Standards of programming in R

R style guide

Stanislav Katina[1]

[1]Institute of Mathematics and Statistics, Masaryk University
Honorary Research Fellow, The University of Glasgow

March 1, 2020

1. R is **open source software**. It has many advantages of other commercial statistical platforms such as MATLAB, SAS and SPSS.
2. R has its roots in the **statistics community**, being **created by statisticians for statisticians**. This is reflected in the design of the programming language: many of its core language elements are geared toward statistical analysis.
3. **The amount of code** that we need to write in R **is very small compared to other programming languages.** There are many high-level data types and functions available in R that hide the low-level implementation details from the programmer. Although there exist R systems used in production with **significant complexity**, for most data analysis tasks, we need to write only a few lines of code.

4. R's history is inexorably tied to its domain specific predecessors and cousins, as it is **100 percent focused and built for statistical data analysis and visualization**.
5. R can access and manipulate various file types and databases (and was also **designed for flexibility and extensibility**)
6. R focus on **foundational analytics-oriented data types**.
7. R makes it remarkably simple to **run extensive statistical analyses on your data and then generate informative and appealing visualizations with just a few lines of code**.
8. More modern R **libraries/packages** extend and enhance these base capabilities and are **the foundations of many of mind- and eye-catching examples of cutting-edge data analysis and visualization.** Vast package library called the **Comprehensive R Archive Network**, or more commonly known as CRAN

9. R also provides **an interactive execution shell** that has enough basic functionality for general needs.
10. The desire for even more interactivity sparked the development of , which is a combination of **integrated development environment** (**IDE**), **data exploration tool**, and **iterative experimentation environment** that exponentially enhances R's default capabilities.

Click below to see more:

The Comprehensive R Archive Network

R

RStudio – Open source and enterprise-ready professional software for R

Both links provide full installation details for **Linux**, **Windows**, and **macOS** systems.
RStudio comes in two flavors: ***Desktop*** and ***Server***.

Statistics
Why R?

RStudio core features:

- Built-in IDE
- Data structure and workspace exploration tools
- Quick access to the R console
- R help viewer
- Graphics panel viewer
- File system explorer
- Package manager
- Integration with version control systems

The primary difference is that one runs as a standalone, single-user application (RStudio Desktop) and the other (RStudio Server) is installed on a server, accessed via browser, and enables multiple users to take advantage of the compute infrastructure.

Statistics
R – reading in data

R abstract quite a bit of complexity when it comes to reading and parsing data into structures for processing. See functions:

- `read.table()` – reads a `*.txt` file in **table format** and creates a **data frame** from it
- `read.csv()` – reads a `*.csv` file in **table format** and creates a **data frame** from it (check also argument `encoding`, e.g. `"Windows-1250"`, `"UTF-8"` or other)
- `read.delim()`

See `help()` arguments `header`, `sep` and `delim`.

- `download.file(url,destfile)` – to download a **single file** from the `url` and store it in `destfile`; the `url` must start with a scheme such as `http://`, `https://`, `ftp://` or `file://`
- `getURL(url)` – to download a **single file** from the `url` directly to R and then use function `read.table()` to read data – in `library(RCurl)`

Statistics
R – the statistician – data cleaning

First ***set*** a **working directory** to `dir` using function `setwd(dir)`. You can check an absolute `filepath` representing the **current working directory** using function `getwd()`.

```
## reading *.txt file
DATA <- read.table("DATA.txt",header = TRUE)
## reading *.csv file
DATA <- read.csv("DATA.csv",encoding = "Windows-1250",
                 header = TRUE)
## reading from the web
URL <- "http://www.math.muni.cz/.../DATA.txt"
download.file(URL,destfile = "DATA.txt",method = "libcurl")
DATA <- read.table("DATA.txt",header = TRUE)
## reading from the web
install.packages("RCurl")
library(RCurl)
URL <- getURL(URL)
DATA <- read.table(textConnection(URL))
head(DATA)
```


Statistics
R – reading in data

R functions for reading data from other statistical software:

- `readMat()` – package `R.matlab`
- `read.spss()` – reads a file stored by the SPSS save or export commands – also in library `foreign`
- `read.ssd()` – generates a SAS program to convert the content of `ssd` data file to SAS transport format and then uses `read.xport()` to obtain a `data.frames()` – library `foreign`
- `read.xport()` – reads a file as a SAS XPORT format library and returns a list of `data.frames()` – library `foreign`

R also provides extensive support for accessing data stored in various **SQL and NoSQL databases**. For SQL databases, use e.g. `library(RPostgreSQL)`.

