

Introduction to R

MSCI 523

January 12, 2016

Contents

1	Introduction	2
2	Installation	2
2.1	Installing R	2
2.2	Installing RStudio	2
2.3	R vs RStudio	3
3	The R environment	3
4	Working Directory	4
5	Performing Basic Operations	5
6	Creating Variables	5
7	Scripts	6
8	Deleting a Variable	8
9	Data Types	9
9.1	Vector	9
9.2	Matrix	12
9.3	Array	15
9.4	TS	15
10	Functions	15
11	Packages	17
11.1	Installing a Package	17
11.2	Loading a Package	20
11.3	Suggested Packages	22
12	Importing Data	23
12.1	.TXT files	23
12.2	.CSV Files	23
12.3	Excel Files	23
12.4	Other types of files	24

13 Exporting Data	24
13.1 .TXT files	24
13.2 .CSV files	24
13.3 Excel files	25
14 Workspace	25

1 Introduction

This tutorial serves as an introduction to R. R is a free statistical programming language that is widely used in the statistical community. The tutorial here assumes **NO** background in either R or programming, and aims at guiding you step-by-step to basic data manipulation functions. Most of the instructions will be accompanied by screenshots, in order to clarify any confusion that you might encounter. No forecasting will be present in this workshop; however it will start as of the next workshop.

2 Installation

The installation procedure is written for a Windows platform. It shouldn't differ much for the MAC or any Operating System.

2.1 Installing R

In order to install R, you need to go to the following link: <https://cran.r-project.org/bin/windows/base/>. On that web page, just click on the *Download R 3.2.2 for Windows* at the top as shown in the below figure.



Figure 1: How to download R

Download the installer by selecting Save. Once done, open the installer file, and The installation process should be straightforward.

2.2 Installing RStudio

After installing R, the next step is to install RStudio. This can be found at the following link: <https://www.rstudio.com/products/rstudio/download/> Once you reach the web page, scroll down to *Installers for Supported Platforms* as shown in Figure 2. There, choose the installer based on the Operating System (once again in this example the mouse is pointing to Windows as the tutorial is written on a Windows).

Installers for Supported Platforms

Installers

Click Here

RStudio 0.99.486 - Windows Vista/7/8/10

73.9 MB

2015-10-07

175330dd242b017a978f279b371dedfc

RStudio 0.99.486 - Mac OS X 10.6+ (64-bit)

56.2 MB

2015-10-07

12f9b2eb5dd4c2946fd73e9f9e3dd058

RStudio 0.99.486 - Ubuntu 12.04+/Debian 8+ (32-bit)

77.4 MB

2015-10-07

8512948585a904489eb1a3e4e029a1c8

RStudio 0.99.486 - Ubuntu 12.04+/Debian 8+ (64-bit)

83.9 MB

2015-10-07

6a9f9951f36c7272ccb1d5267670a5cf

RStudio 0.99.486 - Fedora 19+/RedHat 7+/openSUSE 13.1+ (32-bit)

76.8 MB

2015-10-07

52137c865858ebb4235fef63c009cf3c

RStudio 0.99.486 - Fedora 19+/RedHat 7+/openSUSE 13.1+ (64-bit)

77.7 MB

2015-10-07

df74f9ed1ab98bfff7bf9cf9c631af8166

Zip/Tarballs

Zip/tar archives

RStudio 0.99.486 - Windows Vista/7/8/10

105.5 MB

2015-10-07

4c059ede6a0f21958bd9276806cc4941

RStudio 0.99.486 - Ubuntu 12.04+/Debian 8+ (32-bit)

78.1 MB

2015-10-07

a09614f5244c569ac49a6ebb005b8072

RStudio 0.99.486 - Ubuntu 12.04+/Debian 8+ (64-bit)

84.8 MB

2015-10-07

9ab5a5ea8448f927d0ae6266346e380e

RStudio 0.99.486 - Fedora 19+/RedHat 7+/openSUSE 13.1+ (32-bit)

77.4 MB

2015-10-07

6ebfa84b3bdc7f2d7ada3005281e5639

RStudio 0.99.486 - Fedora 19+/RedHat 7+/openSUSE 13.1+ (64-bit)

78.4 MB

2015-10-07

faf082fb7e04c779f1a667afbb264b2e

Source Code

Figure 2: How to download RStudio

2.3 R vs RStudio

So now you have successfully downloaded and installed both R and RStudio. But what's the difference between these two? In a nutshell, R is the basic programming language that we'll be using. RStudio is an Integrated Development Editor, which in layman's terms is just a front end for R which is more user friendly as it has editing and debugging functionalities. Throughout this module, you will be using RStudio for the workshops.

3 The R environment

Now that the software is installed, you can open Rstudio. This is the screen you are greeted with:

