BITS R tutorial for the RNA-Seq prep course

Software for
Statistical Computing and Graphing

April, 2017

Janick Mathys

VIB-BITS
Contents

1. Introduction: What is R? ..................................................................................................................... 4
   1.1. Good things about R ...................................................................................................................... 4
   1.2. Bad things about R ....................................................................................................................... 4
2. Installation ........................................................................................................................................... Error! Bookmark not defined.
   2.1. Installing R .................................................................................................................................. Error! Bookmark not defined.
   2.2. Installing RStudio ........................................................................................................................ Error! Bookmark not defined.
   2.3. User interface ............................................................................................................................... Error! Bookmark not defined.
   2.4. Installing packages ....................................................................................................................... Error! Bookmark not defined.
3. Documentation .................................................................................................................................... 8
4. Working in RStudio ............................................................................................................................ 9
5. Expressions in R ............................................................................................................................... 10
6. Creating and running scripts .......................................................................................................... 11
   6.1. Assigning variables ....................................................................................................................... 11
      6.1.1. Using operators ....................................................................................................................... 11
      6.1.2. Using functions ....................................................................................................................... 12
   6.2. Creating a new script .................................................................................................................... 13
   6.3. Opening an existing script .......................................................................................................... 13
   6.4. Running a script ........................................................................................................................... 14
   6.5. Recommendations for writing scripts in R .................................................................................. 15
      6.5.1. Adding comments to your scripts ......................................................................................... 15
      6.5.2. Removing old variables ........................................................................................................ 15
      6.5.3. Adding section headings to your scripts ............................................................................... 15
7. Saving info .......................................................................................................................................... 17
   7.1. Saving a script ............................................................................................................................. 17
   7.2. Saving the environment ............................................................................................................... 17
   7.3. Saving a plot ............................................................................................................................... 17
8. Data types in R ..................................................................................................................................... 18
   8.1. Vectors .......................................................................................................................................... 18
      8.1.1. Creating a vector ...................................................................................................................... 18
      8.1.2. Referring to the elements of a vector ..................................................................................... 18
   8.2. Data frames ............................................................................................................................... 19
      8.2.1. Creating a data frame ............................................................................................................. 19
8.2.2. Referring to the elements of a data frame ................................................................. 20
8.3. Factors ......................................................................................................................... 20
9. Operations on variables ................................................................................................. 22
  9.1. General operations ..................................................................................................... 22
  9.2. Functions helpful for working with large data sets .................................................... 22
  9.3. Functions for finding the location of specific elements ............................................ Error! Bookmark not defined.
  9.4. Checking and converting types of variables ............................................................... 23
10. Reading and writing files .............................................................................................. 24
11. Creating graphics using the ggplot2 package ............................................................. 26
  11.1. Histograms ................................................................................................................ 29
  11.2. Scatter plots .............................................................................................................. 30
  11.3. Adding a line to a plot ............................................................................................... Error! Bookmark not defined.
  11.4. Defining the titles of the plot ................................................................................... Error! Bookmark not defined.
  11.5. Scales for controlling the visual elements of the plot .............................................. 30
  11.6. Themes ..................................................................................................................... 30
1. Introduction: What is R?

R is a computer language that allows manipulation of data, statistical analysis and visualization.

It is not easy to learn the language if you haven't done any programming before but it is worth taking the time as it can be a very useful tool. An enormous variety of statistical analyses are available and R allows you to produce graphs exactly as you want them with publication quality.

1.1. Good things about R

- It's free
- It works on Windows, Mac and Linux
- It can deal with very large datasets (compared to Excel for example)
- A lot of freedom: graphs can be produced to your own specification
- Supports all statistical analyses: from the basics to very complex statistical analyses

1.2. Bad things about R

- It can struggle with extremely large datasets
- Difficult if you don't have any programming experience
- Open source: many people contribute therefore consistency can be low
- Open source: documentation of packages can be poor or written by/for experts
1. Installation

1.1. Installing R
R is free software, available at the CRAN website. It can be installed on Linux, Mac and Windows.

1.2. Installing RStudio
Although you can work directly in the R editor, most people find it easier to use RStudio on top of R. RStudio is free. It’s available for Windows, Mac and Linux. You need to have R installed to run Rstudio.

1.3. User interface
Video tutorial on RStudio user interface: https://www.youtube.com/watch?v=o0Y478jOjGk

The user interface consists of 4 windows:

**Script editor**: create and run R scripts. R scripts are programs consisting of R commands. The editor will highlight the syntax in your scripts making it easier to find and prevent errors... You can run the commands in the script editor, which means that they are sent to the console for execution.

**Console**: execute commands and show their output. You can type commands directly in here. This is the actual R session where computation is done. In the console you can scroll through previous command by using the up arrow ↑ on the keyboard.
The > symbol shows that R is ready for something to be entered, e.g. type `10+3` and press return.

```
> 10 + 3
[1] 13
```

The answer is printed in the console. If you don’t see the > symbol reappearing upon execution of a command it means that R crashed (or is still calculating). To terminate a command you press Esc.