Explore: Load flat files in to R with the `readr` package, which is part of the core `tidyverse` package. Most of `readr`'s functions are concerned with turning flat files into data frames:

1. `read_csv()` reads comma-delimited files
2. `read_csv2()` reads semicolon-separated files (common in countries where "comma" is used as the decimal place)
3. `read_tsv()` reads tab-delimited files
4. `read_delim()` reads files with any delimiter
5. `read_fwf()` reads fixed-width files
6. `read_table()` reads a common variation of fixed-width files where columns are separated by white space

Compared to Base R (there are a few good reasons to favor `readr` functions over the `base` equivalents):

1. They are typically **much faster** ($\approx 10\times$) than their `base` equivalents. Long-running jobs have a **progress bar**, so you can see what is happening. If you are looking for raw speed, try `data.table::fread()`.
2. They produce **tibbles**, and they do not convert character vectors to factors, use row names, or munge the column names. These are common sources of frustration with the base R functions.
3. They are **more reproducible**. Base R functions inherit some behavior from your operating system and environment variables, so import code that works on your computer might not work on someone else's.

Tibbles are data frames, but they tweak some older behaviors to make life a little easier. R is an old language, and some things that were useful 10 or 20 years ago now get in your way. Its difficult to change base R without breaking existing code, so most innovation occurs in *packages*.

Before we get into the details of how `readr` reads files from disk, we need to take a little detour to talk about the `parse_*()` functions. These functions are useful in their own right, but are also an important building block for `readr`. Using **parsers** is mostly a matter of understanding what is available and how they deal with different types of input.

1. `parse_logical()` **parse logicals** and `parse_integer()` **parse integers**. There is basically nothing that can go wrong with these parsers so I wont describe them here further.
2. `parse_double()` is **a strict numeric parser**, and `parse_number()` is a **flexible numeric parser**. These are more complicated than you might expect because different parts of the world write numbers in different ways.
3. `parse_character()` seems so simple that it should not be necessary. But one complication makes it quite important: **character encodings**.
4. `parse_factor()` **creates factors**, the data structure that R uses to represent categorical variables with fixed and known values.
5. `parse_datetime()`, `parse_date()`, and `parse_time()` allow you to **parse various date and time specs** – the most complicated – there are so many different ways of writing dates.

- `readr` has the notion of a locale, an object that specifies parsing options that differ from place to place. When parsing numbers, the most important option is **the character you use for the decimal mark (separator)**. You can override the default value of **decimal point** to, e.g. **decimal comma**, by creating a new locale and setting the `decimal_mark` argument
- `readr`'s default locale is **US-centric**, because generally R is US-centric (i.e., the documentation of `base` R is written in **American English**). An alternative approach would be to try and guess the defaults from your operating system. This is hard to do well, and, more importantly, makes your code fragile – even if it works on your computer, it might fail when you email it to a colleague in another country.

- seems like `parse_character()` should be really simple – it could just return its input. Unfortunately life is not so simple, as there are multiple ways to represent the same string.
- **ASCII** does a great job of representing English characters, because its the **American Standard Code for Information Interchange**.
- Things get more complicated for languages other than English. In the early days of computing there were many competing standards for **encoding non-English characters**, and to correctly interpret a string you needed to know both the values and the encoding – e.g. two common encodings are **Latin1** (ISO-8859-1, used for Western European languages) and **Latin2** (ISO-8859-2, used for Eastern European languages) – and coding a particular byte could be different. Fortunately, today there is one standard that is supported almost everywhere: **UTF-8**. UTF-8 can encode just about every character used by humans today, as well as many extra symbols (like emoji).