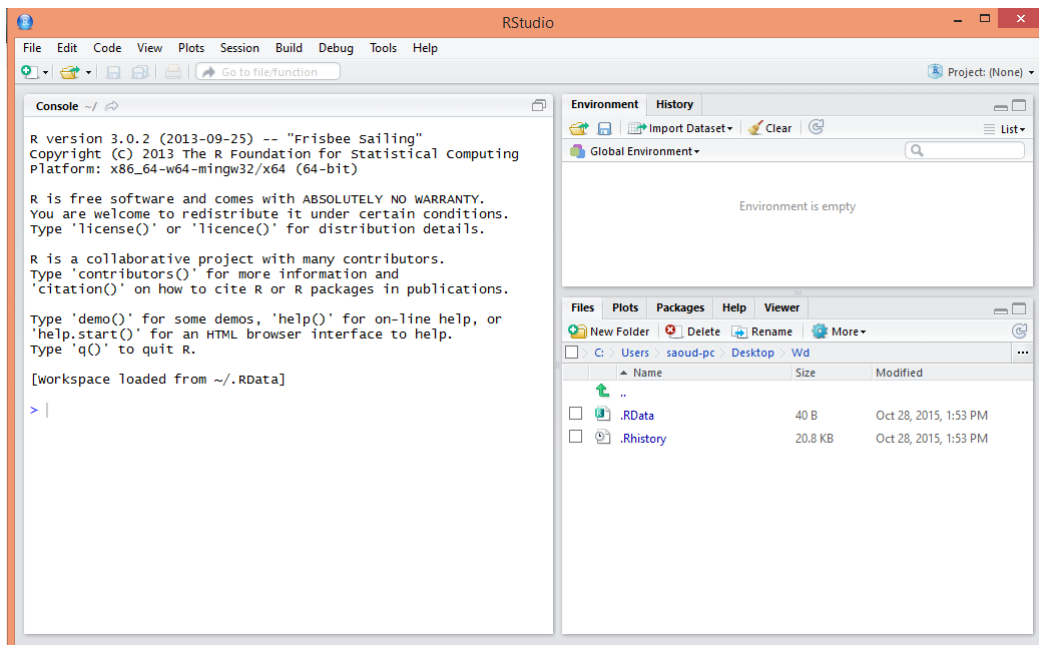


Figure 3: RStudio Screen

- **Console:** The **Console** section is at the left of the screen. Here you can type commands and statements, and you can also see the resulting output.
- **Environment:** At the top right, we have the **Environment**, also called **Workspace**. This is where the variables created are stored. Adjacent to the **Environment** tab is the **History** tab, which shows all the commands entered so far.
- **Files:** The **Files** tab is at the bottom right of the screen. Here you will see what is in our current directory (this will be explained in the subsequent section). Next to this tab you can find the **Plot** tab, where you will be able to see your plots. The other available tabs are the **Packages** tab, where all the packages can be seen (explained in Section 7) and the **Help** tab where additional information and help is displayed.

4 Working Directory

In order to start working on R, you need to specify where you want your data to be stored. This is called the working directory. For this example, create a New Folder on your Desktop and name it "R Working Directory". This is where you want all my R data to be located. Now in order to set this as your current directory, click on *Session* → *Set Working Directory* → *Choose Directory* as indicated in the below figure.

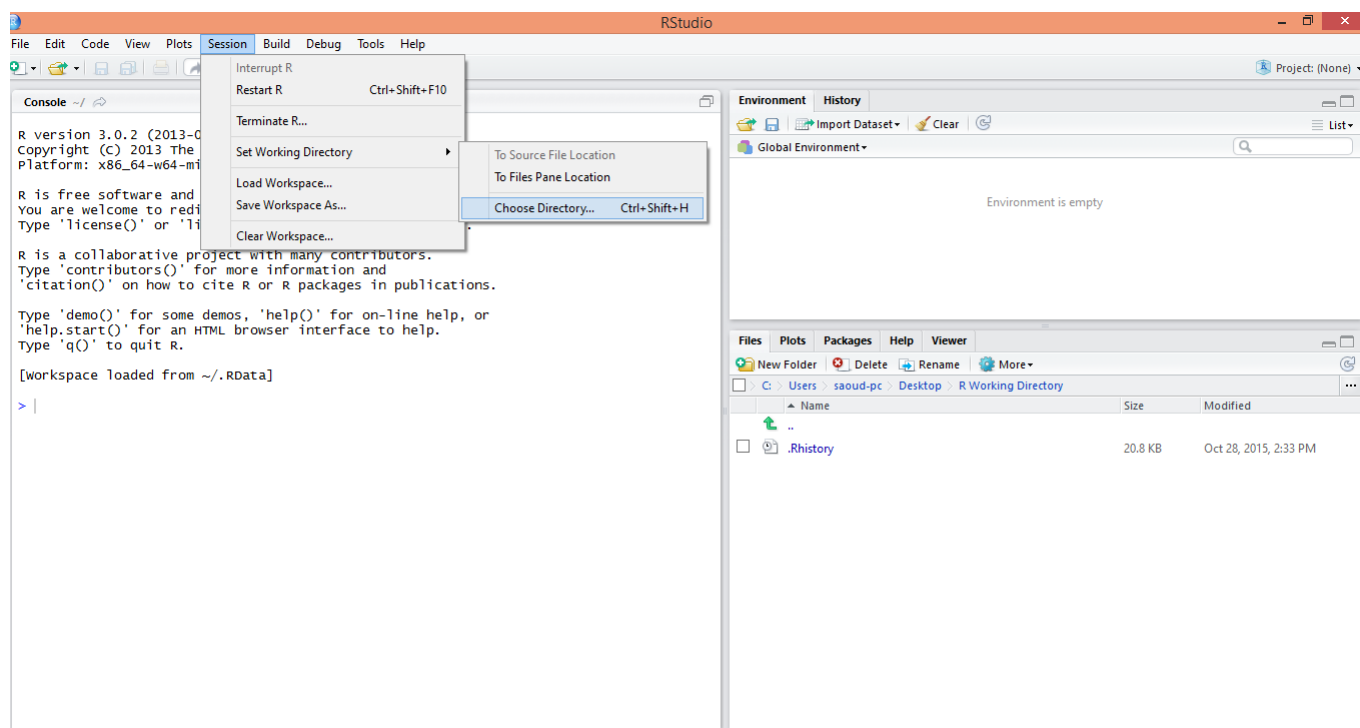


Figure 4: Changing the Directory

A window will open, and there you can look for the folder you want and set it as your directory. So once this menu will appear, go to the Desktop and select the "R Working Directory" folder you created.

5 Performing Basic Operations

In this section, you will see how to perform some basic calculations on R. This will serve as a prelude for the more advanced material that will follow later in the module. As a first example, try to add 5 and 2 together. In order to do that go to the console and type $5 + 2$. As you may notice the font is blue. After entering this statement, press Enter and in the following line you will get the answer (see below).

