice1` and `micedf`. What are the dimensions of `newmice3`?

```
newmice3 <- read.csv(paste(mydir, "micedf1.csv", sep="/"),
                     row.names = 1, stringsAsFactors = F)
```

```
str(newmice3)

## 'data.frame': 8 obs. of  2 variables:
##  $ color : chr  "purple" "yellow" "red" "brown" ...
##  $ weight: int  23 21 18 26 25 22 26 19

newmice3

##         color weight
## mouse1 purple     23
## mouse2 yellow     21
## mouse3    red     18
## mouse4  brown     26
## mouse5  green     25
## mouse6 purple     22
## mouse7    red     26
## mouse8 purple     19
```

The new object `newmice3` now has row names where in `newmice1` they were treated as a variable in the data frame. The `color` variable was not converted to a factor, unlike in `newmice1`. The new object `micedf` is now (nearly) identical to the original `micedf`. It has the same row names and dimensions as `micedf`. The only difference is the class of `weight` is an integer instead of just "numeric" (more specific). The dimensions are 8 x 2.

1.2 Use R to do the following exercises on the `Puromycin` data.

1.2.1 Load the `Puromycin` data using the `data()` function.

```
data(Puromycin)
```

1.2.2 What is the data structure of `Puromycin`? What are the dimensions? Do not just display the data (this is not convenient for large datasets).

```
str(Puromycin)

## 'data.frame': 23 obs. of  3 variables:
##  $ conc : num  0.02 0.02 0.06 0.06 0.11 0.11 0.22 0.22 0.56 0.56 ...
##  $ rate : num  76 47 97 107 123 139 159 152 191 201 ...
##  $ state: Factor w/ 2 levels "treated","untreated": 1 1 1 1 1 1 1 1 1 1 ...
##  - attr(*, "reference")= chr "A1.3, p. 269"
```

Using `str` we see that `Puromycin` is a data frame with dimensions 23 x 3, with variables `conc` and `rate` numeric vectors and `state` a factor with 2 levels.

1.2.3 What are the names of `Puromycin`? Use a function other than `str`.

```
names(Puromycin)

## [1] "conc"  "rate"  "state"
```

1.2.4 What are the levels of the `state` variable? Use a function other than `str`.

```
levels(Puromycin$state)

## [1] "treated"   "untreated"
```

1.2.5 Display the rate for all concentrations less than 0.10 in the treated group.

```
Puromycin[Puromycin$conc < .10 & Puromycin$state == "treated", "rate"]

## [1]  76  47  97 107

## or
Puromycin$rate[Puromycin$conc < .10 & Puromycin$state == "treated"]
```

```
## [1]  76  47  97 107

## or
Puromycin[["rate"]][Puromycin$conc < .10 & Puromycin$state == "treated"]

## [1]  76  47  97 107
```

1.2.6 What are the row indices for the concentrations of 0.22?

```
which(Puromycin$conc == 0.22)

## [1]  7  8 19 20
```

If you want to save your work: save your R session and/or source code!

Basic Course on R: Basic Plotting

Karl Brand* and Elizabeth Ribble†

14-18 May 2018

Contents

*brandk@gmail.com
†emcclel3@msudenver.edu

1 Basic Plotting

R is a very powerful tool for producing custom graphics ranging from basic scatterplots to 3D plots. In this introduction to plotting we will learn how to create basic graphs like scatterplots, line graphs and bar charts, as well as histograms and boxplots, all of which are available in the base package. Different methods for saving plots will be introduced. We will use the data `ToothGrowth` throughout the following examples:

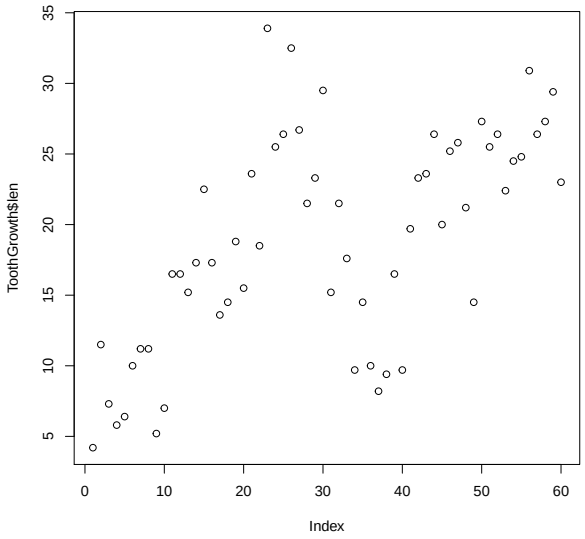
```
data(ToothGrowth)
str(ToothGrowth)

## 'data.frame': 60 obs. of  3 variables:
##  $ len : num  4.2 11.5 7.3 5.8 6.4 10 11.2 11.2 5.2 7 ...
##  $ supp: Factor w/ 2 levels "OJ","VC": 2 2 2 2 2 2 2 2 2 2 ...
##  $ dose: num  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 ...
```

1.1 Scatterplots and Line Graphs

1.1.1 Scatterplots

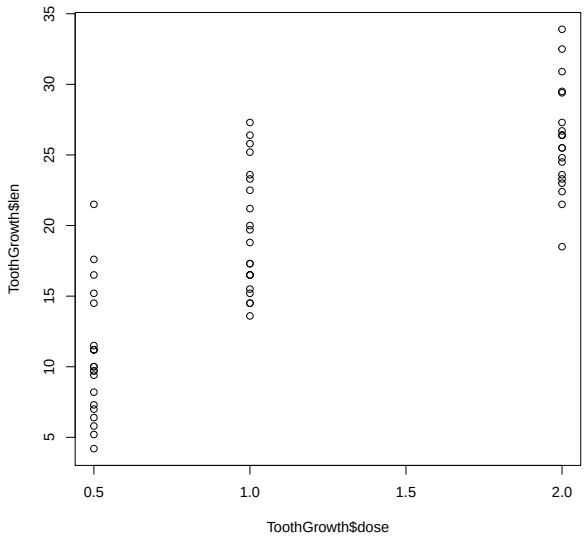
The `plot` function is the most basic function for plotting continuous data. If we use `plot` on one variable, the values of the variable will be plotted against their index, i.e., the order of the data within the object they're stored:

```
plot(ToothGrowth$len)
```



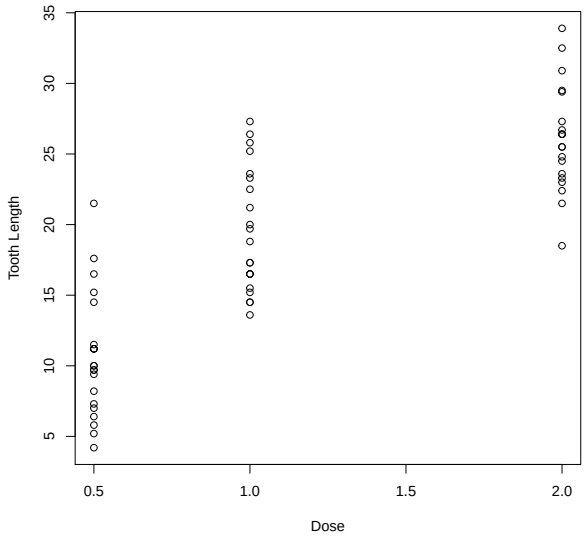
With two variables `plot` puts the first variable on the x-axis and the second variable on the y-axis (y versus x):

```
plot(ToothGrowth$dose, ToothGrowth$len)
```



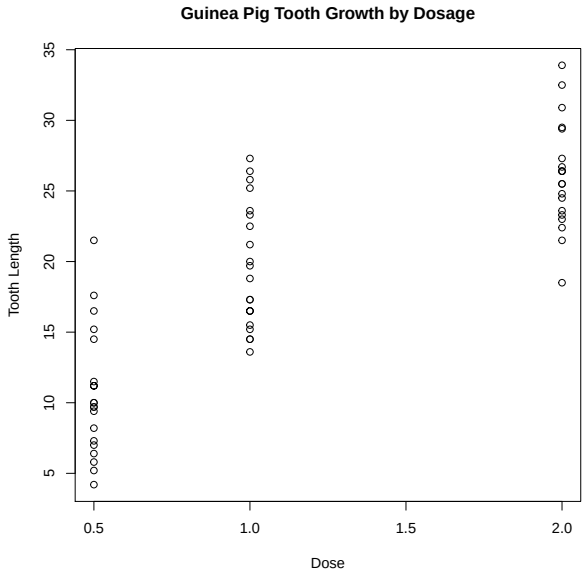
Many parameters are available for customizing your plot. See `?par` for an extensive list. We will just look at a couple here, like changing the axis labels:

```
plot(ToothGrowth$dose, ToothGrowth$len, xlab = "Dose",
     ylab = "Tooth Length")
```



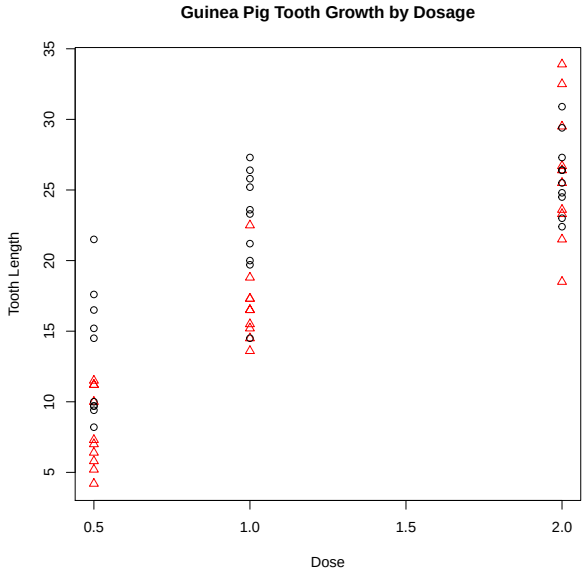
We can also add a title:

```
plot(ToothGrowth$dose, ToothGrowth$len, xlab = "Dose",
     ylab = "Tooth Length", main = "Guinea Pig Tooth Growth by Dosage")
```



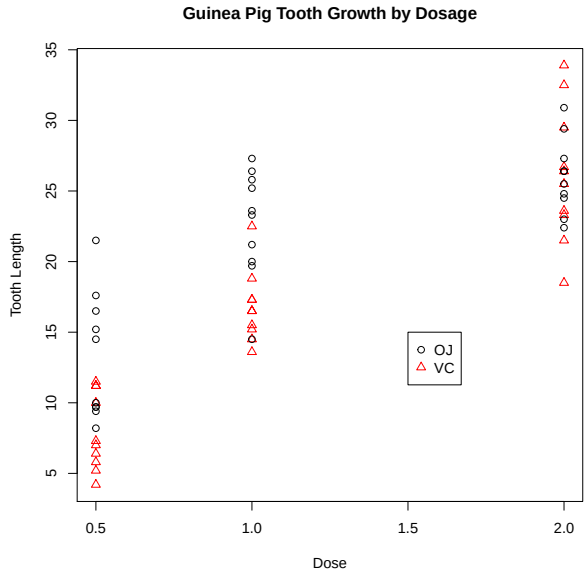
and even change the colors and characters of specific points:

```
plot(ToothGrowth$dose, ToothGrowth$len, xlab = "Dose",
     ylab = "Tooth Length", main = "Guinea Pig Tooth Growth by Dosage",
     col = ToothGrowth$supp, pch = as.numeric(ToothGrowth$supp))
```



The `legend` function adds a legend so we can easily identify which points represent which groups:

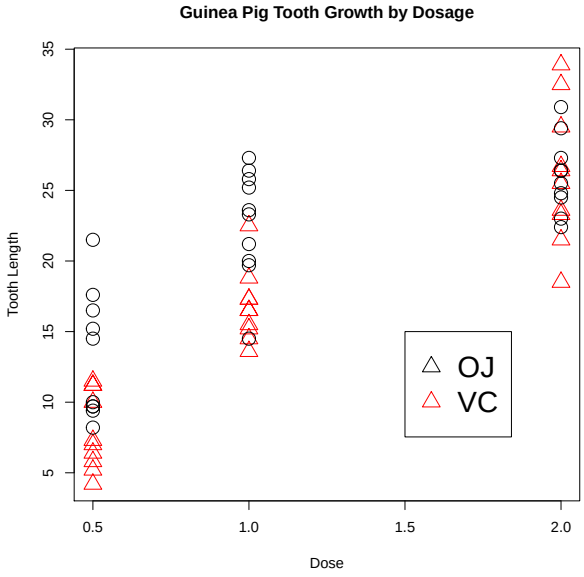
```
plot(ToothGrowth$dose, ToothGrowth$len, xlab = "Dose",
     ylab = "Tooth Length", main = "Guinea Pig Tooth Growth by Dosage",
     col = ToothGrowth$supp, pch = as.numeric(ToothGrowth$supp))
legend(1.5, 15, c("OJ", "VC"), col = 1:2,
       pch = 1:2)
```



The location of the legend can also be specified by stating a region of the plot e.g. `"bottomright"` to place it in the very bottom righthand corner of the plot.

The argument `cex` stands for "character expansion" and will enlarge the size of points:

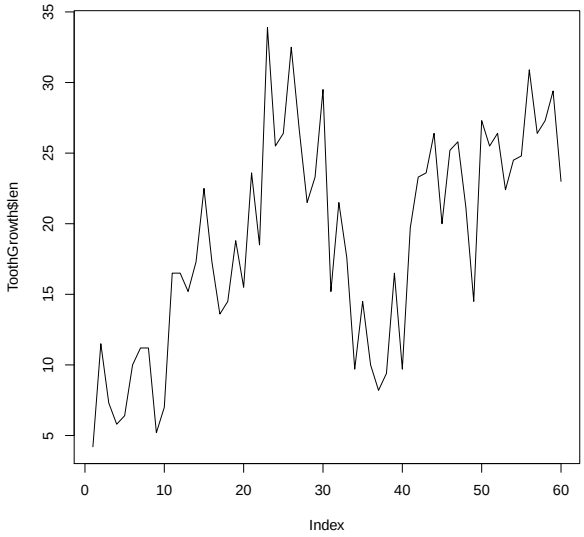
```
plot(ToothGrowth$dose, ToothGrowth$len, xlab = "Dose",
     ylab = "Tooth Length", main = "Guinea Pig Tooth Growth by Dosage",
     col = ToothGrowth$supp, pch = as.numeric(ToothGrowth$supp),
     cex = 2)
legend(1.5, 15, c("OJ", "VC"), col = 1:2,
       pch = as.numeric(ToothGrowth$supp), cex=2)
```



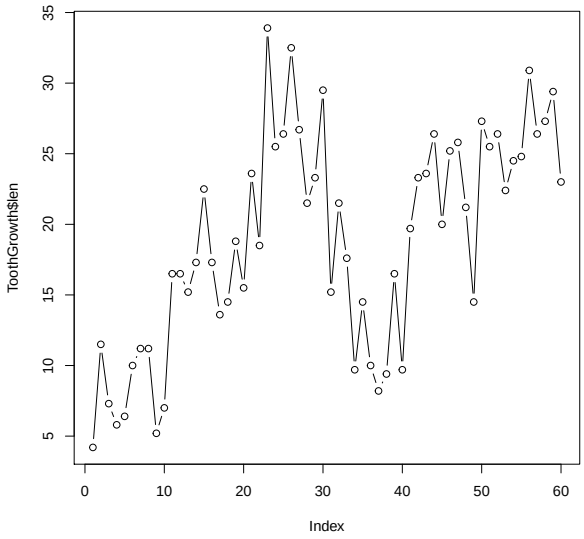
1.1.2 Line Graphs

To create a plot using lines instead of points, we can actually still use `plot`, but we need to specify the type of plot we want, e.g.:

```
plot(ToothGrowth$len, type = "l") # note type is the letter l for "line"
```

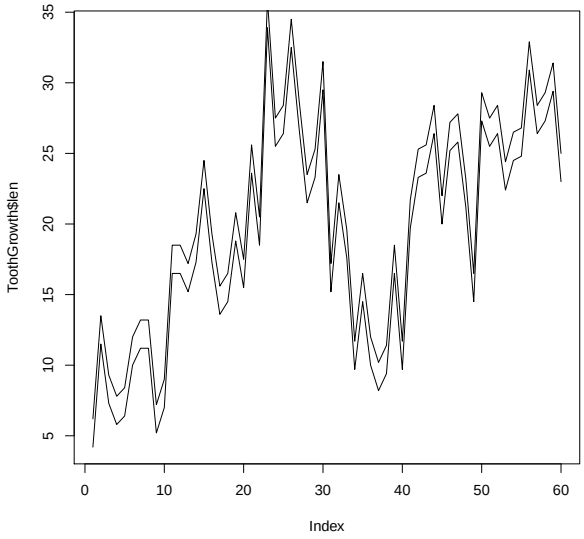


```
plot(ToothGrowth$len, type = "b") # note type is the letter b for "both"
```



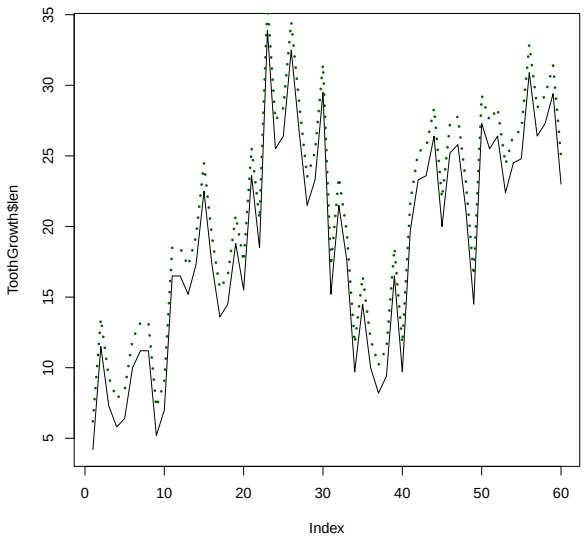
We can add additional lines by calling `lines`:

```
plot(ToothGrowth$len, type = "l")
lines(ToothGrowth$len + 2)
```



The line type, width, and color can be adjusted as follows:

```
plot(ToothGrowth$len, type = "l")
lines(ToothGrowth$len + 2, lty = 3, lwd = 3, col = "darkgreen")
```

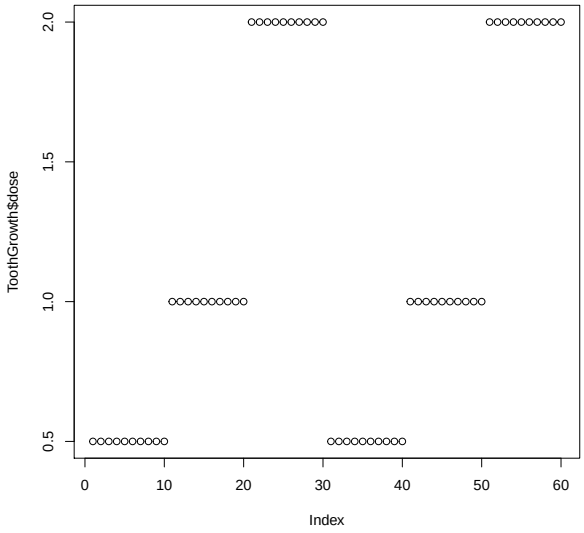


1.2 Bar Charts and Histograms

1.2.1 Bar Charts

Viewing categorical data in a scatterplot often doesn't make sense. For example, if we want to see the number of guinea pigs who received each dosage, the following doesn't provide this information in the most intuitive format:

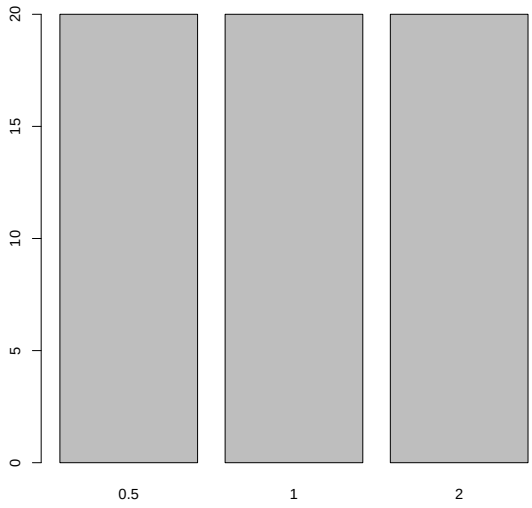
```
plot(ToothGrowth$dose)
```



Instead we can use bar charts to visualize the frequencies or proportions within each category. Note that we have to use a frequency table of the counts (using `table`):

```
table(ToothGrowth$dose)

##
## 0.5   1   2
##  20  20  20
```

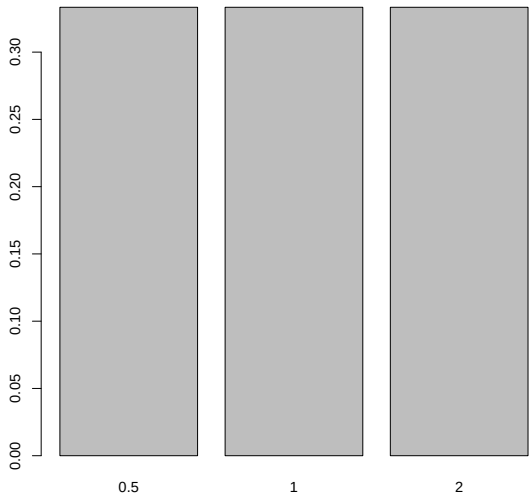
```
barplot(table(ToothGrowth$dose))
```



We can plot proportions instead of frequencies, as is often helpful (though not in this case):

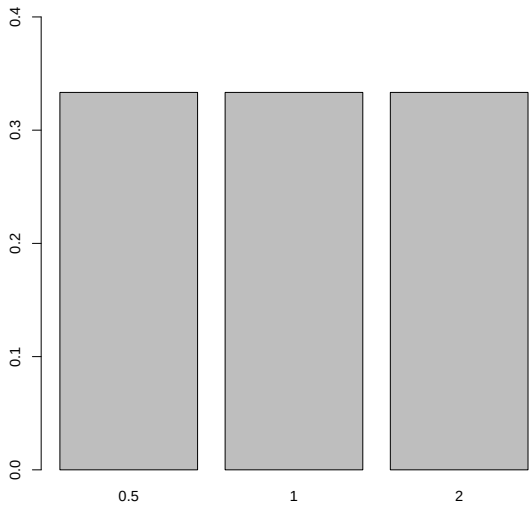
```
props <- table(ToothGrowth$dose)/nrow(ToothGrowth)
props

##
##       0.5         1         2
## 0.3333333 0.3333333 0.3333333
```

```
barplot(props)
```



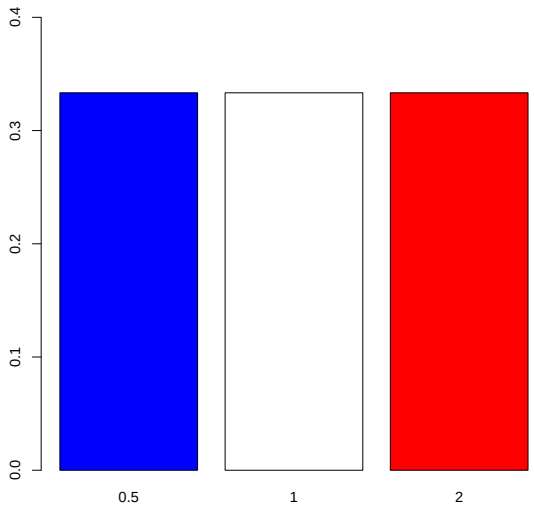
Note that the y-axis doesn't quite extend to the height of the bars. The range of the y-axis is easily change in plots by adjusting the parameter `ylim` (similarly `xlim` for the x-axis):

```
barplot(props, ylim = c(0, .4))
```



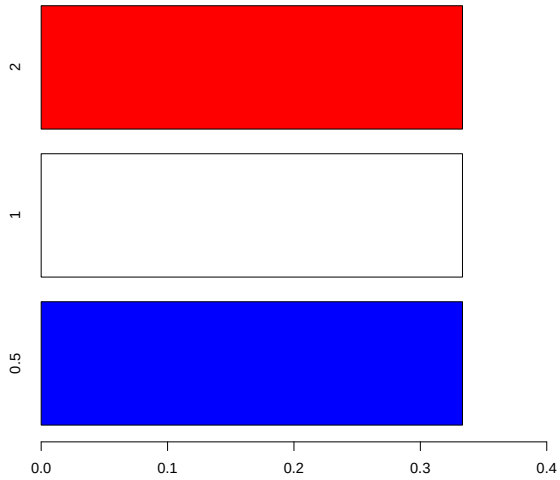
Yet again, there are many parameters available for customizing the plot. Let's try changing the width and colors of the bars:

```
barplot(props, ylim = c(0, .4),
        width = .2, col = c("blue", "white", "red"))
```



And we can plot the bars horizontally, if preferable:

```
barplot(props, xlim = c(0, .4),
        width = .2, col = c("blue", "white", "red"),
        horiz = TRUE)
```

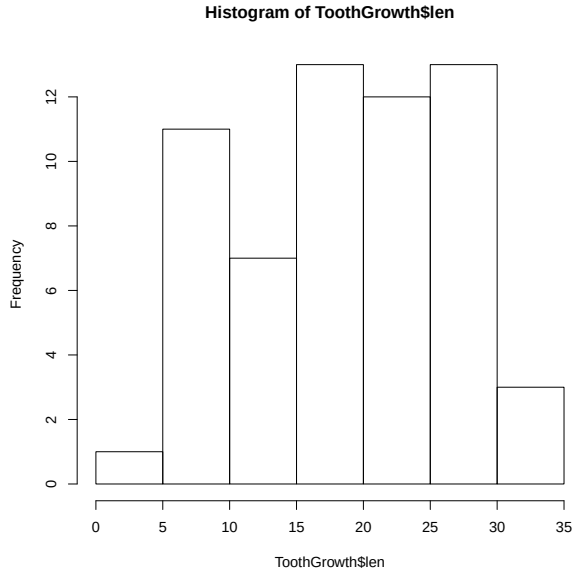


Note that we adjusted the range of the x-axis now instead of the y-axis.

1.2.2 Histograms

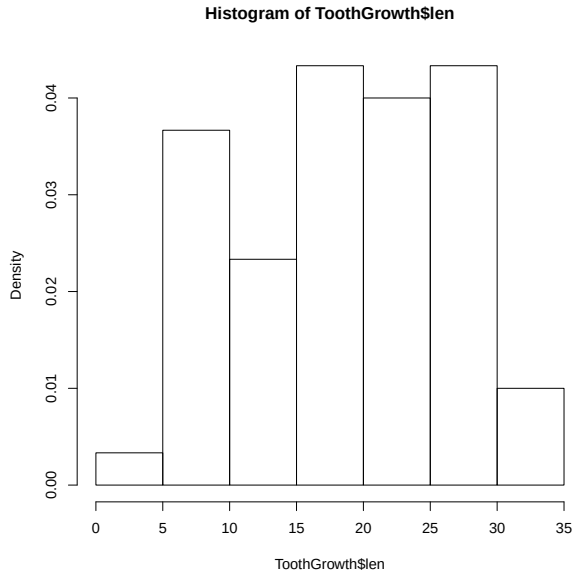
A bar chart is slightly different from a histogram. We use bar charts to plot frequencies (or proportions) of values present in categories. Continuous data don't have categories, so to make a similar plot we have to create "categories". They are often called bins or cells and are created by defining ranges in which the frequencies are calculated. Thankfully, `R` will do this, and plot the histogram, when we call the function `hist`:

```
hist(ToothGrowth$len)
```



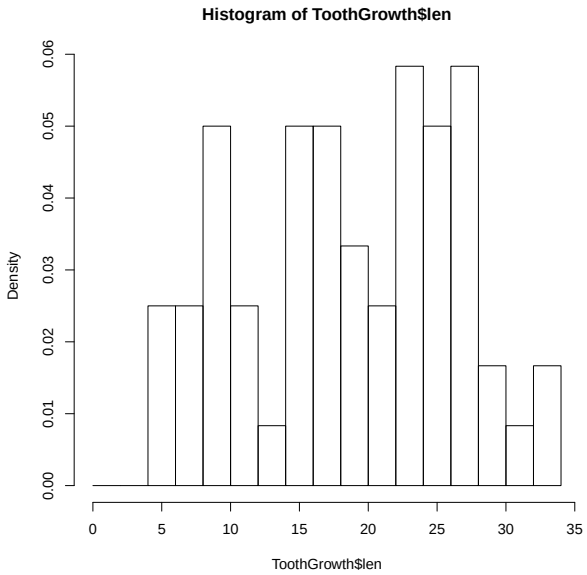
Note that in a histogram the sides of the cells touch, whereas they do not in a bar chart. We can also plot the proportions, now called "density", by changing a single parameter:

```
hist(ToothGrowth$len, freq = FALSE)
```



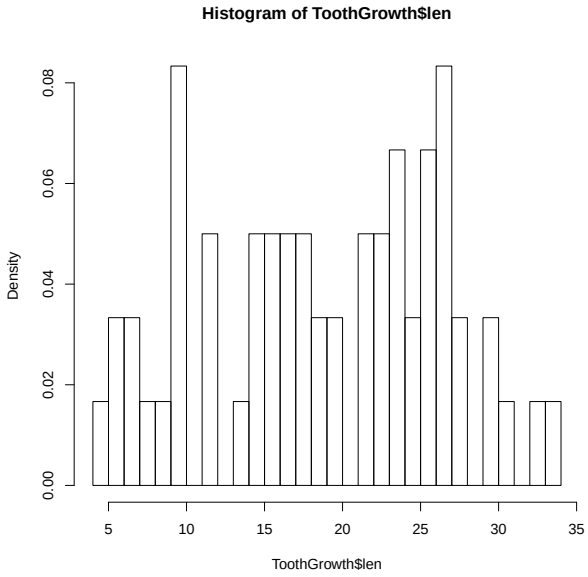
In the next graph we plot the histogram with a different bin width by specifying where to make the breakpoints between the bars:

```
hist(ToothGrowth$len, freq = FALSE, breaks = seq(0, 35, 2))
```



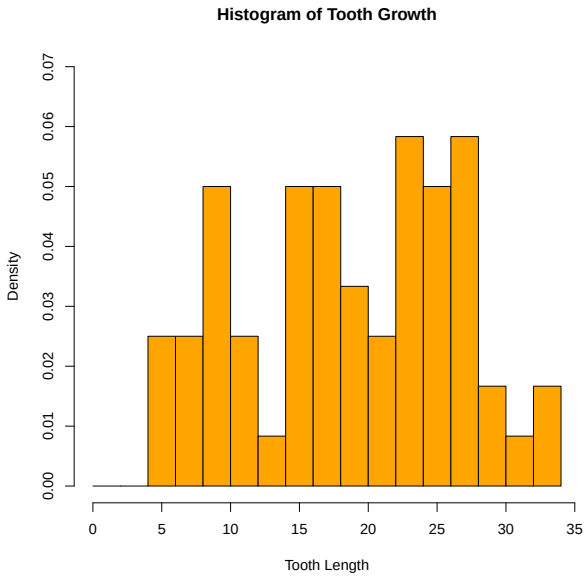
Note how the granularity of the plot changes with the width of the bin. We can also change the breaks by defining the number of cells to use:

```
hist(ToothGrowth$len, freq = FALSE, breaks = 25)
```



The same parameters that change the axes and labels are applicable here:

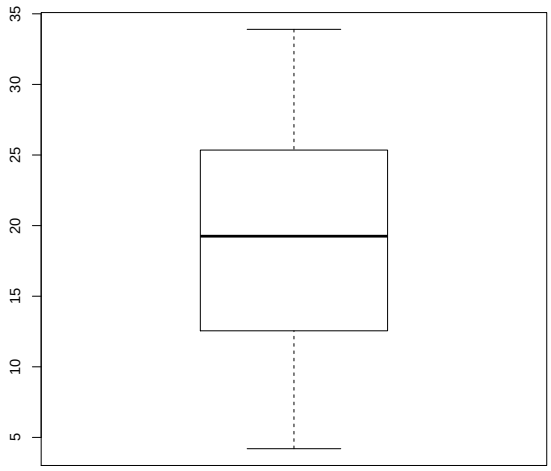
```
hist(ToothGrowth$len, freq = FALSE, breaks = seq(0, 35, 2),
     main = "Histogram of Tooth Growth", xlab = "Tooth Length",
     ylim = c(0, .07), col = "orange")
```



1.3 Boxplots

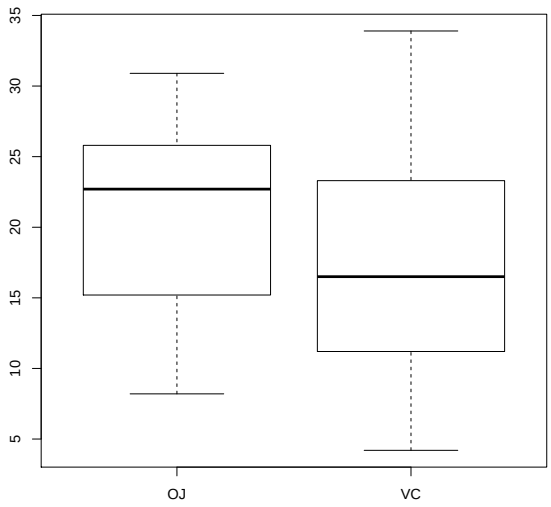
A boxplot is another graph we use to view the distribution of a continuous variable. It displays the specified quantiles of the data in the following way:

```
boxplot(ToothGrowth$len)
```



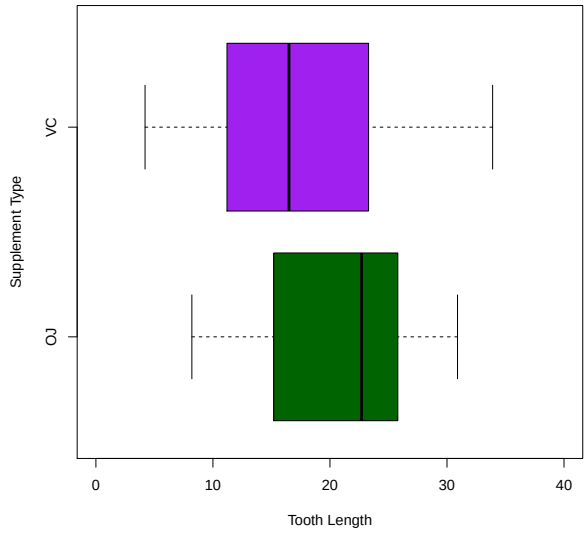
A single boxplot is nice for visualizing the distribution of one variable, but plotting several side by side allows for a simultaneous comparison of distributions. Here we will use the formula notation. The format `y ~ x`, where `y` is a numeric vector and `x` is a factor, tells the boxplot function that we want to separate the continuous `y` values into as many boxplots as there are levels of the factor `x`.

```
boxplot(ToothGrowth$len~ToothGrowth$supp)
```



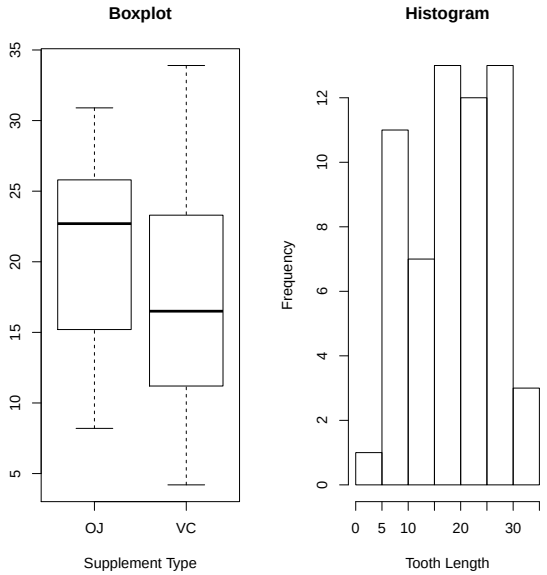
Just like in the bar chart, we can plot the boxes horizontally and change the colors, labels and axes:

```
boxplot(ToothGrowth$len~ToothGrowth$supp,
        col = c("darkgreen", "purple"),
        ylab = "Supplement Type", xlab = "Tooth Length",
        ylim = c(0, 40), horizontal = TRUE)
```



The settings of the parameters of a plot, including the margins, can be specified by calling `par()` before creating a plot. One nice setting we'll introduce here is how to put multiple plots into one graphics device:

```
par(mfrow = c(1, 2))
boxplot(ToothGrowth$len~ToothGrowth$supp, main = "Boxplot",
        xlab = "Supplement Type")
hist(ToothGrowth$len, main = "Histogram", xlab = "Tooth Length")
```



The settings will stay fixed until we close the device (e.g. by calling `dev.off()`) or reset `par` (e.g. `par(mfrow = c(1, 1))`). It is highly recommended to get familiar with the help page for `par` to control graphing parameters to create custom graphs.