**Overview of all RStudio keyboard shortcuts**
Workspace browser

Files + Plots + Packages + Help + Viewer

The latter two windows consist of multiple tabs:

Environment: list of all variables (plots, models) that have been imported or generated. The variables that R creates and manipulates are known as objects.

History: list of all commands which you have ever entered into the console. More info on how to use the history: https://support.rstudio.com/hc/en-us/articles/200526217-Command-History

Connections: interface to easily connect to databases in R. Read more on http://db.rstudio.com/ (not used in this training)

Files: list of files and folders in the work directory

Plots: visualizes the plots (graphs) that you generate

Packages: list of installed packages

Help: links to R manual and other useful info

Viewer: views HTML files that are located on your computer (not used in this training)

Every time you launch RStudio, it has the same text at the top of the console. Below that text is the prompt >. This prompt is a request for a command. You can type and execute commands in the console but it is recommended to write and execute commands from the script editor. The script editor allows to save the scripts making it easier to reproduce and reuse sequences of commands.

Features of the console that make life easier: autocompletion, retrieving previous commands...

1.4. Installing packages

R is popular because of the enormous diversity of packages, collections of R programs that are able to perform a certain analysis, e.g. Matrix is a package that contains all the R-code you need for creating and working with matrices. Packages are available at the CRAN and Bioconductor websites.

To install a package go to the Packages tab and click the Install button:

Once you start typing the name of a package, RStudio tries to autocomplete it:
Select the package you want to install – in our case ggplot2 - and click the **Install** button. In the **console** you can see if the installation was successful.

```console
> install.packages("ggplot2")
Installing package(s) into 'D:/R-2.15.2/library'
(as 'lib' is unspecified)

There is a binary version available (and will be installed) but the source version is later:
  binary source
  ggplot2 0.9.3.1 1.0.0

trying URL 'http://cran.rstudio.com/bin/windows/contrib/2.15/ggplot2_0.9.3.1.zip'
Content type 'application/zip' length 2667652 bytes (2.5 Mb)
opened URL 2.5 Mb

package 'ggplot2' successfully unpacked and MDS sums checked

The downloaded binary packages are in
  C:\Users\Janick\AppData\Local\Temp\RtmpQvbxhh\downloaded_packages
>
```

You can also install a package by typing the following command:

```r
> install.packages("name_of_package")
```

You only need to install a package once. But each time you want to use a package you have to load it (activate its functions). Loading a package is done by selecting it in the list of installed packages or by typing the following command directly in the console or as part of a script:

```r
> library("name_of_package")
```

Note that when you have made changes to the right side of the user interface: the workspace browser, the packages, the files... R is sometimes slow to show these changes. In that case you can hit the refresh button:

![Refresh button](image)

For bioinformatics analyses you often have to install Bioconductor packages. Bioconductor is an R package that provides tools for the analysis of high-throughput data, e.g. NGS data.

To install Bioconductor itself you can type the following commands in RStudio:

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

To install specific Bioconductor packages e.g., "GenomicFeatures" and "AnnotationDbi" you type the following commands in RStudio:

```r
biocLite(c("GenomicFeatures", "AnnotationDbi"))
```

You can find an overview of all available Bioconductor packages on the following web pages:

https://www.bioconductor.org/help/workflows/
https://www.bioconductor.org/packages/release/BiocViews.html#___Software

**Further reading:**

[Video tutorial on installing packages in RStudio](link)
2. Documentation

You can find a lot of documentation online:
- The documentation section of the R website
Unfortunately this section is nor easily accessible nor well-structured and it can be quite a challenge to consult the help files of different R packages and functions online. By far the most user-friendly interface for searching the R documentation is the Rdocumentation website.
- Documentation of RStudio
- Quick R: for those who would like to make the transition to R (from SAS, SPSS, Stata)
- R-bloggers: R-news and tutorials contributed by bloggers
- Rseek: R has an unhelpful name when it comes to googling problems. That’s why rseek.org was created: it’s a google for R.
- Google’s R style guide: Programming rules for R designed to make R code easier to read, share, and verify. The rules were designed in collaboration with the entire R user community at Google.

You can access the R documentation by using R commands. For all help-related commands we refer to the Getting help section of the R Reference Card.
4. Working in Rstudio

The working directory is a folder on your computer where R will store all things you save or write and where R will search for files to load or read. The default working directory is the home directory (C:/Users/Your_Name/ in Windows) typically referenced using ~ in R. Since R will search for files in this folder it is advised to put all your input files there. If you store them elsewhere you have to write out the full path each time that you want to read in files.

You can specify the working directory. To check which directory you’re working in at the moment use the following command:

```r
> getwd()
```

To change the working directory use the following command:

```r
> folder = "D:/trainingen/zelfgegeven/R/"
> setwd(folder)
```

**Exercise 1:** Set the working directory to the following location:

```
/home/VIBTrainingX/data/bulk/example_data/Rprep/
```

Use `getwd()` to see if the working directory was correctly set.

Changing your working directory will make relative file references in your code invalid so you need to type this in the console at the start of the analysis or type these lines at the start of each script you generate (see chapter 5).

Alternatively you can change the working directory in the Files tab, expand More and select Set As Working Directory.

Note that you need to use / or \ in paths. Either will work but \ will not since R sees it as a special character.