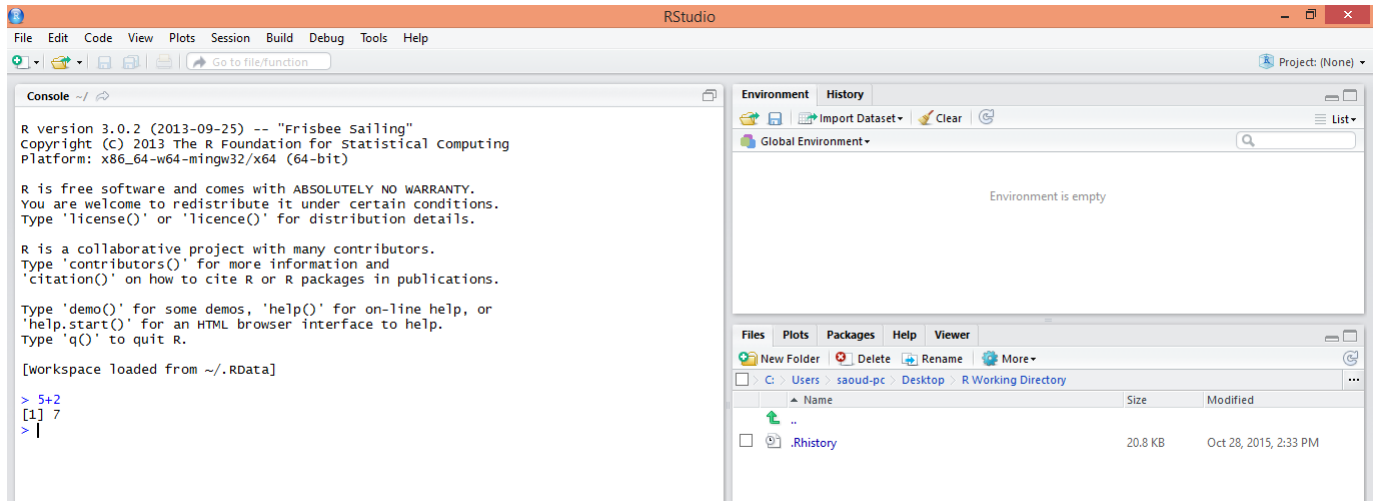


Figure 5: Addition in R

Similarly we can also do basic operations such as multiplication, subtraction or division. As explained before, the statement is entered in the console, and this is followed by the output.

6 Creating Variables

Given that this is a programming language, you can also create variables and store values in them. So create a variable x which will contain the value 100 by entering the following line:

```
x <- 100
```

If you look at the top right, in the environment box, you can see that a new variable has been created, which has the value of 100. By typing the formula the variable is created itself, without having to be defined, and the command is passed to store the value in it. The `<-` sign is what assigns the value to the variable. It is the equivalent of the `=` operator in many programming languages. In R, the `<-` sign is used interchangeably with the `=` sign, but both perform the same task. So if you enter for example:

```
x = 100
y <- 200
```

You will see that both commands are valid, as both variables are created in the workspace. The convention in R is to use the `<-` sign, and throughout the workshops.

In order to check if the variable x has been created, look at the environment (or workspace) box on the right. If the variable x appears, that means that the command has gone through. Another way to check would be to actually type x in the console. The following message will appear:

```
Console ~/
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[workspace loaded from ~/.RData]

> x=100
> x
[1] 100
>
```

Figure 6: Checking a Variable from the Console

Note: In order to write comments in R, simply start the sentence with the `#` sign. So for example if you type:

```
# My first comment
x <- 100
```

You should end up with a screen like in Figure 7. It is always good practice to write comments as you progress, as you are bound to forget later on why you have written a statement or what the function you are using does.

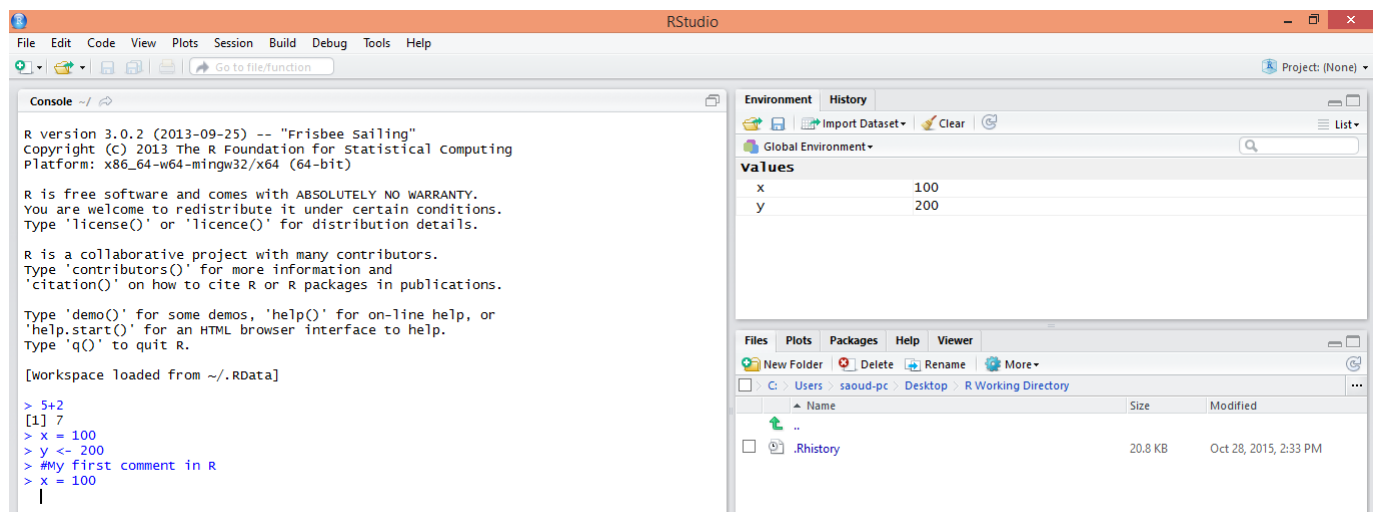


Figure 7: Variables in R

7 Scripts

Now that you have been introduced to the console and workspace, the next step would be to see how to create a script. Basically, in a script you will store several statements so you can run them all in one go, or run chunks of the code, depending on the task. The advantage of the script is that you get to save the code you entered, so that you don't have to write it again the next time you use R.