Another helpful function is `axis`, which allows the specification of custom axis labels (e.g. where to put them and at what angle). See `?axis` for more information.

1.4 Saving Plots

Up until now we have been creating graphs in a single graphics window that gets overwritten every time we create new plots. If we want to save the images created, we can use "Export" in RStudio to specify what type of file we want to write, its size, location, and name.

Alternatively we can use functions such as `png`, `pdf`, and `jpeg`. Since we can save our code and not our mouse clicks, the use of these functions avoids any confusion about how an image was written, allows for quick simple changes, and provides a convenient way to reproduce multiple, similar plots. For example:

```
# note: run getwd() to see the working directory -
# that is the directory to which files will be written
pdf("plot1.pdf", width = 6, height = 9)
boxplot(ToothGrowth$len~ToothGrowth$supp,
        col = c("darkgreen", "purple"),
        xlab = "Supplement Type", ylab = "Tooth Length",
        ylim = c(0, 40), horizontal = TRUE)
dev.off()

## pdf
##   2

pdf("plot2.pdf", width = 4, height = 8)
boxplot(ToothGrowth$len~ToothGrowth$supp,
        col = c("darkgreen", "purple"),
        xlab = "Supplement Type", ylab = "Tooth Length",
        ylim = c(0, 40), horizontal = TRUE)
dev.off()

## pdf
##   2
```

Note that you have to call `dev.off()` to finish writing the image to the file. If more than one device is open, the return of that function will display a number larger than 1.

The units for the width and height arguments of `pdf` are in inches (7 by 7), but the default for e.g. `png` is pixels (there is an argument `units` to change it to `in`, `cm`, or `mm`):

```
png("plot1.png", units = "in", res = 300,
    width = 6, height = 9)
boxplot(ToothGrowth$len~ToothGrowth$supp,
        col = c("darkgreen", "purple"),
        xlab = "Supplement Type", ylab = "Tooth Length",
        ylim = c(0, 40), horizontal = TRUE)
dev.off()

## pdf
##   2
```

Basic Course on R: Basic Plotting Practical

Karl Brand* and Elizabeth Ribble†

17-21 December 2018

Contents

*brandk@gmail.com
†emcclel3@msudenver.edu

1 Basic Plotting

1.1 Use R to do the following exercises on the BOD data.

1.1.1 Display the built-in dataset called `BOD` by running `BOD`.

1.1.2 What is the data structure of `BOD`? What are the dimensions?

1.1.3 What are the names of `BOD`? Use a function other than `str`.

1.1.4 Make a line graph of demand versus time, where the line is a deep pink dot-dashed line [Hint: run `?par` and look for the parameter `lty` to see the line types]. Add a blue dashed line of 1.1 times the demand and give it a thickness of 2 using the line width parameter `lwd`. Make sure both lines are entirely visible by adjusting the range of y using the parameter `ylim` in the original plot.

1.2 Use R to do the following exercises on the chickwts data.

1.2.1 Display the built-in `chickwts` data.

1.2.2 What is the data structure of `chickwts`? What are the dimensions?

1.2.3 What are the names of `chickwts`? Use a function other than `str`.

1.2.4 What are the levels of `feed`?

1.2.5 Make the following plots in one 2 x 2 image:

- A bar chart of the feed types, each bar a different color.
- A bar chart of the proportions of feed types, each bar a different color.
- A boxplot of the weights by feed type, each box a different color.
- A horizontal boxplot of the weights by feed type, each box a different color.

1.3 Use `R` to do the following exercises on the `Puromycin` data.

1.3.1 Display the built-in `Puromycin` data.

1.3.2 Make a scatterplot of the rate versus the concentration. Describe the relationship.

1.3.3 Make a scatterplot of the rate versus the log of the concentration. Describe the relationship.

1.3.4 Make a scatterplot of the rate versus the log of the concentration and color the points by treatment group (`state`). Describe what you see.

1.3.5 Make a scatterplot of the rate versus the log of the concentration, color the points by treatment group (`state`), label the x-axis "Concentration" and the y-axis "Rate", and label the plot "Puromycin".

1.3.6 Add a legend to the above plot indicating what the points represent.

1.3.7 Make a boxplot of the treated versus untreated rates. Using the function `pdf`, save the image to a file with a width and height of 7 inches.

1.3.8 Make a histogram of the frequency of concentrations. What is the width of the bins?

1.3.9 Make a histogram of the frequency of concentrations with a bin width of 0.10. How is this different from the histogram above?

1.3.10 Plot the histograms side by side in the same graphic window and make sure they have the same range on the y-axis. Does this make it easier to answer the question of how the two histograms differ?

If you want to save your work: save your `R` session and/or source code!

Basic Course on R: Basic Plotting Practical Answers

Karl Brand* and Elizabeth Ribble†

17-21 December 2018

Contents

*brandk@gmail.com
†emcclel3@msudenver.edu

1 Basic Plotting

1.1 Use R to do the following exercises on the BOD data.

1.1.1 Display the built-in dataset called BOD by running BOD.

```
BOD
```

```
##   Time demand
## 1    1    8.3
## 2    2   10.3
## 3    3   19.0
## 4    4   16.0
## 5    5   15.6
## 6    7   19.8
```

1.1.2 What is the data structure of BOD? What are the dimensions?

```
str(BOD)
```

```
## 'data.frame': 6 obs. of  2 variables:
##  $ Time  : num  1 2 3 4 5 7
##  $ demand: num  8.3 10.3 19 16 15.6 19.8
##  - attr(*, "reference")= chr "A1.4, p. 270"
```

Using str we see that BOD is a data frame with dimensions 6 x 2, each variable (Time and demand) a numeric vector.

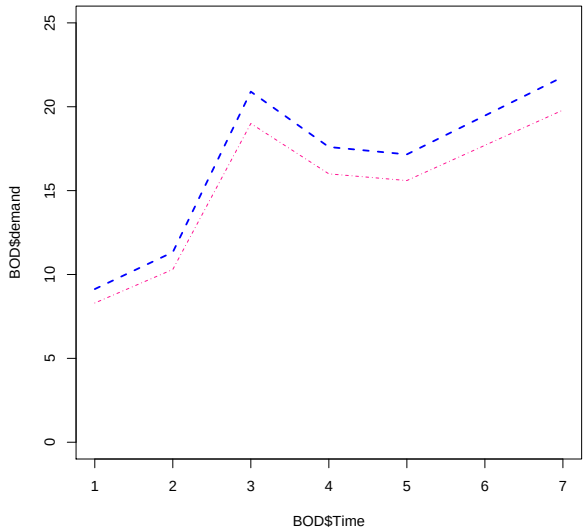
1.1.3 What are the names of BOD? Use a function other than str.

```
names(BOD)
```

```
## [1] "Time"   "demand"
```

a line graph of demand versus time, where the line is a deep pink dot-dashed line [Hint: run ?par and look for the parameter lty to see the line types]. Add a blue dashed line of 1.1 times the demand and give it a thickness of 2 using the line width parameter lwd. Make sure both lines are entirely visible by adjusting the range of y using the parameter ylim in the original plot.

```
plot(BOD$Time, BOD$demand, type = "l", lty = 4,
     col = "deeppink", ylim = c(0, 25))
lines(BOD$Time, 1.1 * BOD$demand, lwd = 2, lty = 2, col = "blue")
```



1.2 Use R to do the following exercises on the chickwts data.

1.2.1 Display the built-in **chickwts** data.

```
chickwts
```

1.2.2 What is the data structure of **chickwts**? What are the dimensions?

```
str(chickwts)
```

```
## 'data.frame': 71 obs. of  2 variables:
##  $ weight: num  179 160 136 227 217 168 108 124 143 140 ...
##  $ feed  : Factor w/ 6 levels "casein","horsebean",..: 2 2 2 2 2 2 2 2 2 2 ...
```

Using **str** we see that **chickwts** is a data frame with dimensions 71 x 2, with variable **weight** a numeric vector and **feed** a factor with 6 levels.

1.2.3 What are the names of **chickwts**? Use a function other than **str**.

```
names(chickwts)

## [1] "weight" "feed"
```

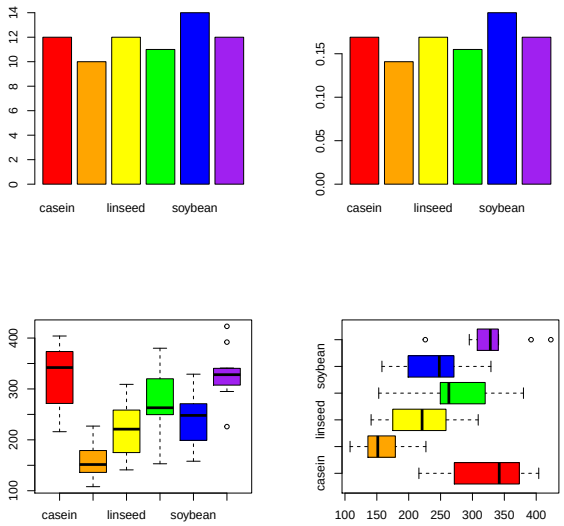
1.2.4 What are the levels of `feed`?

```
levels(chickwts$feed)

## [1] "casein"    "horsebean" "linseed"   "meatmeal"  "soybean"   "sunflower"
```

1.2.5 Make the following plots in one 2 x 2 image:

- A bar chart of the feed types, each bar a different color.
- A bar chart of the proportions of feed types, each bar a different color.
- A boxplot of the weights by feed type, each box a different color.
- A horizontal boxplot of the weights by feed type, each box a different color.

```
par(mfrow = c(2, 2))
barplot(table(chickwts$feed),
        col = c("red", "orange", "yellow",
                "green", "blue", "purple"))
barplot(table(chickwts$feed)/length(chickwts$feed),
        col = c("red", "orange", "yellow",
                "green", "blue", "purple"))
boxplot(chickwts$weight~chickwts$feed,
        col = c("red", "orange", "yellow",
                "green", "blue", "purple"))
boxplot(chickwts$weight~chickwts$feed,
        col = c("red", "orange", "yellow",
                "green", "blue", "purple"),
        horizontal = TRUE)
```



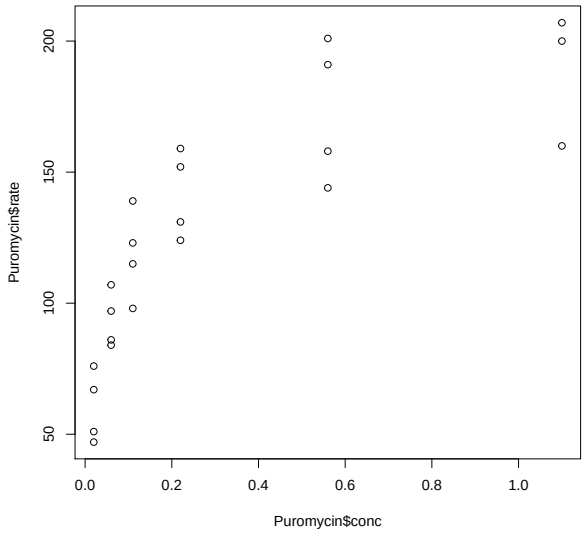
1.3 Use R to do the following exercises on the Puromycin data.

1.3.1 Display the built-in Puromycin data.

```
Puromycin
```

1.3.2 Make a scatterplot of the rate versus the concentration. Describe the relationship.

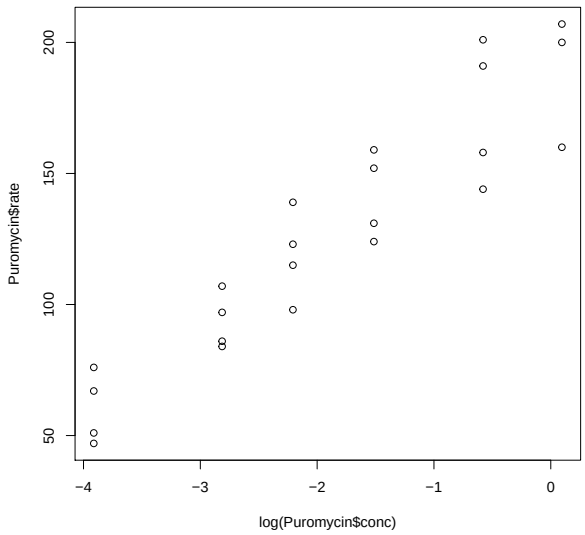
```
plot(Puromycin$conc, Puromycin$rate)
```



The rate increases faster at lower concentrations than at higher concentrations.

1.3.3 Make a scatterplot of the rate versus the log of the concentration. Describe the relationship.

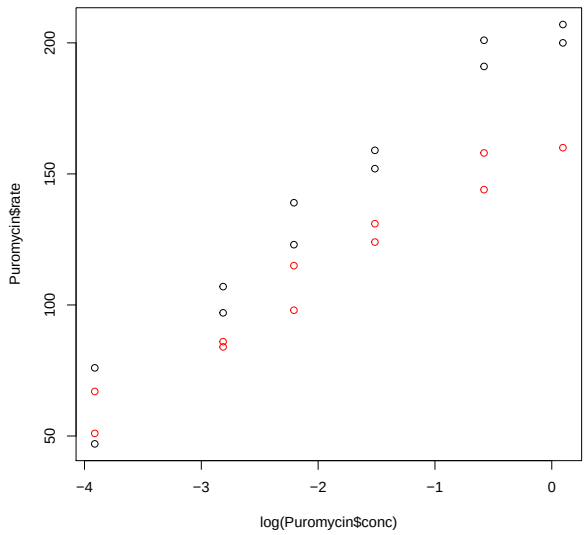
```
plot(log(Puromycin$conc), Puromycin$rate)
```



The two variables have a linear relationship.

1.3.4 Make a scatterplot of the rate versus the log of the concentration and color the points by treatment group (`state`). Describe what you see.

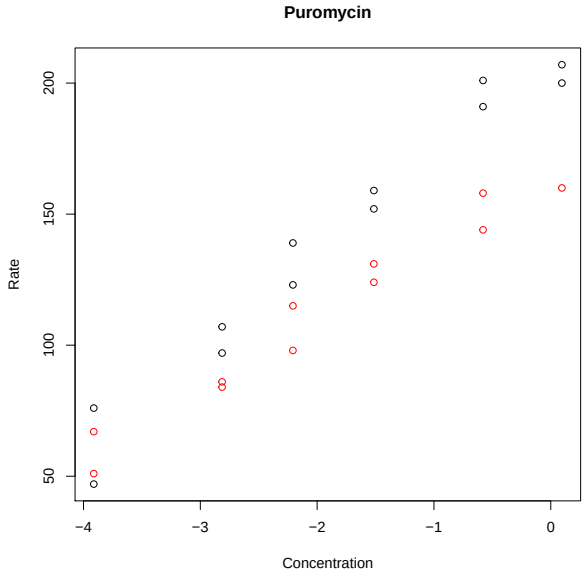
```
plot(log(Puromycin$conc), Puromycin$rate, col = Puromycin$state)
```



It appears that the the treated group has higher rates than the untreated group, on average. (Note that default colors are black for the first level and red for the second level).

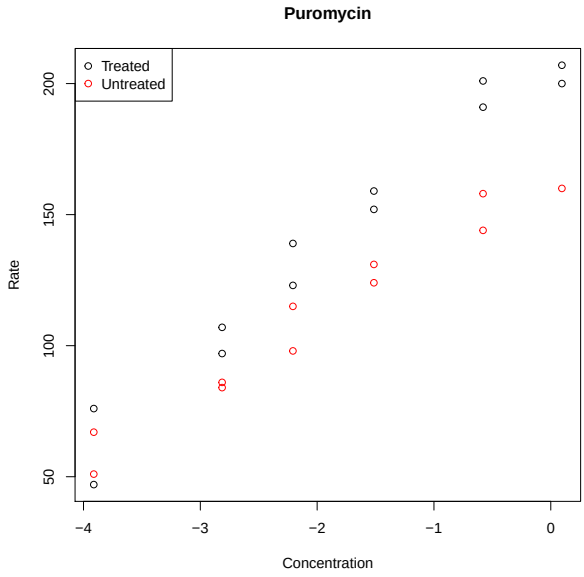
1.3.5 Make a scatterplot of the rate versus the log of the concentration, color the points by treatment group (`state`), label the x-axis "Concentration" and the y-axis "Rate", and label the plot "Puromycin".

```
plot(log(Puromycin$conc), Puromycin$rate, col = Puromycin$state,
     xlab = "Concentration", ylab = "Rate", main = "Puromycin")
```



1.3.6 Add a legend to the above plot indicating what the points represent.

```
plot(log(Puromycin$conc), Puromycin$rate, col = Puromycin$state,
     xlab = "Concentration", ylab = "Rate", main = "Puromycin")
legend("topleft",c("Treated", "Untreated"), col = 1:2, pch = 1)
```

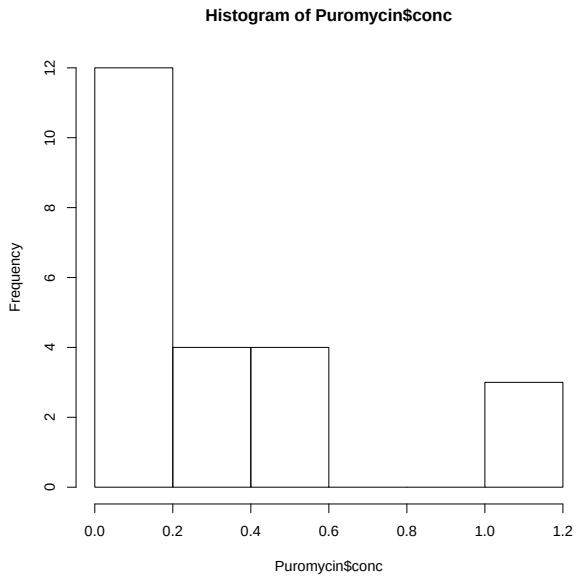


1.3.7 Make a boxplot of the treated versus untreated rates. Using the function `pdf`, save the image to a file with a width and height of 7 inches.

```
pdf("puromycin.pdf",width = 7, height = 7)
boxplot(Puromycin$rate~Puromycin$state)
dev.off()

## pdf
##   2
```

1.3.8 Make a histogram of the frequency of concentrations. What is the width of the bins?

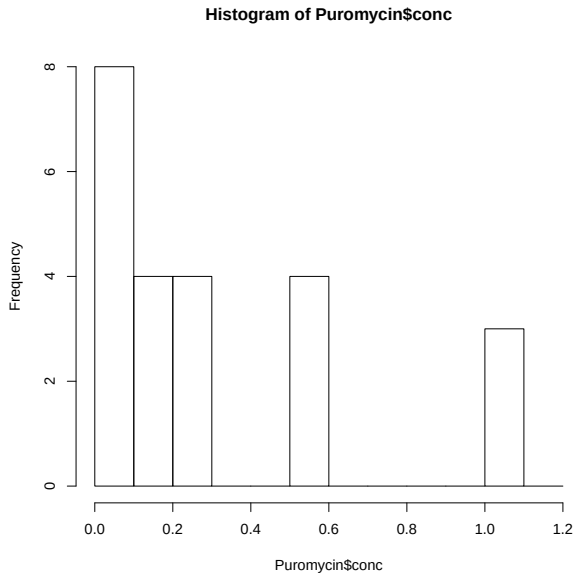
```
hist(Puromycin$conc)
```



The bin width is 0.20.

1.3.9 Make a histogram of the frequency of concentrations with a bin width of 0.10. How is this different from the histogram above?

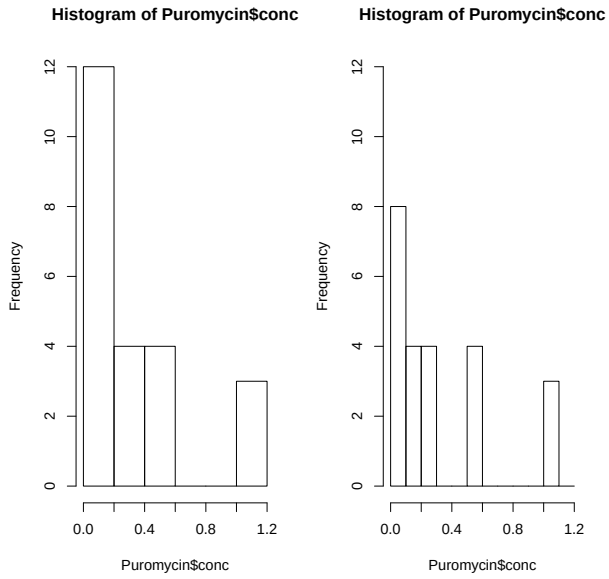
```
hist(Puromycin$conc, breaks = seq(0, 1.2, .10))
```



The bins are narrower, so we see in finer detail the distribution of the concentrations.

1.3.10 Plot the two histograms (bin widths 0.20 and 0.10) side by side in the same graphic window and make sure they have the same range on the y-axis. Does this make it easier to answer the question of how the two histograms differ?

```
par(mfrow = c(1, 2))
hist(Puromycin$conc, ylim = c(0, 12))
hist(Puromycin$conc, breaks = seq(0, 1.2, .10), ylim = c(0, 12))
```



In some situations it may be of use to view plots simultaneously. In this case, on the right we see clearly that more values are between 0 and 0.10 than 0.10 and 0.20 whereas the plot on the left does not display this information. In the histogram on the right we see that no concentrations fall between 0.30 and 0.50, whereas this is not apparent in the histogram on the left.

If you want to save your work: save your `R` session and/or source code!

Basic Course on **R**: Using ggplot2

David Nieuwenhuijse

May 18th - 24th, 2017

Contents

1 Using ggplot2: Lecture

The basic plotting functions of R are nice to quickly visualize data, however, the strength of R is that it has many packages with extra functionality created by others that you can use. ggplot2 is one of those packages, created by Hadley Wickham. ggplot2 allows you to build more sophisticated plots in R using a "Grammar of Grappics" (therefore GGplot).

If ggplot2 is not yet available for you we need to install and load the ggplot2 library like so.

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

```
## Warning: package 'ggplot2' was built under R version 3.3.3
```

ggplot2 is a very extensive package, so it is impossible to show all the possibilities during this course. If you would like to experiment further, the ggplot manual at http://ggplot2.tidyverse.org/reference/index.html is the place to go. A large variety of functionallities are listed there, including example plots.

1.1 Grammar of graphics

The grammar of graphics consists of a number of "layers", which generate a ggplot when correctly put together. These five basic layers will be discussed here:

- Data
- Aesthetics
- Geometries
- Facets
- Scales
- Themes

1.2 Data

The basis of each plot is the data. To plot data with ggplot the data needs to be of the class "data.frame". The structure of your data.frame is important when you want to plot it with ggplot. Each variable you want to use has to be in a separate column in your data.frame. Take for example the "Indometh" dataset which comes with R.

```
?Indometh
```

The "Indometh"" dataset describes the pharmacokinetics of indomethacin after intravenous injection of indomethacin in six subjects. The data.frame has three columns one containing the subject number, one containing the time of sampling in hours, and one containing the indomethacin concentration ug/ml. This dataset is easily plotted with ggplot because it is a data.frame and there is a good separation of variables in the columns.

```
##   Subject time conc
## 1       1 0.25 1.50
## 2       1 0.50 0.94
## 3       1 0.75 0.78
## 4       1 1.00 0.48
## 5       1 1.25 0.37
## 6       1 2.00 0.19
```

An example of a badly formatted dataset for visualization with ggplot is a dataset of monthly deaths from lung diseases in the UK:

```
?ldeaths
```

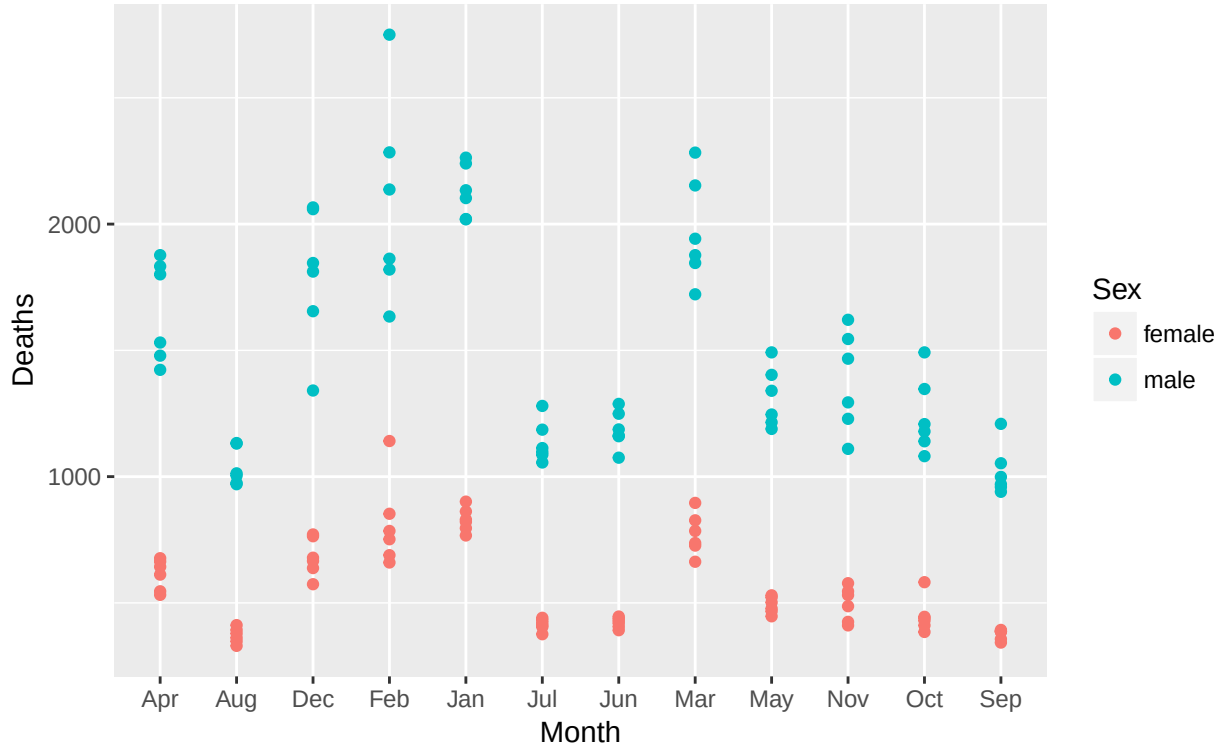
This dataset contains three separate datasets for female, male and both male and female deaths (ldeaths, fdeaths, and mdeaths). The columns contain values per month and the rows indicate the year. (also they are not data.frames)

```
##       Jan  Feb  Mar  Apr  May  Jun  Jul  Aug  Sep  Oct  Nov  Dec
## 1974 3035 2552 2704 2554 2014 1655 1721 1524 1596 2074 2199 2512
## 1975 2933 2889 2938 2497 1870 1726 1607 1545 1396 1787 2076 2837
## 1976 2787 3891 3179 2011 1636 1580 1489 1300 1356 1653 2013 2823
## 1977 3102 2294 2385 2444 1748 1554 1498 1361 1346 1564 1640 2293
## 1978 2815 3137 2679 1969 1870 1633 1529 1366 1357 1570 1535 2491
## 1979 3084 2605 2573 2143 1693 1504 1461 1354 1333 1492 1781 1915
```

Since there are two separate datasets, there is no column indicating the sex. There is also no column indicating the year and no column indicating the month. This means that we cannot use these variables in ggplot to visualize, for instance, the difference between the number of deaths in men and women in a certain year. We can use some more advanced R data reshaping to get these datasets in the right shape.

```
##   Deaths Year Month    Sex
## 1    901 1974   Jan female
## 2    689 1974   Feb female
## 3    827 1974   Mar female
## 4    677 1974   Apr female
## 5    522 1974   May female
## 6    406 1974   Jun female
```

```
ggplot(data, aes(x=Month, y=Deaths, color=Sex)) + geom_point()
```



However, it is much more easy to keep in mind that any variable that you want to use as a part of your ggplot should get its own column when you are designing your dataset (in Excel). With the data in the right class and the right format we can easily call the ggplot function, and add a scatter plot layer to it.

