A Beginners Guide to R's Galaxy

Michal J. Czyz

2018-02-07

Contents

Chapter 1

In the beginning there was only darknes...

R (R Core Team, 2017) is one of the most common used languages in Data Science. It is so called fourth-generation programming language (4GL), meaning it is *user-friendly*, while still quite powerful. **R** is powered by huge open-source oriented community. Thanks to their work, during many years of development, enormous number of *packages* (also called *libraries*) were established, making using **R** for common works related to Data Science easy even for Beginners.

The purpose of this document is to familiarize with **R** people who have at least some basics in statistics or modelling and no knowledge on programming. Thus examples you will find in this book are driven by making life easier for all of those who struggle with date in their work.

To give you an example and how awesome and powerful **R** is, I wrote whole this book in **R** using package *bookdown* (Xie, 2016, 2017). Hoping this short description encouraged you to dive into **World of R**, we can start learning opportunities of this programming language.

Chapter 2

Introduction

2.1 RStudio

Before we jump into coding, you should first get familliar with **RStudio** (RStudio Team, 2016). It is so called *Integrated Development Environment* (IDE), which has built-in fuctionalities to make work easier. This IDE is typically used with 4 different windows:

- *Source* - where you can write scripts;
- *Console* - where scripts are executed;
- *'Environmental'* - it's adjustable window, usually containing *Environemnt*, *History* and Verison Control panes;
- *'Files'* - also adjustable, usually you will find here *File*, *Packages*, *Help* and *Plots* panes.

2.2 Few tips to make life easier

From menu choose *Tools* > *Global options*. Now choose *Code* and *Editing* pane, tick box *Insert spaces for Tab* and assure that Tab width is set to 2. Next, in *Display* pane, check following tickboxes:

- *Highlight selected word*
- *Show line number*
- *Show margin* (and set margin column to 80)
- *Show whitespace characters*
- *Highlite R function calls*

Generally speaking those options, do not influence how your code is performed, but will allow you to write cleaner and read easier. You can also change colors of your environment in *Appearance*.

2.3 Installing packages

In your *'Files'* winodow, you will find *Packages* pane, which contains *Install* button. You can use it now, to install packages needed to perform excercies from this book. The packages are:

- **devtools** (Wickham and Chang, 2017)
- **tidyverse** (Wickham et al., 2017a)
- **fitdistrplus** (Delignette-Muller et al., 2017)
- **e1071** (Meyer et al., 2017)
- **truncdist** (Novomestky and Nadarajah, 2016)

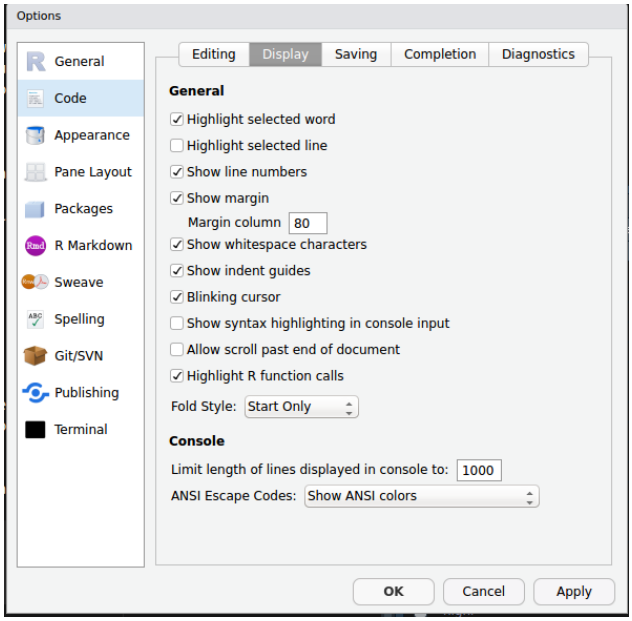

Figure 2.1: Code display options

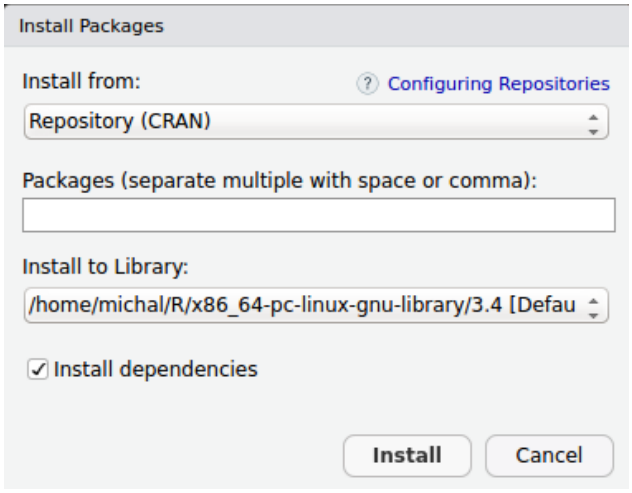

Figure 2.2: Installation window

Now everytime you need finctions from specyfic library you can just tick box next to package name, and RStudio will load it for you.

2.4 Conventions

In this book, we will use following conventions:

- Names of programs and packages are in **Bold**.
- All other names e.g. names of panes menu items as well as things that needs to be stressed are in *italics*.
- Function names and variables are always writen in inline code e.g. `t.test()` or `x`.
- File names are wirten in inline code e.g. `foo.txt`.
- Citations are in APA style, and 'clickable' e.g.click on the name and year of **knitr** package citation (Xie, 2015).
- Code chunks are in blocks and result lines start with ##

```
rnorm(10, 1, 0.5)
```

```
##  [1] 0.2569541 2.0800798 0.7579734 0.9364796 1.2213334 1.3123241 1.1047721
##  [8] 1.1495709 0.9991266 2.1900720
```

- There are no > (*prompt*) signs in code chunks.
- Figures are floating - meaning, that they are not always imediately after they are mentioned in text.
- Tables are in *longtable* format (meaning they are not floating and might be multipage) e.g.

```
knitr::kable(
  head(iris, 25), caption = 'Example table',
  booktabs = TRUE, longtable = TRUE
)
```

Table 2.1: Example table

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa
4.6	3.4	1.4	0.3	setosa
5.0	3.4	1.5	0.2	setosa
4.4	2.9	1.4	0.2	setosa
4.9	3.1	1.5	0.1	setosa
5.4	3.7	1.5	0.2	setosa
4.8	3.4	1.6	0.2	setosa
4.8	3.0	1.4	0.1	setosa
4.3	3.0	1.1	0.1	setosa
5.8	4.0	1.2	0.2	setosa
5.7	4.4	1.5	0.4	setosa
5.4	3.9	1.3	0.4	setosa
5.1	3.5	1.4	0.3	setosa
5.7	3.8	1.7	0.3	setosa
5.1	3.8	1.5	0.3	setosa

5.4	3.4	1.7	0.2	setosa
5.1	3.7	1.5	0.4	setosa
4.6	3.6	1.0	0.2	setosa
5.1	3.3	1.7	0.5	setosa
4.8	3.4	1.9	0.2	setosa

- Tabels and figures are references are clickable e.g. see Table 2.1 or see Figure 2.1.