- `readr` uses **UTF-8** everywhere – **it assumes your data is UTF-8 encoded when you read it, and always uses it when writing**. This is a good default, but will fail for data produced by older systems that do not understand UTF-8.
- **How do you find the correct encoding?** If you are lucky, it will be included somewhere in the data documentation. Unfortunately, thats rarely the case, so `readr` provides `guess_encoding()` to help you figure it out. **Its not foolproof**, and it works better when you have lots of text, but it is a **reasonable place to start**.
- Encodings are a rich and complex topic. If you would like to learn more I would recommend reading the detailed explanation at http://kunststube.net/encoding/.

- R uses factors to represent categorical variables that have a known set of possible values.
- Give `parse_factor()` a vector of known levels to generate a warning whenever an unexpected value is present.

You pick between three parsers depending on whether you want a **date** (the number of days since 1970-01-01), a **date-time** (the number of seconds since midnight 1970-01-01), or a **time** (the number of seconds since midnight). When called without any additional arguments:

- `parse_datetime()` expects an ISO8601 date-time. ISO8601 is an international standard in which the components of a date are organized from biggest to smallest: **year**, **month**, **day**, **hour**, **minute**, **second**. This is the most important date-time standard (for more details see https://en.wikipedia.org/wiki/ISO_8601).
- `parse_date()` expects a four-digit **year**, a - or /, the **month**, a or /, then the **day**.
- `parse_time()` expects the **hour**, :, **minutes**, optionally : and **seconds**, and an optional **a.m./p.m.** specifier.

`Base` R does not have a great built-in class for time data, so `readr` use the one provided in the hms package. You can also supply your own date-time format.

`readr` uses a heuristic to figure out the type of each column – **it reads the first 1000 rows and uses some (moderately conservative) heuristics to figure out the type of each column**. You can emulate this process with a character vector using `guess_parser()`, which returns `readr`'s best guess, and `parse_guess()`, which uses that guess to parse the column. The heuristic tries each of the following types, stopping when it finds a match:

- `logical` – contains only F, T, FALSE, or TRUE
- `integer` – contains only numeric characters (and -)
- `double` – contains only valid doubles (including numbers like 4.5e-5)
- `number` – contains valid doubles with the grouping mark inside

- `time` – matches the default `time_format`
- `date` – matches the default `date_format`
- `date-time` – matches any ISO8601 date

If none of these rules apply, then the column will stay as a vector of strings. It is always a good idea to explicitly pull out the `problems()`, so you can explore them in more depth. It is highly recommended always supplying `col_types`, building up from the printout provided by `readr`. This ensures that you have a consistent and reproducible data import script. If you rely on the default guesses and your data changes, `readr` will continue to read it in. If you want to be really strict, use `stop_for_problems()` – that will throw an error and stop your script if there are any parsing problems.

The consistency in the record format makes the consumption of the data equally as straightforward in each language. In each language/environment, we follow a typical pattern of:

1. Reading in data
2. Assigning meaningful column names (if necessary)
3. Using built-in functions to get an overview of the data structure
4. Taking a look at the first few rows of data, typically with the `head()` or `tail()` function

Statistics

R – reading in data – data entry errors

Most common **data entry errors** (errors can arise from human sloppiness, whereas others are due to machine or hardware failure):

1. **redundant whitespace** – leading and trailing spaces [solved by database programming]
2. **capital letters mismatches** [solved by database programming]
3. **deviation from a code book** [solved by database programming]
4. **different units of measurement** [solved by database programming]
5. **impossible values** and **sanity checks** – physically or theoretically impossible values (can be directly expressed with rules, if present – reference ranges should be used here) [solved by database programming]
6. **possible outliers** [solved by statistical programming]

Statistics

R – writing data to a file – **.csv**

`readr` also comes with two useful functions for writing data back to disk – `write_csv()` and `write_tsv()`. This is about twice as fast as `write.csv()`, and never writes row names. Both functions increase the chances of the output file being read back in correctly by:

1. Always encoding strings in UTF-8.
2. Saving dates and date-times in ISO8601 format so they are easily parsed elsewhere.