1.3 Aesthetics

After figuring out what data to plot it is nescessary to indicate how to plot it. Aesthetics are used to indicate how to plot what. As we saw in the plot of the ldeaths data we start with indicating what data.frame to plot and use aes() to indicate what to plot on the x axis, the y axis and what to use as color. We do not have to indicate what to plot where and which color to give to which category. We simply map a column of our data.frame to an aesthetic. Depending on the geom we use there are a number of aesthetics that can be mapped to a variable:

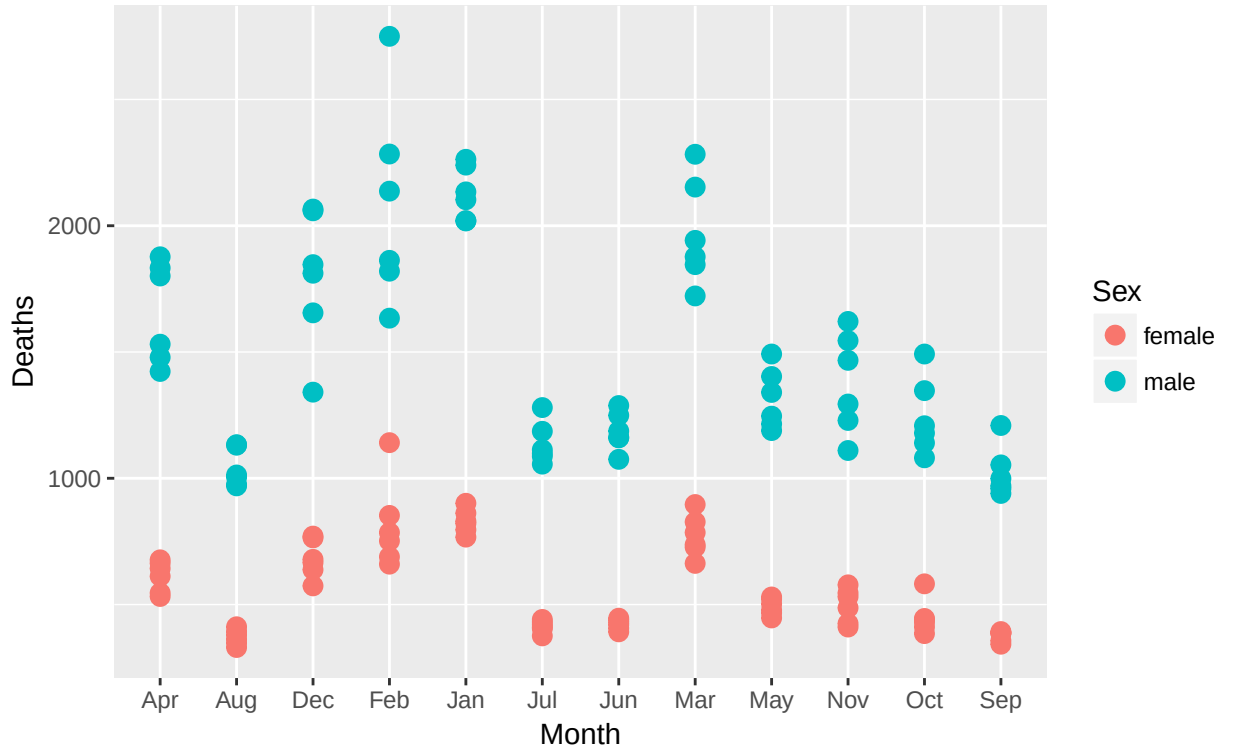
- x
- y
- color
- fill
- size
- alpha
- linetype
- labels
- shape
- group

Aesthetics can also be specified per geom layer. This allows plotting multiple geom layers with different aesthetics. However, when using one data source and one geom layer this will result in exactly the same plot.

```
ggplot(data) + geom_point(aes(x=Month, y=Deaths, color=Sex))
```

We can also use plotting parameters without mapping them to a variable to change all dots in the geom_point. This can only be done in the geom itself.

```
ggplot(data, aes(x=Month, y=Deaths, color=Sex)) + geom_point(size=3)
```



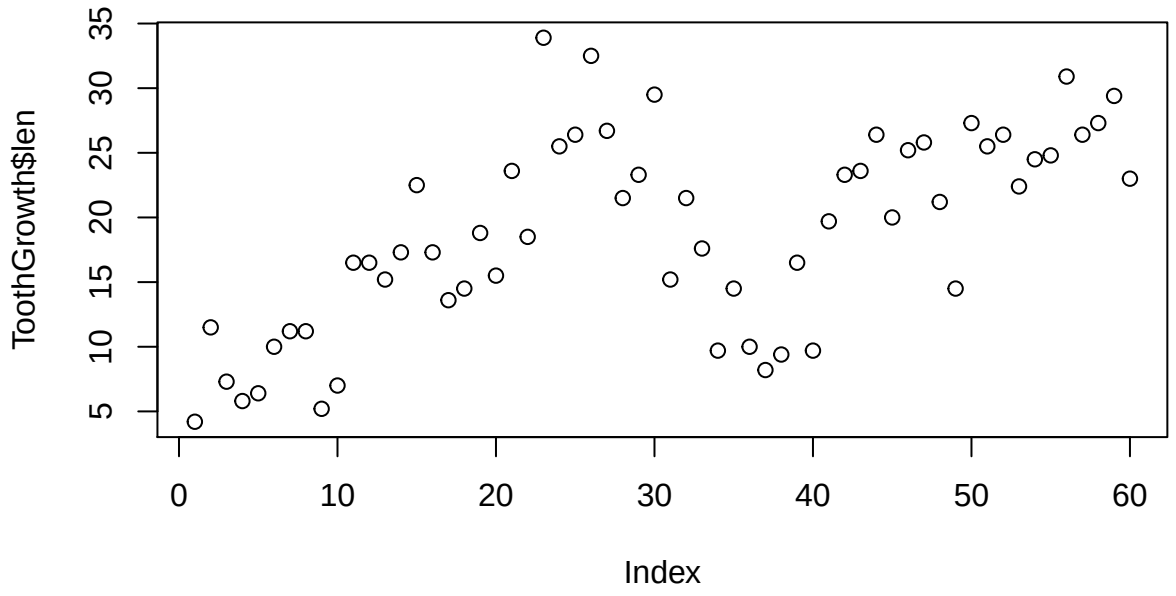
1.4 Geometries

For the previous ggplots we have used the geom_point function which generates a scatter plot layer that can be added to the ggplot. "point" is one of the many geomertry layers available in the ggplot library. There are over thirty different geoms in ggplot which can be found at http://docs.ggplot2.org/. We will have a look at the plot types that we have already encountered using the basic R plotting functions and replace them with their ggplot counterparts.

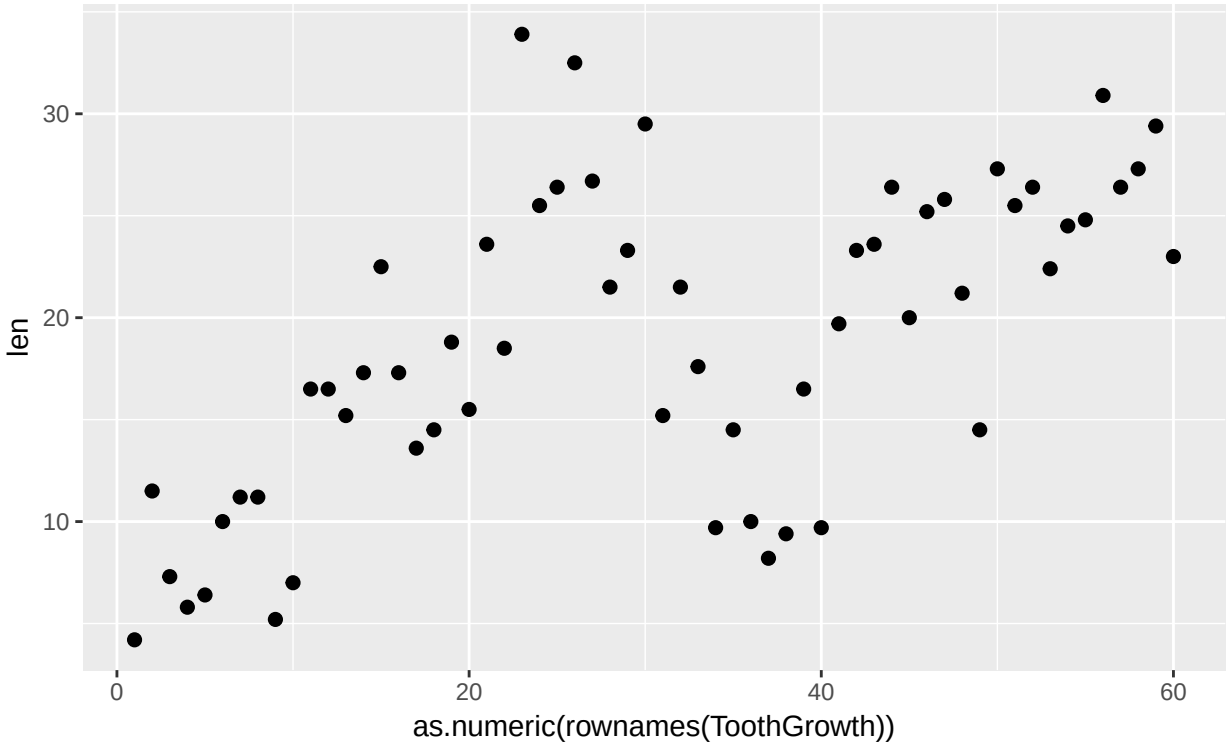
1.4.1 Scatter plots

Just like with other objects in R a ggplot object can be saved to a variable using "<-". This makes it possible to store the basic plot in a variable and add different geoms to it. Geometries are added to a plot using "+".

```
#Basic graphics:
plot(ToothGrowth$len)
```



```
p <- ggplot(ToothGrowth)
p + geom_point(aes(x=as.numeric(rownames(ToothGrowth)), y=len), size=2)
```



We now map the numeric rownames of ToothGrowth to the x aesthetic, however, we could also create a new column called index in the data.frame.

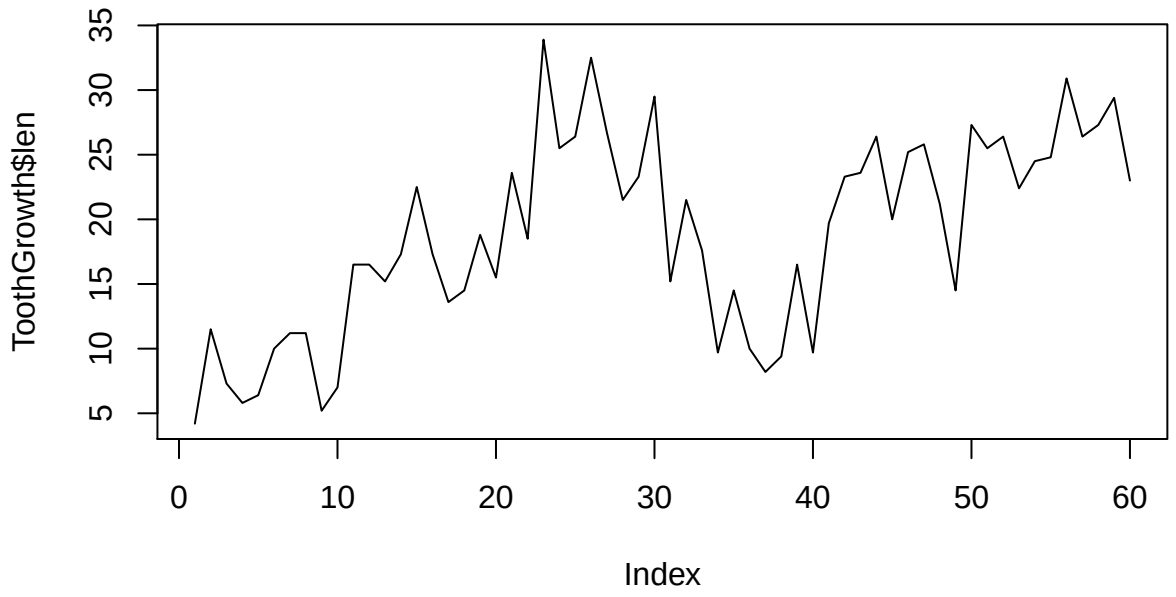
```
ToothGrowth$index <- as.numeric(rownames(ToothGrowth))
```

When we change the data of the ggplot object we saved in variable p we need to recreate the ggplot object with the new dataset.

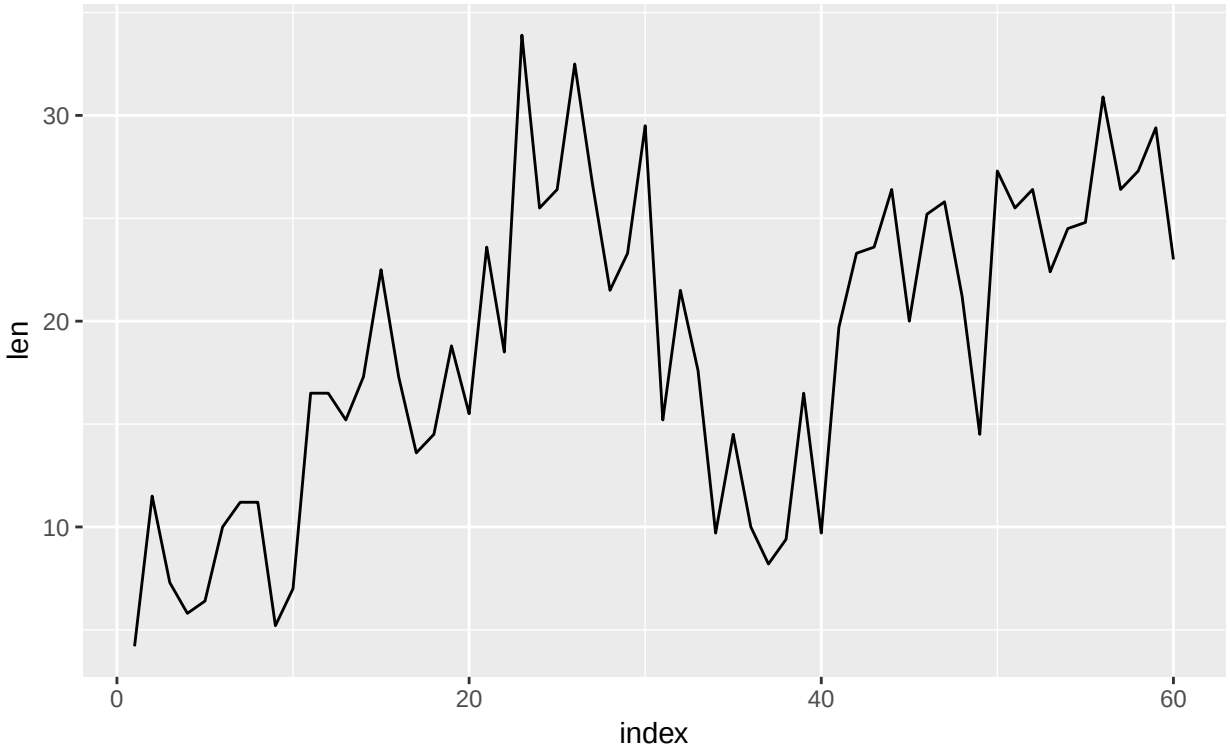
```
p <- ggplot(ToothGrowth)
```

1.4.2 Line graphs

```
#Basic graphics:
plot(ToothGrowth$len, type = "l")
```

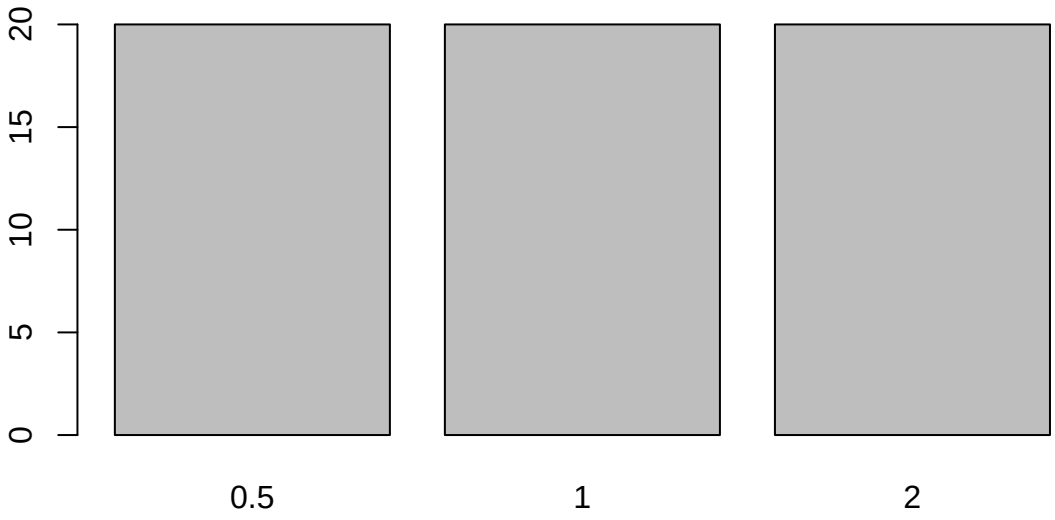



```
p + geom_line(aes(x=index, y=len))
```

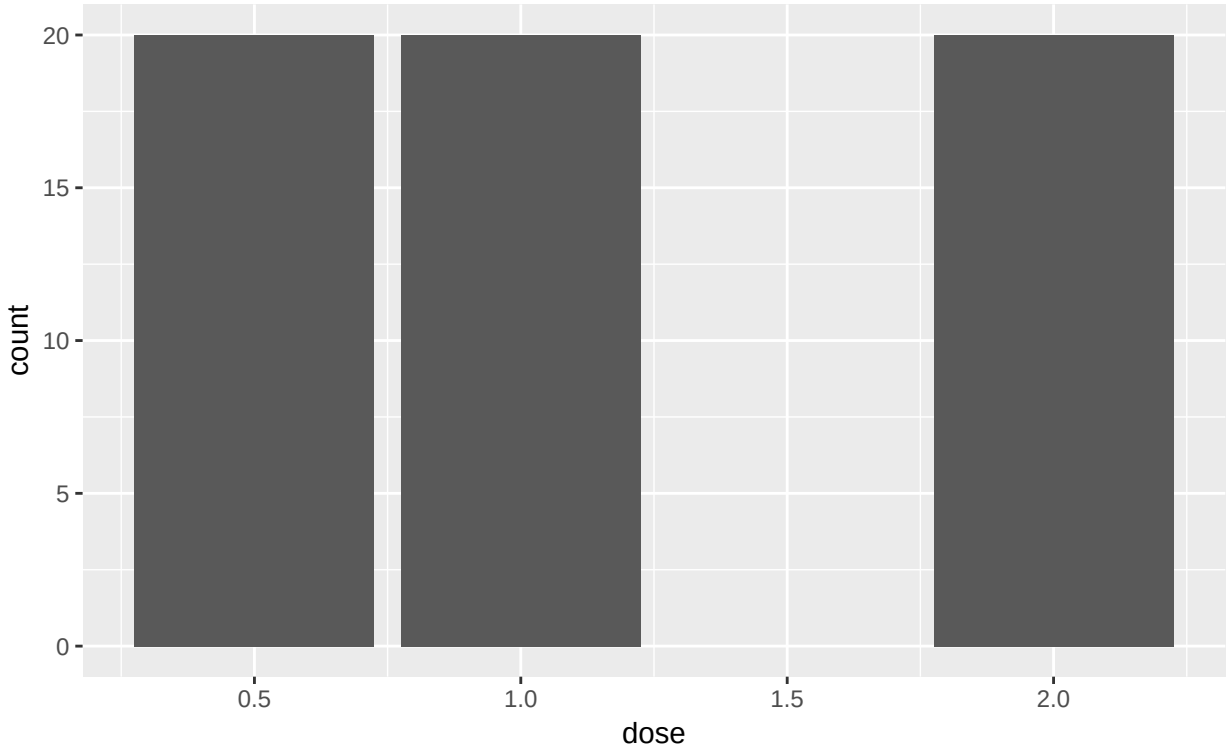


1.4.3 Bar charts

```
#Basic graphics:
barplot(table(ToothGrowth$dose))
```

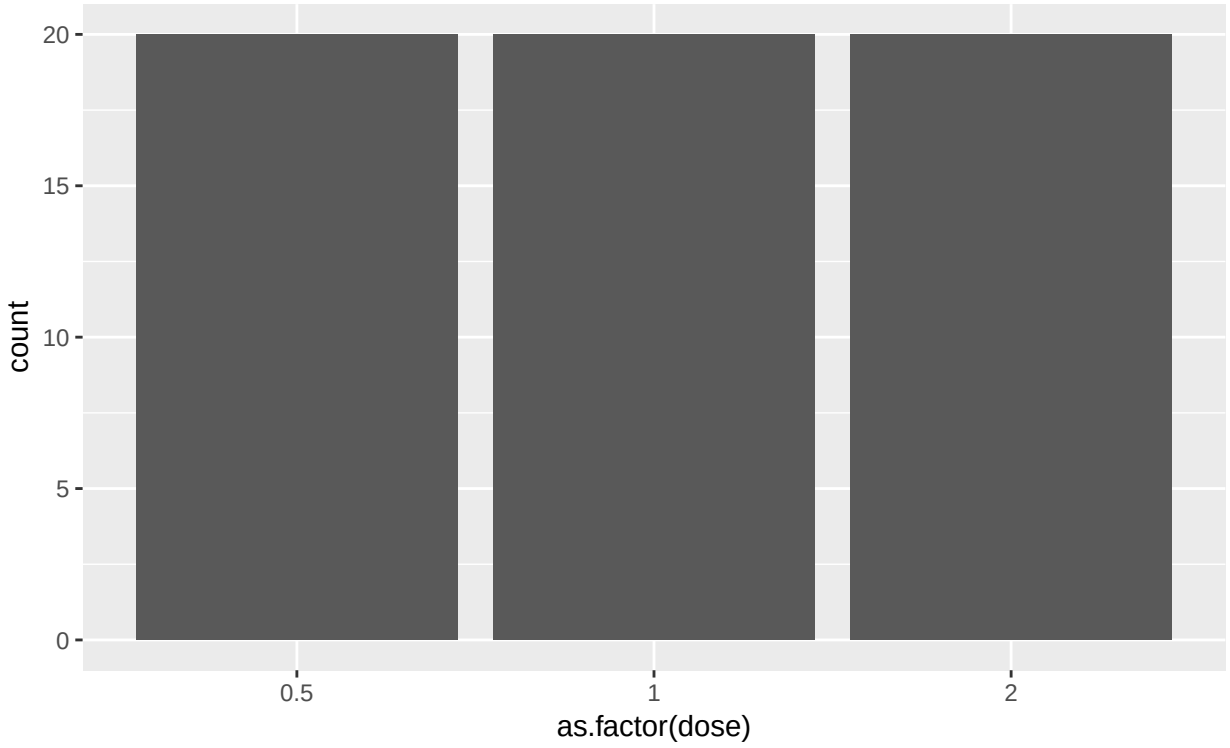



```
p + geom_bar(aes(x=dose))
```



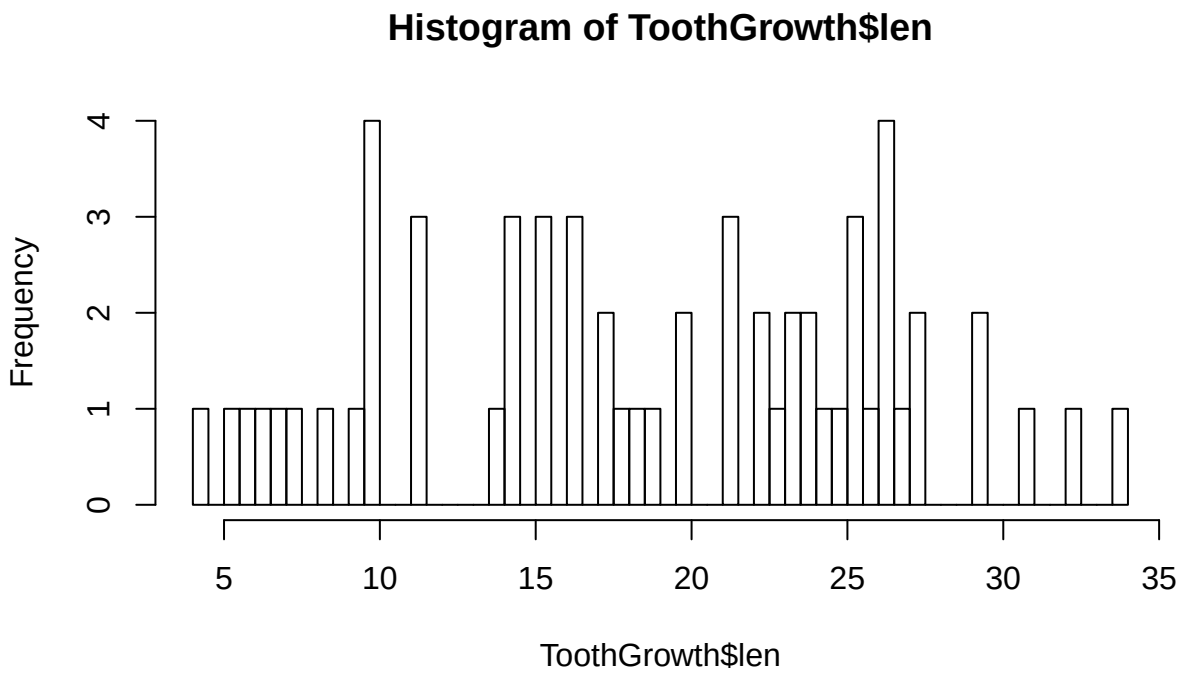
Because dose is a continuous variable, ggplot creates a x axis with a continuous scale, which is why there is an empty spot for the 1.5 dose. If we want to use a continuous value as a catagorical value we can change it into a factor on the fly in the geom function:

```
p + geom_bar(aes(x=as.factor(dose)))
```

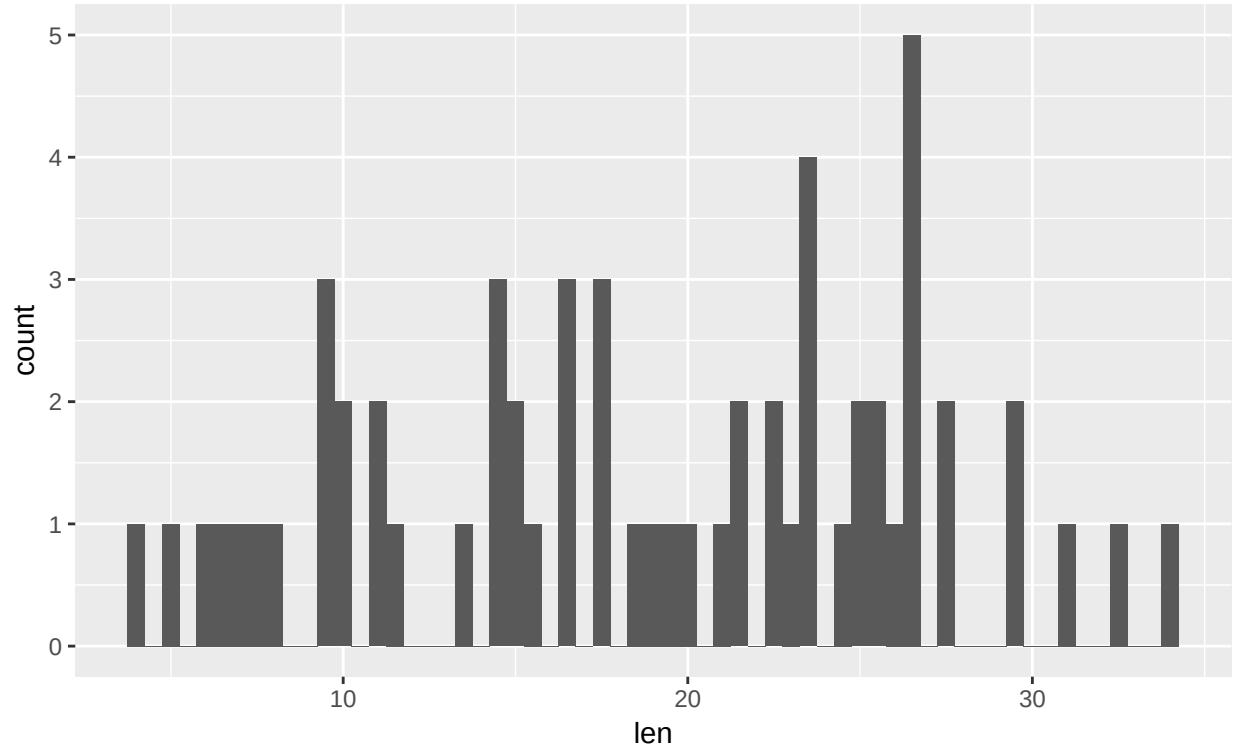


1.4.4 Histograms

```
#Basic graphics:
hist(ToothGrowth$len,breaks = 50)
```

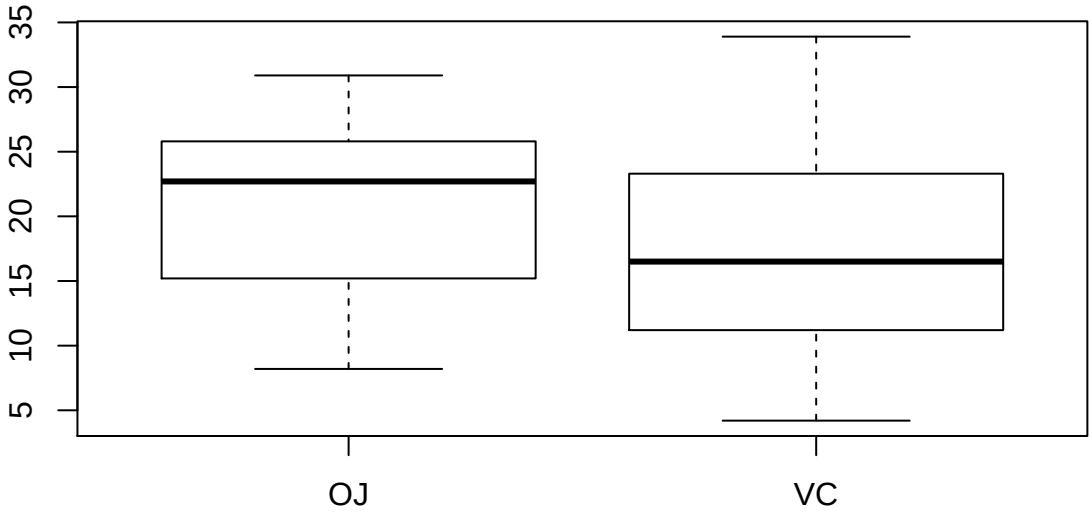



```
p + geom_histogram(aes(x=len), binwidth = 0.5)
```