In order to create a New Script, go to *File* → *New File* → *R Script* as shown in Figure 7, or simply press *Ctrl+Shift+N*.

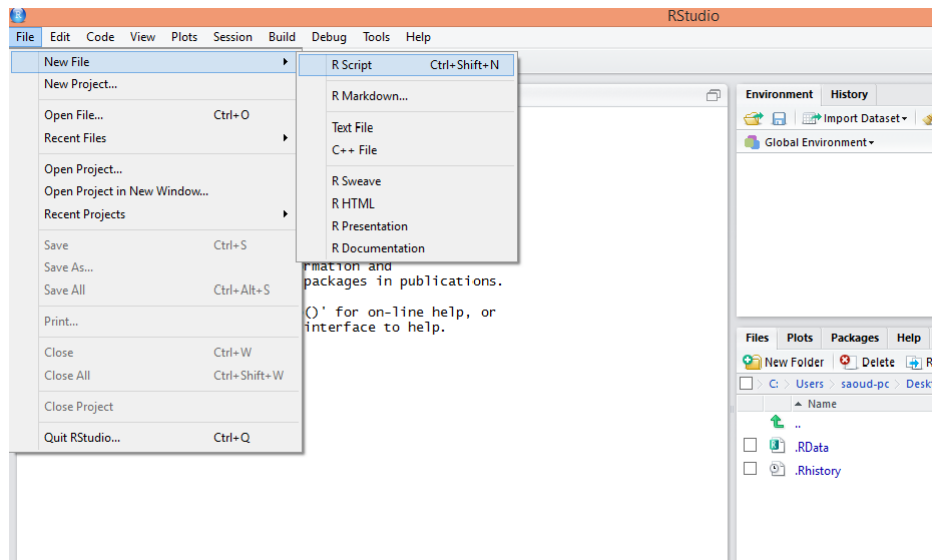


Figure 8: Create a New Script

The screen should look something like in Figure 9

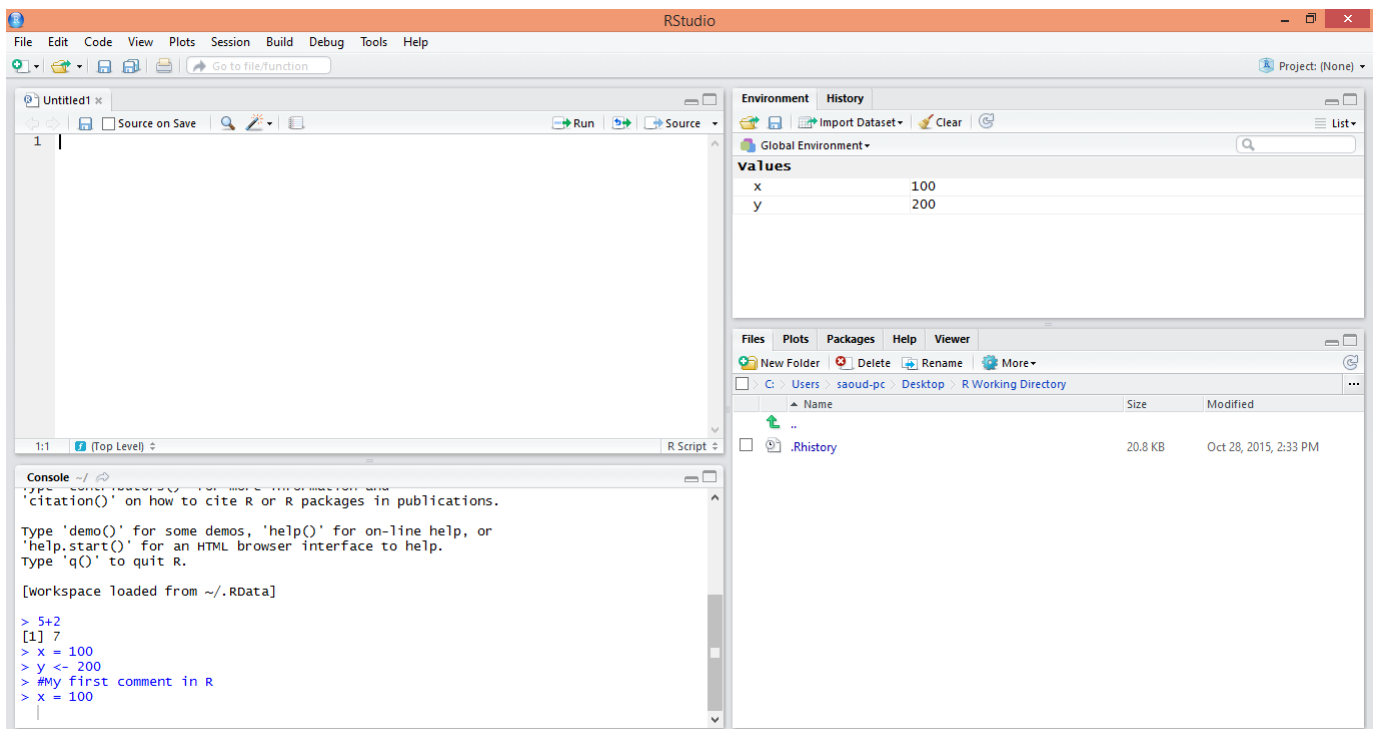


Figure 9: New Script

Now that the script has been, type the following lines in the **SCRIPT** as shown in the figure below:

```

#My first script
a <- 20
b <- 30
c <- a + b
  
```

Remember that the `#` sign represents a comment and is **NOT** an actual executable statement! Now if you press *Enter*, notice that nothing will happen. This is because executing a statement from the script is different than executing it from the console. In order to run these lines from the script, highlight all the lines of code you want to run (as shown in Figure 10), and press *Ctrl+Enter*.

