Learn R for Data Wrangling

GA for TC

2020-04-28

Contents

Chapter 1

Prerequisites

1.1 What to expect

This site will cover basic and intermediate topics in learning the use of R for **data wrangling** and visualization. This will serve as an introduction for data science. The basics of R and RStudio will be revisited, with some tips and tricks included, before moving on to using the Tidyverse library for manipulating, exploring and visualizing data.

The main tools that will be used are RStudio and the packages in the Tidyverse library, including dplyr, tidyr, ggplot2 and readr.

The general mode of operation will be to introduce R topics in sequence and to provide hand-picked apropriate online resources for those topics. However, some chapters, particularly the introductory ones, will have relatively extensive hand-written material. Other chapters might only provide summaries and supplementary comments to the provided online resources.

1.2 Resources Needed:

- R
- RStudio
- Web Browser

This tutorial assumes that R and RStudio have already been installed, but a link to installation instructions is provided.

1.3 What is R?

Such a simple question, but one worth answering. In short, the creators of the language originally defined it as 'a programming language for data analysis and graphics' (Ihaka and Gentleman 1996). However, it has grown to much more than that. R can now be called *a powerful integrated environment for statistical and data analysis, data processing, graphics and reporting.* R is *open-source*, meaning that the R codebase is open to users, who can contribute to the development and improvement of the language, mainly through the creation of packages for different purposes and domains. This has led to an expansion of the use of R accross academia and the industry, with applications in official statistics, data science, machine-learning, web-scraping, interactive visualizations and more! R is a powerful tool that continues to improve and expand.

1.4 Is it worth learning R for data wrangling/data science as opposed to other languages like Python?

It is very much worth learning R. On the statistics side of things, R is used so extensively across academia because its tools and packages are very strongly developed and validated against standards for statistical practice and analysis.

> "R is probably the most thoroughly validated statistics software on Earth." – Uwe Ligges, CRAN maintainer (useR!2017).

Therefore, for somebody that works in Official Statistics or academia, R is an easy choice. However, many people working in the industry or private sector prefer other tools such as Python, especially when it comes to data science and data analytics. Are these tools better than R? Not necessarily. Although all tools, softwares and languages have their pros and cons, R offers some very compelling advantages.

Firstly, R is more specialized. R was *created* for statisitcal analysis and data manipulation. It was built by statisticians for statisticians and therefore considered the best language for statistical analysis and modeling. For example, R provides many easy ways to deal with missing values, and it has many well-built packages for survey data analysis. R is also believed to be the best language out there for data visualization through graphs, plots, etc.

Secondly, R has perhaps the best data-wrangling software on earth: Tidyverse. Tidyverse is a library containing a set of packages that vastly improve the data processing and data-wrangling capabalities of R. For example, it allows for easy reshaping, filtering, cleaning and querying of large datasets (dplyr and tidyr).

It offers tools for quickly reading and writing files from different formats such as text, Excel, SPSS, STATA and more (readr package). Ggplot2 is perhaps the most popular graphing and visualization software there is, that allows plots and graphs to be build layer by layer in a very logical way.

Third, R has many other extended features that will help you in your data analysis workflow. Do you need to write a report with graphs, illustrations and tables? There is RMarkdown for that. Do you need an interactive website for running statistical models and visualizations? R Shiny was created for that purpose.

All of these advantages are unified by what is believed to be one of the best overall best programming IDE* out there: RStudio. RStudio is extremely powerful and flexible development environment, and it integrates all of what makes R so great. At the end of the day, it is all these advantages together that mmakes R such a commpelling sofware to learn.

That beig said, Python is considered to be better at general programming, machine learning and webscraping, and the data-science field on a whole leans more towards Python. However, R is still a major player in data science and it is absolutely worth learning data science with R. In addition, you will still be able to use Python modules and commands in R using the R Package 'Reticulate'. Reticulate allows you to import and run Python code right in R. You can have the best of both worlds!

In conclusion, if you are a statistician or in academia, R is the way to go; and when you learn programming in R, you can apply these concepts later on in any other language, such as Python.

Now, let's take a look at starting to use R and RStudio.

*An IDE (Integrated Developoment Environment) is a program that provides many facilities for programmers to write, test and develop programs. It usually provides tools to manage the workflow and extensions to make coding easier. RStudio is considered by many the best IDE there is.