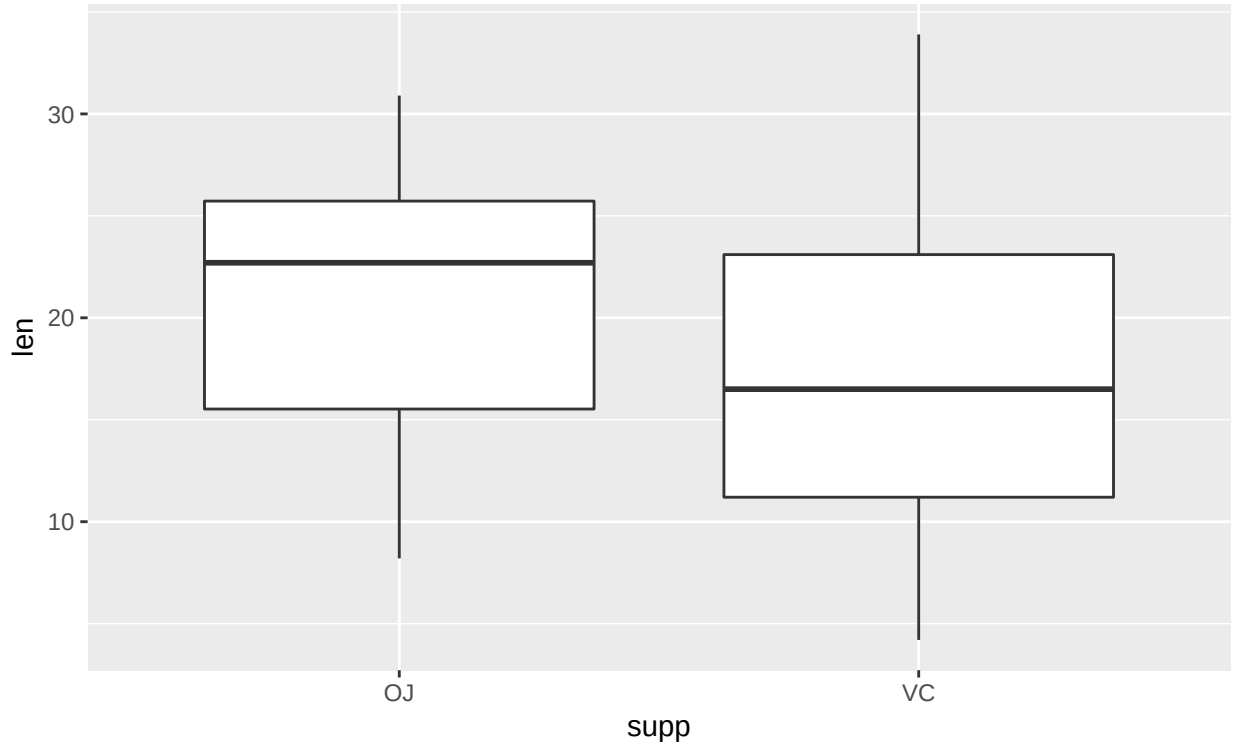


1.4.5 Box plots

```
#Basic graphics:
boxplot(ToothGrowth$len~ToothGrowth$supp)
```

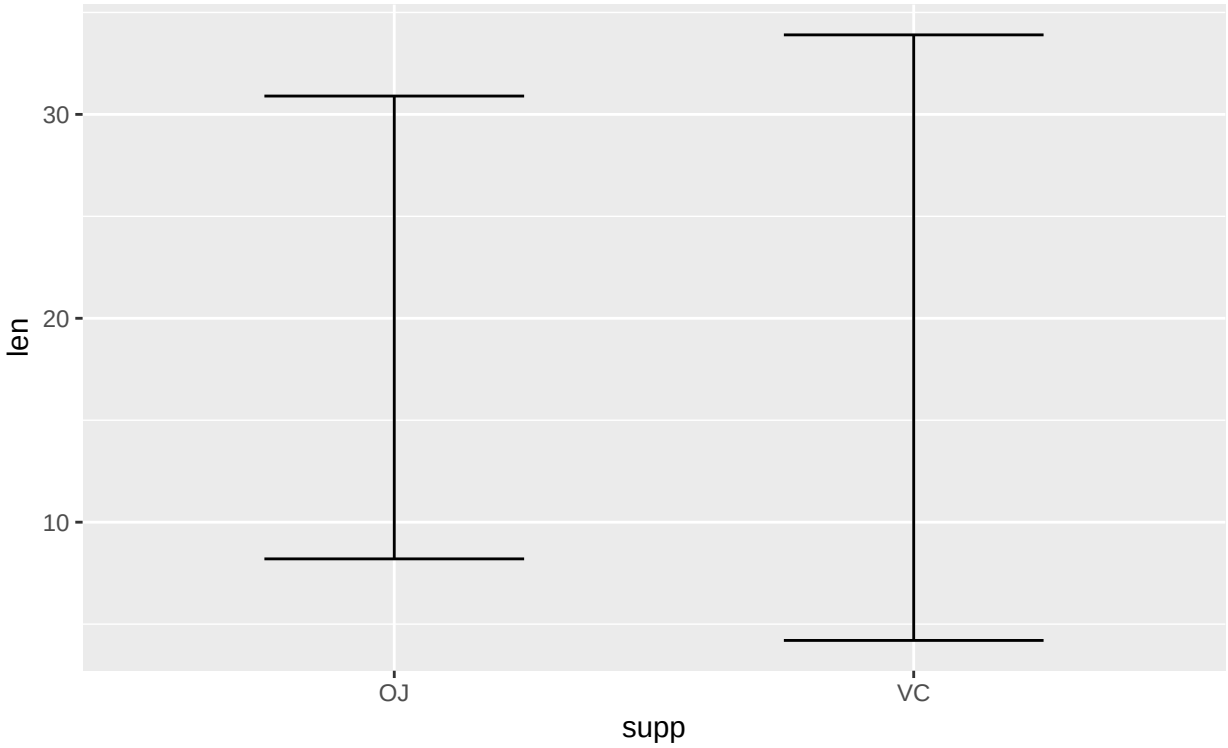


```
p + geom_boxplot(aes(x=supp,y=len))
```



If we want whiskers in our plot we can add an errorbar geom

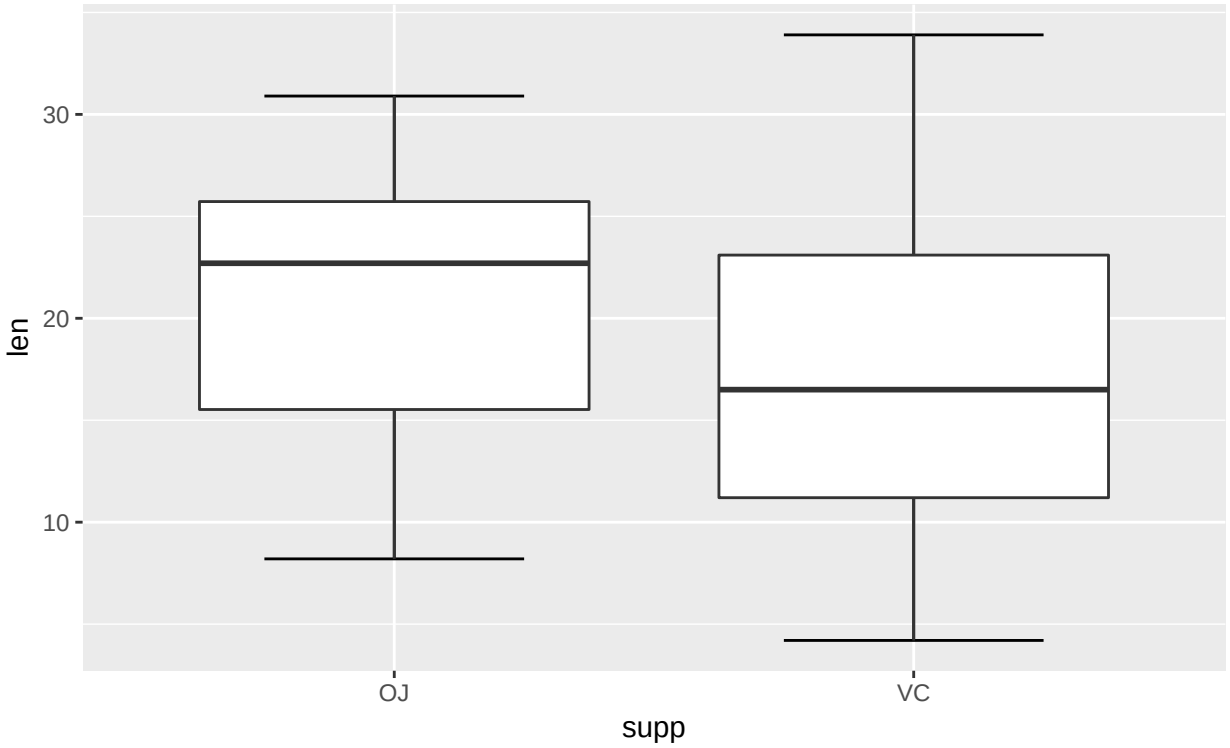
```
p + geom_errorbar(aes(x=supp, ymin=..., ymax=...),width=0.5)
```



Howerver, as you can see, this geom requires ymin and ymax as aesthetics and does not calculate the values by itself. We could calculate the min and max values for toothlength grouped by supplement, but we will use a trick using ggplot's boxplot statistics function. *(if you want to know more, please read more about these "stat" functions online)*

First, we add the errorbars with the "stat_boxplot" function, and then add the boxplots on top of them.

```
p + stat_boxplot(aes(x=supp,y=len), geom="errorbar", width=0.5) +
    geom_boxplot(aes(x=supp,y=len))
```



As we can see the aesthetics of both geoms have to be defined separately in this plot. To reduce typing we could also write the ggplot as follows.

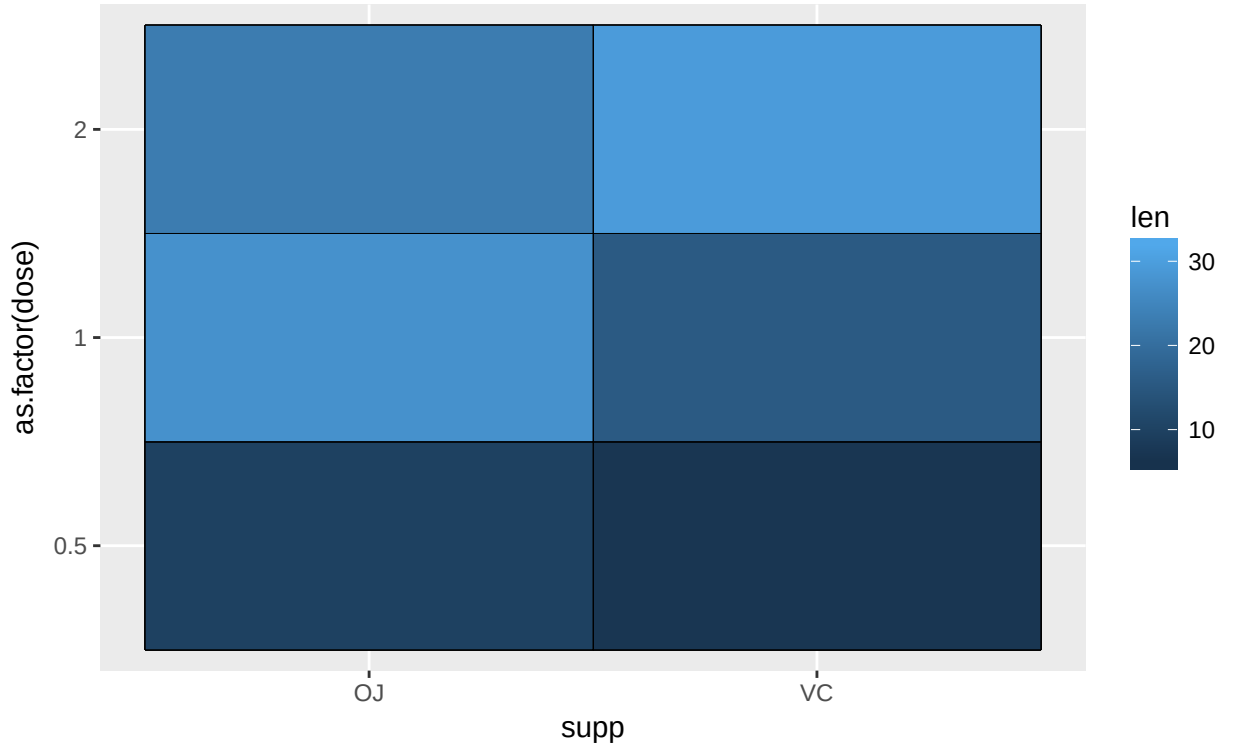
```
p <- ggplot(ToothGrowth, aes(x=supp,y=len))
p + stat_boxplot(geom="errorbar", width=0.5) + geom_boxplot()
```

When the aesthetics are specified in the base ggplot object it is not neccesary to specify them in the added layers.

1.4.6 Heatmaps

One popular way of diplaying large grids of data is using a heatmap. The tile geom generates a tile at each provided x and y value and uses the fill aesthetic to fill the tiles according to another variable. The border color of the tiles are can be changed by using the color aesthetic.

```
p <- ggplot(ToothGrowth)

p + geom_tile(aes(x = supp, y = as.factor(dose), fill = len), color="black")
```



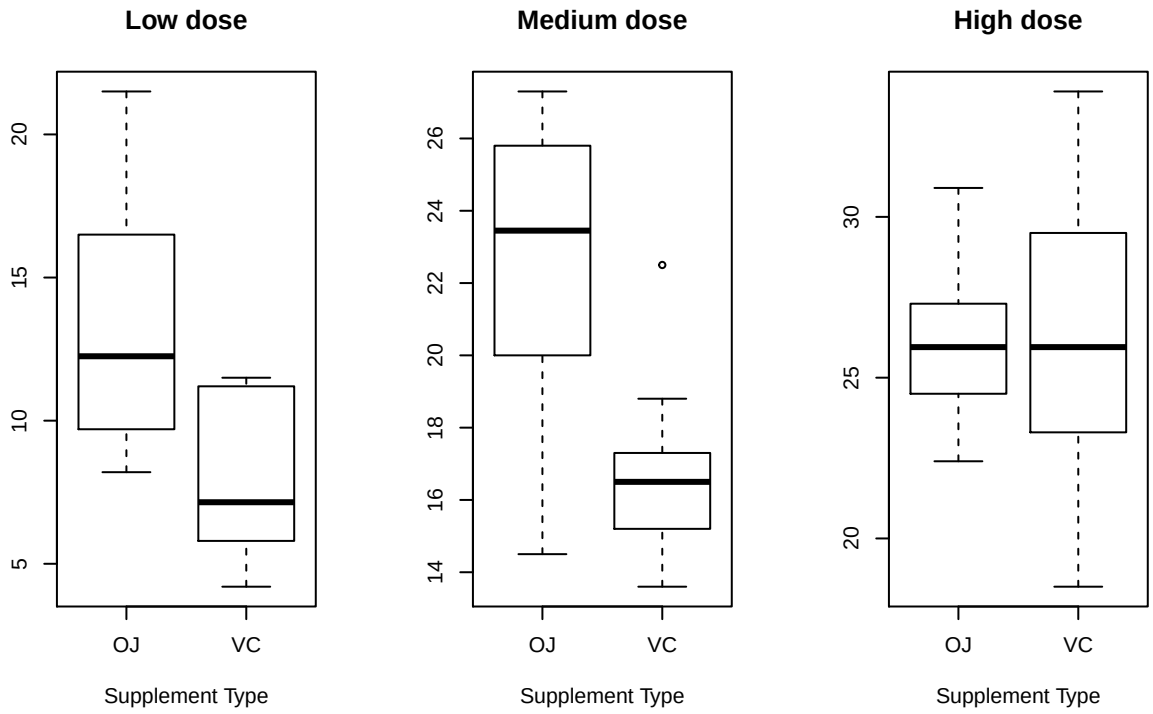
There are dedicated packages for plotting heatmaps, which also perform horizonal and vertical clustering of the tiles. These are, however, out of the scope of this course, but it should be easy to find them online.

1.5 Facets

Apart from using aesthetics to map variables to, ggplot has another feature which can help visualizing data called "facets". We have added multiple plots to the viewing window before, using "par(mfrow=c(...,...))". However, after specifying how many plots should be printed in the plot viewer we have to call the plot function multiple times to populate the viewer.

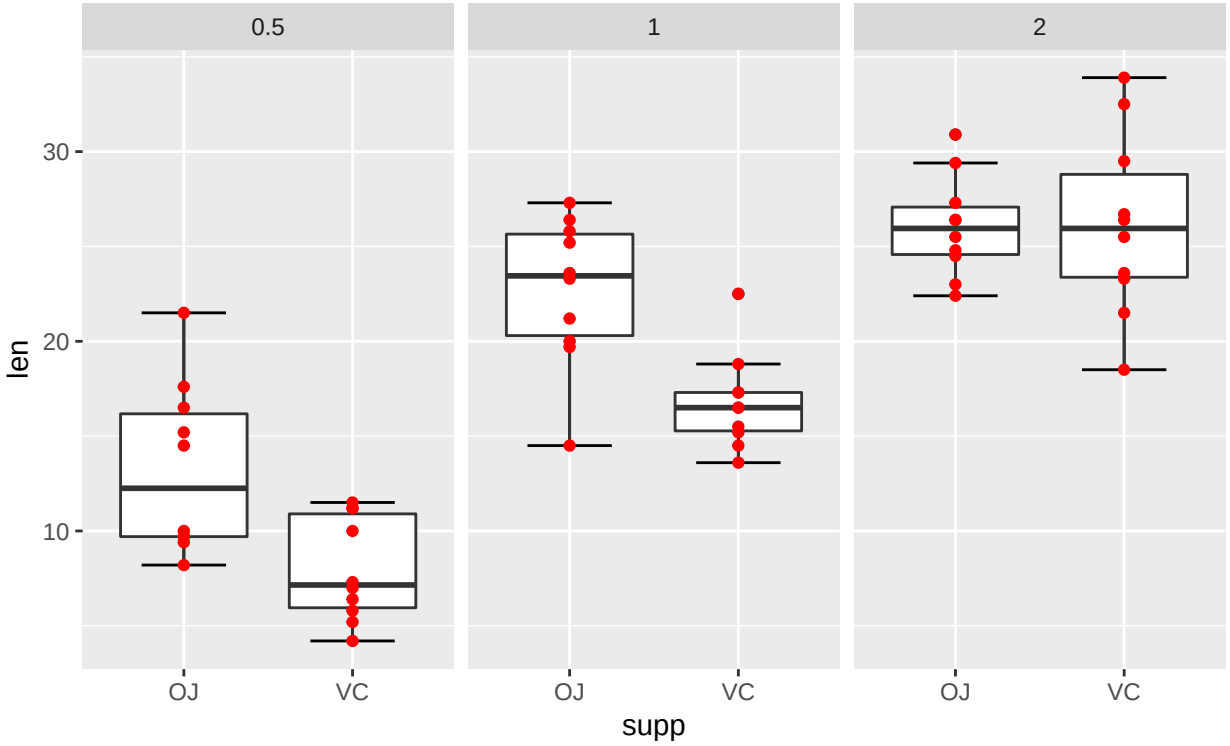
If we want to compare the effect of orange juice versus vitamin C depending on the dose we can plot three boxplots next to eachother.

```
par(mfrow = c(1, 3))

low <- ToothGrowth[which(ToothGrowth$dose==0.5),]
med <- ToothGrowth[which(ToothGrowth$dose==1),]
high <- ToothGrowth[which(ToothGrowth$dose==2),]

boxplot(low$len~low$supp, main = "Low dose", xlab = "Supplement Type")
boxplot(med$len~med$supp, main = "Medium dose", xlab = "Supplement Type")
boxplot(high$len~high$supp, main = "High dose", xlab = "Supplement Type")
```



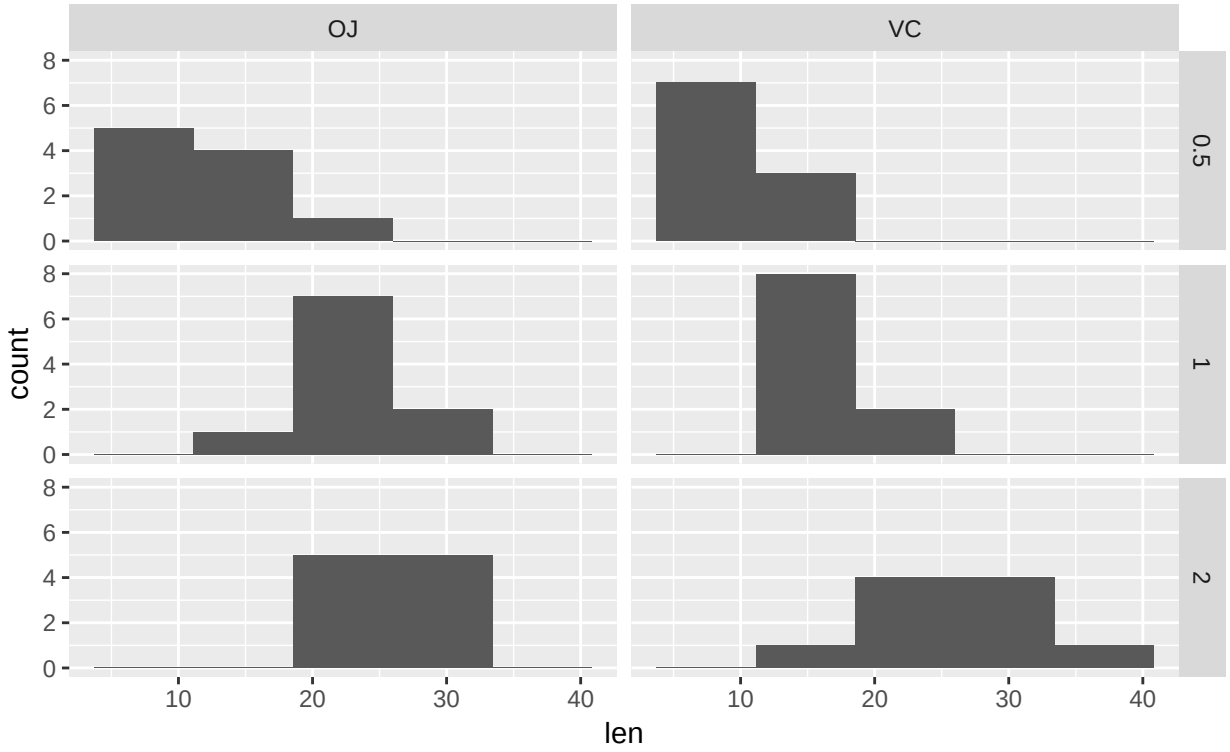
However, using ggplot we can simply map the dose column to a ggplot facet grid. The mapping of variables is not done via a aes(), but in formula form. Variables on the right indicate the columns in the grid and variables on the left indicate the rows of the grid. A "." can be used if we want to use only one variable for facetting. This will also preserve the scales, making it more easy to compare the three plots. For fun, let's add the actual datapoints in an additional geom on top of the boxplot.

```
ggplot(ToothGrowth, aes(x=supp,y=len)) +
    stat_boxplot(geom="errorbar", width=0.5) +
    geom_boxplot() + geom_point(color="red") +
    facet_grid(. ~ as.factor(dose))
```



We can do the same with two variables in a grid.

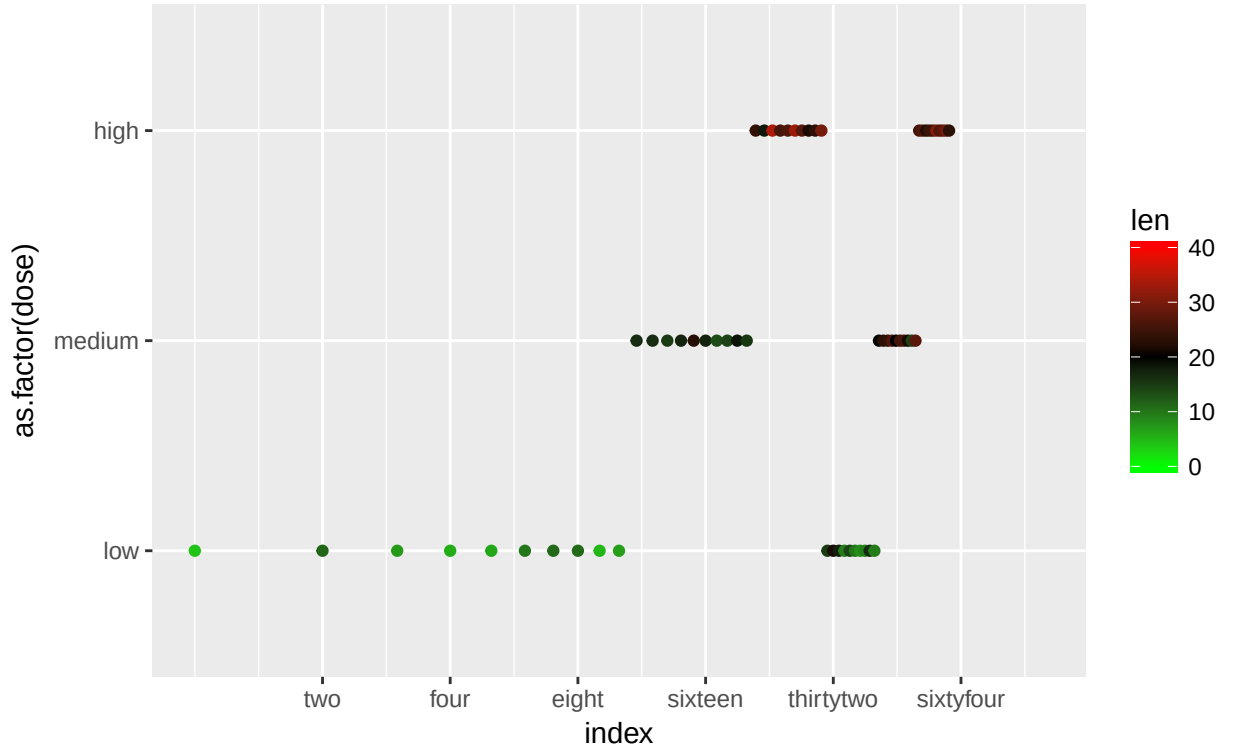
```
ggplot(ToothGrowth, aes(x=len)) +
    geom_histogram(bins = 5) +
    facet_grid(dose ~ as.factor(supp))
```



1.6 Scales

When mapping a variable to one of the aesthetics of a ggplot, ggplot uses the default scales. It is however possible to adjust the scales of the plot and their color using the scales functions. The scales that are generally used the most are those for the x, y, color and fill of the plot. The type of function to adjust the scale of one of the aesthetics depends on the aesthetic and the type of variable (continuous or discrete). Some of the possible scale adjustments, such as limits, transformation, breaks, labels, and color, are exemplified by the plot below.

```
ggplot(ToothGrowth, aes(x=index, y=as.factor(dose), color=len)) +
    geom_point() +
    scale_x_continuous(limits=c(1,100), trans = "log2",
        breaks=c(2,4,8,16,32,64),
        label=c("two","four","eight","sixteen","thirtytwo","sixtyfour")) +
    scale_y_discrete(label=c("low","medium","high")) +
    scale_color_gradient2(limits=c(0,40), low = "green", mid = "black", high = "red",
        midpoint = 20)
```



Have a look at the scales chapter at http://ggplot2.tidyverse.org/reference/ to explore the other possibilities!

1.7 Themes

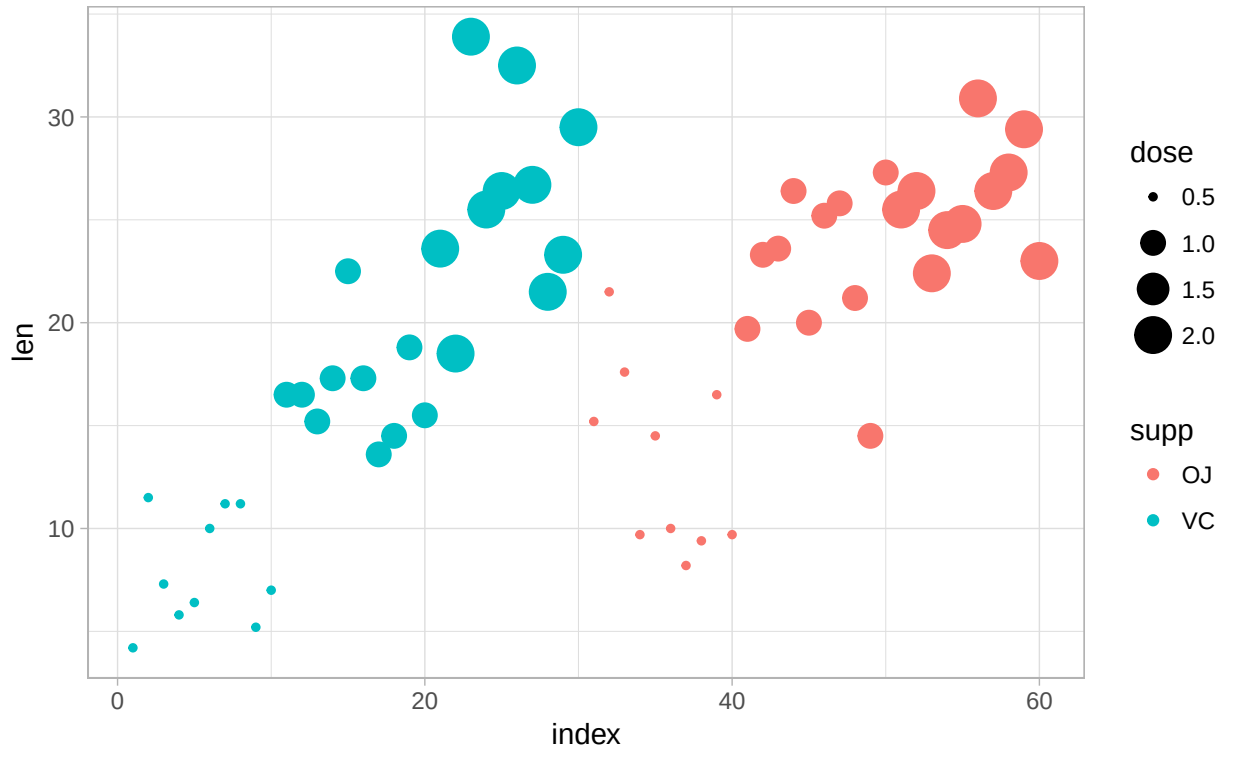
Untill now we have simply mapped a variable from our dataset to an aesthetics parameter, potentially change the scales, and let ggplot handle the styling of the graph. However, ggplot has many options to customize the style of your plot using themes. There are several standard themes that we can use, however it is possible to create our own style from scratch.

The standard themes available in the ggplot package are:

- theme_gray
- theme_bw
- theme_linedraw
- theme_light
- theme_dark
- theme_minimal
- theme_classic
- theme_void

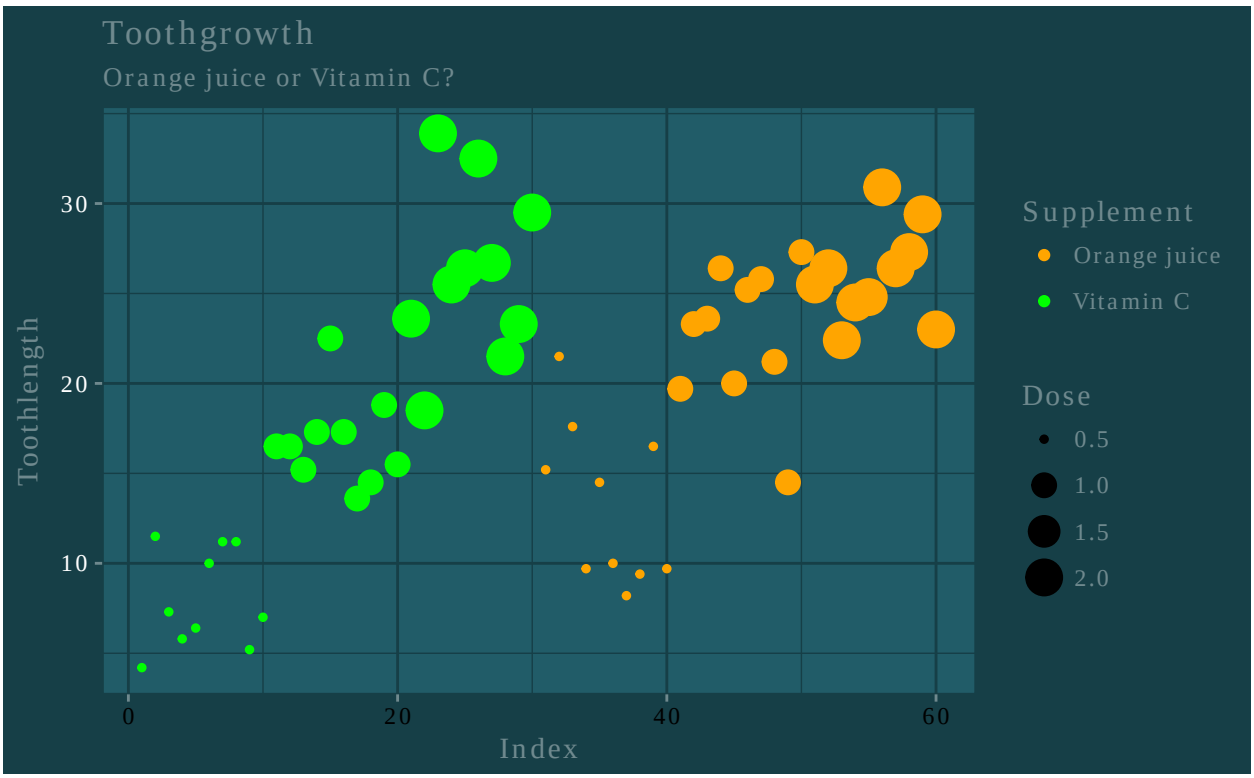
To add a standard theme we simply add the theme to the ggplot like we would with a geom.

```
p <- ggplot(ToothGrowth)
p + geom_point(aes(x = index, y = len, color = supp, size = dose)) + theme_light()
```



Custom themes can be made from scratch using the “theme()” function. To have a look at all customizable parameters visit http://ggplot2.tidyverse.org/reference/theme.html. Labels can be edited by adding “labs()” to the plot.

```
p <- ggplot(ToothGrowth)
p + geom_point(aes(x = index, y = len, color = supp, size = dose)) +
    theme(text = element_text(family = "Bookman", colour = "#6f898e"),
      line = element_line(color = "#163f47"),
      rect = element_rect(fill = "#163f47", color = "#163f47"),
        axis.text.x = element_text(color="black"),
        axis.text.y = element_text(color="white"),
      axis.ticks = element_line(color = "#6f898e"),
      axis.line = element_line(color = "#163f47", linetype = 1),
        legend.background = element_blank(),
        legend.key = element_blank(),
        panel.background = element_rect(fill = "#215c68", colour = "#163f47"),
        panel.border = element_blank(),
      panel.grid = element_line(color = "#163f47"),
      panel.grid.major = element_line(color = "#163f47"),
      panel.grid.minor = element_line(color = "#163f47"),
      plot.background = element_rect(fill = NULL, colour = NA, linetype = 0)
    ) +
    labs(title="Toothgrowth",
        subtitle = "Orange juice or Vitamin C?", x="Index", y="Toothlength",
        size="Dose", color="Supplement") +
    scale_color_manual(label=c("Orange juice","Vitamin C"),
        values = c("VC"="green","OJ"="orange"))
```



1.8 Saving ggplots

We can save out plots by using the ggsave() function. ggsave takes two argument, the first argument being the ggplot and the second being the location where the file should be saved. If we do not specify the complete path, it will save the plot in our current working directory (getwd()). The type of the file depends on the file extension we use. Possible file extensions are “eps”, “ps”, “tex” (pictex), “pdf”, “jpeg”, “tiff”, “png”, “bmp”, “svg” or “wmf” (windows only).

```
p <- ggplot(ToothGrowth)

myplot <- p + geom_point(aes(x = index, y = len, color = supp, size = dose))

ggsave("my_plot.pdf",myplot)
```

2 Using ggplot: Practicals

1. **Load the ggplot2 library (install it if you have to) and the diamonds dataset using data()**
2. **Explore the dataset using dim(), str() and help(), which variables are continuous, which variables are discrete? Is this dataset ready for plotting with ggplot?**
3. **Use ggplot to plot a scatterplot of the relationship between the diamonds' carat and their price**
4. **Make all dots darkblue and set the alpha value to 0.1**
5. **Visualize the influence of the color of a diamond on its price by mapping the diamond color to the color aesthetic**
6. **Use a ggplot barplot to visualize diamond clarity depending on color, map diamond color to x and diamond clarity to fill**
7. **Create a boxplot of the carat of a diamond based on its clarity and add whiskers using stat_boxplot**
8. **Add a geom_point layer to the previous plot mapping the diamonds price to the color**
9. **Create a histogram of the price of the diamonds and separate the histograms into facets using diamond color, choose a good binwith or number of bins**
10. **Create a grid of facets of the same histogram by comparing both color and cut**
11. **Use 'aggregate(diamonds, by = list(cut = diamonds$cut, color = diamonds$color), mean)' to calculate the mean of all variables by cut and color. Create a heatmap of the mean prices by cut and color using geom_tile**
12. **Change the title of the heatmap to "Average prices"**
13. **Change the gradient of the fill scale using 'scale_fill_gradient2'. Have it go from darkblue to white to darkred, set the midpoint to 4500**
14. **Choose and add a theme to the heatmap, or create a theme yourself using the options listed at http://ggplot2.tidyverse.org/reference/theme.html**

3 Using ggplot: Answers

1. **Load the ggplot2 library (install it if you have to) and the diamonds dataset using data()**

```
# if needed install.packages("ggplot2")
library(ggplot2)
data("diamonds")
```

2. **Explore the dataset using dim(), str() and help(), which variables are continuous, which variables are discrete? Is this dataset ready for plotting with ggplot?**

```
dim(diamonds)
str(diamonds)
help(diamonds)
```

Cut, color, and clarity are factors, and therefore discrete. The others are numeric continuous variables.