Chapter 3

Basics

3.1 Getting started

3.1.1 Help

There are just few things you really need to remamber and follow when you want to start using **R**. First, there is very good *help* build in. To access it, you use `?` sign with name of function: eg: `?t.test`. After executing command, in your window with *Help* pane, a page dedicated to this function will pop up. You will get information on syntax, options to use with this function and in most caseses some code examples. However, with single question mark you are telling **R** to only look into functions from packages that are *currently loaded* and have this *precise name*. If you want to tell **R** to look for proper function in *all packages* or you are not sure what the exact name is you can use double question mark e.g. `??mutate`. In effect, in the same pane and window as previously you will get a list of results that match your query. Finally, using *Packages* pane, you can click on one of the packages names, to display all of the functions within it. Than by clicking on the name of functions you are interested in, you will be taken to proper page with description.

3.1.2 Internet is a great source of information

Anytime you feel lost or need help that is beyond the scope of manuals, just ask Google. For instance you can use this query: *how to make density plot in R*. Thanks to huge community you will find a lot of answers. The most reliable ones can be found on *StackOverflow*, *StatsExchange* and *RBloggers*. If you don't know if there is a library to perform particular task also ask uncle Google. For instace, if you want to use random numbers from Dirichlet distribution, you can use this query: *dirichlet distribution r*.

3.1.3 More on internet sources

A good practice, when you want to learn programming language is to read what other people do and how the code. In the begginig it might be a bit overwhealming or confusing to read all the stuff. However, reading others work will get you used to syntax and workflow, and will qive you great basics to invent your own code. Hopefuly, you don't need to spend hours for searching some interesting blogs. There is great blog agregator R weekly that gathers in one place best posts, podcasts, etc. on **R**, every week.

3.2 Syntax

3.2.1 Common operators

There are three main *signs* used in **R's** syntax. First two are assigment symbols: `<-` and to `=`; for convention we use them in different cases. Third one is `#`. It is a symbol used for comments. Everything following this symbol to the end of code line will not be executed. There are also other signs (or symbols) which are building blocks of language, however their use is very precisely defined and reserved for certain events. Below you find a table with reference for most common operators used. You will faster grasp it while you write your own code, than by reading about it. Thus, I suggest we go deeper into variable types in **R** language.

Table 3.1: Common operators in R

sign	type	action
+	maths	addition
-	maths	substraction
*	maths	multiplication
/	maths	division
%%	maths	modulo
^	maths	power
>	relations	left greater
>=	relations	left greater of equal
<	relations	right greater
<=	relations	right greater or equal
==	relations	left equal right
!=	relations	left uneqal right
!	logics	not
&	logics	and
|	logics	or
~	model	left relates to right
<- or ->	assignment	assignes value to variable
$	address	extracts values with 'element name' from variable
:	sequence	creates sequence of numbers from 'left value' to 'right value'
%>% or %<>%	piping	pipes results(from left) as arguments to functionon right

3.2.2 Variables

Concept of variable is crucial for programming. In **R** variables can contain many things: vectors, data frames, results of statistical analyses etc. Each of variables have some characteristic properties. They are defined by *class* of the variable. Thanks to *class* atribute, **R** knows, how to deal with variable – what is the internal structure and what operations can be performed over variable. Data can be stored in variables in different manners. To assign something to variable we use `<-` operator, which tells **R** to store right side of arrow under name on the left side of arrow. The simplest variable is *vector*, which can be of *class*: *character*, *integer*, *numeric* or *logical*. For instance:

```
characterVector <- c('a', 'b', 'c')
class(characterVector)
```

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

```
integerVector <- c(1L, 2L, 3L)
class(integerVector)
```

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

```
numericVector <- c(2.5, 3.5, 4.5)
class(numericVector)
```

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

```
logicalVector <- c(TRUE, FALSE)
class(logicalVector)
```

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

For more complex data, we have three basic classes: *lists*, *data frames* and *matrices*. *Matrix* is simillar to *data frame*. The most obvious difference is that *matrix* contains only one *class* of variables (usually *numeric* or *integer*), while *data frame* can store *numeric*, as well as *characters* and *factors* (for now, you can assume that *factor* class is used to store catagorical variables) in seprate columns. Also *matrices* are used when programmers want to achive great speed in mathematical computation. *Data frames* are resembling tables from popular spreadsheet software. Lets look:

```
matrixVariable <- matrix(c(1:10), nrow = 2)
matrixVariable
```

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

```
class(matrixVariable)
```

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

```
dfVariable <- data.frame(x1 = 1:5, x2 = 6:10)
dfVariable
```

```
##   x1 x2
## 1  1  6
## 2  2  7
## 3  3  8
## 4  4  9
## 5  5 10
```

```
class(dfVariable)
```

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

Lists are... lists of variables. Each *list* element can be of different *class* and length. To grasp the idea of *lists* it will be best to present it with example:

```
listVariable <- list(x1 = c("a", "b"), x2 = 1:4, x3 = matrix(c(1:6), nrow = 2))
listVariable
```

```
## $x1
## [1] "a" "b"
##
## $x2
## [1] 1 2 3 4
##
## $x3
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
class(listVariable)
```

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

```
class(listVariable$x1)
```

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

```
class(listVariable$x2)
```

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

```
class(listVariable$x3)
```

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

There are plenty of other *classes*, e.g. for *time variables*, however mentioned above are the basic ones you will deal mostly. Also because they are so often used, you should learn how to recognize their structure at a glance. Later on, I will present you how (and when) each of this variables types can be used in work.

3.2.3 Naming Variables

First of all all names are *case-sensitive*, which means that **R** recognize variables named `RVariable`, `rVariable` and `Rvariable` as three different objects. Second thing to remamber is that variable name *have to* start with a letter and may contain only letters, numbers and symbols: . (dot) and _ (underscore). There are also some *good practices* in naming variables (after Hadley Wickham Style guide):

- use lowercase to names variables (and functions)
- use nouns to name variables (and verbs for functions)
- try to be precise when naming
- try to be concise when naming
- use underscore _ to separate words (snake_case) e.g. `first_variable`
- *some other guidelines suggest using camel cases e.g.* `firstVariable`

And the golden rule should be - whatever guideline you follow – be consequent!

