Train MALTA Horizon 2020

HTS Summer School
12-16th September 2016

A beginners guide to solving biological problems in R

Romina Petersen (rp520@medschl.cam.ac.uk)
Luigi Grassi (lg490@medschl.cam.ac.uk)

Original slides by Ian Roberts and Robert Stojnić

About this introduction

0

What will we do today?

- unfortunately I can’t teach you R in a few hours
- luckily you wont need me to

- you will get a lot of errors
- you might be frustrated
- you might need to google

… just like all R users

- today is more about providing the vocabulary you will need for google when alone (use “R:”)

Introduction to R and its environment

1

What is R and why do we use it?

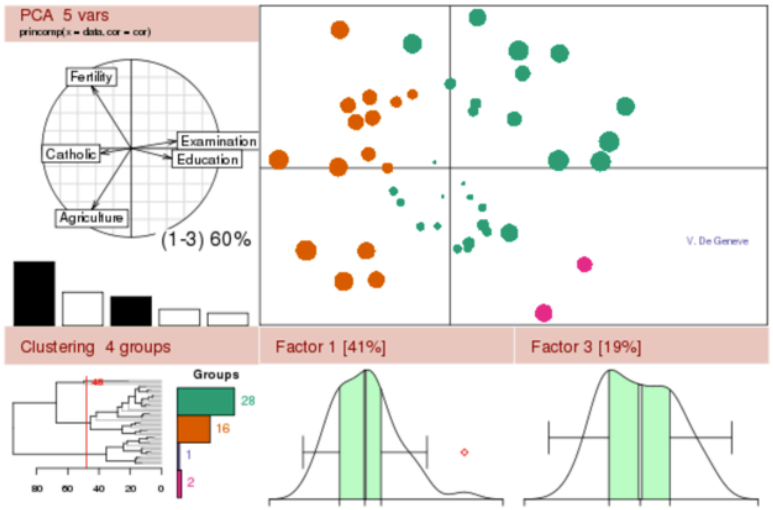
- a statistical programming environment
- based on the statistical programming language S
- suited to high level data analysis
- open source & cross platform
- extensive graphics capabilities
- diverse range of add-on packages
- active community of developers
- thorough documentation

The R Project for Statistical Computing

www.r-project.org

Apple Read Later Subscribe Yahoo! Google Maps YouTube Wikipedia Hermes Library Search Employee self-service R Course

The R Project for Statistical Computing



About R
What is R?
Contributors
Screenshots
What's new?

Download, Packages
CRAN

R Project
Foundation
Members & Donors
Mailing Lists
Bug Tracking
Developer Page
Conferences
Search

Documentation
Manuals
FAQs
The R Journal
Wiki
Books
Certification
Other

Misc
Bioconductor
Related Projects
User Groups
Links

Getting Started:

- R is a free software environment for statistical computing and graphics. It compiles and runs on a wide variety of UNIX platforms, Windows and MacOS. To **download R**, please choose your preferred CRAN mirror.
- If you have questions about R like how to download and install the software, or what the license terms are, please read our answers to frequently asked questions before you send an email.

News:

- **The R Journal Vol.5/1** is available.
- **R version 3.0.1** (Good Sport) has been released on 2013-05-16.
- **R version 2.15.3** (Security Blanket) has been released on 2013-03-01.
- **useR! 2013**, will take place at the University of Castilla-La Mancha, Albacete, Spain, July 10-12 2013. .

This server is hosted by the Institute for Statistics and Mathematics of WU (Wirtschaftsuniversität Wien).

www.r-project.org

Getting Started

- R is a program which, once installed on your system, can be launched and is immediately ready to take input directly from the user
- there are two ways to launch R:

1) from the command line (particularly useful if you're quite familiar with Linux)