3. **Use ggplot to plot a scatterplot of the relationship between the diamonds' carat and their price**

```
ggplot() + geom_point(data=diamonds, aes(x=carat, y=price))

#or

ggplot(diamonds) + geom_point(aes(x=carat, y=price))

#or

ggplot(diamonds, aes(x=carat, y=price)) + geom_point()
```

4. **Make all dots darkblue and set the alpha value to 0.1**

```
ggplot(diamonds, aes(x=carat, y=price)) + geom_point(color="darkblue", alpha=0.1)
```

5. **Visualize the influence of the color of a diamond on its price by mapping the diamond color to the color aesthetic**

```
#The color of the dots will be overwritten if we specify it statically
#in the geom_point function itself
ggplot(diamonds, aes(x=carat, y=price, color=color)) + geom_point(alpha=0.1)
```

6. **Use a ggplot barplot to visualize diamond clarity depending on color, map diamond color to x and diamond clarity to fill**

```
ggplot(diamonds, aes(x=color, fill=clarity)) + geom_bar()
```

7. **Create a boxplot of the carat of a diamond based on its clarity and add whiskers using stat_boxplot**

```
ggplot(diamonds, aes(x=clarity, y=carat)) +
    stat_boxplot(geom="errorbar", width=0.5) +
    geom_boxplot()
```

8. **Add a geom_point layer to the previous plot mapping the diamonds price to the color**

```
ggplot(diamonds, aes(x=clarity, y=carat)) +
    stat_boxplot(geom="errorbar", width=0.5) +
    geom_boxplot() +
    geom_point(aes(color=price))
```

9. **Create a histogram of the price of the diamonds and separate the histograms into facets using diamond color, choose a good binwith or number of bins**

```
ggplot(diamonds, aes(x=price)) +
    geom_histogram(binwidth = 100) +
    facet_grid(color ~ .)
```

10. **Create a grid of facets of the same histogram by comparing both color and cut**

```
ggplot(diamonds, aes(x=price)) +
    geom_histogram(binwidth = 100) +
    facet_grid(color ~ cut)
```

11. **Use 'aggregate(diamonds, by = list(cut = diamonds$cut, color = diamonds$color), mean)' to calculate the mean of all variables by cut and color. Create a heatmap of the mean prices by cut and color using geom_tile**

```
#Aggregate uses a function (in this case mean) to aggregate all variables
#in a given data.frame by a list of variables given in "by"
mean.price <- aggregate(diamonds,
    by = list(cut = diamonds$cut, color = diamonds$color),
    mean)
ggplot(mean.price, aes(x=cut, y=color, fill=price)) +
    geom_tile()
```

12. **Change the title of the heatmap to "Average prices"**

```
ggplot(mean.price, aes(x=cut, y=color, fill=price)) +
    geom_tile() +
    labs(title="Average prices")
```

13. **Change the gradient of the fill scale using 'scale_fill_gradient2'. Have it go from darkblue to white to darkred, set the midpoint to 4500**

```
ggplot(mean.price, aes(x=cut, y=color, fill=price)) +
    geom_tile() +
    labs(title="Average prices") +
    scale_fill_gradient2(low="darkblue", mid="white", high="darkred", midpoint = 4500)
```

14. **Choose and add a theme to the heatmap, or create a theme yourself using the options listed at http://ggplot2.tidyverse.org/reference/theme.html**

```
ggplot(mean.price, aes(x=cut, y=color, fill=price)) +
    geom_tile() +
    labs(title="Average prices") +
    scale_fill_gradient2(low="darkblue", mid="white", high="darkred", midpoint = 4500) +
    theme_minimal()
```

Using ggplot: Practical

1. **Load the ggplot2 library (install it if you have to) and the diamonds dataset using data()**
2. **Explore the dataset using dim(), str() and help(), which variables are continuous, which variables are discrete? Is this dataset ready for plotting with ggplot?**
3. **Use ggplot to plot a scatterplot of the relationship between the diamonds" carat and their price**
4. **Make all dots darkblue and set the alpha value to 0.1**
5. **Visualize the influence of the color of a diamond on its price by mapping the diamond color to the color aesthetic**
6. **Use a ggplot barplot to visualize diamond clarity depending on color, map diamond color to x and diamond clarity to fill**
7. **Create a boxplot of the carat of a diamond based on its clarity and add whiskers using stat_boxplot**
8. **Add a geom_point layer to the previous plot mapping the diamonds price to the color**
9. **Create a histogram of the price of the diamonds and separate the histograms into facets using diamond color, choose a good binwith or number of bins**
10. **Create a grid of facets of the same histogram by comparing both color and cut**
11. **Use 'aggregate(diamonds, by = list(cut = diamonds$cut, color = diamonds$color), mean)' to calculate the mean of all variables by cut and color. Create a heatmap of the mean prices by cut and color using geom_tile**
12. **Change the title of the heatmap to "Average prices"**
13. **Change the gradient of the fill scale using 'scale_fill_gradient2'. Have it go from darkblue to white to darkred, set the midpoint to 4500**
14. **Choose and add a theme to the heatmap, or create a theme yourself using the options listed at http://ggplot2.tidyverse.org/reference/theme.html**

Using ggplot: Answers

1. **Load the ggplot2 library (install it if you have to) and the diamonds dataset using data()**

```
# if needed install.packages("ggplot2")
library(ggplot2)
data("diamonds")
```

2. **Explore the dataset using dim(), str() and help(), which variables are continuous, which variables are discrete? Is this dataset ready for plotting with ggplot?**

```
dim(diamonds)
str(diamonds)
help(diamonds)
```

Cut, color, and clarity are factors, and therefore discrete. The others are numeric continuous variables.

3. **Use ggplot to plot a scatterplot of the relationship between the diamonds' carat and their price**

```
ggplot() + geom_point(data=diamonds, aes(x=carat, y=price))

#or

ggplot(diamonds) + geom_point(aes(x=carat, y=price))

#or

ggplot(diamonds, aes(x=carat, y=price)) + geom_point()
```

4. **Make all dots darkblue and set the alpha value to 0.1**

```
ggplot(diamonds, aes(x=carat, y=price)) + geom_point(color="darkblue", alpha=0.1)
```

5. **Visualize the influence of the color of a diamond on its price by mapping the diamond color to the color aesthetic**

```
#The color of the dots will be overwritten if we specify it statically
#in the geom_point function itself
ggplot(diamonds, aes(x=carat, y=price, color=color)) + geom_point(alpha=0.1)
```

6. **Use a ggplot barplot to visualize diamond clarity depending on color, map diamond color to x and diamond clarity to fill**

```
ggplot(diamonds, aes(x=color, fill=clarity)) + geom_bar()
```

7. **Create a boxplot of the carat of a diamond based on its clarity and add whiskers using stat_boxplot**

```
ggplot(diamonds, aes(x=clarity, y=carat)) +
    stat_boxplot(geom="errorbar", width=0.5) +
    geom_boxplot()
```

8. **Add a geom_point layer to the previous plot mapping the diamonds price to the color**

```
ggplot(diamonds, aes(x=clarity, y=carat)) +
    stat_boxplot(geom="errorbar", width=0.5) +
    geom_boxplot() +
    geom_point(aes(color=price))
```

9. **Create a histogram of the price of the diamonds and separate the histograms into facets using diamond color, choose a good binwith or number of bins**

```
ggplot(diamonds, aes(x=price)) +
    geom_histogram(binwidth = 100) +
    facet_grid(color ~ .)
```

10. **Create a grid of facets of the same histogram by comparing both color and cut**

```
ggplot(diamonds, aes(x=price)) +
    geom_histogram(binwidth = 100) +
    facet_grid(color ~ cut)
```

11. **Use 'aggregate(diamonds, by = list(cut = diamonds$cut, color = diamonds$color), mean)' to calculate the mean of all variables by cut and color. Create a heatmap of the mean prices by cut and color using geom_tile**

```
#Aggregate uses a function (in this case mean) to aggregate all variables
#in a given data.frame by a list of variables given in "by".
#Note: excluding columns 2,3,4 since they are factors
#and therefore don't have means
mean.price <- aggregate(diamonds[, -c(2,3,4)],
    by = list(cutAgg = diamonds$cut, colorAgg = diamonds$color),
    mean)
ggplot(mean.price, aes(x=cutAgg, y=colorAgg, fill=price)) +
    geom_tile()
```

12. **Change the title of the heatmap to "Average prices"**

```
ggplot(mean.price, aes(x=cut, y=color, fill=price)) +
    geom_tile() +
    labs(title="Average prices")
```

13. **Change the gradient of the fill scale using 'scale_fill_gradient2'. Have it go from darkblue to white to darkred, set the midpoint to 4500**

```
ggplot(mean.price, aes(x=cut, y=color, fill=price)) +
    geom_tile() +
    labs(title="Average prices") +
    scale_fill_gradient2(low="darkblue", mid="white", high="darkred", midpoint = 4500)
```

14. **Choose and add a theme to the heatmap, or create a theme yourself using the options listed at http://ggplot2.tidyverse.org/reference/theme.html**

```
ggplot(mean.price, aes(x=cut, y=color, fill=price)) +
    geom_tile() +
    labs(title="Average prices") +
    scale_fill_gradient2(low="darkblue", mid="white", high="darkred", midpoint = 4500) +
    theme_minimal()
```

Basic Course on R: Linear Regression

Elizabeth Ribble*

18-24 May 2017

Contents

*emcclel3@msudenver.edu

Most of the following examples use the data "R_data_January2015.csv" which contains variables on mothers whose babies are either intellectually disabled or developmentally normal.

```
babies <- read.csv("R_data_January2015.csv",header=T,row.names=1)
names(babies)
```

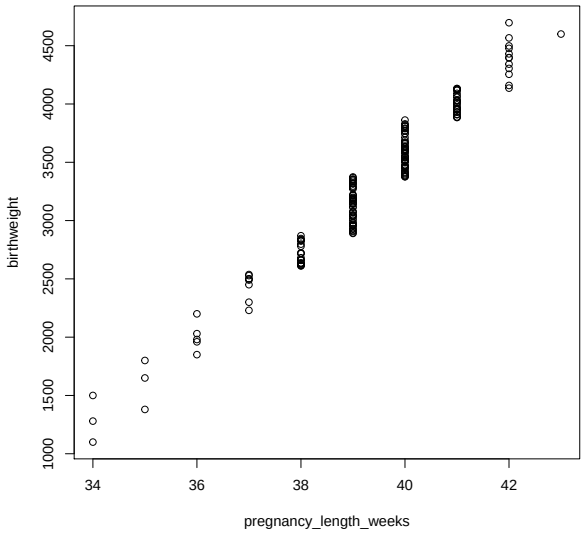
```
##  [1] "Status"                "iodine_deficiency"
##  [3] "BMI"                   "educational_level"
##  [5] "alcohol"               "smoking"
##  [7] "medication"            "birthweight"
##  [9] "pregnancy_length_weeks" "pregnancy_length_days"
## [11] "SAM"                   "SAH"
## [13] "homocysteine"          "cholesterol"
## [15] "HDL"                   "triglycerides"
## [17] "vitaminB12"            "folicacid_serum"
## [19] "folicacid_erys"
```

```
attach(babies)
```

1 Linear Regression Basics

There is a high positive correlation between birth weight and gestational age, but this says nothing about predictive power of the variables. We would like to explain how gestational age influences changes in birth weight.

```
plot(pregnancy_length_weeks,birthweight)
```



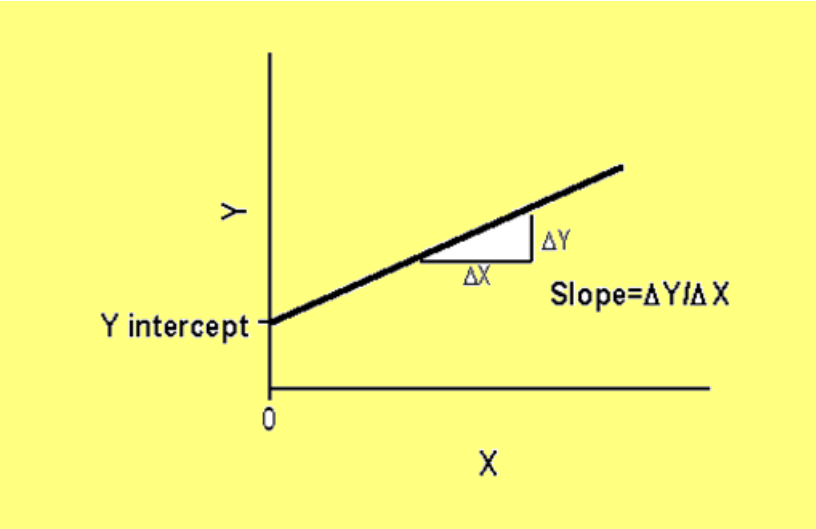

```
cor(pregnancy_length_weeks,birthweight)

## [1] 0.9785784
```

We'll quantify this relationship using linear regression, distinguishing between an independent, or predictor or explanatory, variable (gestational age) and a dependent, or response or outcome, variable (birth weight). Simple linear regression uses the following model:

$$\text{response}_i = (\text{model}_i) + \text{error}_i$$
$$y_i = b_0 + b_1 X_i + \epsilon_i$$

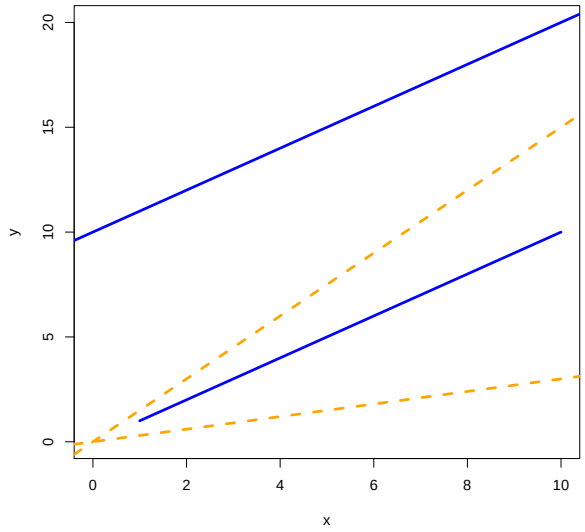
where $1 \leq i \leq n$, model is a straight line, and error is remaining variation which cannot be explained by the model.

The parameters b_0 and b_1 are the intercept and slope of a straight line, respectively:



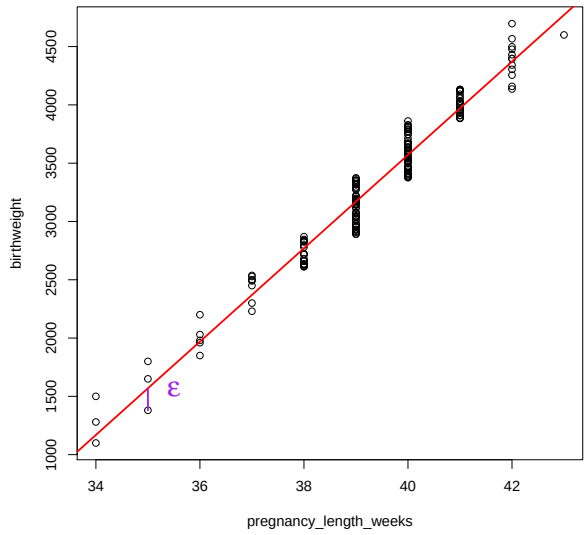
Note that intercept and slope represent different things:

```
plot(1:10, 1:10, type="l", col="blue", lwd=3,
     ylim=c(0,20), xlim=c(0,10), xlab="x", ylab="y")
abline(0, 1.5, lty=2, col="orange", lwd=3)
abline(0, .3, lty=2, col="orange", lwd=3)
abline(10, 1, lty=1, col="blue", lwd=3)
```



The error term ϵ_i denotes the residuals: the differences between observed values and the fitted line.

```
plot(pregnancy_length_weeks,birthweight)
abline(-12441.4, 400.3, col="red", lwd=2)
lines(c(35,35), c(1380,1568.921), lwd=2, col="purple")
text(35.5, 1560, expression(epsilon), cex=2, col="purple")
```



The b_0 and b_1 of the one straight line that best fits the data is estimated via the method of least squares. The "best" line is the one that has the lowest sum of squared residuals. The command to get these estimates in R is `lm`:

```
lm1 <- lm(birthweight ~ pregnancy_length_weeks)
lm1

##
## Call:
## lm(formula = birthweight ~ pregnancy_length_weeks)
##
## Coefficients:
##            (Intercept)  pregnancy_length_weeks
##                -12441.4                   400.3
```

and that's how I knew the line in the above plot should have intercept -12441.4 and slope 400.3! Also, the point on the line for 35 weeks is $-12441.4 + 400.3 * 35 = 1568.921$, which can be more accurately provided by

```
predict(lm1)[pregnancy_length_weeks==35]

##       12       73      141
## 1568.921 1568.921 1568.921
```

You can also use `predict` to predict y for x's that are not already in your data:

```
predict(lm1, newdata=data.frame(pregnancy_length_weeks=c(seq(25,50,5))))

##          1          2          3          4          5          6
## -2434.0284  -432.5538  1568.9207  3570.3952  5571.8698  7573.3443
```

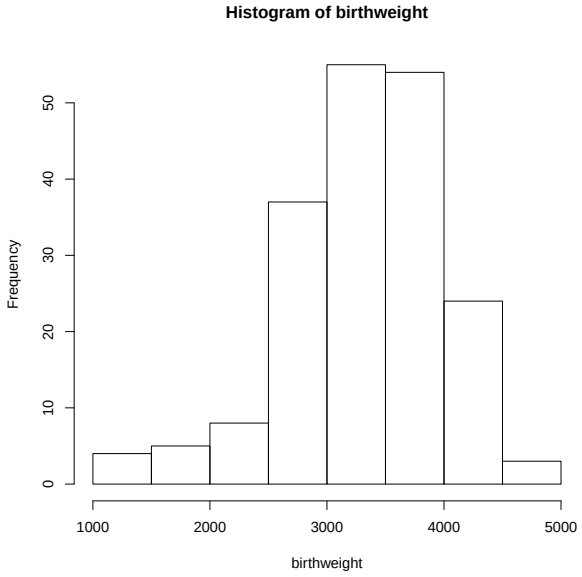
but watch out for extrapolating (predicting outside the range of your data) - clearly we can't have negative birth weights!

Now, before we make inference on or, actually, even find and use this line, we **must** check that the following assumptions hold, otherwise we will not obtain trustworthy results:

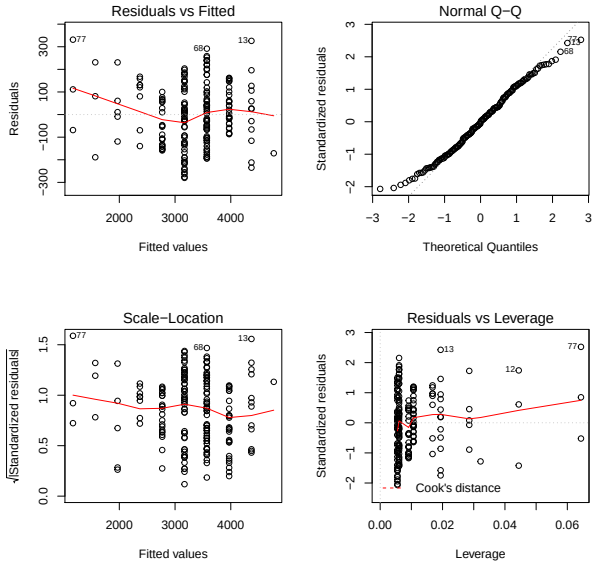
- relationship between x and y can be described by a straight line
- outcomes y are independent
- variance of residuals is constant across values of x
- residuals follow a normal distribution

To get diagnostic plots in R, we can check a histogram of the data and additionally plot our model to check that the residuals are normal and homoscedastic (have constant variance) across the weeks:

```
hist(birthweight)
```

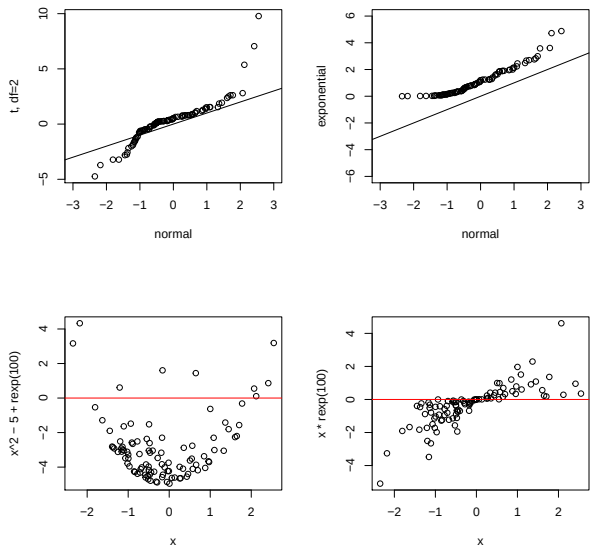



```
par(mfrow=c(2,2))
plot(lm1)
```



The top left plot tells us if our residuals are homoscedastic, and the top right plot displays a quantile-quantile (QQ) plot to check for normality. Here are examples of bad QQ plots and heteroscedasticity:

```
set.seed(1234)
par(mfrow=c(2,2))
x <- sort(rnorm(100))
y1 <- sort(rt(100,2))
plot(x, y1, xlim=c(-3,3), xlab="normal", ylab="t, df=2")
abline(0, 1)
y2 <- sort(rexp(100))
plot(x, y2, xlim=c(-3,3), ylim=c(-6,6), xlab="normal", ylab="exponential")
abline(0, 1)
plot(x, x^2-5+rexp(100))
abline(0, 0, col="red")
plot(x, x*rexp(100))
abline(0, 0, col="red")
```



However, the assumptions reasonably hold for our baby data, so we'll go ahead and use the model fit to make inference on the slope b_1.

Actually, R has already done the inference...we just need to extract it from the model:

```
summary(lm1)

##
## Call:
## lm(formula = birthweight ~ pregnancy_length_weeks)
```

```
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -280.1 -106.7   -2.9  101.2  331.4
##
## Coefficients:
##                           Estimate Std. Error t value Pr(>|t|)
## (Intercept)            -12441.401    242.033  -51.40   <2e-16 ***
## pregnancy_length_weeks    400.295      6.142   65.17   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 135.7 on 188 degrees of freedom
## Multiple R-squared:  0.9576,Adjusted R-squared:  0.9574
## F-statistic:  4248 on 1 and 188 DF,  p-value: < 2.2e-16
```

R has performed a one-sample t-test on the intercept b_1 and slope b_0 to determine if they are each statistically significantly different from 0. The probabilities are quite small, so we can reject the null hypothesis that they are equal to 0 and conclude that birthweight significantly increases, on average, by 400 grams per every additional week of gestation. The intercept is (usually) unimportant and we don't really care that it is different from 0. If the p-value for the slope is not small (e.g. greater than 0.05) then we would say "we do not have enough evidence to reject the null hypothesis that the slope is 0."

Now, how good does the model actually fit our data? How well does x predict y? In a previous lecture, I told you that the square of Pearson's correlation coefficient, r, is a measure of goodness of fit. It is the proportion of variance in y that can be explained by the model (so, x). In our example, r^2 is:

```
cor(pregnancy_length_weeks,birthweight)^2
```

```
## [1] 0.9576157
```

```
summary(lm1)$r.squared
```

```
## [1] 0.9576157
```

which means that 96% of the variability in birth weight can be explained by gestational age.

2 Multiple Linear Regression

Simple linear regression models one y on one x. If we have multiple predictor variables, we use multiple linear regression to determine if the variability in y can be explained by this set of variables. In addition to the assumptions required for a valid simple linear regression, we now include that the covariates have no perfect multicollinearity, that is there is no strong correlation between the multiple x's. The model is

$$y = b_0 + b_1X_1 + b_2X_2 + ... + b_kX_k + \epsilon_i$$

In R, the addition of an extra variable is quite straightforward:

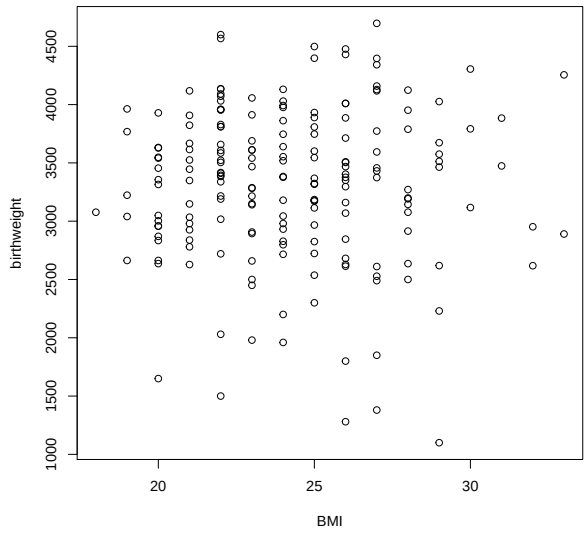
```
cor(pregnancy_length_weeks, BMI)

## [1] 0.01068054

lm2 <- lm(birthweight ~ pregnancy_length_weeks + BMI)
lm2stats <- summary(lm2)
lm2stats

##
## Call:
## lm(formula = birthweight ~ pregnancy_length_weeks + BMI)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -279.39 -105.78   -1.85  105.25  333.50
##
## Coefficients:
##                          Estimate Std. Error t value Pr(>|t|)
## (Intercept)           -12377.339    253.444 -48.837   <2e-16 ***
## pregnancy_length_weeks    400.351      6.147  65.133   <2e-16 ***
## BMI                        -2.738      3.190  -0.858    0.392
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 135.8 on 187 degrees of freedom
## Multiple R-squared:  0.9578,Adjusted R-squared:  0.9573
## F-statistic:  2121 on 2 and 187 DF,  p-value: < 2.2e-16
```

Since the 2 variables are uncorrelated, we can add BMI to the model. We see that it does not significantly predict birth weight, but gestational age still does. We use the adjusted r^2 to check goodness of fit:

```
lm2stats$adj.r.squared

## [1] 0.9573304
```

So adding BMI does not help explain any of the variability in birth weight (since r^2 was previously already 0.96). This is also confirmed by visualization:

```
plot(BMI,birthweight)
```



Note that the `summary` function returns a lot of information. If, for example, you wanted to extract only the p-values you could do the following:

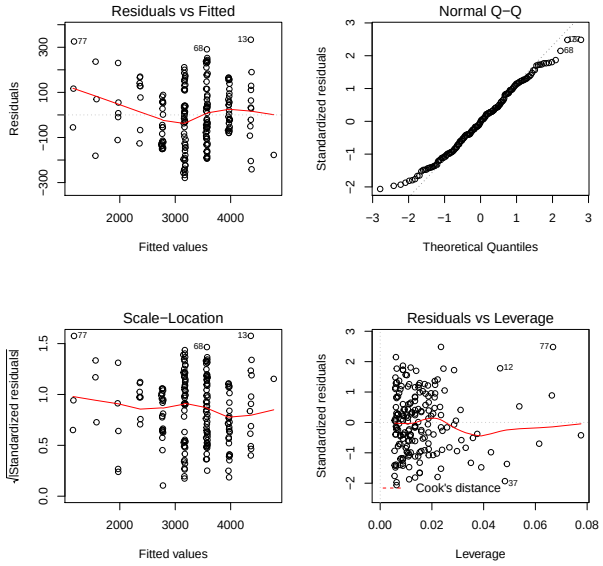
```
names(lm2stats)

##  [1] "call"          "terms"         "residuals"     "coefficients"
##  [5] "aliased"       "sigma"         "df"            "r.squared"
##  [9] "adj.r.squared" "fstatistic"    "cov.unscaled"

lm2stats$coef[,4]

##            (Intercept) pregnancy_length_weeks                     BMI
##          2.179929e-108          1.815592e-130           3.918778e-01
```

We can predict birthweights with new data:

```
predict(lm2, newdata=data.frame(pregnancy_length_weeks=32,
                                BMI=30))

##       1
## 351.771

## same as
lm2coefs <- lm2$coef
lm2coefs[1] + lm2coefs[2]*32 + lm2coefs[3]*30

## (Intercept)
##     351.771
```

But we cannot forget to check assumptions!

```
par(mfrow=c(2,2))
plot(lm2)
```



Basic Course on R: Linear Regression Practical

Elizabeth Ribble*

18-24 May 2017

Contents

*emcclel3@msudenver.edu

1 Baby Data

1. Read in the data "R_data_January2015.csv" with a header and row names from the first column. Assign it to the object `babydata` and allow strings be converted to factors. Attach the data to the environment.

2. We previously saw that there was an association between `vitaminB12` and `homocysteine`. We will now quantify the magnitude of this relationship and see if we can explain the variabiliy of `homocysteine` with values of `vitaminB12`. Answer the following questions:

   (a) First plot the data: a scatterplot to visualize the association (should be linear) and a histogram of the dependent variable to check for normality (normal errors are required - checking the distribution of the dependent variable is good enough for now). Do these particular assumptions appear to hold?

   (b) You should have noticed that the dependent variable `homocysteine` is right-skewed. We need to see if a log-transformation of the data is more normally distributed. Assign the log of `homocysteine` to `loghc` and make a histogram of `loghc`. Check that it is normal by plotting it against a normal distribution (use `qqnorm()`). How does it look? More normal?