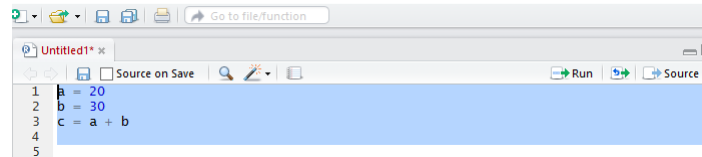


Figure 10: Highlight

Now you can observe in the **Workspace** that all variables *a*, *b* and *c* have been created. Now that the script has been created, you can open it or save your progress.

- **Saving:** It is always a good practice to always save your progress as you go along. To save your script either go to *File* → *Save* or press *Ctrl + S*.
- **Opening:** Once you have saved your script, you can always continue your work later on. In order to load your script go to *File* → *Open File* or press *Ctrl + O*.

8 Deleting a Variable

Sometimes you notice that you no longer need a variable. In our case, suppose you no longer want the variable *c* any more. In order to erase it from the **Workspace**, you will need the *rm* command as follows:

```
rm(c)
```

After entering the statement in either the console and pressing *Enter*, or in the script and highlighting the statement and pressing *Ctrl + Shift + Enter*, look at the **Workspace**. You will see that the variable *c* no longer exists.

Similarly, if you want to get rid of all the variables, just click on the Clear button (the one with the brush icon) on the right as shown in Figure 10. Now you can see that all variables have been removed.

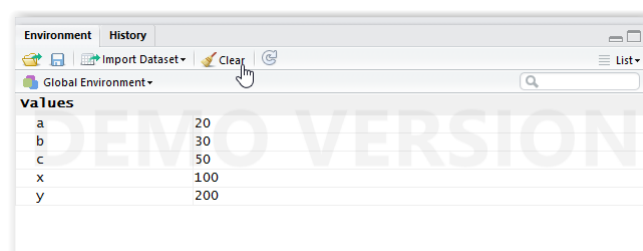


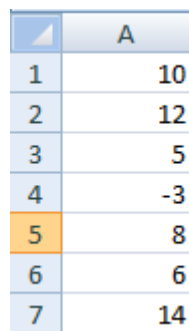
Figure 11: Clear all Variables

9 Data Types

Now that you have seen how to create variables, it is time to introduce the data formats that exist in R. Given that many data types exist in R, and that this is a forecasting module and not an R module, the three most encountered data types will be discussed in this section: vector, matrix and ts.

9.1 Vector

In programs such as Microsoft Excel or SPSS, the data is stored in a spreadsheet or table, and you enjoy full visibility over it. You have the data as a list of numbers which follow a certain order, and using that as input you generate a desired output based on the underlying task or requirement. For example, suppose your data consists of the following seven values: 10, 12, 5, -3, 8, 6 and 14. In Excel, the data will look something like this :



	A
1	10
2	12
3	5
4	-3
5	8
6	6
7	14

Figure 12: Excel Version

In R, the data is not displayed in tables or spreadsheets as in Excel. This is because R is a statistical programming language, which in contrast with Excel, lacks the visibility of the latter. Instead the values are stored in the background, under a variable name. In Excel, the data was entered by typing the values in each cell. In R, they will be entered as the values that a variable takes. Go to the console, and type the following:

```
data <- c(10,12,5,-3,8,6,14)
```

Note that the vector is not a row or column vector, as the vector type in R is only defined by the number of entries. Now you will see that a variable called *data* has been created, and that it contains seven entries. This syntax is telling R to create a variable that contains numerical entries, which are the seven numbers. In the **Workspace**, you will see next to the values of data *num[1:7]*. The *num* indicates that your vector contains numerical data, while the [1:7] indicates that your contains 7 entries.

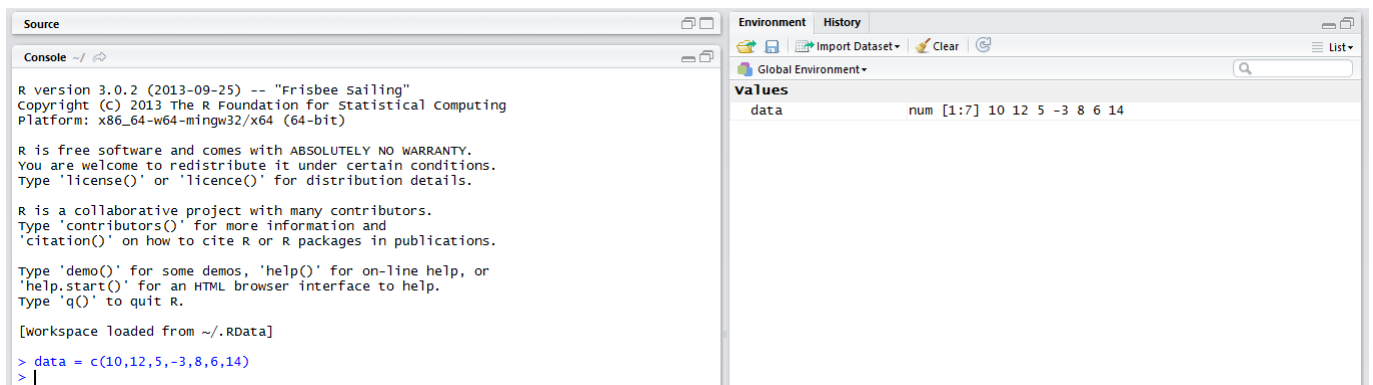


Figure 13: R Version

If you want to see the values on a separate sheet, go to the console and type the following:

```
View(data)
```

The function is called *View*, and it only works if the first letter is in **Capital**! So typing *view* instead will return an error message saying that the function does not exist.