You can list the files in the working directory by typing `list.files()`
5. Expressions in R

R can handle any kind of data: numerical, character, logical... **Character data** like “green”, “male”, “cytoplasm”... has to be typed in between double quotes. **Boolean** values are **TRUE** and **FALSE**. Although you write them as text, they are Booleans so you don’t have to use double quotes. Very important to know is that R is case sensitive: true and false are not recognized as Booleans; they have to written in capitals.

Missing values are represented by NA. Although you write NA as text you don’t use double quotes.

Arithmetic operators follow the standard order of priority, with exponentiation the highest and addition and subtraction the lowest priority, but you can always control the order with parentheses.

**Logical operators** (= < > !) can be used to create logical expressions that take values TRUE or FALSE. Logical expressions may be combined using the logical operators: | (OR) and & (AND).

An overview of the basic R syntax can be found in the **Basic syntax** section of the **R Reference Card**.

| Exercise 2A: What's the difference between x=2 and x==2? Use the console to find out. |
| As you can see the = operator attributes a value to a variable (see next section), x becomes 2. The == is a logical operator, testing whether the logical expression x equals 2 is TRUE or FALSE. |

| Exercise 2B: Check if the words UseR and user are equal. |
| Note that R is indeed case-sensitive ! |

| Exercise 2C: Check if TRUE equals 1. |
| Note the R indeed represents TRUE internally by 1 and FALSE by 0. |
6. Creating and running scripts

A script is a text file that contains all the commands you will use. You cannot only write and run scripts but you can also save them so next time you need to do a similar analysis you can change and re-run the script with minimal effort. An R project can contain multiple scripts.

6.1. Assigning variables

A variable allows you to save a value (e.g. 3) or an object (e.g. a plot, a matrix…) in R. You can then use the variable’s name to easily access the value or the object that is stored in the variable.

A value or object is assigned to a variable by the assignment operator <- consisting of the two characters < (less than) and - (minus).

E.g. v <- 4  now the value of variable v is 4

In most contexts the = operator can be used as an alternative (v <- 4 or v=4 is the same).

If R has performed the assignment you will not see any output, as the value 4 has been saved to the variable called v. You can access and use this variable at any time and print the value of v into the console by typing its name in the console.

> v
[1] 4

You can use v in expressions instead of 4

> v*v
[1] 16

You can re-assign any value to a variable at any time.

> v <- “a cool variable”
> v
[1] “a cool variable”

Very important to know is that R is case sensitive: v.one, V.one and v.One are different variables.

Variables can be named anything, though they cannot begin with a number or symbol. Having informative names is useful and this often involves using more than one word. Providing there are no spaces between your words you can join them in various ways, using dots, underscores and capital letters though the Google R style guide recommends that names are joined with a dot.

Exercise 3A: create a variable patients with value 42 and print the value of patients divided by 2

Exercise 3B: create a variable patients_gr2 with value 24 and print the total number of patients

Exercise 3C: create patients_gr3 with value “twenty” and print the total number of patients

6.1.1. Using operators

R allows you to combine variables into a new variable by using operators.
6.1.2. Using functions
R allows you to combine variables into a new variable by invoking functions.

A function is a named set of commands to manipulate data. There are lots of built in functions in R and you can also write your own.

Commands in R have a certain format:
output <- function(list of arguments)

For example:

    p <- ggplot(mtcars, (aes(wt,mpg)))

In this example ggplot() is the function. It generates a plot so the plot p is the output of the function. The brackets () are always needed. Before a function can start the actions and calculations that it encodes, it needs prior information: input data and parameter settings. These are called the arguments of the function. In this example the arguments are:
- mtcars: a data frame consisting of 9 columns containing the input data
- aes(wt,mpg): defines the two columns you want to plot: weight (wt) and miles/gallon (mpg) and how you want to plot them: wt along the X-axis and mpg along the Y-axis.

To find the arguments a function you need to use ? or the help() function in the Console:

    ? ggplot
    help(ggplot)

This opens the documentation of the function in the Help tab including info about the arguments of the ggplot function, at the bottom of the page you find examples on how to use the function.

**Exercise 3D**: Check the arguments of the mean() function.

You see that the mean() function has multiple arguments and each argument has a default value. If you want to stick to these default values you don’t have to specify them in your script. So you only have to define the arguments for which you want to use a value other than the default.

To bring up the examples of usage for a function instead of the full documentation page:

    > example(min)

If you don’t know the name of the function name e.g. for finding the standard deviation:

    > help.search("deviation")
    > ??deviation

You can also search the internet but the search term 'R' isn’t specific enough for returning relevant information so go to http://www.rseek.org/ to do your searches.

**Exercise 3E**: Calculate and print the sum of patients and patients_gr2 using the sum() function.

Sometimes functions from different packages have the same name. In that case you can use package::function to specify the package to take the function from, e.g. ggplot2::ggplot where ggplot2 is the name of the package and ggplot is the name of the function.
6.2. Creating a new script
Click **File** in the top menu: **New File > R Script**

As you can see there are many other file types you can create, like an R markdown file (allows to incorporate R-code and the results the code generates in a report, you can find more info on http://rmarkdown.rstudio.com/) or an R Sweave file (allows to incorporate R-code and the results the code generates in a LateX report).