   (c) We'll just assume everything looks good. Set up (i.e. write down) the linear regression model for modeling `loghc` against `vitaminB12` and then run it in R. Check the assumptions by plotting the residuals (should be no patterns) versus the fitted values and looking at a QQ plot of the residuals (should be straight line). Do the assumptions hold?

(d) Assuming the assumptions hold (even if they don't), we'll make inference on the slope. Is `vitaminB12` statistically significant in the model? What percent variability does it explain in `loghc`? Write down the model with the estimates.

(e) What is the predicted `loghc` level for a person with `vitaminB12` equal to 650? What is this value when unlogged (exponentiated)? Does it fall within the original range of values of `homocysteine` (can you guess what the name of the function is to find the range of a vector?)?

3. Now let's consider a framework where we want to use more than one predictor. We want to build a regression model for `SAM` using `vitaminB12`, `cholesterol`, `homocysteine` and `folicacid_erys` (folic acid red blood cells). Answer the following questions:

   (a) Set up (i.e. write down) the linear regression model and then run it in R. Check the assumptions by plotting the residuals versus the fitted values and looking at a QQ plot of the residuals. Do the assumptions hold?

(b) Assuming the assumptions hold (even if they don't), we'll make inference on the slopes. Are any of the variables statistically significant in the model? What percent variability do the variables explain in `SAM`? Write down the model with the estimates.

(c) What is the predicted `SAM` level for a person with the following:

$$\begin{aligned} \texttt{vitaminB12} &= 650 \\ \texttt{cholesterol} &= 17 \\ \texttt{homocysteine} &= 16 \\ \texttt{folicacid\_erys} &= 1340 \end{aligned}$$

Basic Course on R: Linear Regression Practical Answers

Elizabeth Ribble*

17-21 December 2018

Contents

*emcclel3@msudenver.edu

1 Baby Data

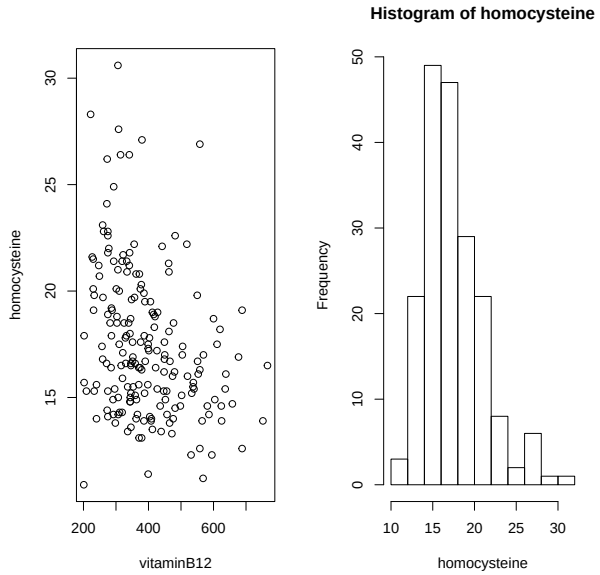
1. Read in the data "R_data_January2015.csv" with a header and row names from the first column. Assign it to the object `babydata` and allow strings be converted to factors. Attach the data to the environment.

```
babydata <- read.csv("R_data_January2015.csv",header=T,row.names=1)
attach(babydata)
```

2. We previously saw that there was an association between `vitaminB12` and `homocysteine`. We will now quantify the magnitude of this relationship and see if we can explain the variabiliy of `homocysteine` with values of `vitaminB12`. Answer the following questions:

   (a) First plot the data: a scatterplot to visualize the association (should be linear) and a histogram of the dependent variable to check for normality (normal errors are required - checking the distribution of the dependent variable is good enough for now). Do these particular assumptions appear to hold?

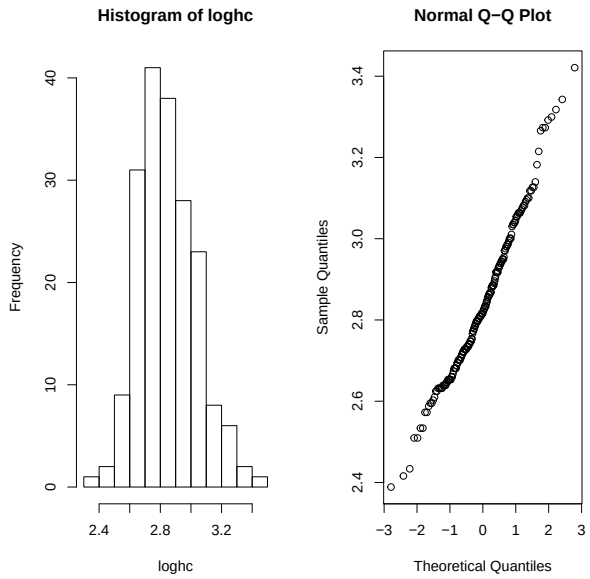
```
par(mfrow=c(1,2))
plot(vitaminB12, homocysteine)
hist(homocysteine)
```



There is a slight linear association but looks like the variable isn't normal...

(b) You should have noticed that the dependent variable `homocysteine` is right-skewed. We need to see if a log-transformation of the data is more normally distributed. Assign the log of `homocysteine` to `loghc` and make a histogram of `loghc`. Check that it is normal by plotting it against a normal distribution (use `qqnorm()`). How does it look? More normal?

```
loghc <- log(homocysteine)
par(mfrow=c(1,2))
hist(loghc)
set.seed(1234)
qqnorm(loghc)
```

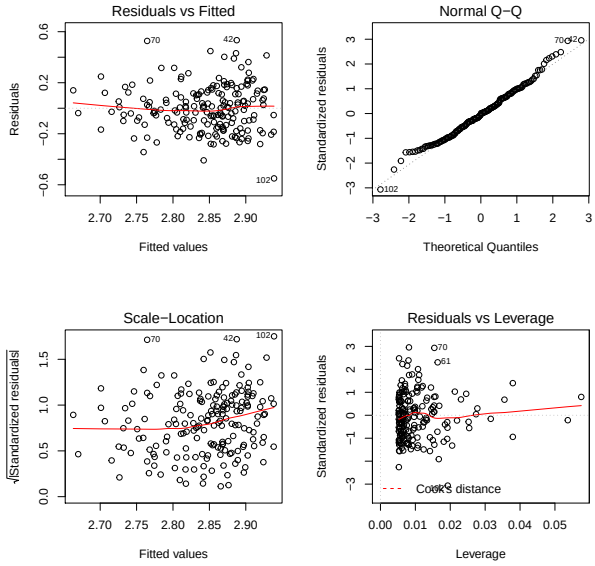


Doesn't look too bad!

(c) We'll just assume everything looks good. Set up (i.e. write down) the linear regression model for modeling `loghc` against `vitaminB12` and then run it in R. Check the assumptions by plotting the residuals (should be no patterns) versus the fitted values and looking at a QQ plot of the residuals (should be straight line). Do the assumptions hold?
The model is

$$\texttt{loghc} \sim b_0 + b_1 * \texttt{vitaminB12} + \epsilon.$$

```
lmbh <- lm(loghc ~ vitaminB12)
par(mfrow=c(2,2))
plot(lmbh)
```



There is a slight fan pattern in the residuals, but it's not too strong. The tails of the distribution deviate from the normal quantile line, but again it's not that bad.

(d) Assuming the assumptions hold (even if they don't), we'll make inference on the slope. Is `vitaminB12` statistically significant in the model? What percent variability does it explain in `loghc`? Write down the model with the estimates.

```
summary(lmbh)

##
## Call:
## lm(formula = loghc ~ vitaminB12)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -0.5501 -0.1288 -0.0041  0.1176  0.5334
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  3.0369329  0.0457556  66.373  < 2e-16 ***
## vitaminB12  -0.0004881  0.0001113  -4.386 1.92e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.1812 on 188 degrees of freedom
```

```
## Multiple R-squared:  0.09281,Adjusted R-squared:  0.08799
## F-statistic: 19.23 on 1 and 188 DF,  p-value: 1.923e-05

lmbh

##
## Call:
## lm(formula = loghc ~ vitaminB12)
##
## Coefficients:
## (Intercept)   vitaminB12
##   3.0369329   -0.0004881
```

`vitaminB12` is statistically significant (p-value less than 0.05). It explains about 9.3 percent of the variability in `loghc`. The model with the estimates is

$$\texttt{loghc} \sim 3.04 - 0.0005 * \texttt{vitaminB12}.$$

(e) What is the predicted `loghc` level for a person with `vitaminB12` equal to 650? What is this value when unlogged (exponentiated)? Does it fall within the original range of values of `homocysteine` (can you guess what the name of the function is to find the range of a vector?)?

```
p650 <- predict(lmbh, newdata=data.frame(vitaminB12=650))
## or
3.04 - 0.0005*650

## [1] 2.715

p650

##        1
## 2.719635

exp(p650)

##        1
## 15.17479

range(homocysteine)

## [1] 10.9 30.6
```

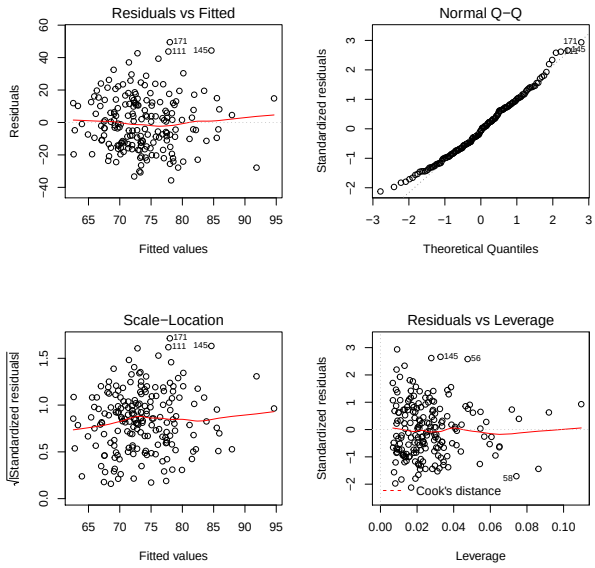
Yes, 15.1 is in the range of the unlogged `homocysteine` values: (10.9, 30.6).

3. Now let's consider a framework where we want to use more than one predictor. We want to build a regression model for `SAM` using `vitaminB12`, `cholesterol`, `homocysteine` and `folicacid_erys` (folic acid red blood cells). Answer the following questions:

   (a) Set up (i.e. write down) the linear regression model and then run it in R. Check the assumptions by plotting the residuals versus the fitted values and looking at a QQ plot of the residuals. Do the assumptions hold?
   The model is

   $$\texttt{SAM} \sim b_0 + b_1 * \texttt{vitaminB12} + b_2 * \texttt{cholesterol} + b_3 * \texttt{homocysteine} + b_4 * \texttt{folicacid\_erys} + \epsilon.$$

```
mlr <- lm(SAM ~ vitaminB12 + cholesterol + homocysteine +
                folicacid_erys)
par(mfrow=c(2,2))
plot(mlr)
```



   The residuals have constant variance and are normal, so the assumptions do hold.

   (b) Assuming the assumptions hold (even if they don't), we'll make inference on the slopes. Are any of the variables statistically significant in the model? What percent variability do the variables explain in `SAM`? Write down the model with the estimates.

```
summary(mlr)

##
## Call:
## lm(formula = SAM ~ vitaminB12 + cholesterol + homocysteine +
##     folicacid_erys)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -35.704 -12.367  -1.047  11.972  49.505
##
## Coefficients:
##                 Estimate Std. Error t value Pr(>|t|)
## (Intercept)     1.800739  27.877405   0.065 0.948566
## vitaminB12     -0.002684   0.011286  -0.238 0.812288
## cholesterol     4.583482   1.369695   3.346 0.000992 ***
## homocysteine   -0.645535   0.413510  -1.561 0.120206
## folicacid_erys  0.005033   0.006812   0.739 0.460939
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 16.94 on 185 degrees of freedom
## Multiple R-squared:  0.09232,Adjusted R-squared:  0.0727
## F-statistic: 4.704 on 4 and 185 DF,  p-value: 0.001225

mlr

##
## Call:
## lm(formula = SAM ~ vitaminB12 + cholesterol + homocysteine +
##     folicacid_erys)
##
## Coefficients:
##    (Intercept)      vitaminB12     cholesterol    homocysteine
##       1.800739       -0.002684        4.583482       -0.645535
## folicacid_erys
##       0.005033
```

Only `cholesterol` is statistically significant (p-value less than 0.05). The 4 variables explain 7.27 percent of the variability in `SAM`. The model with the estimates is

$$\texttt{SAM} \sim 1.8 - 0.003 * \texttt{vitaminB12} + 4.58 * \texttt{cholesterol} - 0.65 * \texttt{homocysteine} + 0.005 * \texttt{folicacid\_erys}.$$

(c) What is the predicted SAM level for a person with the following:

$$\begin{aligned} \texttt{vitaminB12} &= 650 \\ \texttt{cholesterol} &= 17 \\ \texttt{homocysteine} &= 16 \\ \texttt{folicacid\_erys} &= 1340 \end{aligned}$$

```
predict(mlr, newdata=data.frame(vitaminB12=650,
                                cholesterol=17,
                                homocysteine=16,
                                folicacid_erys=1340))
```

```
##        1
## 74.39131
```

Basic Course on R: Logistic Regression

Elizabeth Ribble*

18-24 May 2017

Contents

*emcclel3@msudenver.edu

Most of the following examples use the data "R_data_January2015.csv" which contains variables on mothers whose babies are either intellectually disabled or developmentally normal.

```
babies <- read.csv("R_data_January2015.csv", header = T, row.names = 1)
names(babies)
```

```
##  [1] "Status"                "iodine_deficiency"
##  [3] "BMI"                   "educational_level"
##  [5] "alcohol"               "smoking"
##  [7] "medication"            "birthweight"
##  [9] "pregnancy_length_weeks" "pregnancy_length_days"
## [11] "SAM"                   "SAH"
## [13] "homocysteine"          "cholesterol"
## [15] "HDL"                   "triglycerides"
## [17] "vitaminB12"            "folicacid_serum"
## [19] "folicacid_erys"
```

```
attach(babies)
```

1 When to Use Logistic Regression

There are many research topics for with the dependent variable y is binary (0/1), e.g.

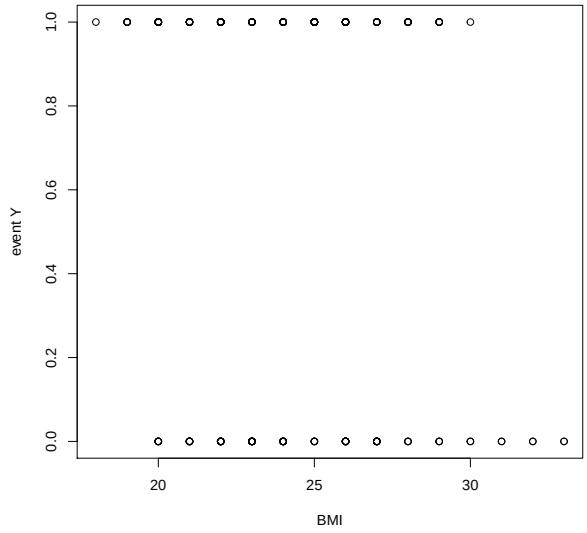
- mortality (dead/alive)
- treatment response (responder/non-responder)
- development of disease (yes/no)

and we want to predict the membership of an individual to one of the two categories based on a set of predictors.

In this situation we have to work with **probabilities**, which are numbers between 0 and 1. A value $P(y)$ that is close to **0** means that y is very **unlikely** to occur, while a value close to **1** means that y is very **likely** to occur.

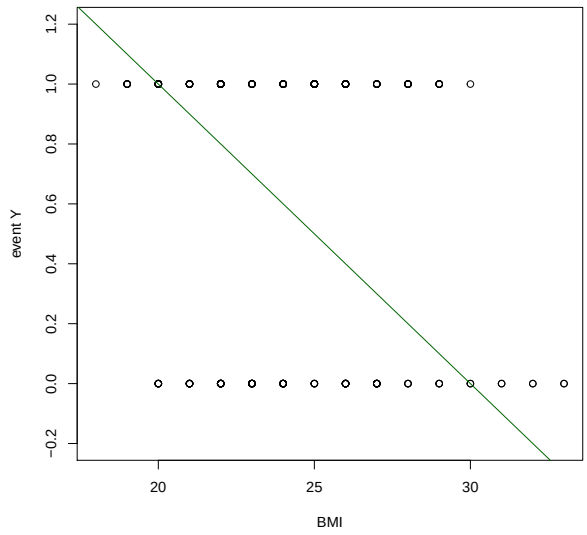
In simple/multiple linear regression a continuous variable y is predicted by continuous/categorical x(s), but if y can only have 2 values (e.g. $y = 0$ or $y = 1$), how do we predict the probability that $y = 1$ given one or more predictors? Could we apply a linear regression model...?

```
plot(BMI, as.numeric(Status)-1, ylab = "event Y")
```



If we try to fit a **linear** regression model...

```
plot(BMI, as.numeric(Status)-1, ylab = "event Y", ylim = c(-.2,1.2))
abline(3, -.1, col = "darkgreen")
```



we would try to fit $P(y = 1) = b_0 + b_1 x$, which doesn't work. In linear regression we assume the relationship between x and y is linear, but in the case of binary outcomes this assumption is no longer valid. Results obtained from a linear regression model wouldn't makes sense! Probabilities beyond the interval (0,1) are not interpretable.

Instead we're going to use a logistic curve! First, note that our factor `Status` has levels that are ordered alphabetically:

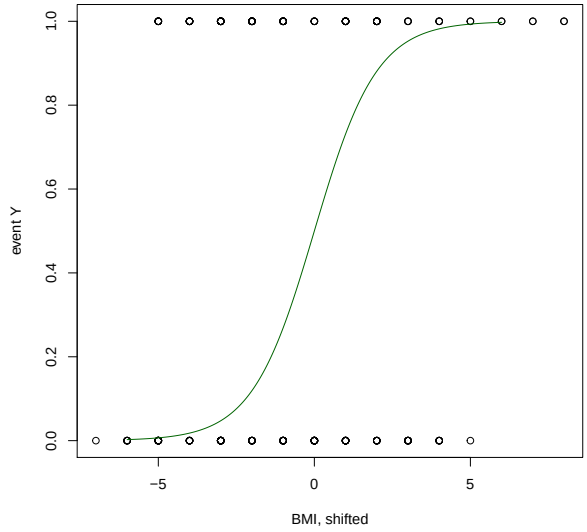
```
levels(Status)

## [1] "intellectual disability"  "normal brain development"

Status[1:3]

## [1] intellectual disability  normal brain development
## [3] intellectual disability
## Levels: intellectual disability normal brain development

as.numeric(Status)[1:3]

## [1] 1 2 1
```

But we would prefer the non-event to be the baseline so we will change `Status` so the event $y = 1$ is intellectual disability and the non-event $y = 0$ is normal brain development. Note that this re-coding does not change the variable in `babies`).

```
newstatus <- 2-as.numeric(Status)
newstatus[1:3]

## [1] 1 0 1
```

So now our visualization should make more sense:

```
plot(BMI-25, newstatus, ylab = "event Y",xlab = "BMI, shifted")
x <- seq(-6, 6, 0.01)
lines(x, exp(x)/(1+exp(x)), type = "l", col = "darkgreen", cex = 2)
```



The forumula we used for the line is `exp(x)/(1+exp(x))`, which is called the logistic function and it is the probability of an event, $P(y = 1)$. Let's see how to derive it...

2 Odds

We're going to start by introducing odds. The odds is the ratio of the probability that the event of interest occurs to the probability that it does not:

$$\text{odds} = \frac{P(\text{event})}{P(\text{no event})} \Leftrightarrow P(\text{event}) = \frac{\text{odds}}{1 + \text{odds}}$$

and note that P(no event)=1-P(event). Describing the probability of event y in terms of odds has a very convenient property:

- as odds increases, $p(y = 1)$ approaches 1
- as odds decreases, $p(y = 1)$ approaches 0

The odds ratio (OR) is a way of comparing whether the probability of a certain event is the same for two groups.

Let's go back to our example from earlier:

	event	no event	total
treatment	20	80	100
placebo	50	50	100
total	70	130	200

then

Total Group			
$P(y=1)$	$= x/n$	$= 70/200$	$= 0.35$
$\text{odds}(y=1)$	$= p/(1-p)$	$= 0.35/0.65$	$= 0.54$
Treated Group			
$P_1(y=1)$	$= x_1/n_1$	$= 20/100$	$= 0.20$
$\text{odds}_1(y=1)$	$= p_1/(1-p_1)$	$= 0.20/0.80$	$= 0.25$
Placebo Group			
$P_0(y=1)$	$= x_0/n_0$	$= 50/100$	$= 0.50$
$\text{odds}_0(y=1)$	$= p_0/(1-p_0)$	$= 0.50/0.50$	$= 1.00$

So $\text{OR}(\text{event}) = \text{odds}_1/\text{odds}_0 = 0.25/1.00 = 0.25$. Thus the odds of an event in the treatment group is 0.25 times lower than the odds of an event in the placebo group. Conversely, $1/0.25 = 4$, so we can also say the odds of an event in the placebo group is 4 times higher than the odds of an event in the treatment group. In R:

```
treatment <- c(20, 80)
placebo <- c(50, 50)
mytable <- rbind(treatment, placebo)
mytable[1,1]*mytable[2, 2]/(mytable[1, 2]*mytable[2, 1])

## treatment
##      0.25
```

3 Simple Logistic Regression Model

We are going to use odds in the following way, where $p = P(y = 1)$ is the probability of an event:

$$\ln\left(\frac{p}{1-p}\right) = b_0 + b_1 x$$

The function $\ln(p/(1-p))$ is called a logit of p and it is this function of y that is linear in x instead of y itself.

This model formulation assures that the predicted probability of an event falls between 0 and 1, unlike a linear regression model. Note:

$$\ln\left(\frac{p}{1-p}\right) = b_0 + b_1 x \Leftrightarrow p = \frac{e^{(b_0+b_1 x)}}{1+e^{(b_0+b_1 x)}}$$

so we've completed our derivation of the logistic function formula from the beginning.

Also note that we've made no assumptions about linearity, normality or homoscedasticity!

The values of the intercept and slope are estimated using the maximum likelihood method, which finds the values of the coefficients that make the observed data most likely to occur.

Statistical significance of estimate coefficients is testing with the Wald test, which is based on the χ^2 distribution (though R calls it "z"). The goodness of fit is assessed by deviance, which is based on the differences observed-expected principle. Also, the Akaike information criterion (AIC) gives a measure of the quality of the model.

The coefficients are most usefully interpreted with the following:

$$e^{b_1} = \frac{\text{odds after a unit change in the predictor}}{\text{original odds}}$$

and when x is binary, e^{b_1} is the odds ratio from the 2 by 2 contingency table!

In R:

```
lr1 <- glm(newstatus ~ BMI, family = binomial(logit))
summary(lr1)

##
## Call:
## glm(formula = newstatus ~ BMI, family = binomial(logit))
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.2759  -1.0805  -0.9168   1.2680   1.4627
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) -2.40591    1.18095  -2.037   0.0416 *
```

```
## BMI          0.08782    0.04822   1.821   0.0685 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 259.83  on 189  degrees of freedom
## Residual deviance: 256.44  on 188  degrees of freedom
## AIC: 260.44
##
## Number of Fisher Scoring iterations: 4
```

so $\text{logit}(p) = -2.40591 + 0.08782 * \text{BMI}$. Thus the probability of having a baby with an intellectual disability when BMI is large, say 32, is:

```
logit_p1 <- -2.40591+0.08782*32
logit_p1
```

```
## [1] 0.40433
```

```
## or
predict(lr1, newdata = data.frame(BMI = 32), se.fit = TRUE)
```

```
## $fit
##         1
## 0.4043502
##
## $se.fit
## [1] 0.3995839
##
## $residual.scale
## [1] 1
```

```
p1 <- exp(logit_p1)/(1 + exp(logit_p1))
p1
```

```
## [1] 0.5997275
```

The coefficient $b_1 = 0.08782$ can be exponentiated to obtain the odds ratio:

```
summary(lr1)$coef
```

```
##                Estimate Std. Error   z value   Pr(>|z|)
## (Intercept) -2.40591055 1.18095490 -2.037259 0.04162413
## BMI          0.08782065 0.04821621  1.821393 0.06854720
```

```
b1 <- summary(lr1)$coef[2, 1]
b1
```

```
## [1] 0.08782065
```

```
exp(b1)
```

```
## [1] 1.091792
```

Like multiple linear regression, we can add variables into the model here as well:

```
lr2 <- glm(newstatus ~ BMI + smoking, family = binomial(logit))
summary(lr2)$coef
```

```
##                Estimate Std. Error   z value    Pr(>|z|)
## (Intercept) -2.52420359 1.22600372 -2.058887 0.039505027
## BMI          0.08128428 0.04997051  1.626645 0.103812552
## smokingyes   1.34305855 0.38824022  3.459349 0.000541482
```

Thus the probability of having a baby with an intellectual disability when BMI is large, say 32, and the mother is a smoker is:

```
mynewdata <- data.frame(BMI = 32, smoking = factor("yes"))
logit_p2 <- predict(lr2, newdata = mynewdata)
p2 <- exp(logit_p2)/(1+exp(logit_p2))
p2
```

```
##         1
## 0.8053309
```

Basic Course on R: Logistic Regression Practical

Elizabeth Ribble*

14-18 May 2018

Contents

*emcclel3@msudenver.edu

1 Baby Data

1. Read in the data "R_data_January2015.csv" with a header and row names from the first column. Assign it to the object `babydata` and allow strings be converted to factors. Attach the data to the environment.

2. We would like to know if `smoking` predicts `Status`. Since `Status` is a binary variable (intellectual disability or normal brain development) we need to use logistic regression. Answer the following questions:

   (a) Write down the model with $\text{logit}(p)$, $\ln(p/(1-p))$, on the lefthand side, but instead of writing p write $P(\text{intellectual disability})$. Then write down the formula for this probability (the probabilty of having a baby with an intellectual disability).

   (b) If we run the model on the data as it is now, R will consider "normal brain development" as the event because it is second in the levels of `Status`:

```
levels(Status)

## [1] "intellectual disability"  "normal brain development"
```

   So we need to first change these factor levels so we treat "intellectual disability" as the event and "normal brain development" as the baseline. Run the following to change all the 1s to 2s and 2s to 1s:

```
table(Status)

## Status
##  intellectual disability normal brain development
##                       82                      108

newstatus <- factor(3-as.numeric(Status),
                    labels=c("normal brain development",
                             "intellectual disability"))
```

And to show that it worked:

```
levels(newstatus)

## [1] "normal brain development" "intellectual disability"

table(newstatus)

## newstatus
## normal brain development  intellectual disability
##                      108                       82
```

(c) Now run the regression model you set up above in R using `newstatus`. Then write down the model and probability of event with the estimates.

(d) Can `smoking` significantly predict `newstatus`? [Hint: use `summary`.]

(e) What is the probability of having a baby with an intellectual disability given the mother smokes?

(f) Our estimate of b_1 is the element in the 2nd row and 1st column of the coefficients from the `summary` call. What is the value of e^{b_1}? [Hint: use `exp`.]

(g) Is the e^{b_1} that you just calculated an odds ratio? How do you interpret it?

(h) What do you think e^{b_1} would have been if we didn't change the levels of `Status`? Re-run the model using `Status` to check your answer. How does it relate to your answer from (f)?

(i) There is another way to calculate an odds ratio without using logistic regression. Suppose we have the following 2×2 contingency table:

	event	no event
predictor yes	a	b
predictor no	c	d

then the odds ratio is $(a * d)/(b * c)$. Create a contingency table of `smoking` and `newstatus` [Hint: use `table`] and then calculate the odds ratio from that. Do you get the same answer as in (f)?

3. We would like to know if `smoking` and `vitaminB12` can be used to predict `newstatus`. Answer the following questions:

   (a) Set up (i.e. write down) the logit model and run it in R. Write the model with the estimates.

   (b) Can either variable significantly predict `newstatus`?

   (c) What is the probability of having a baby with an intellectual disability given the mother smokes and has a vitaminB12 level of 400? What is the probability of having a baby with an intellectual disability given the mother smokes and has a vitaminB12 level of 650?

Basic Course on R: Logistic Regression Practical Answers

Elizabeth Ribble*

17-21 December 2018

Contents

*emcclel3@msudenver.edu

1 Baby Data

1. Read in the data "R_data_January2015.csv" with a header and row names from the first column. Assign it to the object `babydata` and allow strings be converted to factors. Attach the data to the environment.

```
babydata <- read.csv("R_data_January2015.csv",header=T,row.names=1)
attach(babydata)
```

2. We would like to know if `smoking` predicts `Status`. Since `Status` is a binary variable (intellectual disability or normal brain development) we need to use logistic regression. Answer the following questions:

   (a) Write down the model with logit(p), $\ln(p/(1-p))$, on the lefthand side, but instead of writing p write P(intellectual disability). Then write down the formula for this probability (the probability of having a baby with an intellectual disability).
   The model is

$$\ln\left(\frac{P(\text{intellectual disability})}{1-P(\text{intellectual disability})}\right) = b_0 + b_1 * \text{smoking}$$

   and the probability of an event is

$$P(\text{intellectual disability}) = \frac{e^{b_0+b_1*\text{smoking}}}{1+e^{b_0+b_1*\text{smoking}}}.$$

   (b) If we run the model on the data as it is now, R will consider "normal brain development" as the event because it is second in the levels of `Status`:

```
levels(Status)

## [1] "intellectual disability"  "normal brain development"
```

   So we need to first change these factor levels so we treat "intellectual disability" as the event and "normal brain development" as the baseline. Run the following to change all the 1s to 2s and 2s to 1s

```
table(Status)

## Status
##  intellectual disability normal brain development
##                       82                      108

newstatus <- factor(3-as.numeric(Status),
                    labels = c("normal brain development",
                               "intellectual disability"))
```

And to show that it worked:

```
levels(newstatus)

## [1] "normal brain development" "intellectual disability"

table(newstatus)

## newstatus
## normal brain development  intellectual disability
##                      108                       82
```

(c) Now run the regression model you set up above in R using `newstatus`. Then write down the model and probability of event with the estimates.

```
lrs1 <- glm(newstatus ~ smoking, family = binomial(logit))
lrs1$coef

## (Intercept)  smokingyes
##   -0.557015    1.367945
```

The model with estimates is

$$\ln\left(\frac{P(\text{intellectual disability})}{1-P(\text{intellectual disability})}\right) = -0.557 + 1.368 * \text{smoking}$$

and the probability of an event is

$$P(\text{intellectual disability}) = \frac{e^{-0.557+1.368*\text{smoking}}}{1+e^{-0.557+1.368*\text{smoking}}}.$$

(d) Can `smoking` significantly predict `newstatus`? [Hint: use `summary`.]

```
summary(lrs1)$coef

##              Estimate Std. Error   z value     Pr(>|z|)
## (Intercept) -0.557015  0.1691109 -3.293786 0.0009884766
## smokingyes   1.367945  0.3859647  3.544224 0.0003937712
```

The p-value is less than 0.05 so `smoking` can be used as a predictor for `newstatus`.

(e) What is the probability of having a baby with an intellectual disability given the mother smokes?

```
mynew1 <- data.frame(smoking = factor("yes"))
logit_p1 <- predict(lrs1, newdata = mynew1)
p1 <- exp(logit_p1)/(1 + exp(logit_p1))
p1
```

```
##         1
## 0.6923077
```

(f) Our estimate of b_1 is the element in the 2nd row and 1st column of the coefficients from the `summary` call. What is the value of e^{b_1}? [Hint: use `exp`.]
We can exponentiate b_1 to get this value:

```
exp(summary(lrs1)$coef[2, 1])

## [1] 3.927273

## or
exp(1.3679)

## [1] 3.927095
```

(g) Is the e^{b_1} that you just calculated an odds ratio? How do you interpret it?
Since `smoking` is binary, $e^{b_1} = e^{1.37} = 4$ is the odds ratio. Thus we can say that smoking mothers have four times higher odds of having a child with an intellectual disability compared to non-smokers.

(h) What do you think e^{b_1} would have been if we didn't change the levels of `Status`? Re-run the model using `Status` to check your answer. How does it relate to your answer from (f)?

```
lrs2 <- glm(Status ~ smoking, family = binomial(logit))
exp(summary(lrs2)$coef[2,1])

## [1] 0.2546296
```

Switching the event to the non-event inverses the odds ratio:

```
1/exp(summary(lrs1)$coef[2,1])

## [1] 0.2546296
```

So we could have just not changed the levels of `Status` and concluded that non-smoking women have 0.25 times smaller odds of having a baby with an intellectual disability compared to smoking women. But wasn't this more fun?