Chapter 2

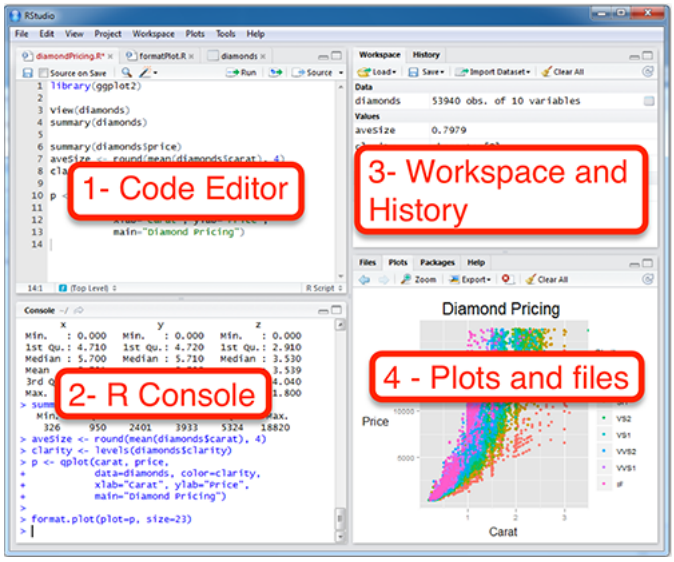
Introduction to R and RStudio

Now let us introduce the basic things needed for working with R: R and RStudio.

For an introduction of how to download R and how to install RStudio, please refer to this link.

2.1 The RStudio interface

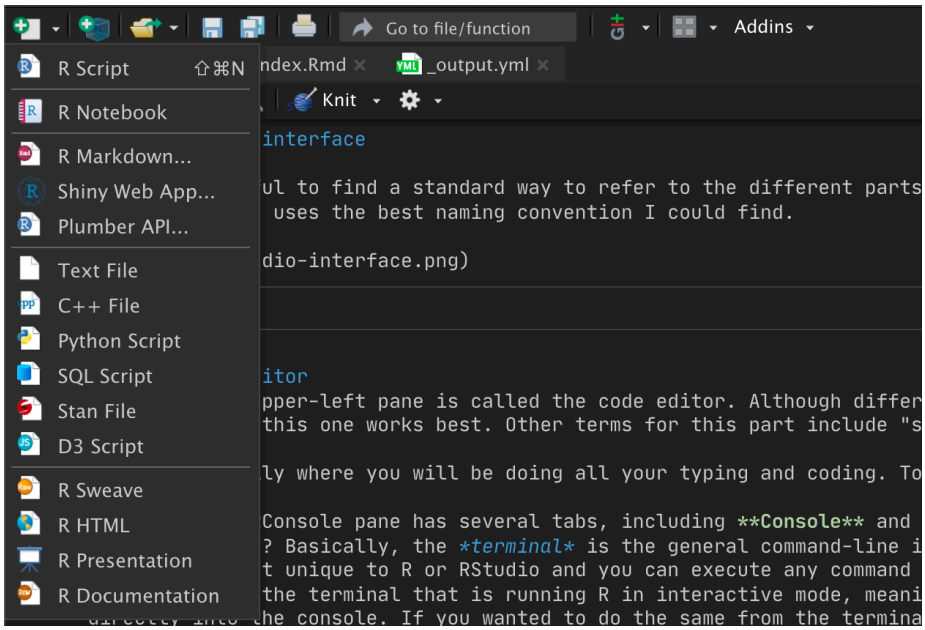
It will is useful to find a standard way to refer to the different parts of the RStudio interface. The following image uses the best naming convention I could find.



2.1.1 The Code Editor

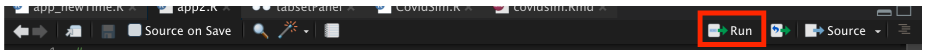
Note that the upper-left pane is called the code editor. Although different people call it by different names, I think this one works best. Other terms for this part include "source editor" or "script editor".

This is basically where you will be doing all your typing and coding. To create a new R file or script, click on the first button on the top menu-bar and select, for example, new R script.