If you want to export a .csv file to MS Excel, use `write_excel_csv()` – this writes a special character (a "byte order mark") at the start of the file, which tells MS Excel that you are using the UTF-8 encoding. Note that the **type** information is lost when you save to .csv. This makes .csv a little unreliable for caching interim results – you need to re-create the column specification every time you load in.

Statistics

R – writing data to a file – **.rds** and **.feather**

Alternatives:

1. `write_rds()` and `read_rds()` are uniform wrappers around the base functions `readRDS()` and `saveRDS()`. These store data in R's custom binary format called .rds.
2. The `feather` package implements a fast binary file format that can be shared across programming languages. `feather` tends to be faster than .rds, is usable outside of R, and .rds supports list-columns (feather currently does not).

Explore other packages for (reading and writing data files): `haven`, `rio`, `readxl`, `xlsx`, `XLConnect`, `xml2`, etc.

Read also the **R data import/export manual** at https://cran.r-project.org/doc/manuals/r-release/R-data.html.

Statistics

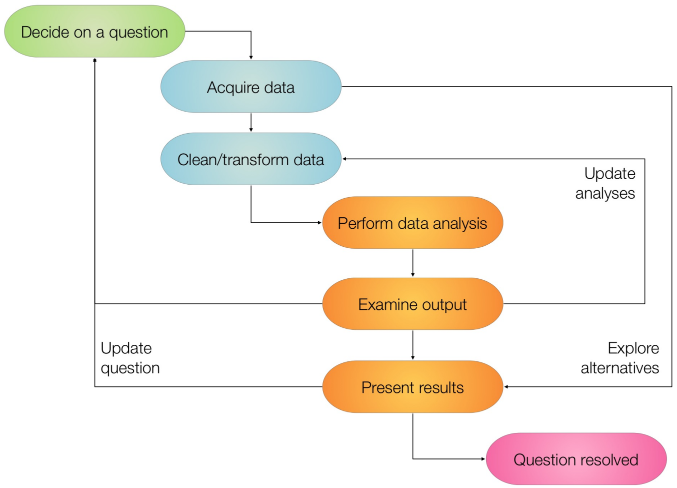
The statistician

Given some of the "**rookie mistakes**" seen in many scientific reports (bio-medical, geographical or other) or industry reports (pharmaceutical, security or other) and **the prevalence of raw counts** in science/industry dashboards, there is a high probability that statistics is the weakest area for science/industry professionals.

You do not need a Ph.D. in statistics to be **an effective data scientist**. However, its important to have an **understanding of the fundamentals of statistical analysis**, even when you are part of a **multidisciplinary team**.

Understanding and applying statistics correctly is more complex than you might imagine, and individuals in disciplines with a rich history of using statistics to solve complex problems oftentimes fall into common traps.

A hallmark of a **good data scientist** is adaptability and you should be continually scouring the digital landscape for emerging tools that will help you solve problems.



The methodology of extracting insights from data is called as **data science**. Historically, data science has been known by different names: in the early days, it was known simply as **statistics**, after which it became known as **data analytics**. There is an important difference between data science as compared to statistics and data analytics.

Data science is a multi-disciplinary subject: it is a combination of statistical analysis, programming, and domain expertise.

Over the last few years, data science has emerged as a discipline in its own right.

Three aspects and their importance:

1. **Statistical skills** are essential in applying the right kind of statistical methodology along with interpreting the results.
2. **Programming skills** are essential to implement the analysis methodology, combine data from multiple sources and especially, working with large-scale datasets.
3. **Domain expertise** is essential in identifying the problems that need to be solved, forming hypotheses about the solutions, and most importantly understanding how the insights of the analysis should be applied.

However, **there is no standardized set of tools that are used in the analysis**. Data scientists use a **variety of programming languages and tools** in their work, sometimes even using a **combination of heterogeneous tools to perform a single analysis**. This increases the learning curve for the new data scientists.

The R programming environment presents a **great homogeneous set of tools for most data science tasks**.