3.2.4 Math operations

In **R** we use standard math oprators `+ - * /` to perform addition, substraction, multiplication and division. Symbol `^` indicates that we want to use power, and sqrt to make square root. Ok, so whats the name of a function to get n^{th} root? Probably you remamber from math lessons that $\sqrt[n]{x} = x^{\frac{1}{n}}$, thus you can just write `x^(1/n)`. To change order of operation (which are following mathematics rules) use brackets `()`. Other important mathematical functions are `%%` for modulo, and %/% for integer division.

```
5 + 2
```

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

```
11 - 3
```

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

```
(4+7)/9*2
```

```
## [1] 2.444444
```

```
14 %/% 3 + 1
```

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

	NA	FALSE	TRUE
NA	NA	FALSE	NA
FALSE	FALSE	FALSE	FALSE
TRUE	NA	FALSE	TRUE

	NA	FALSE	TRUE
NA	NA	NA	TRUE
FALSE	NA	FALSE	TRUE
TRUE	TRUE	TRUE	TRUE

```
8^(1/3) + 10%%6
```

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

To calculate logarithms there is *build in* function `log()`. It uses as a base Eulers number by default, however you can override it i.e. `log(10, base = 10)`. You can calculate exponential function using `exp()` function. There are also trigonometric functions in **R**: `sin()`, `cos()`, `tan()`, `asin()`, `acos()`, `atan()`. Angles are used/expressed in radians. To transform values from degrees to radians multiply by `pi` and divide by 180. To transform values from radians to degrees multiply by 180 and divide by `pi`. By the way, `pi` is a constant in **R**, meaning that its value is build in the language (simmliar as Euler number is `exp(1)`).

```
someArc <- 90*pi/180
sin(someArc)
```

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

```
atanValue <- atan(0.89)
atanValue*180/pi
```

```
## [1] 41.66908
```

3.2.5 Logics

Logical expression are often used in programming. They compare left side with right side arguments of statement. The result of those comparrision might be `TRUE` or `FALSE` (in many other languages those are called *Boolean* values) which belong to *class Logical*. In Table 3.1 you will find list of most common logical operarators used to build statements. Here is a small *cheatsheet tables*:

Below you can see them in action:

```
5 >= 1
```

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

```
10%%2 == 0
```

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

```
!FALSE
```

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

```
5L | 11.1 <= 6
```

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

3.2.6 Functions

When writing code we generally want to perform some actions on our variables. There is a lot of *build in functions* in base **R** distribution, and a whole Galaxy of *functions* provided by community. Function can be literally any action performed on variables. For instance, there are some build in statistical functions like `t.test()` or `chisq.test()`. Other function can be use to draw some charts and plots, e.g. `plot()`. It's easy to recognize function, since its structere is *name* follwoed by parentheses *()*. Inside parentheses user provides arguments and options to function. Lests see how it works with one of *build in* functions:

```
tTestResult <- t.test(numericVector, integerVector)
print(tTestResult)
```

```
##
##  Welch Two Sample t-test
##
## data:  numericVector and integerVector
## t = 1.8371, df = 4, p-value = 0.1401
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.7669579  3.7669579
## sample estimates:
## mean of x mean of y
##       3.5       2.0
```

We used `t.test()` function, with two arguments: `numericVector` and `integerVector`. In second function call, we *ordered* **R** to print out the results of statistical test stored in variable `tTestResult` – which is the argument of this function.

Chapter 4

Somwehere between basic and useful

4.1 Addressing

4.1.1 Vectors

When you deal with variables you often will want to use only a part of it in your work. Other times you will want to get rid of some values which follow certain criteria. In order to do it you need to know an 'addrress' of particular value in a *vector*, *data frame*, *list* etc. From now on, I will use some functions while describing it *at hoc*, since I believe the best way to learn them, is to use them. Lets creat a *vector* and see what we can do with it.

```
addressVec <- seq(from = 1, to = 20, by = 2)
addressVec
```

```
##  [1]  1  3  5  7  9 11 13 15 17 19
```

Instead of typing all the numbers by hand, we can use `seq()` function, to generate it automatically. This function takes two arguments `from` and `to`, however additionaly we can use option `by` which defines increment of the sequence. As your knowledge is growing I can tell you a secret. Often there is no need to name all the arguments and options in functions body. In the beginning you should use the names, but the more experienced you get you will notice that you omit them often. Actually we nearly always omit so called *default* options of a function. Use `?seq` to check the help page for this function. You will see that it can take more options than `from`, `to` and `by`, but since they are predifned we don't neet to bother and type them as long as we are OK with *default* settings. So, lets retype our *variable* definition:

```
addressVec <- seq(1, 20, 2)
addressVec
```

```
##  [1]  1  3  5  7  9 11 13 15 17 19
```

The results are identical. Lets get back to addressing issues. Suppose you want to check the fifth element of a our `addressVec` variable. To do it we use variable name followed by element number in brackets (or square brakcets, if you will).

```
addressVec[5]
```

```
## [1] 9
```

If you want to get rid of fifth element just preced it with -.

```
addressVec[-5]
```

```
## [1]  1  3  5  7 11 13 15 17 19
```

Easy. The thing that is worth to mention right now is that **R Core Team** have reason and dignity of human beings so they start numeration of elements with value 1. In some other languages, because of no sensible reasons numeration starts with 0 – which is horribly annoing. What to do if we want to extract values of more than one element?

```
addressVec[1,5]
```

```
## Error in addressVec[1, 5]: incorrect number of dimensions
```

```
addressVec[-1,-5,-6]
```

```
## Error in addressVec[-1, -5, -6]: incorrect number of dimensions
```

Error occurs with warning that there is incorrect number of dimensions. That's because *vectors* have only one dimension – length. Within square brackets comma separates dimensions. So when we used command `[1,5]` we told R to look for value that address is number 1 in first dimension and number 5 in second dimension. To correct the mistake, we need to provide a vector of numbers from first dimension. We can use `c()` function to create *ad hoc vector* inside square brackets.

```
addressVec[c(1,5)]
```

```
## [1] 1 9
```

```
addressVec[-c(1,5,6)]
```

```
## [1]  3  5  7 13 15 17 19
```

Now something more complicated. Try to figure out what code below does, and what is the result of `which()` function:

```
addressVec[which(addressVec >= 6)] <- 'o..O'
addressVec
```

```
##  [1] "1"    "3"    "5"    "o..O" "o..O" "o..O" "o..O" "o..O" "o..O" "o..O"
```

If you have problems, you can think of this example as a composition of actions. First there is expression `addressVec >=6`, than we use `which()` function with argument from previous expression. Next the result is passed as an address. Last thing is assigment of new value to…