New scripts are opened as a tab in the **script editor**. You can start typing R commands in it. RStudio has many features that will help you write scripts e.g. autocompletion, find/replace, commenting...

For instance, RStudio supports the automatic completion of code, e.g. if you have an object named `relfreq` in your workspace, you type `rel` and RStudio will show a list of possibilities to complete the name.

**Find and replace** can be opened using Ctrl+F.

6.3. Opening an existing script
To open an existing script you select **File -> Open File** in the top menu.

You can open several files in RStudio: they will be available as different tabs in the **script editor**
6.4. Running a script

To run a script you select the code that you want to execute in the script editor and click the Run button at the top right of the script editor. The code will be executed in the console.

**Exercise 4A:**
Type the three commands that you see in the figure above and select them
Click the Run button to execute the commands in the console

You will now see the commands appearing in the Console as they are executed:

```r
> library(ggplot2)
Warning message: package 'ggplot2' was built under R version 2.15.3
> p <- ggplot(mtcars, aes(x = wt, y = mpg))
> p + geom_point()
```

These commands create a plot, you see the plot appearing in the Plots tab.

To run a script you can also use the source("name_script") command e.g.

```
source("plotMeans.R")
```

It is recommended to write the commands you use in a script rather than typing them directly into the console. Creating a script makes it easier to reproduce, repeat and describe the analysis that you have done.
6.5. Recommendations for writing scripts in R

6.5.1. Adding comments to your scripts
In scripts you have to include documentation. You can include comments by preceding them with a # symbol. The comments are displayed in green. The # symbol tells R to ignore everything on this line. The comments make it easy to remember what you did or to tell collaborators what you did. You can comment and uncomment selections of code using Comment/Uncomment Lines:

Exercise 5A: Include some comments in the script that you have created.

6.5.2. Removing old variables
A good idea is to start a script with removing all variables that have been generated up to now to prevent errors such as use of old data.

To remove all variables use the following command:
```
rm(list=ls())
```

Exercise 5B: run the script and then run the command rm(list=ls()) from the Console. Each time look at the Environment tab to see what happens.

You see that the plot p that was generated by the script is removed by the rm() command.

You see that the plot p that was generated by the script is removed by the rm() command.

6.5.3. Adding section headings to your scripts
You can add section headings to your scripts using the following format:
```
#Heading Name####
```
Especially in long scripts this allows you to move quickly between sections

**Exercise 5C:** transform a comment into a section header.
At the bottom of the *Script editor* you can now navigate easily to this section in your script:
7. Saving info

7.1. Saving a script
A newly created script gets a default name e.g. Untitled1. As long as you have not saved the script the name appears in red and is followed by an asterisk (see figure at the top of this page).

To save the script click the Save ( ) button in the toolbar of the script editor.

Once you have saved the script the asterisk disappears and the name is displayed in black.

Exercise 6: save your script as DemoExercise.

7.2. Saving the environment
To save the complete environment click the Save button in the toolbar of the Environment tab of the upper right window.

7.3. Saving a plot
To save a plot click the Export ( ) button in the toolbar of the Plots tab.
8. Data types in R
R has a wide variety of data types including scalars, vectors, matrices, data frames, and lists.

8.1. Vectors
The simplest data structure is the vector, a single row consisting of data values of the same type, e.g. all numbers, characters, Booleans...

8.1.1. Creating a vector
The function `c()` (short for combine) is used to create vectors. The only arguments that need to be passed to `c()` are the data values that you want to combine.

You can create a numeric (a), character (b) or logical (c) vector:

```r
a <- c(1,2,5.3,6,-2,4)
b <- c("janick","jasper","niels")
c <- c(TRUE,TRUE,TRUE,FALSE,TRUE,FALSE)
```

**Exercise 7A:** You’re doing an experiment counting every day how many plants of your initial set of 40 plants developed lesions as a result of a mold infection. Create a vector plants containing the results of your counts: 1, 3, 4, 2, 6

**Exercise 7B:** Let’s assume the experiment started on a Monday. Create a vector days containing the days of the week in the following format: “Mon”, “Tue”, “Wed”, “Thu”, “Fri”.

More functions to create vectors can be found in the Data creation section of the R Reference Card.

If you need a vector with a sequence of consecutive integers you can create it with start:end notation, e.g. a vector with values from 5 through 9

```
5:9 will generate 5 6 7 8 9
```

You can also define a decreasing sequence of integers:

```
9:5 will generate 9 8 7 6 5
```

As you can see, some arguments of a function have a name, e.g. the increment argument is called by. In most cases you do not need to use the names of arguments, we’ll see in the section on data frames an example of when to use the name of an argument.

! Very important: words used as values are to be written between quotes, words used as variable names are not ! If R encounters a word without quotes it will try and find the variable with that name.

More functions to create vectors can be found in the Data creation section of the R Reference Card.