The variable `data` that has been created is a numeric vector, and this is the building block of the data types in R. You can use a few functions to handle the vector at hand.

- **Length of the Vector:** Suppose you are interested in knowing the size of the vector. By looking at this example, you can tell that the data contains seven elements. However, you might need later on to know the length of the vector for other purposes, one of them being that it will be used as an input in a certain code. In order to determine the length, use the *length* function.

```
length(data)
```

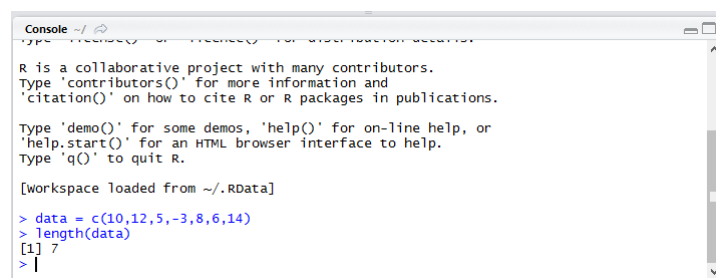


Figure 14: Length of the Vector

- **Creating a vector of consecutive numbers:** Suppose you wanted to create a new vector, `data1`, that contains seven consecutive numbers, such as from 1 to 7. You could type them as:

```
data1 <- c(1,2,3,4,5,6,7)
```

However, there is a shorter way for consecutive items, which is just to use the following syntax:

```
data1 <- 1:7
```

By using `:`, you are asking R to create a vector that starts at 1 and ends at 7, by jumps of 1. This wouldn't work if you wanted a different interval length, say increments of 0.5 instead of 1.

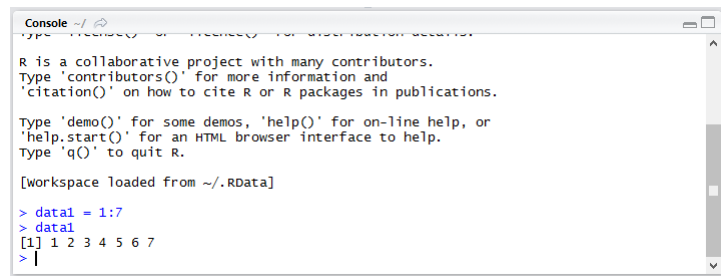


Figure 15: Vector of consecutive numbers

- **Returning an Item from the Vector:** Suppose now that you are only interested in the first item of the *data* vector. This can be done by using the following syntax:

```
data[1]
```

This function will return the first item of the vector. For the second item, the statement would change to:

```
data[2]
```

This function is the counterpart of the INDEX function in Excel, if you are interested in finding one value only. However in some cases, you might be interested in the first three items for example, then the statement would be:

```
data[1:3]
```

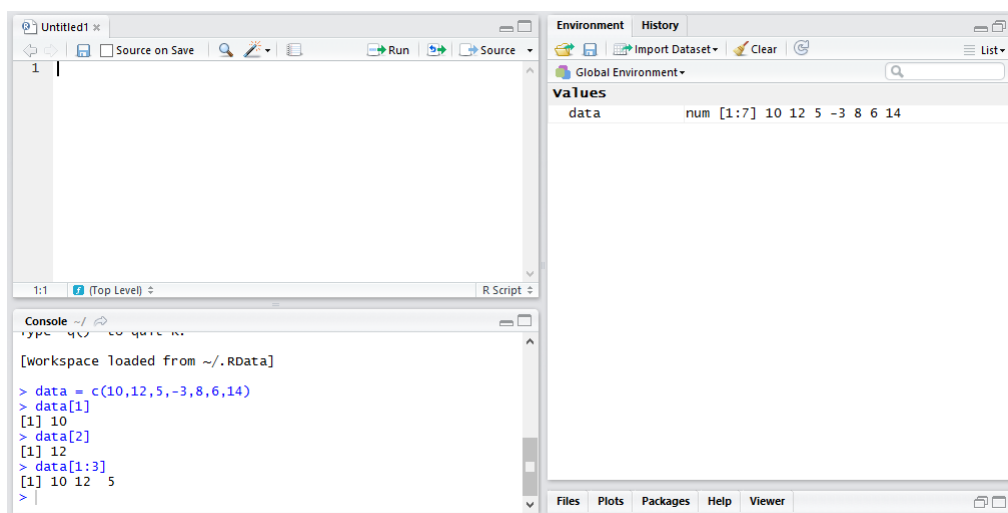


Figure 16: Returning an Item from a Vector

- **Deleting an Item from the Vector:** If you need to eliminate an entry from the vector, e.g. the last element (which corresponds in this example to the seventh entry), then the appropriate statement is:

```
data[-7]
```

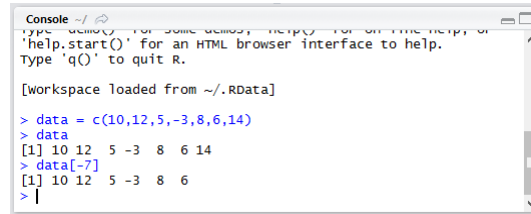


Figure 17: Deleting an Item from a Vector

This will **NOT** delete the point from the variable, but will show you the output if the point was deleted. In order to alter the actual variable, you need to reference it as such:

```
data <- data[-7]
```

This is similar to the syntax you used to find an element; however the `-` sign has appeared now. Always be careful when deleting an element!

- **Other types of vectors:** So far you have seen a numeric vector. But other types of vector do exist such as character (or string) vectors and logical vectors (TRUE or FALSE). However, these are not very relevant to the tasks in the module, and so they will be overlooked.

9.2 Matrix

As seen in the previous section, the vector type is a good way of storing data for a certain variable. However, the vector has one dimension. Sometimes, you might be required to create a two-dimensional vector, where you will need rows and columns. This feature is not permitted in the vector type. This is where the second type of data is needed: the Matrix type. A matrix is defined by the number of rows and columns it possesses. The Matrix can be seen as a concatenation of vectors of the same size, or in other words the vector can be seen as a Matrix with either one row or one column.