(i) There is another way to calculate an odds ratio without using logistic regression. Suppose we have the following 2×2 contingency table:

	event	no event
predictor yes	a	b
predictor no	c	d

then the odds ratio is $(a * d)/(b * c)$. Create a contingency table of `smoking` and `newstatus` [Hint: use `table`] and then calculate the odds ratio from that. Do you get the same answer as in (f)?

```
tss <- table(smoking, newstatus)
tss[1,1]*tss[2,2]/(tss[1,2]*tss[2,1])

## [1] 3.927273
```

Same answer!

3. We would like to know if `smoking` and `vitaminB12` can be used to predict `newstatus`. Answer the following questions:

(a) Set up (i.e. write down) the logit model and run it in R. Write the model with the estimates.
The model is

$$\ln\left(\frac{P(\text{intellectual disability})}{1-P(\text{intellectual disability})}\right) = b_0 + b_1 * \text{smoking} + b_2 * \text{vitaminB12}$$

```
lrs3 <- glm(newstatus ~ smoking + vitaminB12,
            family = binomial(logit))
lrs3$coef

##  (Intercept)   smokingyes   vitaminB12
## -1.397551678  1.435938106  0.002087774
```

The model with estimates is

$$\ln\left(\frac{P(\text{intellectual disability})}{1-P(\text{intellectual disability})}\right) = -1.4 + 1.44 * \text{smoking} + 0.002 * \text{vitaminB12}$$

(b) Can either variable significantly predict `newtatus`?

```
summary(lrs3)$coef

##                 Estimate Std. Error   z value     Pr(>|z|)
## (Intercept) -1.397551678 0.55509528 -2.517679 0.0118130962
## smokingyes   1.435938106 0.39134210  3.669266 0.0002432482
## vitaminB12   0.002087774 0.00130122  1.604474 0.1086096691
```

Yes, `smoking` is still significant (p-value < 0.05), but `vitaminB12` is not.

(c) What is the probability of having a baby with an intellectual disability given the mother smokes and has a vitaminB12 level of 400? What is the probability of having a baby with an intellectual disability given the mother smokes and has a vitaminB12 level of 650?

```
mynew <- data.frame(smoking = factor(c("yes", "yes")),
                    vitaminB12 = c(400, 650))
logit_p <- predict(lrs3, newdata = mynew)
p <- exp(logit_p)/(1+exp(logit_p))
p
```

```
##         1         2
## 0.7054726 0.8014592
```

Basic Course on R: Programming Structures 1

Karl Brand* and Elizabeth Ribble†

17-21 December 2018

Contents

*brandk@gmail.com
†emcclel3@msudenver.edu

1 `if()`, `else`, and `ifelse()` Statements

1.1 Conditional Execution Using `if()` and (Optionally) `else`

- Sometimes we'll want R to execute a statement only if a certain condition is met. This can be accomplished via the `if()` and (optionally) `else` statements:

```
if()    # Used to execute a statement only if the given condition
        # is met
else    # Used to specify an alternative statement to be executed
        # if the condition given in if() isn't met
```

- Such ***conditional execution*** commands have the forms:

```
if (condition) {
  statement1
}
```

and

```
if (condition) {
  statement1
  } else {
    statement2
  }
```

- In both cases above, if `condition` is `TRUE`, `statement1` is executed. If `condition` is `FALSE`, then in the first case nothing happens. In the second case, `statement2` is executed.
- Here's a simple example:

```
x <- 5
if (x < 10) {
  y <- 0
}
y

## [1] 0
```

- Here's another:

```
if (x >= 10) {
  y <- 1
  } else {
    y <- 0
}
y

## [1] 0
```

- There's actually an easier way to accomplish the above task:

```
y <- if(x >= 10) 1 else 0
y

## [1] 0
```

- When using such *conditional assignment* statements, in the absence of `else`, `if()` returns `NULL` if `condition` isn't met. So

```
y <- if(condition) 1
```

is equivalent to

```
y <- if(condition) 1 else NULL
```

- In the next example, `return()` is used to terminate a function call and return a value that depends on whether or not a condition is met:

```
mySign <- function(x) {
    if(x < 0) {
      return("Negative")
    } else {
        return("Non-negative")
    }
}
```

We get:

```
mySign(13)

## [1] "Non-negative"
```

1.2 Using `if()` and `else` with a Sequence of Statements

- `if()` and `else` can be used to conditionally execute whole *sequences* of statements, which we enclose in curly brackets { }. The general form of an `if()` command is:

```
if (condition) {
  statement11
  statement12
      .
      .
      .
  statement1q
}
```

which could be followed by `else` and another sequence of statements (in curly brackets) to be executed if `condition` isn't met.

1.3 Vectorized if-else: The `ifelse()` Function

- Sometimes we'll need to create a vector whose values depend on whether or not the values in another vector satisfies some condition. We use:

```
ifelse()    # Returns a vector whose values depend on whether or
            # not a given condition is met by the elements of
            # another vector
```

- `ifelse()` takes argument `test`, the condition to be met, `yes` the return value (or vector of values) when `test` is `TRUE`, and `no`, the return values (or vector of values) when `test` is `FALSE`.

- Here we convert the values in `ht` to "short" or "tall":

```
ht <- c(69, 71, 67, 66, 72, 71, 61, 65, 73, 70, 68, 74)
htCategory <- ifelse(ht > 69, yes = "tall", no = "short")
htCategory
```

```
##  [1] "short" "tall"  "short" "short" "tall"  "tall"  "short" "short"
##  [9] "tall"  "tall"  "short" "tall"
```

2 Looping

- ***Loops*** are used to ***iterate*** (repeat) an R statement (or set of statements). They're implemented in three ways, `for()`, `while()`, and `repeat`:

```
for()       # Repeat a set of statements a specified number of
            # times
while()     # Repeat a set of statements as long as a specified
            # condition is met
repeat      # Repeat a set of statements until a break command is
            # encountered
```

- Two other commands, `break` and `next`, are used, respectively, to terminate a loop's iterations and to skip ahead to the next iteration:

```
break       # Terminate a loop's iterations
next        # Skip ahead to the next iteration
```

- Here's an example in which each of the three loop types, `for()`, `while()`, and `repeat`, are used to perform a simple task, namely printing the numbers $1^2, 2^2, ..., 5^2$ to the screen:

```
for(i in 1:5) {
  print(i^2)
}
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
## [1] 25
```

```
i <- 1
while(i <= 5) {
  print(i^2)
  i <- i + 1
}
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
## [1] 25
```

```
i <- 1
repeat {
  print(i^2)
  i <- i + 1
  if(i > 5) break
}
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
## [1] 25
```

2.1 `for()` Loops

- `for()` loops are used when we know in advance how many iterations the loop should perform.
- The general form of a `for()` loop is:

```
for(i in sequence) {
  statement1
  statement2
      .
      .
      .
  statementq
}
```

where `sequence` is a vector, `i` (whose name you're free to change) assumes the values in `sequence` one after another, each time triggering another iteration of the loop during which `statements 1` through `q` are executed. The `statements` usually involve the variable `i`.

- Here's an example. Suppose we have the data frame describing someone's coin collection:

```
coins <- data.frame(Coin = c("penny", "quarter", "nickel",
                             "quarter", "dime", "penny"),
                    Year = c(1943, 1905, 1889, 1960, 1937, 1900),
                    Mint = c("Den", "SF", "Phil", "Den", "SF",
                             "Den"),
                    Condition = c("good", "fair", "excellent",
                                  "good", "poor", "good"),
                    Value = c(12.00, 55.00, 300.00, 40.00, 18.00,
                              28.00),
                    Price = c(15.00, 45.00, 375.00, 25.00, 20.00,
                              20.00))
coins
```

```
##      Coin Year Mint Condition Value Price
## 1   penny 1943  Den      good    12    15
## 2 quarter 1905   SF      fair    55    45
## 3  nickel 1889 Phil excellent   300   375
## 4 quarter 1960  Den      good    40    25
## 5    dime 1937   SF      poor    18    20
## 6   penny 1900  Den      good    28    20
```

- If we type:

```
colMeans(coins)
```

```
## Error in colMeans(coins):  'x' must be numeric
```

  we get an error message because some of the columns are non-numeric.

- We can compute the means of the numeric columns by looping over the columns, each time checking whether it's numeric before computing it's mean:

```
means <- NULL
for(i in 1:ncol(coins)) {
  if (is.numeric(coins[ , i])) {
    means <- c(means, mean(coins[ , i]))
    }
}
```

The result is:

```
means

## [1] 1922.33333   75.50000   83.33333
```

2.2 `while()` Loops

- `while()` loops are used when we want the loop to iterate until some condition is no longer met.
- The general form of a `while()` loop is:

```
while(condition) {
  statement1
  statement2
      .
      .
      .
  statementq
}
```

  where `condition` is a logical (`TRUE` or `FALSE`) expression involving a variable whose value changes over the course of the loop iterations.

- Prior to each iteration, R checks whether `condition` is `TRUE` or `FALSE`. If it's `TRUE`, the iteration proceeds, otherwise the iterations are terminated.

2.3 `repeat` Loops

- A `repeat` loop iterates a set of statements until a `break` statement is encountered. The general form is of a `repeat` loop is:

```
repeat {
  statement1
  statement2
      .
      .
      .
  statementq
}
```

where at least one of the `statements` should be of the form

```
if(condition) break
```

where `condition` is a logical (`TRUE` or `FALSE`) expression which may be updated during the loop's iterations.

2.3.1 Terminating an "Endless" Loop

- Once in a while we (mistakenly) write a loop that has no way of stopping, for example:

```
i <- 1
while(i <= 5) {
  print("I Cannot Stop by Myself")
}
```

- To terminate the iterations hit the **Escape** key or select **Terminate R...** in RStudio's **Session** pulldown menu.

2.4 Looping Over List Elements

- In the next example, we loop over the elements of a list, printing a list element and recording it's length during each iteration:

```
myList <- list(
      w = c(4, 4, 5, 5, 6, 6),
      x = c("a", "b", "c"),
      y = c(5, 10, 15),
      z = c("r", "s", "t", "u", "v")
)
lengths <- NULL
for(i in myList) {
  print(i)
  lengths <- c(lengths, length(i))
}
```

```
## [1] 4 4 5 5 6 6
## [1] "a" "b" "c"
## [1]  5 10 15
## [1] "r" "s" "t" "u" "v"
```

```
lengths

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

3 The Logical Operations "And", "Or", and "Not"

3.1 Logical Operations and Compound Logical Expressions

- R has ***logical operators*** (or ***Boolean operators***) corresponding to "and" and "or". They're used to combine two logical expressions together to form a single ***compound*** logical expression. Another logical operator corresponding to "not" is used to negate a logical expression. These are written in R as:

```
&&              # "And" for logical scalars
||              # "Or" for logical scalars
!               # "Not" (for logical scalars or vectors)
&               # "And" for logical vectors
|               # "Or" for logical vectors
```

- These operate on logical (`TRUE` or `FALSE`) expressions and return `TRUE` or `FALSE`.
- The distinction between `&&` and `&`, and between `||` and `|` is this:
  - `&&` and `||` operate on logical *scalars* (single `TRUE` or `FALSE` values).
  - `&&` and `||` are the preferred operators to use in `if()` statements.
  - `&` and `|` operate elementwise on logical *vectors*.
  - `&` and `|` are the preferred operators to use in `ifelse()` statements and in square brackets `[ ]` when extracting subsets from vectors or data frames.

3.2 Logical Operations on Scalar Logical Expressions

- `&&` returns `TRUE` if both of the expressions are `TRUE` and it returns `FALSE` otherwise:

```
TRUE && TRUE

## [1] TRUE

TRUE && FALSE

## [1] FALSE
```

- `||` returns `TRUE` if one or both of the expressions are `TRUE` and it returns `FALSE` otherwise:

```
FALSE || TRUE

## [1] TRUE

FALSE || FALSE

## [1] FALSE
```

- As a practical example, if we want to test whether a variable `x` lies *between* two numbers, say 60 and 70, we type:

```
x <- 75
x > 60 && x < 70

## [1] FALSE
```

  and to test whether it lies *outside* the range 60 to 70, we type:

```
x < 60 || x > 70

## [1] TRUE
```

- Here's an example of using `&&` in an `if()` statement:

```
x <- 3
y <- 5
if(x < 10 && y < 10) {
  print("Both less than 10")
} else {
  print("Not both less than 10")
}

## [1] "Both less than 10"
```

- The negation operator, `!`, returns "the opposite" of a logical expression:

```
!TRUE

## [1] FALSE

!FALSE

## [1] TRUE

!(5 < 6)

## [1] FALSE
```

- Pay attention to the operator precedence for `&&`, `||`, and `!`. It can be found by typing:

```
?Syntax
```

  but parentheses can be used to control the order of operations.

- If we try to apply `&&` or `||` to *vectors*, R only applies it to their first elements:

```
c(TRUE, FALSE, TRUE) && c(TRUE, TRUE, FALSE)

## [1] TRUE
```

3.3 Logical Operations on Logical Vectors

- To apply the operations "and" and "or" elementwise on two logical vectors, use `&` and `|`. For example:

```
c(TRUE, FALSE, TRUE) & c(TRUE, TRUE, FALSE)

## [1]  TRUE FALSE FALSE
```

- `&` and `|` are useful in `ifelse()` statements. (Recall that `ifelse()` operates elementwise on vectors.). For example, consider the systolic and diastolic blood pressure readings:

```
systolic <- c(110, 119, 111, 113, 128)
diastolic <- c(70, 74, 88, 74, 83)
```

A blood pressure is classified as normal if the systolic level is less than 120 *and* the diastolic level is less than 80:

```
classification <- ifelse(systolic < 120 & diastolic < 80,
                         yes = "Normal",
                         no = "Abnormal")
classification

## [1] "Normal"   "Normal"   "Abnormal" "Normal"   "Abnormal"
```

- In the next example, we use `&` in square brackets `[ ]` to extract rows from a data frame:

```
bpData <- data.frame(
  name = c("Joe", "Katy", "Bill", "Kim", "Mark"),
  systolic = c(110, 119, 111, 113, 128),
  diastolic = c(70, 74, 88, 74, 83))
bpData

##   name systolic diastolic
## 1  Joe      110        70
## 2 Katy      119        74
## 3 Bill      111        88
## 4  Kim      113        74
## 5 Mark      128        83
```

```
attach(bpData)
bpData[systolic < 120 & diastolic < 80, ]

##   name systolic diastolic
## 1  Joe      110        70
## 2 Katy      119        74
## 4  Kim      113        74

detach(bpData)
```

4 Writing A Function

- To write your own function, you need to use the function `function()`, specify argument(s) that will be used in the function (with our without defaults), and in curly brackets specify what the function should do (referring to the argument(s)), including whether something is returned. The general form is:

```
myFun <- function(arg1, arg2, ...) {
  ## expressions that use the arguments
  ## the last command is what you want the function to return
}
```

  Each line inside the function is an object assignment, a function call, a subsetting, a conditional statement, an if/else statement, a for loop, etc. - basically anything you have now learned how to do in R that you want the function to do! If you do specify arguments, you should use them.

  To have the function output something, you must return the object (either the last command is just the object name or you can use `return()`). If you have multiple objects to return, I recommend returning everything as a list e.g. put this in the last line: `return(list(object1, object2))`.

- Here are a few examples of functions with no default arguments; note the different outputs:

```
do1 <- function(x, y){
  z <- x + y
  x
  z
}
do1(x = 1, y = 3) ## note that x is not returned

## [1] 4

do2 <- function(x, y){
  z <- x + y
  return(x)
  z
}
do2(x = 1, y = 3) ## note that z is returned

## [1] 1
```

```
do3 <- function(x, y){
  z <- x + y
  return(list(x, z))
}
do3(x = 1, y = 3) ## x and z are returned as a single list

## [[1]]
## [1] 1
##
## [[2]]
## [1] 4
```

- Here is an example of a function with default arguments that returns a vector:

```
do4 <- function(x, y = 2){
  z1 <- x + y
  z2 <- x * y
  c(z1, z2)
}
do4(x = 1) ## uses y = 2

## [1] 3 2

do4(x = 1, y = 3) ## overwrites default value of y

## [1] 4 3
```

- Here is an example of a function that takes no arguments:

```
do5 <- function(){
  sum(2, 4, 6)
  print("Hello World!")
  return(mean(1, 3, 5))
}
do5()

## [1] "Hello World!"
## [1] 1
```

- Recall that you can create functions with variable arguments using `...`. For example, here's a function that returns the mean of all the values in an arbitrary number of vectors:

```
meanOfAll <- function(...) {
  return(mean(c(...)))
}
```

The command

```
meanOfAll(usSales, europeSales, otherSales)
```

would combine the three vectors and take the mean of all the data.

5 The `source()` Function

- `source()` is a nice function for reading in big chunks of R code, e.g. a set of functions that you want to use every time you start a new R session.

```
source()       # Read R commands from a text file.
```

- For example, suppose we have the following commands saved in a text file '**C:\myRcode.txt**':

myFun < − function(message) {
print(message)
}

myFun("Hello World")

We can execute those commands using `source()` by running:

```
source("myRcode.txt")
```

```
## [1] "Hello World"
```

6 Document License

GNU General Public License v2.0 or higher (GPL>=v2)

Basic Course on R: Programming Structures 1 Practical

Elizabeth Ribble*

18-24 May 2017

Contents

*emcclel3@msudenver.edu

1 Part A: `if()`, `else`, and `ifelse()` and Vectorization

1. Write a function `evenOrOdd()` involving `if()` and `else` that takes an argument `x` and returns `"Even"` or `"Odd"` depending on whether or not `x` is divisible by 2. (*Do not* use the `ifelse()` function).

2. Is your function `evenOrOdd()` *vectorized*? Check by passing it the vector:

```
w <- c(3, 6, 6, 4, 7, 9, 11, 6)
```

3. Another way to determine if each element of a vector is even or odd is to use the `ifelse()` function, which serves as a vectorized version `if()` and `else`. Use `ifelse()` to obtain `"Even"` or `"Odd"` for each element of `w`.

2 Part B: Loops

1. How many times will `"Frisbee Sailing"` be printed to the screen in each of the following sets of commands? Try to answer without using R.

a)
```
i <- 5
while(i < 1) {
  print("Frisbee Sailing")
  i <- i + 1
}
```

b)
```
i <- 0
while(i < 5) {
  print("Frisbee Sailing")
}
```

c)
```
i <- 0
while(i < 5) {
  print("Frisbee Sailing")
  i <- i + 1
}
```

2. How many times will `"Masked Marvel"` be printed to the screen in the following set of commands? Try to answer without using R.

```
i <- 1
repeat {
  if(i > 5) break
  print("Masked Marvel")
  i <- i + 1
}
```

3. The file **kennedys.txt** has a command to create a list containing two generations of the famous Kennedy family:

```
Kennedys <- list(
    JosephJr = character(0),
    John = c("Caroline", "JohnJr", "Patrick"),
    Rosemary = character(0),
    Kathleen = character(0),
    Eunice = c("RobertIII", "Maria", "Timothy", "Mark", "Anthony"),
    Patricia = c("Christopher", "Sydney", "Victoria", "Robin"),
    Robert = c("Kathleen", "JosephII", "RobertJr", "David",
               "MaryC", "Michael", "MaryK", "Christopher",
               "Matthew", "Douglas", "Rory"),
    Jean = c("Stephen", "William", "Amanda", "Kym"),
    Edward = c("Kara", "EdwardJr", "Patrick")
)
```

Read in the file with the use of `source()` and type `ls()` to see if the list was created (type `Kennedys` to view the object).

Loop over the list of the first generation of Kennedys, keeping track of how many children each one has in a vector.

3 Part C: Logical Operators &, |, and !

1. What will be the result of the following:

```
(10 < 20 && 15 < 16) || 9 == 10
```

2. One of the following evaluates to `TRUE`, the other to `FALSE`. Which is which?

```
4 < 3 && (5 < 6 || 8 < 9)
(4 < 3 && 5 < 6) || 8 < 9
```

3. The data set below contains the systolic and diastolic blood pressure readings for 22 patients (and can be found in the file **BPressure.txt**).

PatientID	Systolic	Diastolic
CK	120	50
SS	96	60
FR	100	70
CP	120	75
BL	140	90
ES	120	70
CP	165	110
JI	110	40
MC	119	66
FC	125	76
RW	133	60
KD	108	54
DS	110	50
JW	130	80
BH	120	65
JW	134	80
SB	118	76
NS	122	78
GS	122	70
AB	122	78
EC	112	62
HH	122	82

a) Read the data from **BPressure.txt** into a data frame called `bp` using `read.table()`.

b) A person's blood pressure is classified as normal if the systolic level is below 120 and the diastolic level is below 80.Use square brackets `[ ]` to extract from `bp` the rows corresponding to patients with normal blood pressures.

c) Now use square brackets `[ ]` to extract the rows corresponding to patients whose blood pressures *aren't* normal.

Basic Course on R: Programming Structures 1 Practical Answers

Elizabeth Ribble*

18-24 May 2017

Contents

*emcclel3@msudenver.edu

1 Part A: `if()`, `else`, and `ifelse()` and Vectorization

1. Write a function `evenOrOdd()` involving `if()` and `else` that takes an argument `x` and returns `"Even"` or `"Odd"` depending on whether or not `x` is divisible by 2. (*Do not* use the `ifelse()` function).

```
evenOrOdd <- function(x) {
   if(x %% 2 ==0) {
       return("Even")
   } else {
       return("Odd")
   }
}
```

2. Is your function `evenOrOdd()` *vectorized*? Check by passing it the vector:

```
w <- c(3, 6, 6, 4, 7, 9, 11, 6)
```

```
w <- c(3, 6, 6, 4, 7, 9, 11, 6)
evenOrOdd(w)

## Warning in if (x%%2 == 0) {: the condition has length > 1 and only
the first element will be used

## [1] "Odd"
```

The function is not vectorized because it only runs on the first element!

3. Another way to determine if each element of a vector is even or odd is to use the `ifelse()` function, which serves as a vectorized version `if()` and `else`. Use `ifelse()` to obtain `"Even"` or `"Odd"` for each element of `w`.

```
w <- c(3, 6, 6, 4, 7, 9, 11, 6)
ifelse(w %% 2 ==0, "Even", "Odd")

## [1] "Odd"  "Even" "Even" "Even" "Odd"  "Odd"  "Odd"  "Even"
```

2 Part B: Loops

1. How many times will `"Frisbee Sailing"` be printed to the screen in each of the following sets of commands? Try to answer without using R.

a)
```
i <- 5
while(i < 1) {
  print("Frisbee Sailing")
  i <- i + 1
}
```

It will not print because the original i is not less than 1.

b)
```
## not run
i <- 0
while(i < 5) {
  print("Frisbee Sailing")
}
```

It will print indefinitely - until you hit escape on the keyboard or click stop in RStudio because i is always less than 5.

c)
```
i <- 0
while(i < 5) {
  print("Frisbee Sailing")
  i <- i + 1
}
```

```
## [1] "Frisbee Sailing"
## [1] "Frisbee Sailing"
## [1] "Frisbee Sailing"
## [1] "Frisbee Sailing"
## [1] "Frisbee Sailing"
```

The phrase will print five times; it stops once the i+1 value reaches 5.

2. How many times will `"Masked Marvel"` be printed to the screen in the following set of commands? Try to answer without using R.

```
i <- 1
repeat {
  if(i > 5) break
```

```
  print("Masked Marvel")
  i <- i + 1
}

## [1] "Masked Marvel"
## [1] "Masked Marvel"
## [1] "Masked Marvel"
## [1] "Masked Marvel"
## [1] "Masked Marvel"
```

The phrase will print five times; it stops once the i+1 value reaches 5.

3. The file **kennedys.txt** has a command to create a list containing two generations of the famous Kennedy family:

```
Kennedys <- list(
    JosephJr = character(0),
    John = c("Caroline", "JohnJr", "Patrick"),
    Rosemary = character(0),
    Kathleen = character(0),
    Eunice = c("RobertIII", "Maria", "Timothy", "Mark", "Anthony"),
    Patricia = c("Christopher", "Sydney", "Victoria", "Robin"),
    Robert = c("Kathleen", "JosephII", "RobertJr", "David",
               "MaryC", "Michael", "MaryK", "Christopher",
               "Matthew", "Douglas", "Rory"),
    Jean = c("Stephen", "William", "Amanda", "Kym"),
    Edward = c("Kara", "EdwardJr", "Patrick")
)
```

Read in the file with the use of `source()` and type `ls()` to see if the list was created (type `Kennedys` to view the object).

```
source("kennedys.txt")
```

Loop over the list of the first generation of Kennedys, keeping track of how many children each one has in a vector.

```
children <- NULL
for(i in Kennedys){
  children <- c(children, length(i))
```

```
}
children

## [1]  0  3  0  0  5  4 11  4  3
```

3 Part C: Logical Operators &, |, and !

1. What will be the result of the following:

```
(10 < 20 && 15 < 16) || 9 == 10

## [1] TRUE
```

TRUE because the first statement (in parentheses) is TRUE and the second is FALSE.

2. One of the following evaluates to TRUE, the other to FALSE. Which is which?

```
4 < 3 && (5 < 6 || 8 < 9)

## [1] FALSE

(4 < 3 && 5 < 6) || 8 < 9

## [1] TRUE
```

The first one FALSE because the first statment before is FALSE. The second one is TRUE because one of the two statements to the left and right of || is TRUE.

3. The data set below contains the systolic and diastolic blood pressure readings for 22 patients (and can be found in the file **BPressure.txt**).

   a) Read the data from **BPressure.txt** into a data frame called `bp` using `read.table()`.

```
bp <- read.table("BPressure.txt", header=TRUE)
```

b) A person's blood pressure is classified as normal if the systolic level is below 120 and the diastolic level is below 80.Use square brackets `[ ]` to extract from `bp` the rows corresponding to patients with normal blood pressures.

```
bp[(bp$Systolic < 120 & bp$Diastolic < 80), ]
```

```
##    PatientID Systolic Diastolic
## 2         SS       96        60
## 3         FR      100        70
## 8         JI      110        40
## 9         MC      119        66
## 12        KD      108        54
## 13        DS      110        50
## 17        SB      118        76
## 21        EC      112        62
```

c) Now use square brackets `[ ]` to extract the rows corresponding to patients whose blood pressures *aren't* normal.

```
bp[!(bp$Systolic < 120 & bp$Diastolic < 80), ]
```

```
##    PatientID Systolic Diastolic
## 1         CK      120        50
## 4         CP      120        75
## 5         BL      140        90
## 6         ES      120        70
## 7         CP      165       110
## 10        FC      125        76
## 11        RW      133        60
## 14        JW      130        80
## 15        BH      120        65
## 16        JW      134        80
## 18        NS      122        78
## 19        GS      122        70
## 20        AB      122        78
## 22        HH      122        82
```

Basic Course on R: Programming Structures 2

Elizabeth Ribble*

17-21 December 2018

Contents

*emcclel3@msudenver.edu

1 Environment and Scope Issues

- Each function, whether built-in or user-defined, has an associated ***environment***, which can be thought of as a container that holds all of the objects present at the time the function is created.

1.1 The Top-Level (or Global) Environment

- When a function is created on the command line, it's environment is the so-called ***Global Environment*** (or ***Workspace***):

```
w <- 2
```

```
f <- function(y) {
  d <- 3
  return(d * (w + y))
}
```

```
environment(f)

## <environment: R_GlobalEnv>
```

- The function `objects()` (or `ls()`), when called from the command line, lists the objects in the Global Environment:

```
objects()

## [1] "f" "w"
```

1.2 Global and Local Variables

- In the function `f()` defined above, the variable `w` is said to be ***global*** to `f()` and the variable `d`, because it's created *within* `f()`, is said to be ***local***.

- Global variables (like `w`) are visible from within a function, but local variables (like `d`) aren't visible from outside the function. In fact, local variables are *temporary*, and disappear when the function call is completed:

```
f(y = 1)

## [1] 9

d

## Error in eval(expr, envir, enclos):  object 'd' not found
```

- When a global and local variable share the same name, the local variable is used:

```
w <- 2
d <- 4
f <- function(y) {
  d <- 3
  return(d * (w + y))
}
f(y = 1)

## [1] 9
```

- Note also that when an assignment takes place within a function, and the local variable shares its name with an existing global variable, only the local variable is affected:

```
w <- 2
d <- 4                  # This value of d will remain unchanged.
f <- function(y) {
  d <- 3                # This doesn't affect the value of d in the
  return(d * (w + y))   # global environment (Workspace)
}
f(y = 1)

## [1] 9

d

## [1] 4
```

1.3 Nested Functions and the Scope Hierarchy

- For user-defined functions created on the command line, the global variables for that function are those in the Workspace, or ***global environment***. They're listed by typing `ls()` (or `objects()`) on the command line.
- When a function is *created inside another function*, its global variables are the local variables of the outer function *plus* the outer function's global variables.
- Regardless of whether a function is created on the command line or inside another function, its local variables are the variables created inside of it *plus* its formal arguments to which values have been passed.
- For example:

```
w <- 2            # w is global to f() and therefore also to h()
f <- function(y) {
  d <- 3
  h <- function() {
    b <- 5              # b is local to h()
    return(d * (w + y))
    }
  return(h())
}
```

  Above,

  - `w` is global to `f()` and therefore also to `h()`.
  - `y` and `d` are local to `f()`, but global to `h()`.
  - `b` is local to `h()`.

- This ***scope hierarchy*** continues when multiple function definitions are ***nested*** inside of each other.

- We can use a `print(ls())` statement to see which objects are local to `f()`:

```
w <- 2                  # w is global to f() and therefore also to h()
f <- function(y) {
  d <- 3                # y and d are local to f() but global to h()
  h <- function() {
    b <- 5              # b is local to h()
    return(d * (w + y))
    }
  print(ls())
  return(h())
}
f(y = 2)

## [1] "d" "h" "y"
## [1] 12
```

- Likewise we can use a `print(environment(h))` statement to view the environment of `h()`:

```
w <- 2                  # w is global to f() and therefore also to h()
f <- function(y) {
  d <- 3                # y and d are local to f() but global to h()
  h <- function() {
    b <- 5              # b is local to h()
    return(d * (w + y))
    }
  print(environment(h))
  return(h())
}
f(y = 2)

## <environment: 0x000000001979c468>
## [1] 12
```

In the output above, the environment of `h()` is referred to by its memory location. The *environment* of `h()` is the "container" that contains `h()` as well as the objects `d` and `y`.

1.4 Writing "Upstairs" in the Scope Hierarchy

- Sometimes we need to assign a value to a variable in the global environment from within a function. We can do so using either of the following:

```
<<-         # Assign a value to a variable in the global environment
            # (Workspace).
assign()    # Assign a value to a variable in the global environment
            # (Workspace).
```

- Here's an example using the so-called ***superassignment operator*** `<<-`:

```
w <- 2
d <- 4                  # This value of d will be replaced by 3.
f <- function(y) {
  d <<- 3               # This replaces the value 4 of d in the global
  return(d * (w + y)) # environment (Workspace) by 3.
}
f(y = 2)

## [1] 12

d

## [1] 3
```

Above, the assignment of `3` to `d` is done in the global environment, or Workspace, overwriting the previous value (`4`).

- Here's how to accomplish the same thing using `assign()`:

```
w <- 2
d <- 4                  # This value of d will be replaced by 3.
f <- function(y) {
  assign("d", 3, envir = .GlobalEnv)
  return(d * (w + y))
}
f(y = 2)

## [1] 12

d

## [1] 3
```

(Note that when `assign()` is used, `d` is written as a character string (i.e. in quotes as `"d"`) and the global environment is written as `.GlobalEnv`.)

- Be aware that assignment to the variable `d` using `<<-` actually results in a search up the environment hierarchy, stopping at the first level at which the name `d` is encountered. If it's not encountered, then assignment is done in the global environment. For example:

```
w <- 2
d <- 4                      # This value of d will remain unchanged.
f <- function(y) {
    d <- 5                  # This value of d will be replaced by 3.
    h <- function() {
        d <<- 3             # This replaces the value 5 of d in the
        return(d * (w + y)) # h() and f() environments by 3.
    }
    return(h())
}
f(y = 2)

## [1] 12

d

## [1] 4
```

1.5 When Should You Use Global Variables?

- Here are suggestions about using global variables. They're especially important when your code will be shared with other R users:
  - Assignment to the global environment using `<<-` or `assign()` should be used very sparingly (i.e. only when necessary) because it can accidentally overwrite existing variables.
  - The use of a global variable can be justified when that variable needs to be accessed by several different functions (that aren't nested).
  - It's generally preferable to pass variables as arguments to functions rather than accessing them from the global environment.

2 Printing a Warning Message or Terminating a Function Call Using `warning()`, `return()`, or `stop()`

- The following functions are useful for terminating a function call or just printing a warning message:

```
return()      # Terminate a function call and return a value.
stop()        # Terminate a function call and print an error message.
warning()     # Print a warning message (without terminating the
              # function call).
```

2.1 Terminating a Function Call Using `if()` and `return()`

- One way to terminate a function call is with `return()` which, when encountered, immediately terminates the call and returns a value. For example:

```
mySign <- function(x) {
  if(x < 0) return("Negative")
  if(x > 0) return("Positive")
  return("Zero")
}
```

Passing `my.sign()` the value `x = 13` produces the following:

```
mySign(x = 13)

## [1] "Positive"
```

(Note that the last line, `return("Zero")`, was never encountered during the call to `my.sign()`.)

2.2 Terminating a Function Call and Printing an Error Message Using `if()` and `stop()`

- Another way to terminate a function call is with `stop()`, which then prints an error message without returning a value. Here's an example:

```
myRatio <- function(x, y) {
  if(y == 0) stop("Cannot divide by 0")
  return(x/y)
}
```

An attempt to pass the value `0` for `y` now results in the following:

```
myRatio(x = 3, y = 0)

## Error in myRatio(x = 3, y = 0):  Cannot divide by 0
```

(Note that the last line, `return(x/y)`, was never encountered during the call to `myRatio()`.)

2.3 Printing a Warning Message Using `if()` and `warning()`

- `warning()` just prints a warning message to the screen without terminating the function call. Here's an example:

```
myRatio <- function(x, y) {
  if(y == 0) warning("Attempt made to divide by 0")
  return(x/y)
}
```

Now when we pass the value `0` for `y` the function call isn't terminated (the value `Inf` is returned), but we get the warning message:

```
myRatio(x = 3, y = 0)

## Warning in myRatio(x = 3, y = 0):  Attempt made to divide by 0

## [1] Inf
```

3 Recursion

- ***Recursion*** is a programming technique in which a function calls itself.
- Here's an example in which the function `f()` takes a non-negative integer `x` and returns the factorial of `x`, denoted $x!$ and defined as

$$x! = \begin{cases} 1 & \text{if } x = 0 \\ x(x-1)(x-2)\cdots(2)(1) & \text{if } x > 0 \end{cases}$$

- Notice that we can write

$$x! = \begin{cases} 1 & \text{if } x = 0 \\ x(x-1)! & \text{if } x > 0 \end{cases}$$

```
f <- function(x) {
  if(x == 0) {
    return(1)
    } else {
      return(x * f(x - 1))
      }
}
f(0)

## [1] 1

f(1)

## [1] 1

f(5)

## [1] 120
```

- In general, to solve a problem of type X by writing a recursive function `f()`:

  1. Break the original problem of type X into one or more smaller problems of type X.
  2. Within `f()`, call `f()` on each of the smaller problems.
  3. Within `f()`, piece together the results of Step 2 to solve the original problem.

4 Replacement Functions

- Some of R's built-in functions can be used both to *return* a value and to *replace* a value. For example using the data frame:

```
var1 <- c(1, 2, 3)
var2 <- c(19, 20, 16)
var3 <- c("small", "medium", "large")
x <- data.frame(var1, var2, var3)
x

##   var1 var2   var3
## 1    1   19  small
## 2    2   20 medium
## 3    3   16  large
```

`names()` will both return the names of the variables in `x`:

```
names(x)

## [1] "var1" "var2" "var3"
```

and replace them:

```
names(x) <- c("IDNumber", "Weight", "Size")
names(x)

## [1] "IDNumber" "Weight"   "Size"
```

- Such functions are called ***replacement functions***.
- It's possible to create user-defined replacement functions.

Basic Course on R: Programming Structures 2 Practical

Elizabeth Ribble*

14-18 May 2018

Contents

*emcclel3@msudenver.edu

1 Part A: Scope

1. For each of the following sets of commands, give the value that will be returned by the last command. Try to answer without using R.

   a)
   ```
   w <- 5
   f <- function(y) {
     return(w + y)
   }
   f(y = 2)
   ```

   b)
   ```
   w <- 5
   f <- function(y) {
     w <- 4
     return(w + y)
   }
   f(y = 2)
   ```

2. Among the variables `w`, `d`, and `y`, which are global to `f()` and which are local?

   ```
   w <- 2
   f <- function(y) {
     h <- function() {
       d <- 3
       return(w + y)
       }
     return(d * h())
     }
   ```

3. Do the following in R.

   a) Try:

   ```
   myFun1 <- function() {
     a <- 2
     b <- 3
     myFun2(3)
   }
   myFun2 <- function(y) {
     return(y + a + b)
   ```

```
}
myFun1()
```

What happens?

b) Now try:

```
a <- 1
b <- 2
myFun1()
```

What happens?

4. What value for `w` will be printed in the last line below? Try to answer without using R.