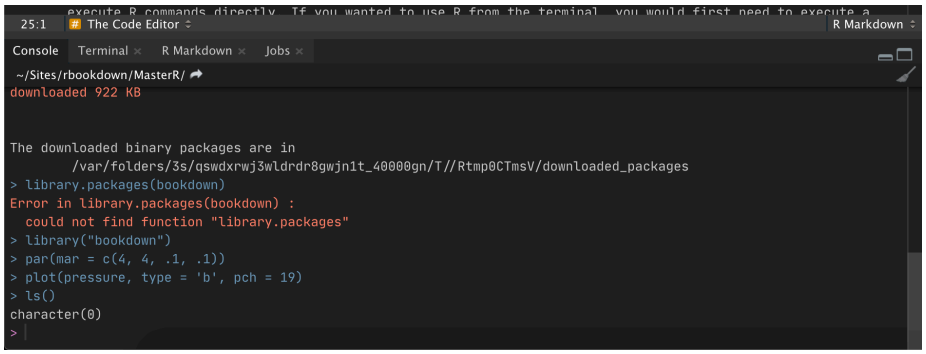
Once this new file is created, you can start writing your R code!

You can run all of the contents of an R script by clicking on the **Run** button in the top middle-right, or you can highlight a chunk of code in the code editor and press cmd+Enter to run that piece of code. This is very useful when testing a piece of code without having to run the entire script!



2.1.2 The R Console

The console is used to execute R code directly by typing R commands into it, and it also displays the results of the executed code. Whenever you execute any command from any other part of RStudio, the console will also produce some feedback. The console is basicaly "R in interactive mode".



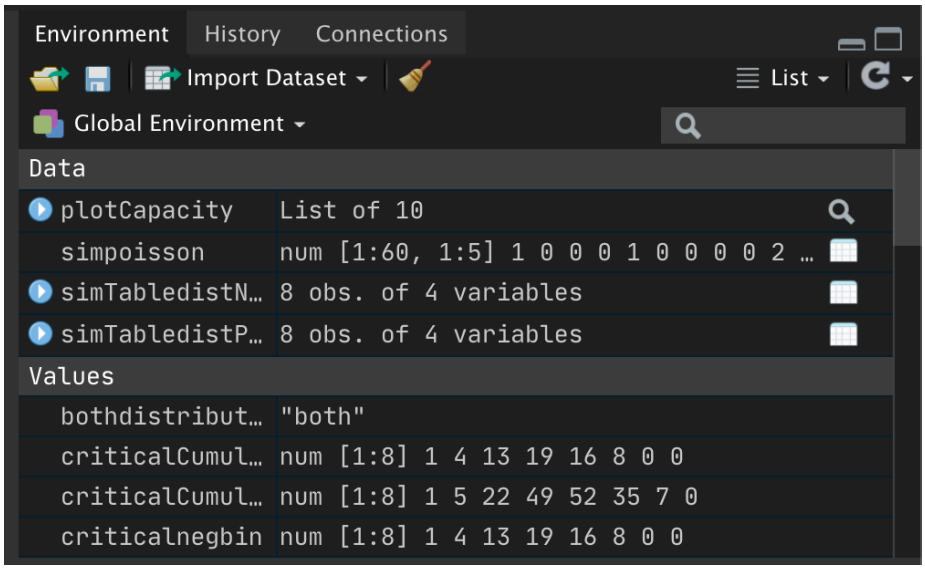
Note that the RConsole pane has several tabs, including **Console** and **Terminal**. What is the difference between the two? Basically,

- The *terminal* is a general command-line interface that all computers have. It is used for executing commands of many types and is not unique to R or RStudio.

- The *Console* is a **subset** of the terminal that is running R in interactive mode, which allows you execute R commands directly. If you wanted to use R from the terminal, you would first need to execute a separate command to call and initiate R.

2.1.3 The Workspace and History pane

This pane gives you an overview of your **working environment**. For one, it tells you how many variables and data objects you currently have loaded into memory. You can click on data objects (such as tables) to view the contents. This is very useful when you import files and save them into R objects. Here is an example of a workspace with some data and variables (*values*) loaded.