8.1.2. Referring to the elements of a vector
To refer to elements of a vector use their indices inside square brackets []
e.g. to retrieve the 2nd element of vector a use:

```
a[2]
```

to retrieve the 2nd, 3rd and 4th element of vector a use:

```
a[2:4]
```
to retrieve the 2\textsuperscript{nd} and 4\textsuperscript{th} element of vector a use:
\[
a[c(2,4)]
\]
You also see this when you look at the output in the console. The numbers in between square brackets tell you the index of the first value on the line.

\[
> v <- c(rep(5,10),rep(10,5))
\]
\[
[1] 5 5 5 5 5 5 5 5 10 10
[13] 10 10 10
\]

There are 12 values on the first line, so on the second line of data, the first value (10) is actually on the 13\textsuperscript{th} position in the vector v. So the [13] refers to the index of the first element on the line.

**Exercise 8A:** Retrieve the 4\textsuperscript{th} and 5\textsuperscript{th} elements from days.

**Exercise 8B:** Retrieve the elements from plants that are larger than 2.

More ways to refer to elements of vectors can be found in the Data extraction section of the R Reference Card.

### 8.1.3. Removing elements from a vector

To remove an element from a vector use negative indices: '-' indicates ‘NOT’ followed by the index of the element you want to remove

e.g. to remove the second element of vector z use
\[
z <- z[-2]
\]

**Exercise 8C:** Create a vector named x containing the numbers 20 to 2. Retrieve elements that are larger than 5 and smaller than 15.

**Exercise 8D:** Remove the first 8 elements from x and store the result in x2.

### 8.2. Data frames

A data frame is a table. In theory they are used when columns contain different data types (numeric, character, logical...). However, in practice data frames are also used to hold data of the same type. They are the typical variables used by statistical functions.

R has a number of built-in data frames, e.g. in chapter 4 we used a built-in data frame, called mtcars.

#### 8.2.1. Creating a data frame

To create a data frame D use

\[
D <- \text{data.frame}(column1, column2, column3,...)
\]

You can provide labels for the columns:

\[
D <- \text{data.frame}(label1=column1, label2=column2, label3=column3,...)
\]

**Exercise 9:** Create data frame Plant\_study containing days and plants. Name the columns Days and Plants.
### 8.2.2. Referring to the elements of a data frame

Referring to elements of a data frame can be done in the same way as for matrices, using row and column indices in between square brackets. The only difference is that in data frames you can also use the labels of the columns to retrieve them (see Data extraction section in the R Reference Card).

To retrieve the element on the second row, first column:

```
> D[2,1]
```

To select all values from one dimension leave out the index value, e.g. all elements of the first column:

```
> D[,1]
```

You can also use the column label for this. Note that column names have to be written between quotes:

```
> D[,"label1"]
```

You can also use the range functions e.g. to select the second, third and fourth element of the first column:

```
> D[2:4,1]
```

The $ symbol can be used to retrieve a specific column based on its label. Note that you do not have to write the label in between quotes, e.g. to retrieve column label1 from D:

```
> D$label1
```

More examples on retrieving elements from a data frame can be found in the Data extraction section of the R Reference Card.

---

### Exercise 10

**A:** On which days did we observe more than 2 infected plants in the plant experiment?

**B:** Retrieve the smoking behavior of the patients in the Drug_study

**C:** Change the treatment of the fourth patient to “A”

**D:** Add a column called activity to Drug_study with following values: 4, NA, 12.1, 2.5

### 8.3. Factors

You can tell R that a variable is categorical by making it a factor. A categorical variable represents a limited number of categories. A continuous variable is the result of a measurement and can correspond to an infinite number of values.

There are two types of categorical data. On the one hand there’s the unranked categorical data, which does not have an implied order. The ranked categorical data, on the other hand, do have a natural ordering.

R will treat factors by default as unranked but you can create ordered factors. Storing data as factors ensures that the graphing and statistical functions in R will treat such data correctly.

To create a factor, the `factor()` function is used:

```
> F <- factor(x,levels=c("b","a"), ordered=TRUE)
```
The only required argument is a vector of values \( x \) which will be factorized. Both numeric and character vectors can be made into factors but factors are typically used for numerical data that represents categories.

By default, the function `factor()` transforms a vector into an unordered factor. To create an ordered factor, you have to add two additional arguments: \texttt{ordered} and \texttt{levels}.

Optional arguments are:

- By setting the argument \texttt{ordered} to \texttt{TRUE}, you indicate that the factor is ordered.
- \texttt{levels}: a vector of the values (as character strings) of the factor. To change the order in which the levels are displayed (default is alphabetical order), the \texttt{levels=} argument can be given a vector of all the values in the correct order.

The levels of a factor are used when displaying the factor’s values.
9. Operations on variables

9.1. General operations
The big difference between R and other programming languages is that functions in R are designed to be applied to variables rather than to individual values, avoiding loops e.g. if we want to log transform a whole dataset we can do this using a single operation:

```r
> v <- c(1,10,100,1000,10000)
> log10(v)
[1] 0 1 2 3 4
```

The log10() function is written in such a way that it can be applied on a vector. This is true for all functions and operations in R:

```r
> v <- 1
[1] 0 9 99 999 9999
```

Arithmetic operations can also be performed on variables. Provided the variables have the same dimensions, you can perform **element-wise** addition, subtraction, multiplication and division of two vectors or tables. Element-wise means that the calculation is performed on the equivalent positions between the two variables: first element + first element, second element + second element...