The syntax for a matrix is:

```
matrix(data, nrow, ncol, byrow, dimnames)
```

The *data* argument specifies the data that will be stored inside the matrix. *nrow* specifies the number of rows in the matrix, while *ncol* specifies the number of columns. The *byrow* is a logical argument which is by default set to FALSE; it determines whether the matrix is filled with columns by default or by rows. For the scope of this class, this won't be used in the workshops. The final argument is the *dimnames*, which allows you to name your rows and columns. This becomes useful when your matrix is big, and you would like to name for example each column as a variable. This point will be elaborated in later workshops.

Practical Tip: When creating a matrix, a good practice is to always set the data to be handled as missing values, and to specify the dimensions of the matrix first. This is done by

setting the data to **NA**, which will populate your matrix with **NA** values. The **NA** value can easily be replaced later on. Any value of NA does not affect your matrix, this is the standard way of saying that a value is Not Available, hence the name NA.

So how do the previously introduced functions for a vector differ from those of a matrix? The differences are simple yet intuitive, and in order to illustrate them better, they will be applied to the following example: **Matrix Example:** For example, if you want to create a 4x3 matrix, which contains in the first column values from 1 to 4, in the second column values from 10 to 13, and in the third column values from 35 to 38. This would look like this:

$$\begin{bmatrix} 1 & 10 & 35 \\ 2 & 11 & 36 \\ 3 & 12 & 37 \\ 4 & 13 & 38 \end{bmatrix}$$

Here is how it could be done on R:

```
#Create the empty Matrix called m
m <- matrix(NA, nrow = 4, ncol = 3)
```

Up till here, you have just created a blank matrix, and specified its dimensions. The matrix is blank as it is filled with NA. The next step consists of filling up these entries.

```
#Populate the first column
m[,1] <- 1:4
#Populate the second column
m[,2] <- 10:13
#Populate the third column
m[,3] <- 35:38
```

So now the matrix has been set up, the differences in the functions can now be highlighted.

- **Length:** The length value now no longer indicates the dimensions of the matrix; instead it shows how many entries are contained inside the matrix. For the example, if you enter:

```
length(m)
```

The answer will be 12, which counts all the entries in the matrix. When retrieving an item from a matrix, you now need both the row and column, instead of the position. In order to find the dimensions of the matrix, the function you should use is *dim*, which will return the rows and columns. The *nrow* function will return the number of rows, while the *ncol* will return the number of column.

```
#Find the dimensions of the matrix
dim(m)
#Find the number of rows
nrow(m)
#Find the number of columns
ncol(m)
#Retrieve the names of the rows and columns of m
dimnames(m)
```

NOTE: The *dim* function returns a vector containing the number of rows in the first entry and the number of columns in the second entry. So an alternative way to find rows and columns would be to retrieve either the first or the second entry as you have done for the vector

```
#Find the dimensions of the matrix
dim(m)
#Find the number of rows
dim(m)[1]
#Find the number of columns
dim(m)[2]
```

- **Returning an item:** If you want the first item of the first row and first column then type:

```
m[1,1]
```

Writing the matrix name, followed by a bracket, indicates that you are retrieving some items from the matrix. The first number between the brackets refers to the row number, while the second number refers to the column number. For example:

```
#Retrieve the element of the second row and first column
m[2,1]
#Retrieve the element of the first row and second column
m[1,2]
#Retrieve the element of the second row and second column
m[2,2]
```

If you are interested in getting more than one element, i.e. a vector or a sub-matrix here are the procedures:

```
#Retrieve the first row entirely
m[1,]
#Retrieve the first column entirely
m[,1]
```

Keeping a blank where the column number must be in the brackets means that you want all the columns to be selected. The same applies for the rows.

If you wanted a sub-matrix from *m*, say a matrix composed of the first two rows and two columns of *m*, then the statement would be:

```
m[1:2,1:2]
```

Equivalently, this statement can also be written as:

```
m[c(1,2),c(1,2)]
```

As `1 : 2` is the same as `c(1,2)`, both of them creating a vector having values 1 and 2!

Notice that many similarities exist with the statements used for the vector; however you need to bear in mind that a matrix has a different structure, thus different requirements.

NOTE: Always make sure that any command you pass on the matrix is one that can actually be executed! For instance, in the example of the matrix *m*, asking for the element in the fifth row is impossible since you only have 4 rows!

- **Deleting an item:** Given that the data is now a matrix, deleting an item isn't as elementary as before. This is because the matrix has a structure defined by the number of rows and columns. So in your example, if you were to delete the first entry as such:

```
m[-1]
```

You will notice that you no longer have a matrix but a long vector. This is because the previous structure of the matrix no longer applies.

```

> #Populate the second column
> m[,2] = 10:13
> #Populate the third column
> m[,3] = 35:38
> dim(m)[1]
[1] 4
> m
      [,1] [,2] [,3]
[1,]    1    10    35
[2,]    2    11    36
[3,]    3    12    37
[4,]    4    13    38
> m[-1]
[1] 2 3 4 10 11 12 13 35 36 37 38
>

```

Figure 18: Deleting an item from a matrix

If however you want to delete a specific column or row, these are the statements to execute:

```

#Delete the second row
m[-2,]
#Delete the second column
m[, -2]

```

9.3 Array

The Matrix data type is specified by its number of columns and rows. In that sense, it is a two-dimensional type of data. However, sometimes you will need to exceed these two dimensions when storing data.

The Array type allows you to store in 3 or more dimensions. Given that you no longer have exclusively rows and columns, its syntax changes. In order to do so, create a 3-dimensional array of dimensions 3x4x5 as such:

```
my_array <- array(NA, dim = c(3,4,5))
```

The *dim* statement has now been used to assign the dimensions of the array.

9.4 TS

TS stands for time series, and it is a type of matrix with special properties such as a seasonal frequency. This will be discussed in the next workshop, when the actual forecasting starts.