4.1.2 Data frames (and matrices)

When dealing with *data frames* and addressing is possible in two ways. First is simmilar to *vector* addressing. However, as *data frames* have two dimentions, you will need to express them like `df[1,2]` – which tells **R** to look up the cell in row 1, column 2. But what if you want to check all the cells in particular row or column? Omit the number, but not the coma – like this: `df[,3]`. It will display all row values from column 3. And of course you can still use *vectors* when addressing. This way is also working for *matrices*.

```
addressDF <- data.frame(C1 = seq(1,10), C2 = seq(11,20), C3 = seq(21,30))
addressDF[5,2]
```

```
## [1] 15
```

```
addressDF[6,]
```

```
##   C1 C2 C3
## 6  6 16 26
```

```
addressDF[c(6,7), 1:3]
```

```
##   C1 C2 C3
## 6  6 16 26
```

```
## 7  7 17 27
```

The second option is tu use `$` sign followed with column name (and it work only for *data frames* and *lists*). This way, we tell **R** to look in whole column. If we want to display values from particular cells, we put them in brackets after column name, the same way as we do for *vectors*:

```
addressDF$C2[5]
```

```
## [1] 15
```

```
addressDF$C3[c(6:9)]
```

```
## [1] 26 27 28 29
```

4.1.3 Lists

As mentioned before each *list* element can have different structure. Thus, addressing is kind of a mixture of all above. First you need to address element of your *list*. You do it with double brakcets containing element number following name of variable. You can also use `$` sign with name of the element. Than you use addressing the same way you address *data frames*, *vectors* or *matrices*.

```
addressList <- list(x1 = c('a', 'b'), x2 = 1:4, x3 = matrix(c(1:6), nrow = 2))
addressList[[2]]
```

```
## [1] 1 2 3 4
```

```
addressList$x2[3]
```

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

```
addressList[[3]]
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
addressList[[3]][,3]
```

```
## [1] 5 6
```

Ok, now simple task for you. There is a function called `colnames()` which allow us to change names in *matrix* variable. To use it you need to put name of *matrix* variable in the parentheses (like this: `colnames(x)`), and assign value to it (e.g. `colnames(x) <- c('one', 'two')). Now change names of of columns in *matrix* that is element of list above to`I‘, ‘Love’, ‘R’.

4.2 Operation on Vectors

Old proverb:

> *The power of R is vectorization*

Indeed, vectorization is one of the most usefull features of **R** environment. In short words, many functions are contructued in such manner that one can avoid using loops (since, they are often very slow). When you begin your journey with programming you will probably not notice the difference in computation speed between *vectorized way* and *loop way*. Nonetheless, what will you find attractive, is that in many cases writting in vectorized form feels more natural. So how it works? Most basic functions are *written and compiled* in very fast, low level languages like **C** or **FORTRAN**. It makes computation way faster, but still we use easy **R** syntax. Instead of running the *precompiled* function on each of the elements of vector we actually can pass the vector into function – and **R** will know what to do. The speed is achieved beacause when we use function

on each element, **R** needs to figure out what to do several (or even hundred or thousands of) times, however when we pass whole vector into function, it needs to find out what to do only once. Sometimes, however we need to use some other functions, wich are not meant to process vectors (or you want to use some arbitrary functions on matrix, data frame or list). You will need to parse your data more than once to function. In **R** you can do it by using loops or by using so called `apply` family functions. When I was learning **R** it was most confusing thing to me, however once you get it, they become fairly easy to use. The most important difference for you as a beginer is that when using loops you should make some memory allocation – in human language, before you run loop, you should create vector which will store results. The loops achieve highest speed when your vector can fit all the values from loop. If you do not know how big your vector will be, you need to relay on **R** ability to re-allocate the memory, which is **S L O W**. `apply` family hopefully takes care of this problem, so everytime you know how to – use it. It will save you time and frustration of using loops. I will cover basic use of this functions later on examples.

4.3 Randomization and distribution

Random sampling is quite easy task. There is nice `sample()` function, that allow you to do that. You can sample from *numeric*, *integer* or even *character vectors*. You can even assign probabilities to particular elements of vector (we will not need that at the moment). So lets make some fun and make virtual casino. We will take two dice and will roll it 1000 times. If the sum of each result is greater odd and smaller than `5` we will earn 30EUR. If it is odd and higher than `5`, we will get 50EUR. However if the sum is even number we loose 70EUR. Lets see if we can beat casino… First lets define our dice and rolls.

```
niceDice <- seq(1,6)
rollDiceOne <- sample(niceDice, 1000, replace = T)
rollDiceTwo <- sample(niceDice, 1000, replace = T)
```

Ok, you heard about power of vectorization, already so lets sum up both vectors:

```
sumDiceRolls <- rollDiceOne + rollDiceTwo
```

Next lets make some use of knowledge we have already and subsitute our results with some cash…

```
cashDiceRolls <- sumDiceRolls
cashDiceRolls[which(cashDiceRolls %% 2 == 1 & cashDiceRolls <= 5)] <- 30
cashDiceRolls[which(cashDiceRolls %% 2 == 1 & cashDiceRolls > 5)] <- 50
cashDiceRolls[which(cashDiceRolls %% 2 == 0)] <- -70
sum(cashDiceRolls)
```

```
## [1] -70000
```

Ok. What happened? We put our substitutoons in very wrong order… After first two substitutions we have only even numbers in our vector… So how to fix it? Start with assigning the highghest absolute value, and after all change it to negative one.

```
cashDiceRolls <- sumDiceRolls
cashDiceRolls[which(cashDiceRolls %% 2 == 0)] <- 70
cashDiceRolls[which(cashDiceRolls %% 2 == 1 & cashDiceRolls > 5)] <- 50
cashDiceRolls[which(cashDiceRolls %% 2 == 1 & cashDiceRolls <= 5)] <- 30
cashDiceRolls[which(cashDiceRolls  == 70)] <- -70
sum(cashDiceRolls)
```

```
## [1] -13260
```

Now, thats not very nice… We lost a lot of cash… But we can also track how are luck changed. Maybe we were above `0` for a while, before things got South? Or maybe we were doomed since the begining? Lets make a plot (Fig. 4.2) to evaluate our progress.