2) as an application called RStudio

Prepare to launch R

From command line

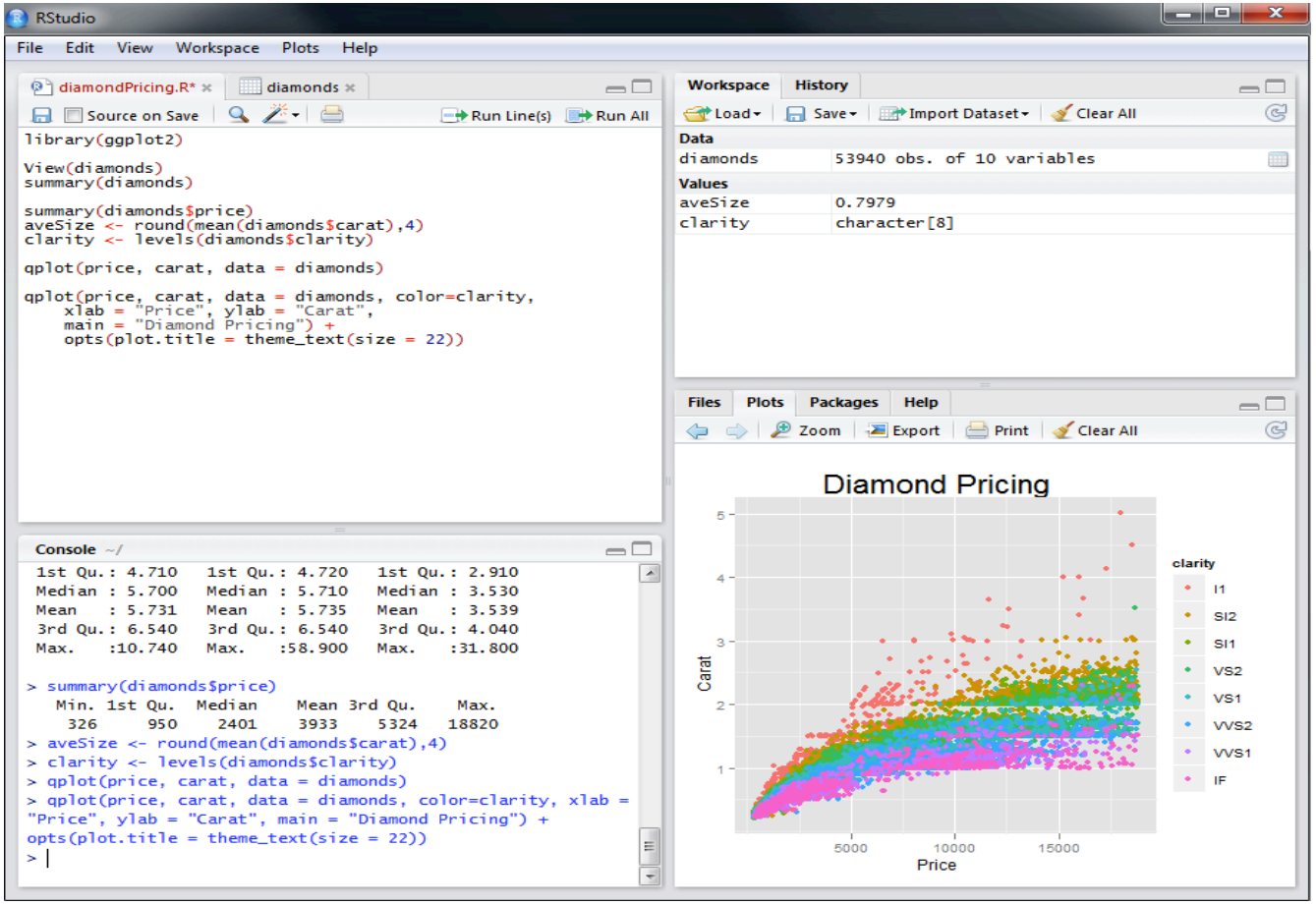
- to start R in Linux we need to enter the Linux console (also called Linux terminal and Linux shell)

- to start R, at the prompt simply type:

```
$ R
```

Prepare to launch R
Using RStudio

- to launch RStudio, go to the web host http://ec2-52-209-201-139.eu-west-1.compute.amazonaws.com:3000 and go to “Connect to RStudio”