To see on the console all the R objects currently loaded into your R Workspace, use the ls() command

```
ls()
```

To remove objects from memory, use the rm() command:

```
#removes all objects
rm(list = ls())

#removes one object
rm(object_name)

#removes multiple objects
rm(c("object1", "object2", "object3"))
```

You can also save the current state of your working environment (all objects etc) so that if you close RStudio and then later you open it again, you can pick up where you left off. To do this:

```
# save all items in workspace to a .RData file
save.image()

# save specified objects to a .RData file
save(object1, object2, file = "myfile.RData")

# load workspace into current session
load("myfile.RData")
```

However, you don't really have to this manually as RStudio automatically asks you if you want to save your workspace, and then it reloads the previously saved workspace whenever you open the project again.

2.1.4 The File and Plots (or Miscellaneous) pane

The bottom right pane contains multiple tabs. The Files tab allows you to see which files are available in your working directory. The Plots tab will display any visualizations that are produced by your code. The Packages tab will list all pack- ages downloaded to your computer and also the ones that are loaded (more on this concept of packages shortly). And the Help tab allows you to search for topics you need help on and will also display any help responses (more on this later as well).

2.2 The working directory

Whenever you create a new project, RStudio automatically sets your **working directory** to that specific folder where you created the project.

What is the working directory and why is it important?

The working directory (wd) is, essentially, the point of reference for the particular project you are working on. The Files pane in RStudio (lower-right pane) will display the files and folders in your current wd. And wenever you want to refer to external files in your R code, your wd will once more be your point of reference and you can use *relative* file-paths (relative to the wd), as opposed to absolute file paths (would need to type the entire path).

For example, lets say you create a new project called *DataWrangling*. And in there, you create an RScript called script.R; this is where you are writing your R code. Now lets say you have a CSV file called meals.csv where you are keeping a meal schedule, and you want to manipulate some of that data in R. Well, if you copy the meals.csv file to your working directory, you can access that file using a **relative path**. For example, to store the contents of the file you would be able to use a command like the following:

```
mealsFrameRelative <- read.csv('meals.csv')
```

The absolute path for the file would look like this (see how much longer it is):

```
mealsFrameAbsolute <- read.csv('Users/gian/sites/R/DataWrangling/meals.csv')
```

In order to see your current working directory, use getwd():

```
getwd()
```

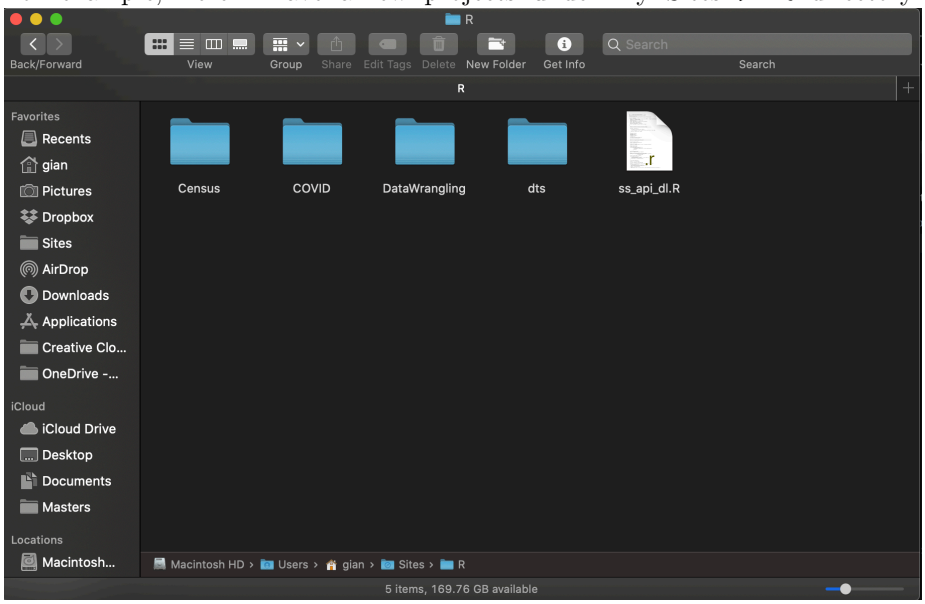
To change it to somewhere else, use setwd():

```
setwd('path/to/new/directory')
```

The main thing to remember is: each project will have its own working directory, usually the folder where you created the project, but this can be changed with setwd().