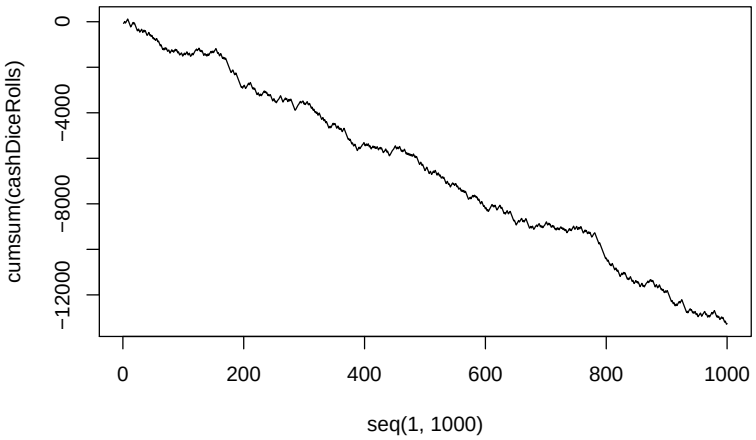

Figure 4.1: How to lose cash in casino

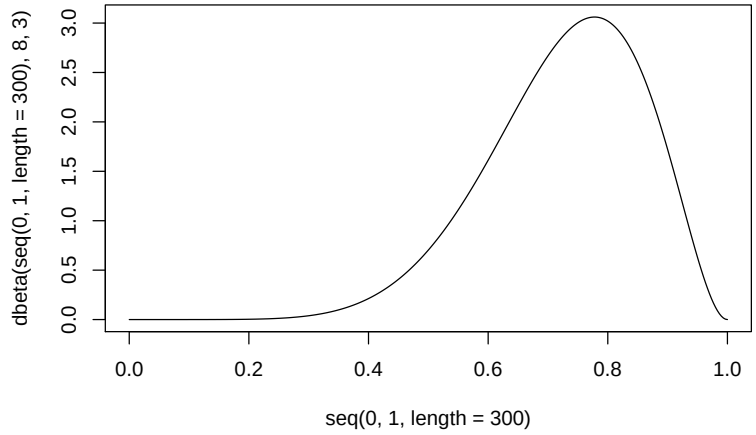

Figure 4.2: Example of beta distribution

I think that your love to **R** is being deeper since I showed you how it can save a lot of your money!

In nowadays stochastic analyses are more, and more popular. With **R** it is quite easy to generate random numbers, sample and do all the *mumbo jumbo* on data. There are few functions that cover *classical* distributions: normal, Poisson, binomial, uniform (and few others), that are actually build in base **R** distribution (full list you will find here). Some more suffisticated stuff is usually covered by some packages you will need to download by yourself. You will easily find them by querying Google like this Pearson distribution in R. By using this technique it is also possible to find other useful libraries for dealing with distributions (try to search for library that alows you to sample from trimmed distribution). All the distribution libraries follow the same schema when naming functions. They use letters: `d`, `p`, `q` and `r` followed by abbreviation of distibution name to generate: Density, distribution function, quantile function and random numbers – respectively. For instance, lets generate ten random numbers from *beta distribution*, whith parameters `10` and `3`:

```
rbeta(25,10,3)
```

```
##  [1] 0.8985423 0.9229588 0.9839358 0.8602257 0.8785151 0.8492233 0.7787028
##  [8] 0.7941326 0.7176832 0.8337257 0.5683864 0.5669949 0.7613561 0.6484569
## [15] 0.8416932 0.8059819 0.8935295 0.6764472 0.8259531 0.7707533 0.8651882
## [22] 0.7652365 0.7049847 0.9383397 0.8686104
```

Or we can make this nice plot (Fig. 4.2) for density of *beta distribution* with parameters `8` and `3`:

4.4 `tidyverse` idea and `dplyr` library

The whole idea of tidy data comes from one of most famous **R** developer – Hadley Wickham. In on of his papers (Wickham, 2014) he described procedures for generating and cleaning data in standardized manner. Many packages right now are designed to work best with data structured according to this publication. Eventualy it lead to `tidiverse` – tools tailored for data science with common syntax and philosophy(Wickham, 2017). One of the most useful packages that are inlcuded in `tidiverse` (Wickham, 2017) are `tidyr` (Wickham and Henry, 2017) and `dplyr` (Wickham et al., 2017b). First heplps us to swap from 'wide' to 'long' table format (and back). The second package contains set of tools to easily manipulate rows, columns or even single cells in *data frame*. It is extremly powerful tool, which speeds up work with datasets so much that after few times dealing with it, you will leave traditional spread sheet forever.

4.4.1 Wide vs. long tables

Lets start with changing wide format table into long format table.

```
wideTable <- data.frame(male = 1:10, female = 3:12)
wideTable
```

```
##    male female
## 1     1      3
## 2     2      4
## 3     3      5
## 4     4      6
## 5     5      7
## 6     6      8
## 7     7      9
## 8     8     10
## 9     9     11
## 10   10     12
```

Than we just make a little *mumbo jumbo* and change it to long format:

```
library('tidyr')
library('magrittr')
```

```
##
## Attaching package: 'magrittr'
```

```
## The following object is masked from 'package:tidyr':
##
##     extract
```

```
library('dplyr')
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
longTable <- gather(wideTable, key = sex,value = number)
longTable
```

```
##       sex number
## 1    male      1
## 2    male      2
## 3    male      3
## 4    male      4
## 5    male      5
## 6    male      6
## 7    male      7
## 8    male      8
## 9    male      9
## 10   male     10
## 11 female      3
## 12 female      4
## 13 female      5
## 14 female      6
## 15 female      7
## 16 female      8
## 17 female      9
## 18 female     10
## 19 female     11
## 20 female     12
```

Thats how easy it But what if table is more complicated?

```
wideTable2 <- data.frame(male = 1:10,
                         female = 3:12,
                         type = rep(c('bacteria','virus'), times = 5),
                         group = rep(c('a','b'), each = 5))
wideTable2
```

```
##    male female     type group
## 1     1      3 bacteria     a
## 2     2      4    virus     a
## 3     3      5 bacteria     a
## 4     4      6    virus     a
## 5     5      7 bacteria     a
## 6     6      8    virus     b
## 7     7      9 bacteria     b
## 8     8     10    virus     b
## 9     9     11 bacteria     b
## 10   10     12    virus     b
```

```
gather(wideTable2, sex, number, -c(3:4))
```

```
##        type group    sex number
## 1  bacteria     a   male      1
## 2     virus     a   male      2
## 3  bacteria     a   male      3
## 4     virus     a   male      4
## 5  bacteria     a   male      5
## 6     virus     b   male      6
## 7  bacteria     b   male      7
## 8     virus     b   male      8
```

```
## 9  bacteria     b   male      9
## 10    virus     b   male     10
## 11 bacteria     a female      3
## 12    virus     a female      4
## 13 bacteria     a female      5
## 14    virus     a female      6
## 15 bacteria     a female      7
## 16    virus     b female      8
## 17 bacteria     b female      9
## 18    virus     b female     10
## 19 bacteria     b female     11
## 20    virus     b female     12
```

In example above, using expression `-(3,4)` we indicated that we do not want to gather columns 3 and 4.