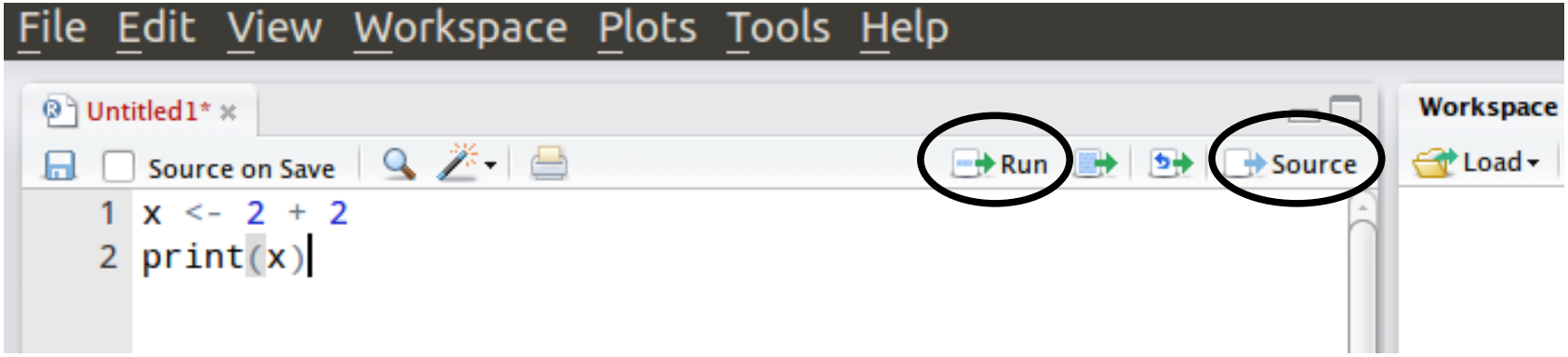


Writing scripts with Rstudio

- Typing lots of commands into R can be tedious. A better way is to write the commands to a file and the load it into R.
- Click on **File -> New** in Rstudio
- Type in some R code, e.g.

```
> x <- 2 + 2
> print(x)
```

- Click on **Run** to execute the **current line**, and **Source** to execute the **whole script**



- Sourcing can also be performed manually with

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

The working directory (wd)

- like many programs R has a concept of working directory (wd)
- it is the place where R will look for files to execute and where it will save files, by default
- for this course we need to set the working directory to the location of the course scripts
- at the command prompt in the terminal or in RStudio console type:

```
> setwd("~/scratch/day1")
```

Interacting with the R console

- R console symbols:
- # comment

  indicates text is a comment and not executed
- + command line wrap

  R is waiting for you to complete an expression
- Ctrl-c or escape to clear input line and try again
- Ctrl-l to clear window
- press q to leave help (using R from the terminal)
- use the TAB key for command auto completion
- use up and down arrows to scroll through the command history
- Alt-- (Alt and minus) gives the assignment operator <-
- extra spaces usually do not matter

Basic concepts in R

Command line calculation

- the command line can be used as a calculator:

```
> 2 + 2
[1] 4

> 20/5 - sqrt(25) + 3^2
[1] 8

> sin(pi/2)
[1] 1
```

- Note: The number in the square brackets is an indicator of the position in the output. In this case the output is a 'vector' of length 1 (i.e. a single number). More on vectors coming up…

Basic concepts in R

Variables

- a **variable** is a letter or word which takes (or contains) a value. We use the assignment 'operator', **<-** (or **=**)

```
> x <- 10
> x
[1] 10
> myNumber <- 25
> myNumber
[1] 25
```

- we can perform arithmetic on variables:

```
> sqrt(myNumber)
[1] 5
```

- we can add variables together:

```
> x + myNumber
[1] 35
```

Basic concepts in R

Variables

- we can change the value of an existing variable:

```
> x <- 21
> x
[1] 21
```

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

```
> x <- myNumber
> x
[1] 25
```

- we can modify the contents of a variable:

```
> myNumber <- myNumber + sqrt(16)
> myNumber
[1] 29
```

Basic concepts in R

Vectors

- the basic data structure in R is a **vector** – an ordered collection of values. The function **c()** *combines* its arguments into a vector:

```
> x <- c(3, 4, 5, 6)
> x
[1] 3 4 5 6
```

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

```
> x[1]
[1] 3
> x[4]
```

- we can even put a vector inside **[]** (vector indexing):

```
> y <- c(2, 3)
> x[y]
[1] 4 5
```

Basic concepts in R

Vectors

- there are shortcuts to create a vector. Instead of:

```
> x <- c(3, 4, 5, 6, 7, 8, 9, 10, 11, 12)
```

- we can write:

```
> x <- 3:12
```

- or we can use the **seq()** function, which returns a vector:

```
> x <- seq(2, 10, 2)
> x
[1] 2 4 6 8 10
> x <- seq(2, 10, length.out = 7)
```

- or the **rep()** function:

```
> y <- rep(3, 5)
> y
[1] 3 3 3 3 3
> y <- rep(1:3, 5)
```

Basic concepts in R

Vectors

- for extracting elements from a vector, we can use shortcuts to make things easier (or more complex!):

```
> x <- 3:12
> x[3:7]
[1] 5 6 7 8 9
> x[seq(2, 6, 2)]
[1] 4 6 8
> x[rep(3, 2)]
```