R is more than a programming language. It is an **interactive environment for doing statistics**. Think of R as *having* a programming language than *being* a programming language. The R language is the scripting language for the R environment. Variables cannot be declared. They come into existence on first assignment (**lexical scoping**) – it is not always easy to determine the scope of a variable.

1. the **assignment operator** in R is "`<-`" (the arrow) with the receiving variable on the left; it is also possible, though uncommon, *to reverse the arrow* and put the receiving variable on the right; it is sometimes possible to use "=" for assignment
2. when supplying *default function arguments* or *calling functions with named arguments*, you must use the "=" operator and cannot use the arrow
3. at some time in the past R used *underscore* as assignment – this meant that the C convention of using underscores as separators in multi-word variable names was not only disallowed but produced strange side effects; however, R allows *underscore as a variable character* and not as an assignment operator
4. don't use *hyphens* "`-`"

5. because the underscore was not allowed as a variable character, the convention arose to use *dot* as a **name separator**
6. unlike its use in many object oriented languages, the dot character in R has no special significance, with two exceptions
   - the `ls()` function in R lists **active variables** but *does not list files that begin with a dot*
   - `...` is used to indicate a **variable number of function arguments**
7. R uses "$" in a manner analogous to the way other languages use dot (identifying the parts of an **object**) – see e.g. `data.frame()` and `list()`
8. R has several **one-letter reserved words**: `c`, `q`, `s`, `t`, `C`, `D`, `F`, `I`, and `T` – actually, these are not reserved, but its best to think of them as reserved

9. the preferred form for ***variable names*** is **all lower case letters and words separated with dots** (`variable.name`), but `variableName` is also accepted
10. ***function names*** have **initial capital letters and no dots** (`FunctionName`)
11. ***constants*** are named like functions but with an initial k (`kConstantName`)
12. **line length** – the maximum line length is 80 characters
13. **indentation** – when indenting your code, use two spaces – never use tabs or mix tabs and spaces (<u>exception</u>: when a line break occurs inside parentheses, align the wrapped line with the first character inside the parenthesis)

14. **spacing**
   - place spaces around all binary operators (`=`, `+`, `-`, `<-`, etc.)
     <u>exception</u>: spaces around ='s are optional when passing parameters in a function call
   - do not place a space before a comma, but always place one after a comma
   - place a space before left parenthesis, except in a function call
   - extra spacing (i.e., more than one space in a row) is okay if it improves alignment of equals signs or arrows (`<-`)
   - do not place spaces around code in parentheses or square brackets
     <u>exception</u>: always place a space after a comma.
15. **semicolons** – do not terminate your lines with semicolons or use semicolons to put more than one command on the same line

16. `attach()` – avoid using it – the possibilities for creating errors when using attach are numerous
17. **commenting** – comment your code
    - **entire commented lines** should begin with "#" and one space
    - **short comments** can be placed after code preceded by two spaces, "#", and then one space
18. **function definitions and calls** – function definitions should first list arguments *without default values*, followed by those *with default values* – in both function definitions and function calls, *multiple arguments per line are allowed*; *line breaks are only allowed between assignments*

19. **function documentation**
    - functions should contain a *comments section* immediately below the function definition line – these comments should consist of a *one-sentence description* of the function
    - a list of the function's **arguments**, denoted by `Args:`, with a description of each (including the data type)
    - and a description of the **return values**, denoted by `Returns:`
    - the comments should be descriptive enough that a caller can use the function without reading any of the function's code

20. **general layout and ordering**
    - copyright statement comment
    - author comment
    - file description comment, including purpose of program, inputs, and outputs
    - `source()` and `library()` statements
    - function definitions
    - executed statements, if applicable (e.g., `print`, `plot`)

For more details see:
Google's R Style Guide and R Coding Conventions

1. built-in function for creating vectors is `c()`
2. "**container vector**" – an ordered collection of numbers with no other structure
    - the **length of a vector** is the number of elements in the container
    - **operations** are applied *componentwise*
3. "**mathematical vector**" – an element of a vector space
    - **length of a vector** is geometrical length determined by an inner product
    - the number of components is called **dimension**
    - **operations** are *not applied componentwise*

Statistics
R – vectors

A vector in R is a **container vector**, a statisticians collection of data, not a mathematical vector. The R language is designed around the assumption that a vector is **an ordered set of measurements** rather than a geometrical position or a physical state. R supports mathematical vector operations, but they are secondary in the design of the language.