Even more complicated?

```
wideTable3 <- data.frame(male = rep(1:10, each = 2),
                         female = rep(3:12, times = 2),
                         type = rep(c('bacteria','virus'), times = 10),
                         group = rep(c('a','b'), each = 10),
                         day = 1:10)
wideTable3
```

```
##    male female     type group day
## 1     1      3 bacteria     a   1
## 2     1      4    virus     a   2
## 3     2      5 bacteria     a   3
## 4     2      6    virus     a   4
## 5     3      7 bacteria     a   5
## 6     3      8    virus     a   6
## 7     4      9 bacteria     a   7
## 8     4     10    virus     a   8
## 9     5     11 bacteria     a   9
## 10    5     12    virus     a  10
## 11    6      3 bacteria     b   1
## 12    6      4    virus     b   2
## 13    7      5 bacteria     b   3
## 14    7      6    virus     b   4
## 15    8      7 bacteria     b   5
## 16    8      8    virus     b   6
## 17    9      9 bacteria     b   7
## 18    9     10    virus     b   8
## 19   10     11 bacteria     b   9
## 20   10     12    virus     b  10
```

```
gather(wideTable3, sex, number, 1:2) %>% spread(group, number)
```

```
##        type day    sex  a  b
## 1  bacteria   1 female  3  3
## 2  bacteria   1   male  1  6
## 3  bacteria   3 female  5  5
## 4  bacteria   3   male  2  7
## 5  bacteria   5 female  7  7
## 6  bacteria   5   male  3  8
## 7  bacteria   7 female  9  9
## 8  bacteria   7   male  4  9
```

```
## 9  bacteria   9 female 11 11
## 10 bacteria   9   male  5 10
## 11    virus   2 female  4  4
## 12    virus   2   male  1  6
## 13    virus   4 female  6  6
## 14    virus   4   male  2  7
## 15    virus   6 female  8  8
## 16    virus   6   male  3  8
## 17    virus   8 female 10 10
## 18    virus   8   male  4  9
## 19    virus  10 female 12 12
## 20    virus  10   male  5 10
```

Here our data set had additional column `group` storing values a and b. But imagine that `group` is not a variable, but it just stores the names of variables - which are a and b. This somehow might be frustrating, to decide if those are really separate variables or not. You might run into problem, that your column named `environmental factor` contains values: *pH*, *conductivity*, and *oxygen concentration*. This would be straightforward as each of this is different variable and you should spread this column into three different variables. On the other hand you might see a column that contains *minimum temperature* and *maximum temperature*. This would not be as straightforward and decision upon spreading this column would strongly depend on the context. Nonetheless, using `spread()` function we were able to transform values from this column as separate columns. We also used *piping operator* `%>%`. It is a shortcut which allows us to pass the left side as an agrument to the function on the right side of operator. In general it means that writing `a %>% funtion(b)` actually is translated into `function(a,b)`. In the begining this idea might be not very usefull for you, but actually it vary helpful, mainly because your code gets better structure and you can perform multiple operations without storing it in variables which you do not want.

There is also one more common case that I should shortly mention - compound variable. It is a variable that stores multiple values in a single column. E.g. city and district, age and sex, sex and smoking, blood type and RH, etc. With `tidyr` is very easy to deal with it.

```
wideTable4 <- data.frame(type = rep(c('bacteria.a','virus.a'), times = 5))
wideTable4
```

```
##          type
## 1  bacteria.a
## 2     virus.a
## 3  bacteria.a
## 4     virus.a
## 5  bacteria.a
## 6     virus.a
## 7  bacteria.a
## 8     virus.a
## 9  bacteria.a
## 10    virus.a
```

```
wideTable4 %>% separate(type, c('organism', 'type'), sep = '\\.')
```

```
##    organism type
## 1  bacteria    a
## 2     virus    a
## 3  bacteria    a
## 4     virus    a
## 5  bacteria    a
## 6     virus    a
## 7  bacteria    a
## 8     virus    a
```

```
## 9  bacteria    a
## 10    virus    a
```

4.4.2 World of `dplyr`

`dplyr` is a library containing several useful functions, designed to ease all kinds of transformations, selections and filtering of your data frame. As the number and possibilities of functions are really huge, here I will concentrate only on some mostly used ones. Of course there is a well described documentation if you want to get deeper.

4.4.2.1 `select` columns and `filter` rows

Those are the most basic operations on data frames. `select()` function allows you to choose which columns you use. The biggest advantege of using this function instead of simple addressing, is that you can use some special functions inside it: `starts_with()`, `ends_with()`, `contains()`, `matches()`, `num_range()` which are helpful when using big data sets with numerous columns. There is also similar function `rename()` which can be used to change variable names, but in results (contrary to `select()`) it keeps all the variables. Lets look how the thing works on some simple examples:

```
head(select(wideTable3, starts_with('gr')), 3)
```

```
##   group
## 1     a
## 2     a
## 3     a
```

```
head(select(wideTable3, -starts_with('gr')), 3)
```

```
##   male female     type day
## 1    1      3 bacteria   1
## 2    1      4    virus   2
## 3    2      5 bacteria   3
```

```
head(select(wideTable3, contains('ale')), 3)
```

```
##   male female
## 1    1      3
## 2    1      4
## 3    2      5
```

```
head(rename(wideTable3, Male = male), 3)
```

```
##   Male female     type group day
## 1    1      3 bacteria     a   1
## 2    1      4    virus     a   2
## 3    2      5 bacteria     a   3
```

Filtering rows is also straightforward. Inside function `filter()` you cand use logical operators (`&`, `|`, `xor`, `!`), comparisions (e.g. `==` or `>=`), or functions (`is.na()`, `between()`, `near()`).

```
filter(wideTable3, group == 'a')
```

```
##    male female     type group day
## 1     1      3 bacteria     a   1
## 2     1      4    virus     a   2
## 3     2      5 bacteria     a   3
## 4     2      6    virus     a   4
```

```
## 5     3      7 bacteria     a   5
## 6     3      8    virus     a   6
## 7     4      9 bacteria     a   7
## 8     4     10    virus     a   8
## 9     5     11 bacteria     a   9
## 10    5     12    virus     a  10
```

```
filter(wideTable3, type != 'bacteria')
```

```
##    male female  type group day
## 1     1      4 virus     a   2
## 2     2      6 virus     a   4
## 3     3      8 virus     a   6
## 4     4     10 virus     a   8
## 5     5     12 virus     a  10
## 6     6      4 virus     b   2
## 7     7      6 virus     b   4
## 8     8      8 virus     b   6
## 9     9     10 virus     b   8
## 10   10     12 virus     b  10
```

```
filter(wideTable3, near(female, 5))
```

```
##   male female     type group day
## 1    2      5 bacteria     a   3
## 2    7      5 bacteria     b   3
```