You can select the labels of a vector or table using the `names()` function. For tables you can use `rownames()` and `colnames()` to either access or set the row and column names.

To sort data frames you can use the `order()` function:

```r
> mtcars[order(mtcars$mpg),]
```

You can sort on two columns (first on mpg, then on cyl):

```r
> mtcars[order(mtcars$mpg,mtcars$wt),]
```

You can sort in descending order by placing by a minus sign in front of the variable:

```r
> mtcars[order(mtcars$mpg, -mtcars$wt),]
```

Most functions operate on numbers but there are also functions for manipulating text, e.g. `paste(x,y,sep=" ")`

concatenates (glues them together into one string) character strings x and y separating them by the string given by sep. Arguments x and y can be character strings but they can also be vectors. If x and y are vectors, they are concatenated element-wise to give a character vector result.

More examples on operations on vectors, matrices or data frames can be found in the Operations on variables section of the R Reference Card.

9.2. Functions helpful for working with large data sets
Research in biology/medicine often generates very large data sets. When you work with very large data sets, it is often useful to show only a small part of the data set.
The function `head()` enables you to show the first 6 elements (vector) or rows (table) while the function `tail()` prints out the last 6 elements (vector) or rows (table).

**Exercise 11A:** View the first 6 rows of the mtcars data frame.
**Exercise 11B:** You repeat the plant study experiment this time with the following numbers of plants developing lesions: 1, 6, 6, 5, 4. Add this as a third column to the data frame.
**Exercise 11C:** Relabel columns in data frame Plant_study to “Day”, “Infected” and “Repeat”
**Exercise 11D:** Calculate for each day the total number of infected plants.

### 9.4. Checking and converting types of variables

To check the data type of an object you can check its structure using the `str()` or the `class()` function.

```r
> class(c(10,12,30))
[1] "numeric"
> class(c("alana","britt","chris"))
[1] "character"
```

You can also use the `is.X` functions e.g. `is.numeric()`, `is.character()`, `is.vector()`, `is.data.frame()`...

The `is.na()` function returns TRUE if x is a missing value and FALSE otherwise.

You can convert the data type of an object by using the `as.X` functions e.g. `as.numeric()`, `as.character()`, `as.vector()`, `as.data.frame()`

**Exercise 12A:** View the IDs of the patients from Drug_study with missing activity info?
**Exercise 12B:** Check the data type of the second column of Drug_study
**Exercise 12C:** Convert the second column of Drug_study into a vector. What is different now?
10. Reading and writing files
In most you have to import data in text format: both tab-delimited text (.txt) and comma-separated value files (.csv) can be opened.

In a script, the generic function for opening a file and creating a data frame from it is:

```r
> read.table(file,header=FALSE, sep = ",", dec=".", skip=0, comment.char= "#",strip.white=FALSE)
```

This function has a long list of arguments, the most important ones are:

- **file** is not simply the name of the file that holds the data but the full path on your computer that leads to the directory where the file is stored e.g. D:/trainingen/Prism/Hormone.csv. If it is stored in your working directory, you can simply use the name of the file. You can also use the function `file.choose()` to browse to the file and select it. Alternatively file can be replaced by a url to load data from the internet.
- **header** defines if the first line is to be read as a header. The default is `FALSE`.
- **dec** defines the decimal separator. The default is “.”
- **sep** defines the column separator. The default is a whitespace “”, which represents one or more blank spaces, tabs, newlines or enters.
- **skip** number of lines to skip before starting to read data. The default is 0.
- **comment.char** defines which lines should be interpreted as a comment and thus ignored during reading. Default is lines starting with “#”.
- **strip.white**: used only when **sep** was specified, allows stripping of leading and trailing white spaces from unquoted character fields (numeric fields are always stripped). Default is `FALSE`.
- **na.strings**: vector of strings to be interpreted as missing values. The default is “NA”.
- **nrows**: maximum number of rows to read in. The default is `-1` which means to read the complete file.
- **row.names**: number of the column that contains the row names. The default is `NULL`: rows are numbered.

See the help file for a full overview of all arguments. The output of all `read` functions is a data frame.

---

**Exercise 13A**: The babies.csv file consists of three columns of values, separated by semicolons. The comma is used as decimal separator. Import the file.

**Exercise 13B**: Check the babies data frame (remember it’s a large data set).

**Exercise 13C**: Import the file GeneEx.csv into a data frame called GeneEx.

**Exercise 13D**: Rename the two last columns Ct1 and Ct2.

**Exercise 13E**: Create a new column containing the mean Ct value per row.

---

Reversely, to write a data frame to a file you can use the generic function

```r
write.table(x,file="",quote=TRUE,sep=" ",row.names=TRUE,col.names=TRUE)
```

This function has a long list of arguments, the most important ones are:

- **x**: object to be written. If x is not a data frame, R attempts to transform x into a data frame.
- **file** is not simply the name of the file that holds the data but the full path on your computer that leads to the directory where the file is stored e.g. D:/trainingen/Prism/Hormone.csv
• **quote**: if TRUE, strings will be surrounded by double quotes. Row and column names are quoted if they are written. If FALSE, nothing is quoted.
• **sep**: defines the column separator
• **row.names**: either a logical value indicating whether the row names of x are to be written or a character vector of row names to be written.
• **col.names**: either a logical value indicating whether the column names of x are to be written or a character vector of column names to be written
• **append = FALSE**: if TRUE the output is added to the file that is defined by the file= argument
• **eol = “\n”**: defines the end-of-line character. The default “\n” represents an enter.
• **na=“NA”**: defines the string to use for missing values in the data
• **dec=“.”**: defines the decimal separator

See the help file for a full overview of all arguments.
11. Apply functions as an alternative for for-loops

If you have to calculate the individuals means of every row in a data frame you can use the rowMeans() function but for all other statistics such a rowStatistic() function is not available.