- we can add an element to a vector:

```
> y <- c(x, 1)
> y
[1] 3 4 5 6 7 8 9 10 11 12 1
```

- we can glue vectors together:

```
> z <- c(x, y)
```

Basic concepts in R

Vectors

- we can remove element(s) from a vector:

```
> x <- 3:12
> x[-3]
 [1]  3  4  6  7  8  9 10 11 12
> x[-(5:7)]
 [1]  3  4  5  6 10 11 12
> x[-seq(2, 6, 2)]
```

- finally, we can modify the contents of a vector:

```
> x[6] <- 4
> x
 [1]  3  4  5  6  7  4  9 10 11 12
> x[3:5] <- 1
```

- Keep in mind: **square** brackets for indexing **[]**, **parentheses** for function arguments **()**!

Basic concepts in R

Vector arithmetic

- when using standard arithmetic operations on vectors, application is element-wise:

```
> x <- 1:10
> y <- x*2
> y
 [1]  2  4  6  8 10 12 14 16 18 20
> z <- x^2
```

- adding two vectors:

```
> y + z
 [1]  3  8 15 24 35 48 63 80 99 120
```

- if vectors are not the same length, the shorter one will be recycled:

```
> x + 1:2
 [1]  2  4  4  6  6  8  8 10 10 12
```

- but be careful if the vector lengths aren't factors of each other:

```
> x + 1:3
```

Basic concepts in R

Character vectors and classes

- all the vectors we have seen so far have contained numbers, but we can also create **character** vectors:

```
> gene.names <- c("Pax6","Beta-actin","FoxP2","Hox9")
```

- BUT: each vector can contain only a single class of elements:

```
> y <- c(3,"F", x)
> y
```

- class can be:

numeric	0.5, 1000, pi
integer	1, 2, 3
character	"actin", "Romina"
logical/Boolean	TRUE or FALSE
factor	read as character, but treated as categorical

- important debugging note: always check the class of your object!

Basic concepts in R
Advanced indexing

- all values in R really are vectors, so indices are actually vectors, and can be numeric or logical:

```
> s <- letters[1:5]
> s[c(1,3)]
[1] "a" "c"
> s[c(TRUE, FALSE, TRUE, FALSE, FALSE)]
[1] "a" "c"
> a <- 1:5
> a < 3
[1]  TRUE  TRUE FALSE FALSE FALSE
> s[a<3]
[1] "a" "b"
> s[a>1 & a<3]
[1] "b”
> s[a==2]
```

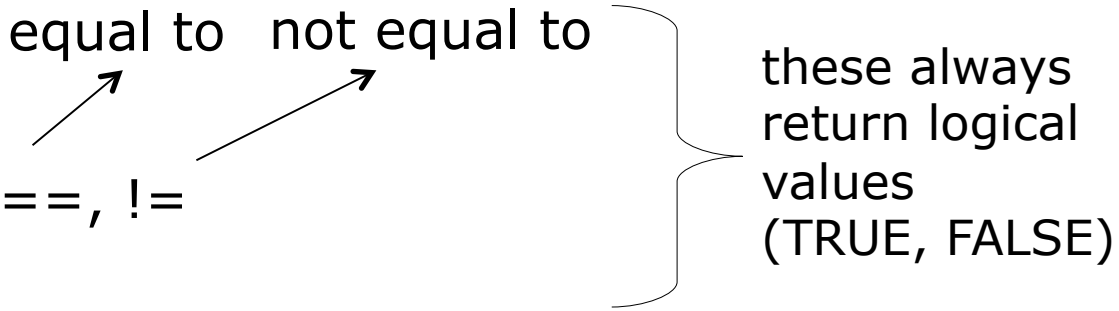
Basic concepts in R
Operators

- **arithmetic**

  +, -, *, /, ^

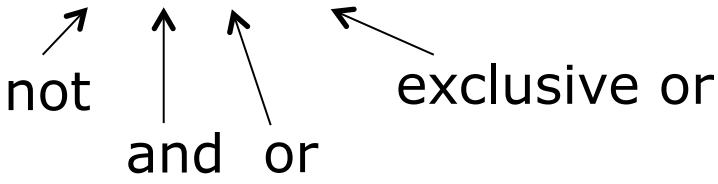
- **comparison**

  <, >, =<, >=, ==, !=



- **logical**

  !, &, |, xor



Basic concepts in R

Functions

- **Functions** in R perform operations on **arguments** (the input(s)). E.g. **sin(x)** has one argument, **x**. Arguments are *always* contained in parentheses **()**, separated by commas:

```
> sum(3, 4, 5, 6)
[1] 18
> max(3, 4, 5, 6)
[1] 6
> min(3, 4, 5, 6)
[1] 3
```