4.4.3 mutate and transmute

Both functions are widly used in data manipulation in **R**. Their main purpose is to create new variable (usually from existing ones) in data frame. Lets look on examples:

```
select(wideTable3, male) %>%
  mutate(cumulativeMaleSum = cumsum(male)) %>%
  head(5)
```

```
##   male cumulativeMaleSum
## 1    1                 1
## 2    1                 2
## 3    2                 4
## 4    2                 6
## 5    3                 9
```

```
select(wideTable3, male, female) %>%
  mutate(cumulativeMaleSum = cumsum(male),
         femaleLog = log(female),
         cumSumFemaleLog = cumsum(femaleLog)) %>%
  head(5)
```

```
##   male female cumulativeMaleSum femaleLog cumSumFemaleLog
## 1    1      3                 1  1.098612        1.098612
## 2    1      4                 2  1.386294        2.484907
## 3    2      5                 4  1.609438        4.094345
## 4    2      6                 6  1.791759        5.886104
## 5    3      7                 9  1.945910        7.832014
```

As you can see in second example, when we use `mutate()` function, the newly created variables (in this example `femaleLog`) are available immediately so we can use them to create another variable (in this case `cumSumFemaleLog`) within one function call. In this example, I also used piping operator, becasue calculating intermediate steps and storing them as a result, which can be used later is useless as we are interested only in final outcome. The biggest advantage of this procedure is that we clean our *Environment* clean and preserve memomry – which is very important in long and memory consuming projects. Last but not least, the difference between `mutate()` and `transmute()` is that the latter do not preserve all variables, only the ones you created.

4.4.3.1 `group_by` and `summarise`

`summarise()` is commonly used function on grouped data. It allows to calculate many typical descriptive statistics (like `mean()` or `quantile`) for particular groups in you data set, as well as it can count number of observations (`n()` function), or number of uniqe observations (`distinct()`). To see how it works in practice lets look back on our example and calculate mean value for `males` and `female`, as well as day range and number of cases for groups derived from `type`.

```
wideTable3 %>%
  group_by(type) %>%
  summarise(meanF = mean(female),
            meanM = mean(male),
            numberOfDays = (range(day)[2]-range(day)[1]),
            numberOfCases = n())
```

```
## # A tibble: 2 x 5
##   type     meanF meanM numberOfDays numberOfCases
##   <fctr>   <dbl> <dbl>        <int>         <int>
## 1 bacteria  7.00  5.50            8            10
## 2 virus     8.00  5.50            8            10
```

Easy.

4.4.3.2 Is there anything more in `dplyr` library?

Yes. Actually I presented here only very very tiny fracture of `dplyr` possibilities just to familiarize you with syntax and using piping operator. When you go deeper into world of data frames transformation, you will find other commonly used functions like: different kinds of joining data frames (simmilar to SQL joining tables), conditional selecting, filtering, renaming rows and columns, extracting values or arranging your data frame. Thankfuly this procedures are so common that even if you won't grasp it immedately from functions description, you will still find hundreds of tutorials on the web, or help in *StackOverflow*.

Chapter 5

Lets do some math!

5.1 Simple statistical model

Ok. There is no such thing as simple statistical model. However there are lots of packages that will make you suffer less. In fact this is one of biggest **R** advantages, that you can make even very suffisticated statistical modelling without any knowledge on programming since you use *black boxes*. When you are dealing with classic statistical models many of them are included in **base R** distribution - like linear or gereralized additive models. You would probably need to use mixed effect models, at some point. Good news is that there is a very nice and quite straightforward to use library called `lme4` . Every time you run into problem, and you do not know what to use for statistical modelling, or how to perform full procedure, just query google. There are hundrets of blogs, web pages and *Stack Overflow* discussion on it.

5.2 Other models

5.2.1 Libraries

Other, usually dynamic models require more knowledge, experience and some libraries. Before you start looking for more tailored solutions, install following libraries: `deSolve`, `fitdistrplus`, `rriskDistributions` and `truncdist`. First one contains tools for solving differential equations sets, second and third provides you tools to deal with distriubutions (such as comparing distributions or estimating its parameters), and the last one allows you to use truncated distributions.

5.2.2 Simple asymptotic model and noise

5.2.3 Mechanistic model in R

5.2.4 Solving differential equations

Chapter 6

Functions

In short, function is a piece of code that takes some arguments, makes *mumbo jumbo* and returns result. All the time, through this book we were using functions that are buil in **base R** or comes from additional packages (like `dplyr`). So you are now quite familiar whit the syntax that resambles typical mathematical syntax – *name of a function followed by arguments in brackets - like f(x).* From time to time you will need to do something in your code for few times with different arguments. In order to not repeat yourself and not *copy – paste* your code multiple times you can wrap your procedure in a function. The best way (as always) to understand how to do it, is to do it. To begin with something simple we will start with makeing basic math functions which later will lead us to simple calculator.

6.1 Simple math functions.

Simple math operations include: *adding*, *substracting*, *dividing* and *multipling*. To make our calculator slightly less boring, we can also add *powers* and *nth rooting*. To not to complicate to much things in the begining, lets say that we want our function to take two and only two arguments. Lets look below how to code our functions:

```
addF <- function(x,y) {
  return(x+y)
}
subF <- function(x,y) {
  return(x-y)
}
divF <- function(x,y) {
  return(x/y)
}
mulF <- function(x,y) {
  return(x*y)
}
powF <- function(x,y) {
  return(x:y)
}
ntrF <- function(x,y) {
  return(x^(1/y))
}
```

6.2 Building your own calculator

Above example is of course useless and boring. So lets quickly get to making calculator, that would use one of above functions, or return results of all of them. So actually this time we will make so called *wraper* around previously made functions. We will use *switch* functionality, so besides our two parameters: `x` and `y` we will also tell our function which of the results we are interested in.

```
simpleCalculator <- function(x, y, mathType) {
  addF <- function(x, y) {
  return(x+y)
  }
  subF <- function(x, y) {
  return(x-y)
  }
  divF <- function(x, y) {
  return(x/y)
  }
  mulF <- function(x, y) {
  return(x*y)
  }
  powF <- function(x, y) {
  return(x^y)
  }
  ntrF <- function(x, y) {
  return(x^(1/y))
  }
  switch(mathType,
         add = addF(x, y),
         substract = subF(x, y),
         multiple = mulF(x, y),
         divide = divF(x, y),
         power = powF(x, y),
         root = ntrF(x, y),
         all = cat('The result of mathematical opereators on two numbers:',
                     paste(x, 'and', y), 'are:', '\naddition:', addF(x, y),
                     '\nsubstraction:', subF(x, y), '\nmultiplication:', mulF(x, y),
                     '\ndvision:', divF(x, y), '\nWhat is more the', y,
                     'th power of', x, 'is', powF(x, y), 'and', x, y,
                     'th root is', ntrF(x, y)))
}

simpleCalculator(25,5, 'root')
```

```
## [1] 1.903654
```

```
simpleCalculator(16,2,'all')
```

```
## The result of mathematical opereators on two numbers: 16 and 2 are:
## addition: 18
## substraction: 14
## multiplication: 32
## dvision: 8
## What is more the 2 th power of 16 is 256 and 16 2 th root is 4
```