A good means of solving this problem are the apply() functions. They are an alternative for for- and while-loops. Generally, experts advise against the use of loops in R since it will slow down your analysis immensely.

The code to use the apply function is:

```r
> apply(D, margin, fun)
```

where

- **D** = data frame
- **margin** = 1:2
  - 1: apply the function on the rows of D
  - 2: apply the function on the columns of D
  - 1:2: apply the function on each element of D
- **fun** = function to apply

For instance to divide all elements of D by 2 you can use the following apply() function:

```r
> apply(D, 1:2, function(x) x/2)
```

**Exercise 14:** Calculate the rowMedians of the GeneEx data frame using apply().
12. Reshape2 for manipulating data sets

To calculate statistics in your data set, R assumes that all data of the data set are located in one single column. Sometimes, your data is not in this format e.g. a column might contain measurements from different groups. This is called long format. In the example below the third columns contains measurements for two groups of patients (group A and group B).

<table>
<thead>
<tr>
<th>Patient</th>
<th>Drug</th>
<th>Hormone conc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>58,6</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>57,1</td>
</tr>
<tr>
<td>3</td>
<td>B</td>
<td>40,6</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>10,5</td>
</tr>
<tr>
<td>5</td>
<td>A</td>
<td>60,3</td>
</tr>
</tbody>
</table>

If you want to calculate the mean Hormone conc. of each group, you cannot directly use mean() on the third column. This will compute the overall mean of the two groups combined. mean() expects the data in wide format, one column for each group.

Sometimes, R expects data in long format, e.g. some graphs need data in this format. To transform data from wide to long format you can use functions from the reshape2 package.

reshape2 is based around two key functions: melt and cast:
- melt(D) takes wide-format data and melts it into long-format data.
- cast(D) takes long-format data and casts it into wide-format data.


This is a file containing normalized expression levels of 10 genes obtained via RNASeq analysis. They are in wide format. We want to make a heat map of the data so we need it in long format.

Exercise 15B: Use a reshape2 function to get the data in long format
13. Creating graphics using the ggplot2 package

Ggplot2 is a package for producing graphics in R. It is based on the theory that a graph is composed of a set of independent components, each with its characteristics. Plots can be built as layers of these components (coordinate system, titles, legend, symbols, error bars...). A carefully chosen set of default settings for the characteristics of these components ensures that most of the time you can produce a publication-quality graphic in seconds, but if you do have special formatting requirements, the philosophy behind the structure of graphs in ggplot2 makes it easy to do what you want.

It is not a default package in R so you need to install it using the following command:

```r
> install.packages("ggplot2")
```

Working with ggplot is not that easy so it’s very useful if you know where to find help/explanations:

- Documentation
- R Graphics Cookbook
- R for dummies

The basic plot command in ggplot2 is as follows:

```r
> ggplot(data, aes(xdata,ydata, fill=second_factor)) +
```

- data: the data to plot, ggplot2 expects a data frame holding the data
- aes(xdata,ydata): defines the column to be mapped to the X-axis (factor or a quantitative variable) and the column that should be mapped to the Y-axis (always quantitative)
- fill=second_factor: defines the coloring scale, setting fill equal to a factor variable uses a discrete color scale with different colors for different levels/categories. Only used when you have two factors: one you group by (xdata) and one you color by (second_factor).

In the example above clarity is the first factor that defines the grouping on the X-axis and cut is the second factor that defines the coloring of the bars.

---

2. [http://docs.ggplot2.org/current/](http://docs.ggplot2.org/current/)
To actually create the plot, you have to add layers to this basic plot. The ‘+’ sign that follows the basic ggplot command is used to combine the different layers of the plot. Between each layer you place a ‘+’ sign in the ggplot command. These layers consist of:

- geoms: define overall look of the plot
- coords: change coordinate system (e.g. switch axes...)
- scales
- ...

**Geoms** tell the plot how you want to display your data. They represent data values and have aesthetic properties, e.g.

- color
- shape
- size
- x,y location

Geoms define the type of plot, e.g. geom_bar creates a bar chart. Geoms are therefore also linked to stats, statistical transformations of the data. Each type of plot has a default stat.