The R language has no provision for **scalars**. The only way to represent a single number in a variable is to use a vector of length one. It is usually clearer and more efficient in R to operate on vectors as a whole.

4. vectors in R are **indexed starting with 1** and matrices in are stored in **column-major order**
5. elements of a vector can be accessed using "`[]`".
6. vectors automatically expand when assigning to an index past the end of the vector

Statistics
R – vectors

7. **five types of indices/subscripts** in R
   - **positive integers** – subscripts that reference particular elements
   - **negative integers** – is an instruction to remove an element from a vector (it makes sense in statistical context)
   - **zero** – is does nothing (it doesn't even produce an error)
   - **Booleans**
     - a Boolean expression with a vector evaluates to a vector of Boolean values, the results of evaluating the expression componentwise (e.g. `x[x > 3]` – the expression `x > 3` evaluates to the vector of TRUE or FALSE)
     - when a vector with a Boolean subscript appears in an assignment, the assignment applies to the elements that would have been extracted if there had been no assignment (`x[x > 3] <- 7`)
   - **nothing** – a subscript can be left out entirely (So `x[]` would simply return `x`)

Statistics
R – sequences, replications

8. **sequences**
   - the expression `seq(a, b, n)` creates *a closed interval* from a to b in steps of size n
   - the notation `a:b` is an abbreviation for `seq(a, b, 1)`
   - the notation `seq(a, b, length = n)` is a variation that will set the step size to `(b - a)/(n - 1)` so that the sequence has n points

```
seq(1, 10, by = 2)  # odd numbers
seq(1, 10, length = 4)
seq(1, 10, by = 0.05) # sufficiently dense sequence (?)
```

9. **replications** – function `rep(x)` replicates the values in `x` – important arguments are `times`, `each` and `length`

```
rep(1:4, 2)
rep(1:4, each = 2)  # not the same as above
rep(1:4, c(2,2,2,2))  # the same as above
rep(1:4, c(2,1,2,1))
rep(1:4, each = 2, len = 4)  # only first four elements
```


Statistics
R – types

10. **the type of a vector** is the type of the elements it contains and must be one of the following `logical`, `integer`, `numeric`, `character`, `factor`, `complex`, `double` (creates a double-precision vector), or `raw` – *all elements of a vector must have the same underlying type* (this restriction does not apply to lists)

```
x1 <- c(TRUE,TRUE,TRUE,FALSE,TRUE,FALSE)  # logical vector
x2 <- c(1,2,5.3,6,-2,4)  # numeric vector
x3 <- c("one","two","three")  # character vector
gender <- c(rep("male", 20), rep("female", 30))
gender <- factor(gender)  # factor vector
```

11. **type conversion functions** have the naming convention `as.xxxx()` for the function converts its argument to type `xxxx`, e.g., `as.integer(4.2)` returns the integer 3, and `as.character(4.2)` returns the string "4.2" (see also `is.xxxx()`)

Statistics
R – Boolean operators

12. **Boolean operators**
    - **true values** – T or TRUE and **false values** – F or FALSE
    - the *shorter form* operators **and** "&" and **or** "|" apply element-wise on vectors (are vectorized)

```r
((-2:2) >= 0) & ((-2:2) <= 0)
# [1] FALSE FALSE  TRUE FALSE FALSE
```

- the *longer form* operators **and** "&&" and **or** "||" are often used in conditional statements (evaluates left to right examining only the first element of each vector)

```r
((-2:2) >= 0) && ((-2:2) <= 0)
# [1] FALSE
```

- the operators will not evaluate their second argument if the return value is determined by the first argument

Statistics
R – lists, matrices, arrays

13. **lists** are like vectors, except *elements need not all have the same type*, e.g. the first element of a list could be an integer and the second element be a string or a vector of Boolean values
    - are created using the `list()` function
    - elements can be access by position using "`[[]]`".
    - named elements of lists can be accessed by dollar sign "$"

```r
A <- list(name = "John", age = 24)
A[[1]]
A$name
```