```
w <- 1
f <- function(y) {
  g <- function() {
    w <<- 3
    return(2)
  }
  return(g())
}
f(y = 1)
w
```

5. What value for `w` will be printed in the last line below? Try to answer without using R.

```
w <- 1
f <- function(y) {
  w <- 2
  g <- function() {
    w <<- 3
    return(2)
    }
  return(g())
}
f(y = 1)
w
```

2 Part B: `if()` Statements, `warning()`, and `stop()`

The functions `warning()` and `stop()` are used to print a warning message and to stop the execution of the function call and print an error message. For example:

```
noNegMean <- function(x) {
  if(all(x < 0)) {
    stop("All values in x are negative")
    }
  if(any(x < 0)) {
    x[x < 0] <- 0
    warning("Negative values in x replaced by zero")
    }
  return(mean(x))
  }
```

1. The file **nonegmean.txt** contains the above code; source it into R and then pass `noNegMean()` a vector containing some negative and some positive values. What happens?

2. What happens when you pass `noNegMean()` a vector containing all negative values?

3. Write a function `ratio()` that takes two arguments, `x` and `y`, and attempts to compute the ratio `x/y`. If both `x == 0 & y == 0`, the function should stop and print an error message about dividing 0 by 0. If `y == 0` (but not `x`), the function should print a warning message about dividing by 0, and then return `x/y` (which will be `Inf`). In all other cases, it should return `x/y`.

   Test your `ratio()` function first using two nonzero values for `x` and `y`, then using a nonzero `x` but `y = 0`, and finally using `x = 0` and `y = 0`.

Basic Course on R: Programming Structures 2 Practical Answers

Elizabeth Ribble*

14-18 May 2018

Contents

*emcclel3@msudenver.edu

1 Part A: Scope

1. For each of the following sets of commands, give the value that will be returned by the last command. Try to answer without using R.

a)
```
w <- 5
f <- function(y) {
  return(w + y)
}
f(y = 2)

## [1] 7
```

This will return 7 because `w` is 5 and we are evaluating the function at `y = 2`.

b)
```
w <- 5
f <- function(y) {
  w <- 4
  return(w + y)
}
f(y = 2)

## [1] 6
```

This will return 6 because `w` is reassigned as 4 inside the function and we are evaluating the function at `y = 2`.

2. Among the variables `w`, `d`, and `y`, which are global to `f()` and which are local?

```
w <- 2
f <- function(y) {
  h <- function() {
    d <- 3
    return(w + y)
    }
  return(d * h())
  }
```

The object `w` is global to `f()` while `d` and `y` are local to `f()`.

3. Do the following in R.

   a) Try:

```
myFun1 <- function() {
  a <- 2
  b <- 3
  myFun2(3)
}
myFun2 <- function(y) {
  return(y + a + b)
}
myFun1()

## Error in myFun2(3):  object 'a' not found
```

   What happens?

   We get an error message because a and b are local to `myFun1` so the function `myFun2` can't find them in the global environment.

   b) Now try:

```
a <- 1
b <- 2
myFun1()

## [1] 6
```

   What happens?

   We get get the value 6 because the values `a` and `b` are global so `myFun2` can find them and use them in its commands.

4. What value for `w` will be printed in the last line below? Try to answer without using R.

```
w <- 1
f <- function(y) {
  g <- function() {
    w <<- 3
    return(2)
  }
  return(g())
}
f(y = 1)
```

```
## [1] 2

w

## [1] 3
```

We get the value 3 because the superassign operator overwrote the original assignment of `w`.

5. What value for `w` will be printed in the last line below? Try to answer without using R.

```
w <- 1
f <- function(y) {
  w <- 2
  g <- function() {
    w <<- 3
    return(2)
    }
  return(g())
}
f(y = 1)

## [1] 2

w

## [1] 1
```

We get the value 1 because the superassign operator only overwrote the assignment of `w` within the `f()` function.

2 Part B: `if()` Statements, `warning()`, and `stop()`

The functions `warning()` and `stop()` are used to print a warning message and to stop the execution of the function call and print an error message. For example:

```
noNegMean <- function(x) {
  if(all(x < 0)) {
    stop("All values in x are negative")
```

```
  }
if(any(x < 0)) {
  x[x < 0] <- 0
  warning("Negative values in x replaced by zero")
  }
return(mean(x))
}
```

1. The file **nonegmean.txt** contains the above code; source it into R and then pass `noNegMean()` a vector containing some negative and some positive values. What happens?

```
source("nonegmean.txt")
noNegMean(c(-1,0,1))

## Warning in noNegMean(c(-1, 0, 1)):  Negative values in x replaced
by zero

## [1] 0.3333333
```

   We get the warning message and it returned 0.3333, which is the average of 0, 0, 1.

2. What happens when you pass `noNegMean()` a vector containing all negative values?

```
source("nonegmean.txt")
#noNegMean(c(-1,-1,-1)) # not run; error message
```

   We get the error message and nothing is returned.

3. Write a function `ratio()` that takes two arguments, `x` and `y`, and attempts to compute the ratio `x/y`. If both `x == 0 & y == 0`, the function should stop and print an error message about dividing 0 by 0. If `y == 0` (but not `x`), the function should print a warning message about dividing by 0, and then return `x/y` (which will be `Inf`). In all other cases, it should return `x/y`.

   Test your `ratio()` function first using two nonzero values for `x` and `y`, then using a nonzero `x` but `y = 0`, and finally using `x = 0` and `y = 0`.

```r
ratio <- function(x,y) {
  if(x == 0 & y == 0) {
    stop("Cannot divide zero by zero.")
  }
  if(y == 0) {
    warning("Cannot divide by zero.")
  }
  ratio <- x/y
  return(ratio)
}

ratio(2,3)

## [1] 0.6666667

ratio(0,0)

## Error in ratio(0, 0):  Cannot divide zero by zero.

ratio(1,0)

## Warning in ratio(1, 0):  Cannot divide by zero.

## [1] Inf
```

Basic Course on R: Object-Oriented Programming

Elizabeth Ribble*

14-18 May 2018

Contents

*emcclel3@msudenver.edu

1 Object Oriented Programming

1.1 Objects and Classes

1.1.1 Objects

- In R, every "thing" is an ***object***. The vectors `x` and `logicVec` and the matrix `X` are examples of objects:

```
x

##  [1]  2  4  2  7  9 11  9  4  2  6  7  9  9  4  3
```

```
logicVec

## [1]  TRUE FALSE  TRUE
```

```
X

##      [,1] [,2] [,3]
## [1,]    2   11    7
## [2,]    4    9    9
## [3,]    2    4    9
## [4,]    7    2    4
## [5,]    9    6    3
```

- Objects can be passed as arguments to functions which perform operations on them and then return other objects.

1.1.2 Classes of Objects

- Each object belongs to a certain ***class*** of objects. We can determine what class an object belongs to using `class()` or `is.cname()`:

```
class()      # Determines the class of an object.  Can also be
             # used to assign a class to an object.
is.cname()   # Returns TRUE if an object belongs to the class
             # "cname" (e.g. is.numeric(), is.data.frame(), etc.)
             # and FALSE otherwise.
```

- For example, the object `x` belongs to the *numeric vector* class, `logic.vec` belongs to the *logical vector* class, and `X` belongs to the *matrix* class:

```
class(x)

## [1] "numeric"

class(logicVec)

## [1] "logical"

class(X)

## [1] "matrix"
```

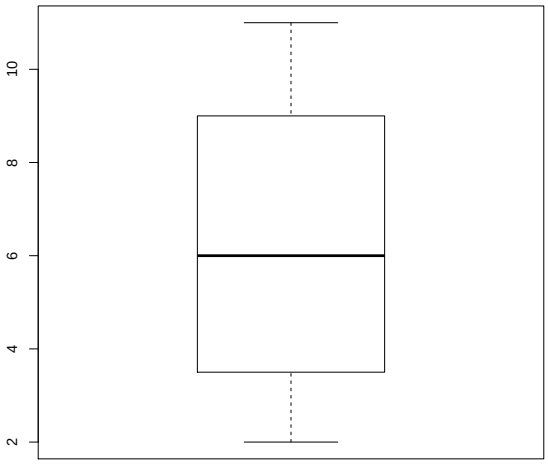
- There are two types of classes in R, **S3** classes and **S4** classes. Nearly all of R's built-in classes are the S3 type, including numeric, character, and logical vectors, matrices and arrays, lists, and data frames.

1.2 Generic Functions and Methods

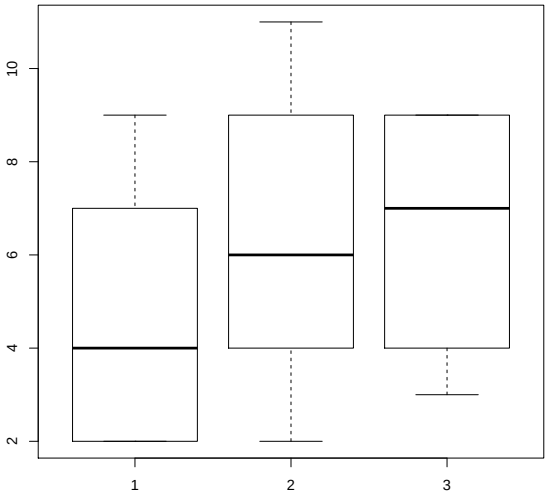
1.2.1 Generic Functions

- Some R functions only accept arguments from a single class. Others accept objects from more than one class.
- Functions that accept objects from more than one class are called ***generic functions***.
- For example, `boxplot()` is a generic function because it can be passed a numeric vector *or* a matrix (among other classes of objects). Below it's first passed a vector and then a matrix:

```
boxplot(x)                # x is a vector.
```




```
boxplot(X)                  # X is a matrix with 3 columns.
```



Notice that `boxplot()` does different things depending on what class of object is passed to it – when passed a vector it produces a single boxplot, but when passed a matrix, it produces side-by-side boxplots of the matrix's columns.

1.2.2 Methods

- There are actually different "versions" of the `boxplot()` function that are referred to as its ***methods***.
- Each `boxplot()` *method* accepts a specific class of arguments. One method accepts vectors and another accepts matrices.
- When we pass an object to a generic function, the function determines the class of the object and then "dispatches" it to the appropriate method.
- If we look at a generic function's definition (by typing its name on the command line), we see that it calls another function, `UseMethod()`, which dispatches the object to the appropriate method. For example:

```
boxplot

## function (x, ...)
## UseMethod("boxplot")
## <bytecode: 0x00000000086d3a90>
## <environment: namespace:graphics>
```

- We can view the methods associated with a given generic function using:

```
methods()      # Determine the S3 methods that are associated
               # with a given generic function
showMethods()  # Determine the S4 methods that are associated
               # with a given generic function
```

- For example, to see the methods for `boxplot()` type:

```
methods(boxplot)

## [1] boxplot.default  boxplot.formula* boxplot.matrix
## see '?methods' for accessing help and source code
```

- The method `boxplot.default()` is the "version" of `boxplot()` that accepts vectors. The `boxplot.matrix()` method is the one that accepts matrices. So typing

```
boxplot(x)
```

is equivalent to typing

```
boxplot.default(x)
```

and typing

```
boxplot(X)
```

is equivalent to typing

```
boxplot.matrix(X)
```

(Actually, in addition to vectors, `boxplot.default()` also accepts several other classes of objects.)

- In general, methods have names of the form `fname.cname()`, where `fname()` is name of the generic function and `cname` is the class of objects that the method accepts.
- "Non-visible" methods (such as `boxplot.formula`) are ones that are "not found" when you type their name on the command line, but R can find them when it needs them. (If you need to view a "Non-visible" method's definition you can do so using `getAnywhere()`, for example `getAnywhere(boxplot.formula)`.)
- For the other ("visible") methods, we can view their definitions by typing their name on the command line, e.g.

```
boxplot.matrix;

## function (x, use.cols = TRUE, ...)
## {
##     groups <- if (use.cols) {
##         split(c(x), rep.int(1L:ncol(x), rep.int(nrow(x), ncol(x))))
##     }
##     else split(c(x), seq(nrow(x)))
##     if (length(nam <- dimnames(x)[[1 + use.cols]]))
##         names(groups) <- nam
##     invisible(boxplot(groups, ...))
## }
## <bytecode: 0x000000000af91688>
## <environment: namespace:graphics>
```

2 Performance Enhancement: Speed and Memory

2.1 Writing Faster R Code

- Computationally intensive tasks, e.g. those involving very large data sets, can require excessive amounts of computing time and/or memory usage.
- Some ways of speeding up your R code are:
  - Optimizing your code by using *vectorization* (and avoiding loops), *bytecode compilation*, and other techniques.
  - Writing the computationally intensive chunks code in C or C^{++} and then calling them from R.
  - Using parallel R computing.
- The amount of time required to run your code will depend on:
  - The computer it's being run on.
  - The number and types of other processes (applications) that are running while your R code is being executed, such as web browsers, word processors, music players, etc.
- To test the speed of a chunk of R code, we use:

```
system.time()      # Returns the computation time required to
                   # execute a chunk of R code (in seconds)
```

- `system.time()` takes as its argument one or more R commands (enclosed by curly brackets if there are more than one), and returns time spent executing them.
- Here's an example in which we time how long it takes to add two vectors together using a loop:

```
x <- runif(100000)
y <- runif(100000)
z <- NULL
system.time(
    for(i in 1:100000) {
       z[i] <- x[i] + y[i]
    }
)


##    user  system elapsed
##    0.03    0.00    0.03
```

- The return value of `system.time()` contains three components, all reported in seconds:
  - ***User time***: The CPU time spent by the *current process* (i.e. the current R session).[1]
  - ***System time***: The CPU time spent by the *operating system* on behalf of the current process (R session) carrying out tasks that must also be carried out on behalf of other processes (applications), e.g. input/output tasks such as accepting keyboard input, printing to the screen, opening files for reading or writing, etc.[2]
  - ***Elapsed time***. This is the time you would get using a stopwatch, and includes time spent on processes (applications) unrelated to the R session (e.g. open web browsers, music players, word processors, etc.).

[1]The ***CPU*** (central processing unit) is the computer's hardware that carries out instructions of processes (applications) and the operating system, such as performing arithmetic and logical operations and input/output tasks such as receiving messages from the keyboard, printing to the screen, and writing to a file.

[2]The ***operating system*** is the computer's software that manages processes (applications), making sure they don't interfere with each other, and performs common services for those applications such as directing input/output, e.g. from the keyboard to the screen, to or from a file, or a to printer.

- Usually we don't care too much whether the CPU is being used by R or by the operating system (on behalf of R). We just want to know how much total CPU time was used. **The sum of the user and system times gives the total CPU time spent executing the R code**.

2.2 Removing Unnecessary Computations from Loops

- Here are two examples, both of which add $\sqrt{2}$ to each of 100,000 numbers. The first is *inefficient*. The second improves upon the first by removing the computation of $\sqrt{2}$ from the loop because it doesn't belong there:

  1. In this code we unnecessarily compute $\sqrt{2}$ during each of 100,000 iterations of the loop:

```
x <- runif(100000)
system.time(
    for(i in 1:100000) {
      y <- sqrt(2)    # This can be moved outside the loop
      x[i] <- x[i] + y
    }
)
```

```
##    user  system elapsed
##    0.01    0.00    0.01
```

2. The code can be made more efficient by moving the command `y <- sqrt(2)` outside the loop:

```
system.time({
    y <- sqrt(2)            # This was pulled from the loop
    for(i in 1:100000) {
        x[i] <- x[i] + y
    }
})
```

```
##    user  system elapsed
##    0.02    0.00    0.01
```

- It turns out that function calls (like `sqrt(2)` above) are time consuming in R, in part because they involve setting up environments, sometimes several nested inside each other, to store local variables. The second loop above only calls `sqrt()` once (as opposed to 100,000 times), so it's faster. (The task can be performed even faster using vectorization).

2.3 Vector Preallocation

- ***Vector preallocation*** refers to creating a vector *prior* to executing a loop, and then replacing its elements during the loop's iterations. Preallocation can speed up R code.

- Here are three examples in which we use a loop to add two vectors `x` and `y` together, storing the result in another vector `z`. The first two don't use preallocation, the third does.

1. Here's an *inefficient* set of code that uses `c()` to *recreate* `z` each iteration of the loop:

```
x <- runif(100000)
y <- runif(100000)
z <- NULL                   # do not preallocate the elements of z
system.time(
    for (i in 1:100000) {
        z <- c(z, x[i] + y[i])  # R has to recreate z entirely
                                # each iteration
    }
```

```
)

##    user  system elapsed
##   14.76    0.29   15.07
```

2. An alternative approach is to *redimension* `z` by increasing its length one element each iteration, but this too is *inefficient* because it turns out that redimensioning a vector takes as much time as recreating it altogether:

```
z <- NULL                    # do not preallocate the elements of z
system.time(
    for (i in 1:100000) {
        z[i] <- x[i] + y[i] # R has to redimension z (change
    }                        # its length) each iteration
)

##    user  system elapsed
##    0.03    0.00    0.04
```

3. Now watch how much faster the code is when we *preallocate* space for the elements of `z` before entering the loop:

```
z <- rep(NA, 100000)       # Now preallocate 100,000 elements of z,
                           # assigning each the value NA
system.time(
    for (i in 1:100000) {
        z[i] <- x[i] + y[i] # R only has to assign to a single
    }                        # element of z, with no redimensioning
)

##    user  system elapsed
##       0       0       0
```

We see that preallocation speeds the code up considerably.

2.4 Using Vectorization and Avoiding Loops

- Recall that many of R's built-in functions (`sqrt()`, `abs()`, etc.) and operators (‘`+`’, ‘`-`’, ‘`*`’, ‘`/`’, ‘`^`’, etc.) are *vectorized*.
- It's usually *much* faster to use *vectorization* than to perform the same task using a loop.

- For example, below we time two sets of code that perform the same task, adding the elements of two vectors `x` and `y` together. The first uses a loop and the second the vectorized property of the '+' operator:

```
x <- runif(100000)
y <- runif(100000)
z <- rep(NA, 100000)
```

1. Here we use a loop, which is *inefficient*:

```
system.time(
    for(i in 1:length(x)) {
        z[i] <- x[i] + y[i]
    }
)

##    user  system elapsed
##       0       0       0
```

2. Now we use vectorization to speed up the code:

```
system.time(
    z <- x + y
)

##    user  system elapsed
##       0       0       0
```

   Note that the vectorized code was *much* faster than the loop.

- The reason why vectorization is (usually) faster than looping is that vectorized computations are carried out behind the scenes in the C language, which is much faster than R. (Recall that underlying R is a suite of C functions that R invokes when it needs them).

  More precisely, vectorization usually involves fewer R function calls, which as noted can be slow. For example, although it may not look like it, in the loop above, the '+' operation actually involves making 100,000 calls to a function `"+"()`:

```
"+"(2, 3)

## [1] 5
```

Using vectorization, the `"+"()` function is only called once, at which point all further computational tasks are passed to the C language.

2.5 Bytecode Compilation

- A ***bytecode compiler*** translates a so-called ***high-level*** language like R, which is closer to human spoken language and therefore easier to understand but relatively slow, into a ***low-level*** language called ***bytecode***, which is closer to the computer's ***machine instruction language*** and therefore harder to understand but much faster.

- R comes equipped with its own bytecode compiler, in the built-in package `compile`, which we can use to try to speed up our code. We'll look at one function from the `compile` package:

```
cmpfun() # Translate (compile) a function from R code to bytecode
```

- Here's an example of how `cmpfun()` is used to speed up the first example of Subsection 2.4:

```
x <- runif(100000)
y <- runif(100000)
z <- rep(NA, 100000)
library(compiler)     # The 'compiler' package comes built-in with R
f <- function() {
    for(i in 1:length(x)) {
        z[i] <<- x[i] + y[i]    # Note the use of '<<-'
    }
}
```

```
cf <- cmpfun(f)
```

```
system.time(cf())
```

```
##    user  system elapsed
##    0.02    0.00    0.01
```

We see by comparison that the compiled code runs faster than the uncompiled code in the first example of Subsection 2.4. (Note, though, that using bytecode still isn't as fast as using vectorization.)

Basic Course on R: Object-Oriented Programming Practical

Elizabeth Ribble*

17-21 December 2018

Contents

*emcclel3@msudenver.edu

1 Part A: Object Oriented Programming

1. The ***geometric mean*** can be defined as the nth root of the product of n positive numbers $x_1, x_2, \ldots, x_n$, i.e.

$$\text{gm} \;=\; (x_1 \cdot x_2 \cdots x_n)^{\frac{1}{n}}$$

   Write a function `gm()` that takes a vector argument `x` containing positive numbers and returns their geometric mean. Your function should include a `stop()` statement that returns an error message if any of the values in `x` are nonpositive. **Hint**: The function `prod()` can be used to compute the product of the values in `x`.

2. Determine the class of your output by running the following:

```
class(gm(1:4))
```

3. Modify your geometric mean function, using `class()`, so that the return value has the class `"geometric"`.

4. Verify the new class of your output is correct by running the following

```
class(gm(1:4))
```

2 Part B: Performance Enhancement: Speed

1. This problem concerns efficiency and timing code.

   Using `system.time(expression)`, explore how time changes with size of the inputs (e.g. use sizes 100, 1000, 10000, 100000, 1000000, 10000000). Plot time versus input size and see if algorithm is linear, polynomial, or exponential.

   - Move expressions within loops that are invariant to compute just once and assigned to a variable.
   - Avoid concatenating vectors by pre-allocating and assigning to the ith element. i.e.

     ```
     x <- c(); for(i in seq(along = y)) x = c(x, g(y[i]))
     ```

   What are your conclusions?

Basic Course on R: Object-Oriented Programming Practical Answers

Elizabeth Ribble*

17-21 December 2018

Contents

*emcclel3@msudenver.edu

1 Part A: Object Oriented Programming

1. The ***geometric mean*** can be defined as the nth root of the product of n positive numbers $x_1, x_2, \ldots, x_n$, i.e.

$$\text{gm} = (x_1 \cdot x_2 \cdots x_n)^{\frac{1}{n}}$$

Write a function `gm()` that takes a vector argument `x` containing positive numbers and returns their geometric mean. Your function should include a `stop()` statement that returns an error message if any of the values in `x` are nonpositive. **Hint**: The function `prod()` can be used to compute the product of the values in `x`.

```
gm <- function(x) {
  if(any(x <= 0)){
    stop("All values in the vector must be positive.")
  }
  else{
    y <- prod(x)
    geom <- y^(1/length(x))
    return(geom)
  }
}
```

2. Determine the class of your output by running the following: ¿ class(gm(1:4))

```
class(gm(1:4))


## [1] "numeric"
```

3. Modify your geometric mean function, using `class()`, so that the return value has the class `"geometric"`.

```
gm <- function(x){
  if(any(x <= 0)){
    stop("All values in the vector must be positive.")
  }
  else{
    geom <- prod(x)^(1/length(x))
    class(geom) <- "geometric"
    return(geom)
  }
}
```

4. Verify the new class of your output is correct by running the following ¿ class(gm(1:4))

```
class(gm(1:4))

## [1] "geometric"
```

2 Part B: Performance Enhancement: Speed

1. This problem concerns efficiency and timing code.

   Using `system.time(expression)`, explore how time changes with size of the inputs (e.g. use sizes 100, 1000, 10000, 100000, 1000000, 10000000). Plot time versus input size and see if algorithm is linear, polynomial, or exponential.

   - Move expressions within loops that are invariant to compute just once and assigned to a variable.
   - Avoid concatenating vectors by pre-allocating and assigning to the ith element. i.e.

     ```
     x <- c(); for(i in seq(along = y)) x = c(x, g(y[i]))
     ```

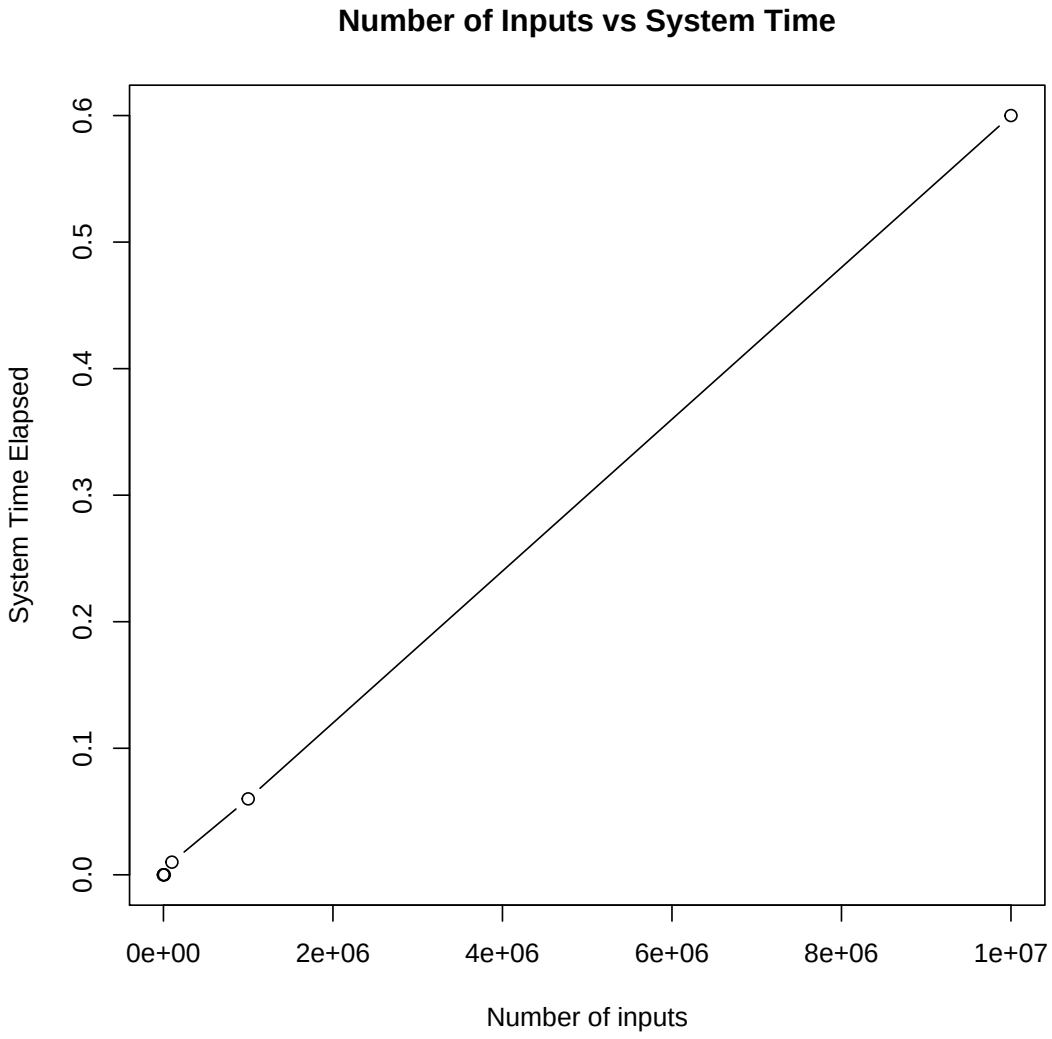
   What are your conclusions?

   Answers may vary.

```
time <- function(n){
  x <- runif(n)
  y <- sqrt(2)
  t <- system.time( for(i in 1:n) {
    x[i] <- x[i] + y
  }
  )
  return(t[["elapsed"]])
}
inputs <- c(100, 10^3, 10^4, 10^5, 10^6, 10^7)
t1 <- time(100)
t2 <- time(10^3)
t3 <- time(10^4)
t4 <- time(10^5)
```

```
t5 <- time(10^6)
t6 <- time(10^7)
timeV <- c(t1, t2, t3, t4, t5, t6)
```

```
plot(x = inputs, y = timeV, main = "Number of Inputs vs System Time",
     xlab = "Number of inputs", ylab = "System Time Elapsed",
     type = "b")
```



The scatterplot of computation time versus size of inputs is somewhat linear.