- Arguments can be named or unnamed, but if they are unnamed they must be ordered:

```
> seq(from=2, to=10, by=2)
[1] 2 4 6 8 10
> seq(2, 10, 2)
[1] 2 4 6 8 10
```

Basic concepts in R

Functions

- to get help on any R function, type `?` followed by the function name. For example:

```
> ?seq
```

- this retrieves the syntax and arguments for the function. You can see the default order of arguments here. The help page also tells you which **package** it belongs to.
- there will typically be example usage, which you can test using the **example** function:

```
> example(seq)
```

- if you can't remember the exact name type `??` followed by your guess. R will return a list of possible ones:

```
> ??rint
```

Basic concepts in R

Functions

- functions can be concatenated:

```
> length(seq(1,100,2))
```

- we can also write our own functions! makes sense if we want to carry out an operation repeatedly

```
> add_1 <- function(number) {
      number + 1
  }

> add_1(1)
  [1] 2
```

- or we can get additional functions by loading R packages (more later)
- Keep in mind: **square** brackets for indexing **[]**, **parentheses** for function arguments **()**!

Questions???

Exercise

Variables, vectors and functions

- Save your code for the answers to a script called "Exercise1.R" in your working directory and add comments

1. Assign the value of `40` to `x`
2. Assign the value of `30` to `y`
3. Make `z` the value of `x` minus `y`
4. Display the value of `z` in the console
5. Create a vector called `vect1` containing the values of `x`, `y` and `z`
6. calculate the `sum`, `mean`, `min` and `max` of `vect1`
7. Use `a:b` notation to create a second vector `vect2` containing the numbers `4` to `6`
8. Subtract `vect2` from `vect1` and look at the result
9. Use `seq` to create a vector `vect3` of 50 values starting at 2 and increasing by 4 each time
10. Extract the values at positions `5`, `10` and `15` of `vect3`

R and experimental data

2

R is designed to handle experimental data

- very often experimental data come as “tables”
- there are two types of “table” in R: matrices and data frames
- a **matrix** can only hold one class of data, e.g. only numeric
- a **data frame** can be a mix of classes (and therefore is more common)
- you can think of individual columns of a data frame as individual vectors
- we will use the iris data set as an example (already loaded):

```
> iris
```

- usually we would load our data with

```
> read.table(“path/to/table”)
> read.csv(“path/to/table”)
> read.delim(“path/to/table”)
```

Data frames

- always check your data before you start working with them!
- useful functions:

```
> head()
> tail()
> dim()
> length()
> summary()
> str()
> colnames()
> class()
```

- we can address individual columns with a `$` sign:

```
> iris$Species
> class(iris$Species)
```

Indexing data frames

Special cases:
a[*i*,] i-th row
a[,*j*] j-th column

- we can index multidimensional data structures like data frames using commas. If you don't provide an index for either rows or columns, all of the rows or columns will be returned:

```
object [ rows , columns ]
> iris[3,2]
[1] 3.2
> iris[1,]
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
> iris[3,"Species"]
[1] setosa
> iris$Sepal.Length[3]
[1] 4.7
```

Data frames

- we can add rows and columns to a data frame (as long as they are vectors of the appropriate length):

```
> year <- rep(2014,150)
> iris_year <- cbind(iris, year)

> new_obs <- c(4.6, 3.6, 1.0, NA, "setosa", 2016)
> iris_year_current <- rbind(iris_year, new_obs)
```

- info: R uses NA to represent a missing value
- we can remove rows and columns from a data frame
- we can change the values in a data frame:

```
> iris[1,1] <- 5.5
```

- we can save our processed data as tables:

```
> write.table(data.frame, "path/to/table")
> write.csv(data.frame, "path/to/table")
```

Plotting

- R comes with some nice plotting functions:

```
> plot()
> barplot()
> boxplot()
> hist()
```

- We can use these functions to look at our data:

```
> plot(iris$Petal.Length)
> plot(iris$Petal.Length, iris$Petal.Width)
> boxplot(iris$Petal.Length, iris$Petal.Width)
> hist(iris$Petal.Length)
> boxplot(iris$Petal.Length~iris$Species)
```