- if you attempt to access a non-existent element of a list, say `A[[3]]` above, you will get an error
- you can assign to a non-existent element of a list, thus extending the list; if the index you assign to is more than one past the end of the list, intermediate elements are created and assigned `NULL` values

Statistics
R – matrices, arrays, data frames

14. **matrix** and **array** – R does not support matrices and arrays, only vectors, but you can *change the dimension of a vector*, essentially making it a matrix (see also `rbind()`,`cbind()`)
    - R **fills matrices by column**
    - to **fill matrix by row**, add the argument `byrow = TRUE` to the call to the `matrix()` function

```r
A1 <- array(c(1,2,3,4,5,6), dim = c(2,3))
A2 <- matrix(c(1,2,3,4,5,6), 2, 3))
A3 <- matrix(c(1,2,3,4,5,6), 2, 3, byrow = TRUE)
```

15. **data frame** – is more general than a matrix, in that different columns can have different modes (numeric, character, factor, etc.)

```r
x1 <- c(1,2,3,4)
x2 <- c("red","white","red",NA)
x3 <- c(TRUE,TRUE,TRUE,FALSE)
mydata <- data.frame(x1,x2,x3)
names(mydata) <- c("ID","Color","Passed") # variable names
```


Statistics
R – missing values and NaNs

16. **missing values and NaNs** – the result of an operation on numbers may return different types **non-number**
    - "**not a number**" – `NaN`
    - "**not applicable**" – NA (to indicate missing data, and is unfortunately fairly common in data sets)
    - the author of an R function, has *no control over the data* his function will receive because NA is a legal value inside an R vector – there is no way to specify that a function takes only vectors with non-null components – you must handle NA values, even if you handle them by returning an error
    - the function `is.nan()` will return `TRUE` for those components of its argument that are `NaN` (see also `!is.nan()`)
    - the function `is.na()` will return true for those components that are NA or `NaN` (see also `!is.na()`)

17. `sessionInfo()` – prints the R version, OS, packages loaded, etc.
18. `help(fctn)` – displays help on any function `fctn`,
19. the function `quit()` or its alias `q()` terminate the current R session
20. `save.image()` is just a short-cut for ”save my current workspace”
21. `ls()` – shows which objects are defined
22. `rm(list=ls())` – clears all defined objects
23. prefixes `d`, `p`, `q`, `r` stand for **density** (probability density function, PDF), **probability** (cumulative distribution function, CDF), **quantile** (CDF^{-1}), and **random sample** – e.g., `dnorm()` is the density function of a normal random variable and `rnorm()` generates a sample from a normal random variable etc.

function	description	function	description
binomial distribution		**Poisson distribution**	
dbinom()	probability mass function	dpois()	probability mass function
pbinom()	distribution function	ppois()	distribution function
qbinom()	quantile	qpois()	quantile
rbinom()	pseudo-random numbers	rpois()	pseudo-random numbers
multinomial distribution		**gamma distribution**	
dmultinom()	probability mass function	dgamma()	density function
pmultinom()	distribution function	pgamma()	distribution function
qmultinom()	quantile	qgamma()	quantile
rmultinom()	pseudo-random numbers	rgamma()	pseudo-random numbers
normal distribution		**Student t distribution**	
dnorm()	density function	dt()	density function
pnorm()	distribution function	pt()	distribution function
qnorm()	quantile	qt()	quantile
rnorm()	pseudo-random numbers	rt()	pseudo-random numbers
χ^2 **distribution**		**Fisher F distribution**	
dchisq()	density function	df()	density function
pchisq()	distribution function	pf()	distribution function
qchisq()	quantile	qf()	quantile
rchisq()	pseudo-random numbers	rf()	pseudo-random numbers
multivatiate normal distribution library mvtnorm		**multivatiate normal distribution** library MASS	
rmvnorm()	pseudo-random numbers	mvrnorm()	pseudo-random numbers

For more details see e.g. R language for programmers.