6.3 It is not over yet... *Calculator shouldn't divide be 0*!

We know that division by 0 is not the best idea in the world, thus we should stop users (or ourselves) from doing it. Thus we will add an `if...else` statment to our function. Next time when you use 0 as a second argument you will see an error. Also, we declare `mathType = 'all'` as a default value, so if we omit this parameter, function will evaluate anyway.

```
simpleCalculator <- function(x, y, mathType = 'all') {
  if (mathType == 'divide' & y == 0) {
    return('You cannot divide by 0, please change y value.')
  } else if (mathType == 'root' & y == 0) {
    return('Root denominator is 0, cannot perform operation, please change y value.')
  } else if (mathType == 'all' & y == 0) {
    return('Y value needs to be different from 0 to make division and nth root.')
  }
  addF <- function(x, y) {
  return(x+y)
  }
  subF <- function(x, y) {
  return(x-y)
  }
  divF <- function(x, y) {
  return(x/y)
  }
  mulF <- function(x, y) {
  return(x*y)
  }
  powF <- function(x, y) {
  return(x^y)
  }
  ntrF <- function(x, y) {
  return(x^(1/y))
  }
  switch(mathType,
         add = addF(x, y),
         substract = subF(x, y),
         multiple = mulF(x, y),
         divide = divF(x, y),
         power = powF(x, y),
         root = ntrF(x, y),
         all = cat('The result of mathematical opereators on two numbers:',
                   paste(x, 'and', y), 'are:', '\naddition:', addF(x, y),
                   '\nsubstraction:', subF(x, y), '\nmultiplication:', mulF(x, y),
                   '\ndvision:', divF(x, y), '\nWhat is more the', y,
                   'th power of', x, 'is', powF(x, y), 'and', x, y,
                   'th root is', ntrF(x, y)))

}
simpleCalculator(25,0)
```

```
## [1] "Y value needs to be different from 0 to make division and nth root."
```

```
simpleCalculator(25,0, 'root')
```

```
## [1] "Root denominator is 0, cannot perform operation, please change y value."
```

```
simpleCalculator(25,0, 'divide')
```

```
## [1] "You cannot divide by 0, please change y value."
```

```
simpleCalculator(25,5)
```

```
## The result of mathematical opereators on two numbers: 25 and 5 are:
## addition: 30
## substraction: 20
## multiplication: 125
## dvision: 5
## What is more the 5 th power of 25 is 9765625 and 25 5 th root is 1.903654
```

Chapter 7

Graphics

What would be our work value without visualisation? Not much. **R** provides use with some tools to make plots, charst and other visual stuff. However base version has very limited graphic design by default and making it pretty needs a lot of time and code. Nowadays, however, in `tidyverse` there is a very powerfull library with dozens of extensions - `ggplot2`. *GG* stands for *Gramar of Graphics* and in parctice it means that we build our visualisation layer after layer.

Chapter 8

Final indications

8.1 Use Projects

To organize your work, not only in **RStudio**, but also on hard drive you should use projects. It is extremly easy in this IDE. Just click button – *Create new project* – choose *New Directory* and type of project you need. **RStudio** will take care of everything for you. Later you can easily add files of subfolders with specifing content into your project and use ralational links.

8.2 Use RMarkodow

To make your work more reproductible, and also to make nice looking output (in html or pdf) it is a good idea to work in RMarkdown files instead of RScripts. The syntax of **RMarkdown** is nearly identical to original **Markdown**, but allows you to execute and store **RCode**. You can find RMarkdown introduction here. There is also nice cheat sheet which contains all commands you will need. As there is strong pressure to make reasearch and science more open and reproductible, it is strongly adviced that you work with proper tools to do it like **RMarkdown** or **Project Jupyter**. If you want to learn more, there is very nice guide by British Ecological Society you should read.

8.3 Use .rds files

Usually we work with plain text files like csv or tsv in **R**. Howevere, there is also a highly valuable format of files caled rds, that makes work even more pleasant. It not only preserves all classes of variables, but also takes less space than plain text file, due to compression algorithms. Also when using `saveRDS()` you don't need to define hundrets of parameters, just name of object you want to save and its file name. The downside is that it is format to use directly with R.

Bibliography

Delignette-Muller, M.-L., Dutang, C., and Siberchicot, A. (2017). *fitdistrplus: Help to Fit of a Parametric Distribution to Non-Censored or Censored Data*. R package version 1.0-9.

Meyer, D., Dimitriadou, E., Hornik, K., Weingessel, A., and Leisch, F. (2017). *e1071: Misc Functions of the Department of Statistics, Probability Theory Group (Formerly: E1071), TU Wien*. R package version 1.6-8.

Novomestky, F. and Nadarajah, S. (2016). *truncdist: Truncated Random Variables*. R package version 1.0-2.

R Core Team (2017). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

RStudio Team (2016). *RStudio: Integrated Development Environment for R*. RStudio, Inc., Boston, MA.

Wickham, H. (2014). Tidy data. *Journal of Statistical Software*, 59(10):1–23.

Wickham, H. (2017). *tidyverse: Easily Install and Load the 'Tidyverse'*. R package version 1.2.1.

Wickham, H. and Chang, W. (2017). *devtools: Tools to Make Developing R Packages Easier*. R package version 1.13.4.

Wickham, H., Francois, R., Henry, L., and Müller, K. (2017a). *dplyr: A Grammar of Data Manipulation*. R package version 0.7.4.

Wickham, H., Francois, R., Henry, L., and Müller, K. (2017b). *dplyr: A Grammar of Data Manipulation*. R package version 0.7.4.

Wickham, H. and Henry, L. (2017). *tidyr: Easily Tidy Data with 'spread()' and 'gather()' Functions*. R package version 0.7.2.

Xie, Y. (2015). *Dynamic Documents with R and knitr*. Chapman and Hall/CRC, Boca Raton, Florida, 2nd edition. ISBN 978-1498716963.

Xie, Y. (2016). *bookdown: Authoring Books and Technical Documents with R Markdown*. Chapman and Hall/CRC, Boca Raton, Florida. ISBN 978-1138700109.

Xie, Y. (2017). *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.5.