Plotting

- plotting functions accept a lot of arguments, so we can tailor our plots to our liking (colours, labels, background,…) – check out the help page (and google!)
- for example, there are whole colour palettes available
- more advanced plots can be created with the ggplot2 CRAN R package
- we can save our plots for example as a pdf either by clicking "Export" on the "Plots" tab, or using the `pdf()` function
- we can also save them in various other graphical formats

Lists

- If we want to store data of multiple types, or vectors of different lengths in one object, we can use a **list**. Lists can contain objects of any type:

```
> vect1 <- c(1,2,3,4)
> vect2 <- c("a", "b", "c”)
> data.frame(vects1, vect2)
   Error in data.frame(vect1, vect2)
> list1 <- c(vect1, vect2)
   [[1]]
   [1] 1 2 3 4
   [[2]]
   [1] “a” “b” “c”
```

- you can recognise lists by the double square brackets **[[]]**
- some functions return the result as a list (for me often undesired), **`unlist()`** can help (but be careful!)

Questions???

Exercise

Data frames and plotting

- Save your code for the answers to a script called "Exercise2.R" in your working directory and add comments

1. How many observations are in the iris data frame?
2. What is the mean petal length?
3. How many species were analysed, and how many samples of each species?
4. What is the minimum sepal width for species versicolor?
5. Create a scatterplot of the sepal length of all samples
6. Create a boxplot showing the Sepal.Length for each of the species, and give each of the boxes a different colour
7. Add a meaningful title, x-axis label and y-axis label to your boxplot (hint: look at ?plot)
8. Save your boxplot to your working directory

Advanced data analysis

3

Three steps to data analysis

1. **reading in data**
- read.table()
- read.csv(), read.delim()

2. **analysis**
- manipulating & reshaping the data
- any maths you like
- plotting the outcome

3. **writing out results**
- write.table()
- write.csv()

R packages

- R comes ready loaded with various libraries of functions called **packages**
- there are 1000s of additional packages provided by third parties, and the packages can be found in numerous server locations on the web called **repositories**
- the two repositories you will come across the most are **The Comprehensive R Archive Network (CRAN)** and **Bioconductor**
- to install a CRAN R package, simply type

```
> install.packages("PackageName")
```

- to install a Bioconductor package, set the Bioconductor package download tool by typing

```
> source("http://bioconductor.org/biocLite.R")
```

and then load the package with the biocLite function:

```
> biocLite("PackageName")
```

R packages

- 5400+ packages on CRAN:
  use CRAN search to find functionality you need:
  http://cran.r-project.org/search.html
  or look for packages by theme:
  http://cran.r-project.org/web/views/
- 750 packages on Bioconductor
  specialised in genomics:
  http://www.bioconductor.org/packages/release/bioc/
- Other repositories:
- 1700+ projects on R-forge: http://r-forge.r-project.org/
- R graphical manual: http://rgm3.lab.nig.ac.jp/RGM
- **always** look first if there is already an R package available that does what you want before trying to implement it yourself!
- every package comes with documentation (manuals/vignettes)

R packages

- packages we regularly use in genomics (very subjective list):
- **GenomicRanges**: very useful when working with genomic regions, e.g. if we want to find out whether our transcription factor binding sites overlap promoters or enhancers
- **biomaRt**: to extract information from databases, e.g. to find the Ensembl gene ID for a list of our genes of interest
- **reshape2**: to remodel data frames prior to plotting with ggplot2
- **ggplot2**: advanced graphics
- **DESeq2**: popular for differential expression analysis

Exercise
Installing swirl

- we will install swirl, a package for learning R interactively
- swirl is a CRAN package
- Use the **`install.packages()`** function – only required once for each package:

```
> install.packages("swirl")
```

- R needs to be told to load the newly installed functions from the new package – required for every R session in which you want to use the package

```
> library("swirl")# loads swirl functions
```

- one of the new functions is **`swirl()`**

Exercise
Using swirl

{swirl}
Learn R, in R.

- type `swirl()`
- follow the instructions to choose a name and install the **R programming** course
- type `info()` at the prompt to get swirl options
- go through as many lessons as you like in whatever order you prefer (you can also easily continue at home!)
- very relevant lessons: 1,3,4,5,6,7,8,9,12,15
- not as relevant: 13,14
- advanced: 2,10,11
- you can also load additional courses (check out the homepage)
- swirl can be very exact about the answers
- sometimes swirl gives up on you too easily and prints the answer for you – in that case try to understand it!
- the individual lessons do not always link well to the previous ones

Thank you!
Questions???