<table>
<thead>
<tr>
<th>Geom</th>
<th>Description</th>
<th>Default Stat</th>
</tr>
</thead>
<tbody>
<tr>
<td>geom_bar()</td>
<td>Bar chart</td>
<td>stat_bin()</td>
</tr>
<tr>
<td>geom_point()</td>
<td>Scatterplot</td>
<td>stat_identity()</td>
</tr>
<tr>
<td>geom_line()</td>
<td>Line diagram, connecting observations in order by x-value</td>
<td>stat_identity()</td>
</tr>
<tr>
<td>geom_boxplot</td>
<td>Box-and-whisker plot</td>
<td>stat_boxplot()</td>
</tr>
<tr>
<td>geom_path</td>
<td>Line diagram, connecting observations in original order</td>
<td>stat_identity()</td>
</tr>
<tr>
<td>geom_smooth</td>
<td>Add a smoothed conditioned mean</td>
<td>stat_smooth()</td>
</tr>
<tr>
<td>geom_histogram</td>
<td>An alias for geom_bar() and stat_bin()</td>
<td>stat_bin()</td>
</tr>
</tbody>
</table>

### 13.1. Histograms

To make a histogram you use the geom_histogram() function.

**Exercise 16A:** Create a histogram of birth weights of all babies.

For an overview of the aesthetics of geom_histogram() we refer to the slides. In the previous exercise you saw that R notifies you that the default number of bins (30) can be changed by setting the **bins** argument.

**Exercise 16B:** Create a histogram of birth weights of all babies. Set the number of bins to 20.

**Exercise 16C:** Create a histogram of birth weights of all babies. Color the borders of the bins black and color the bins blue.
13.2. Scatter plots
To make a scatter plot you use the `geom_point()` function.

Exercise 16E: Create a scatter plot of Hormone conc. (Y axis) for each drug treatment (X-axis).
Exercise 16F: Create the same scatter plot but now with red dots.
Exercise 16G: Create the same scatter plot but now color the dots according to drug treatment.
Exercise 16H: Create the same scatter plot but now use different shapes of dots for each treatment.

13.3. Scales for controlling the visual elements of the plot
Scales can be used to control the appearance of axes e.g. to set a title for an axis, to control the tick marks on an axis, to set a range for an axis. But you can also set scales for shapes (points) on the plot, e.g. `scale_shape(solid = TRUE)` produces solid shapes (this is the default).

And you can also use scales to change the appearance of the legend e.g. by using `scale_fill_manual` and setting its arguments:

- **name**: title of the legend
- **breaks**: used in combination with **values**, breaks defines the different categories that are defined in the legend
- **values**: colors of the categories, breaks lists the categories, values defines which color each category gets
- **labels**: names of the categories like you want them to appear in the legend

An overview can be found on the slides and the [http://www.cookbook-r.com/Graphs/](http://www.cookbook-r.com/Graphs/) website.

Exercise 17A: Create a scatter plot of mtcars with cyl on the X-axis and mpg on the Y-axis.
Color the dots according to the transmission type (am) the car has.
Change the label of the X-axis into “# of cylinders”.
Change the label of the Y-axis into “gas usage”.
Change the label of the legend into “transmission”.
Change the colors of the dots: red for cars with transmission 0, black for cars with transmission 1.
Change the labels of the categories: 0 is “automatic”, 1 is “manual”

Exercise 17B: Create a scatter plot of mtcars with disp on the X-axis and mpg on the Y-axis.
Use a logarithmic (log10) X-axis. Set the following tick marks on the X-axis: 100, 200, 300

13.4. Themes

A theme is simply a structure containing a list of options. These options describe the visual properties of the axes, legends, panels, strips, and the overall plot, like the color of the background, the absence/presence of a legend. Note that these options define everything that is not linked to data in the plot, so not the layers. If you create a layer to draw red dots and blue lines, the theme will not change these settings.

Ggplot2 provides several built-in themes (see slides), for instance:

- `theme_grey()` - the default theme, with a grey background
- `theme_bw()` - a theme with a white background
Additionally, ggplot2 allows you to create a theme yourself. There is an enormous amount of properties that you can set, e.g.

- `axis.text.x` appearance of labels on the x axis: colour, size, line height...
- `axis.text.y`
- `axis.ticks` appearance of ticks: color, size
- `axis.title.x` appearance of title of the x axis: colour, size
- `axis.title.y`
- `legend.background` background of the legend: colour
- `legend.text` appearance of the labels in the legend: colour, size
- `legend.title` appearance of the title of the legend: colour, size, bold or not...
- `legend.position` position of the legend: can be used to hide legend
- `panel.background` background of the plot: colour, colour of border
- ...

When you set one of these properties, you set them for all plots on a figure (when using faceting).

To set these properties you can use a defined set of elements:

- `element_blank()` nothing (used to hide components)
- `element_text()` text - arguments: colour, size, lineheight, vjust, face, hjust, angle
- `element_line()` lines e.g. tick marks – arguments: colour, size
- `element_rect()` rectangles e.g. panel of the plot – arguments: fill, colour

For example

```r
ggplot(mtcars, aes(x = factor(gear), y = mpg)) + geom_point() + theme(panel.background = element_rect(colour="black",fill ="white"))
```

creates a plot with a white rectangle with black borders that is used as a background panel.
**Exercise 18A:** Create the same plot with grid lines.
Tip: Use one of the built-in themes (see slides).

**Exercise 18B:** Create the same plot with “# gears” as label of the X-axis.
Set the size of the labels of the axes to 6.