If you have a Mac, I suggest that you keep your R projects under User > Sites > R > [ProjectName]

For example, here I have a few projects under my Sites > R directory:



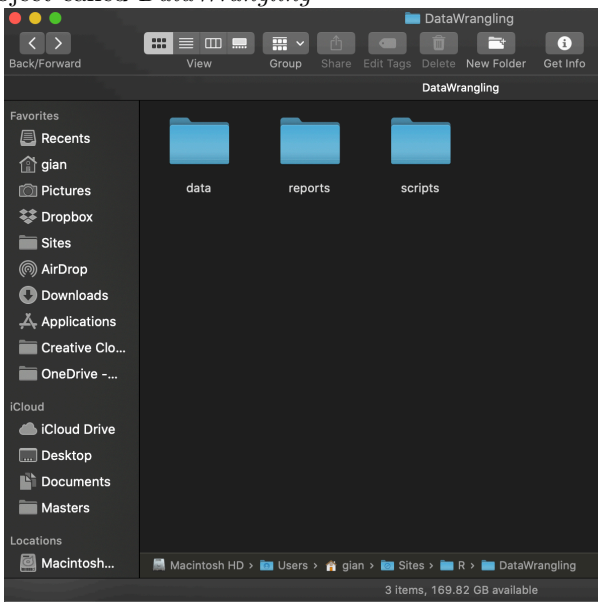
2.3 R and RStudio tips

2.3.1 Organizing your working directory

Usually, when you start an R projects, you will be working with a variety of different files, often of different types. For example, you have your R scripts which end with the file extension *.R*. You also likely will use data files such as CSV or Excel or SPSS files. If you write reports, you will also likely have PDF and Word files. A good way to organize all of these files in a logical manner is to, inside of your working directory, create the following folders:

- **scripts** - to keep your .R and .RMd (markdown) files.
- **data** - to keep all your data files that you will use to work with.
- **reports** - to keep your output documents and reports.

This is how it would look in practice, if I have a project called *DataWrangling* and I have the previously mentioned folder structure:



2.3.2 Other ways to use R

Even though RStudio is the main and preferred way to interact with R, it is not the only way.

Once R is installed, it can be run from the Mac/Windows terminal using the command *R*, which will trigger the interactive mode and you will be able to type R commands. This is useful to remember because you can also type

```
R myScript.R
```

From the terminal and it will automatically execute whatever code you have in the file called myScript.R. In the programming world, we call this *working from the command line* (meaning, working without an interface, directly into the terminal or console).

Also remember, to write R code (or any programming language), all you need is a text editor. You don't need to use RStudio to write R, although it is preferred, and you can view/edit R files with the default text editor, or more specialized coding-oriented text editors such as VS Code. Whatever you use to write R

code, you can later open it and edit it with RStudio if you prefer. This is a useful tip if you ever have to write R in a computer that does not have RStudio installed. Just remember to save the file with the *.R* extension.

Chapter 3

R Language Basics

3.1 Objectives

1. To get a refresher on the basics of R programming.
2. To understand objects and functions.
3. Learn how to create and use vectors.
4. Learn how to deal with missing data.
5. Learn how to use the help and documentation.

3.2 Glossary of terms

- **Project**: An R project consisting of a main folder (as the working directory), and all the files within.
- **Object**: An R variable. This can be a single value, a vector, list, matrix, data frame or other. Almost everything in R is referred to as an object, though there are several types of objects.
- **Function**: A command in R that executes a specific action and usually returns a value. For example, the `sum()` function usually takes values for addition as arguments, and returns the sum.
- **Arguments**: These are values or inputs passed into functions. For example in the function `sum(a, b)`, a and b are *arguments* to the sum function. These will be added and the sum will be returned. Functions usually have documentation where they specify which arguments are required for the function to work properly.
- **Package**: A package, sometimes called a **library**, is a set of related precreated functions in R. It is code that has been written by other people and then packaged together so that others can download the code and reuse it. For example, if somebody were to write code about how to simulate

and model the behavior of an epidemic or infectious disease, they could create a package with the related code and documentation, so that others can benefit and use it as well.

3.3 Creating a new R project

To start, lets create a new project in R and call it DataWrangling. As mentioned, it is recommended to create this project in the directory `user/sites/R/`.

Click on the *File > New Project* and then select *New Directory > New Project > [Select directory and enter project name] > Create*.

Once the project is created, create a new R script by going to *File > New File > R Script*.

3.4 Getting started with code

3.4.1 Creating objects

For the purpose of learning the basics of R coding, we will refer to this excellent online resource. In addition to that, I will provide a summmary of the important points.

3.5 Objects and assignment of values

To create an objects and assign values:

```
object1 <- 5
object2 <- 10

object3 <- object1 + object2
```

3.5.1 Which assignment operator to use

The traditional assignment operator in R is the left-facing arrow `<-`. With time, the language also evolved to use the equal-sign `=` for this same purpoose. However, the standard practice is to always use the arrow operator `<-` for assigning values to objects, and to use the equal sign `=` to specify function arguments. For example:

```
newPerson <- createNewPerson(name = "John Doe", age = 45)
```

In the previous example, the function `createNewPerson()` is called and two arguments (`name` and `age`) are provided. The result of the function is then assigned to the object `newPerson`. Note that the values provided to the arguments were specified using the equal-sign `=` while the assignment of the resulting value to the newPerson object was donw with the assignmment arrow operator `<-`.

Tip: Use `option` + `-` in Mac or `alt` + `-` in PC and RStudio will type in the arrow operator for you. This is very convenient and saves time and keystrokes.

3.6 Comments

Use the pound or hash sign `#` to write comments in your code. Comments are non-executable pieces of writing that help explain your code.

Sometimes, it is useful to "comment out" R code that is not currently being used, or when you want intend to delete a piece of code but are not sure yet if it is the write decision. You can add # to the beginning of each line of code that you want to be skipped during execution.

Tip: To comment out large chunks of code at a time, select/highlight the code and then use `commmand` + `shift` + c (or `ctrl` + `shift` + c in Windows). Select the same code and press the same sequence of keys to un-comment the code later, if necessary.

3.7 Functions and arguments

A function takes inputs and returns a value - the result of the function. A function executes pre-written commands using the arguments (inputs) provided, and returns the result. For example executing the code `sum(1,3,5)` will return the value `9`.

Many functions have arguments that need to be provided explicitly. For example, the following function:

```
createNewPerson(name = "John Doe", age = 45)
```

Has two arguments that must be explicitly specified (written out) when ___calling*___ the function.

*Calling a function is the term used when a function is executed in the code.

3.8 Types of objects

3.8.1 Vectors

Vectors are the most common type of object in R, and consist of a group or sequence of values (in other programming languages, vectors are called 'arrays'). A vector is defined using the c() function. For example:

```
ageVector <- c(19,22,20,18,18,19,22,21)
```

In the previous code, I created a vector called ageVector by using the c() function to combine a group of values into a vector.

There are several R functions that help you get information on vectors (and other objects), such as:

- `length()`: Tells you the number of elements (distinct values) in the vector.
- `class()`: The type of data contained in the vector (such as numeric or string/character)
- `str()`: Detailed information about the structure and contents of the vector, including length and class, but other details as well.

3.8.1.1 Subsetting vectors

Subset vectors by using brackets [] after the object name and then specifying the index of the element/value you are interested in.

```
ageVector[1]
```

Returns the first value in the vector.

```
ageVector[c(1,2,4)]
```

Returns the first, second and fourth values.

```
ageVector[-1]
```

Returns all values *except* the first.

```
ageVector[-c(1,3,5)]
```

Returns all values *except* the first, third and firth values.

```
ageVector[2:5]
```

Returns all values between the second and fifth, inclusive.

3.8.2 Matrix

A matrix consists of rows and columns. All columns must be of the same length as well as all rows (Eg: cant have a row with 3 entries and another with 5). To construct a matrix, we use a function conveniently called matrix().

```
y <- matrix(1:20, nrow=5,ncol=4) # generates 5 x 4 numeric matrix
```

Subset a matrix with [row , column]:

```
y[,4]       # 4th column of matrix
y[3,]       # 3rd row of matrix
y[2:4,1:3]  # rows 2,3,4 of columns 1,2,3
```

3.8.3 List

Lists can have elements of any type and length. To construct one, use the list() function:

```
myl <- list(id="ID_1", a_vector=animals, a_matrix=y, age=5.3) # example of a list with 4 componer
myl[[2]] # 2nd component of the list
myl[["id"]] # component named id in list
```

3.8.4 Data frame

Data frames in R is like a mixture of lists and matrices. You have a set named columns each of which can be of different type (like in a list), but all columns must be of same length (like in a matrix). A way to think about a data frame is like a table in Excel and is commonly used to store tabular-type data.

Here is how you could construct a data frame.

```
mydf <- data.frame(ID=c(1:4),
                   Color=c("red", "white", "red", NA),
                   Passed=c(TRUE,TRUE,TRUE,FALSE),
                   Weight=c(99, 54, 85, 70),
                   Height=c(1.78, 1.67, 1.82, 1.59))

mydf
```

3.9 Missing data

R automatically codes missing data with the `NA` value. Datasets that have missing data can be processed with a variety of ways that will help you decide

how to deal with these missing values. Dealing with missing values is important when operating with rows and columns of data. For example, you might want to add all values in a row to get a total, but what happens if you have a bunch of NAs?

There are two main ways:

1. Using the **na.rm** argument. Many functions such as sum() and mean() have an argument called **na.rm** (basically: NA remove) which you can set to TRUE in order to ignore missing values. For example:

```
heightVector <- c(110, 120, NA, NA, 100, NA)

avgHeight <- mean(heightVector, na.rm = TRUE)
```

2. Use functions such as `na.omit()` or `is.na()`. For example:

```
heightVector <- c(110, 120, NA, NA, 100, NA)

avgHeight <- mean(na.omit(heightVector))
```

or

```
heightVector <- c(110, 120, NA, NA, 100, NA)

avgHeight <- mean(!is.na(heightVector))
```

Note the exlamation point before !is.na(), which makes sure that the function returns the values that are **not** NA. Otherwise, is.na() returns values that **are** NA. The exclamation point negates or reverses the operation.

3.10 Help and documentation

For general R help, look at the 'Help' tab in the Files and Miscellaneous pane. For quick help relating to any function or package, use the question mark `?` before the name of a function or package.

```
?sum
```

Will open up the documentation for the sum() function and tell you how to use it.

3.11 Extra topic: Naming and notation

Objects or variables can have practically any namme, but there are some rules:

1. Cannot start with a number, but can have numbers in the name.
    - `1time` is not valid, but `time1` and `time1result` are valid.
2. Cannot be one of key-words or reserved words in R. For example, terms such as `if`, `else`, `for`, and `function` already have specific uses in R and cannot be used for names of objects.
3. Must be descriptive, but succint.
4. Are case sensitive: `value1` is not the same as `Value1`, these are different objects.

In addition, there are several styles for naming and writing object names, especially if the name consists of more than one "English" word. For example, I previously referred to an object called `newPerson`. A different programmer maybe would have preferred to name it `new_person` instead. This is what we refer to when we talk about naming-styles or naming-conventions. The main styles are as follows:

- Camel-case: the first word is small-letters, but all other words start with a capital letter.
    - Eg: `newPerson`, `verySmallChild`, `totalSum`.
- Underscore sepatator: Uses underscores `_` to separate the words.
    - Eg: `new_person`, `very_small_child`, `total_sum`.
- All small: The entire name is in small letters.
    - Eg; `newperson`, `verysmallchild`, `totalsum`.

The important thing about notation styling is to just be consistent. I personally preferr to use `camelCase` for object and variable notation in programming language. However, when working with databases and the SQL language, it is recommended to use `underscore_separators` to name your tables and table-fields (columns). Therefore, when creating tables (or data frames) in R, I will typically use `underscore_separators` when naming the columns; but I will use `camelCase` for the names of any other kind of object. Eg:

```
salesTable <- data.table(value_one, value_two, total_sales)
```

The previous example is not really valid R code, but it illustrates the naming convention principles that were previously discussed.

3.12 Challenge

> **Challenge**
>
> 1. Create a data frame that holds the following information for yourself, your right and your left neighbor:
>
> - first name
> - last name
> - lucky number
>
> 2. There are a few mistakes in this hand-crafted `data.frame`, can you spot and fix them? Don't hesitate to experiment!
>
> ```
> animal_data <- data.frame(animal=c("dog", "cat", "sea cucumber", "sea urchin),
>                           feel=c("furry", "squishy", "spiny"),
>                           weight=c(45, 8 1.1, 0.8))
> ```