10 Functions

So far you have seen R perform certain functions such as being a calculator or creating variables and vectors. These functions are embedded within the software, as they have been implemented by the developers of R. So these are the basic functions found in R. A wide variety of basic functions are included in R, as it is a statistical computing programming language. Below are some useful functions that you might use throughout the workshops.

```

#Create the data
data = c(10,12,5,-3,8,6,14)
#Find the minimum value
min(data)
#Find the maximum value
max(data)
#Find the average
mean(data)
#Find the median
median(data)
#Find the variance
var(data)
#Find the standard deviation
sd(data)
#Find summary statistics (minimum, maximum, mean, median, 1st and 3rd quartile)
summary(data)

```

Note that all these functions are for the sample and not for the population. If you were confused about what a function does, and what arguments it takes, then do the following:

1. Click on the **Help** tab on the right.

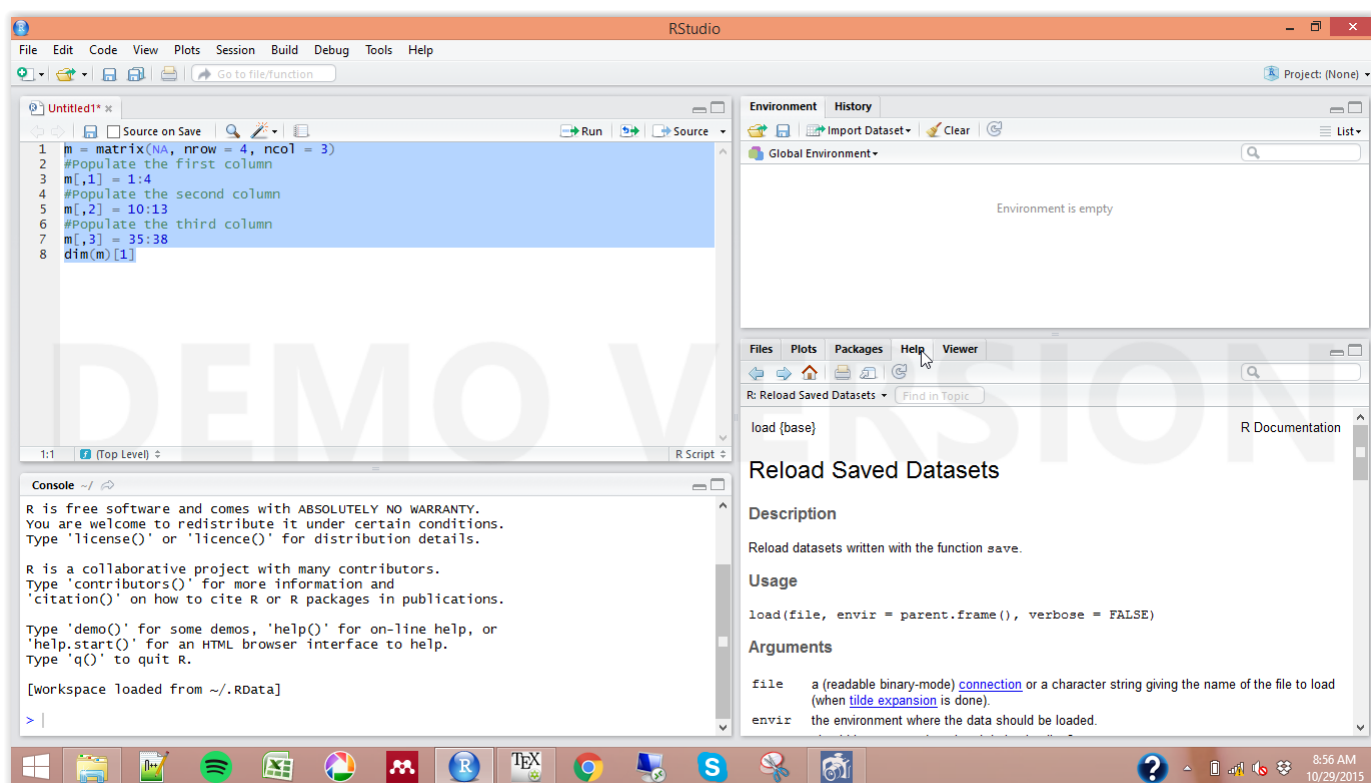


Figure 19: Click on the Help Tab

2. Type the name of the function in the search bar. For example, you want to explore the *matrix* function. So type matrix and click on the drop down or press Enter.



Figure 20: Type the function name

3. Now you will see the help page for the *matrix* function.

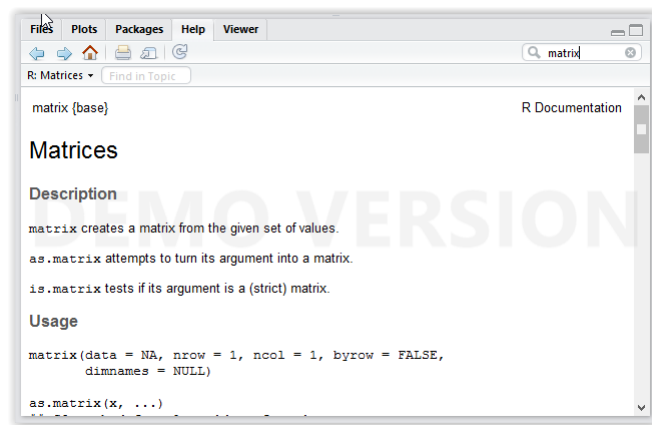


Figure 21: Help page screen

11 Packages

Some other functions will be introduced in the workshops, which are essential for forecasting time series. Those functions are not built-in; rather they have been developed by researchers separately who have then compiled them into packages and shared them for all users. One such package is the *forecast* package.

11.1 Installing a Package

In order to install the *forecast* package, do the following:

1. Click on the **Packages** tab on the right in the lower pane.

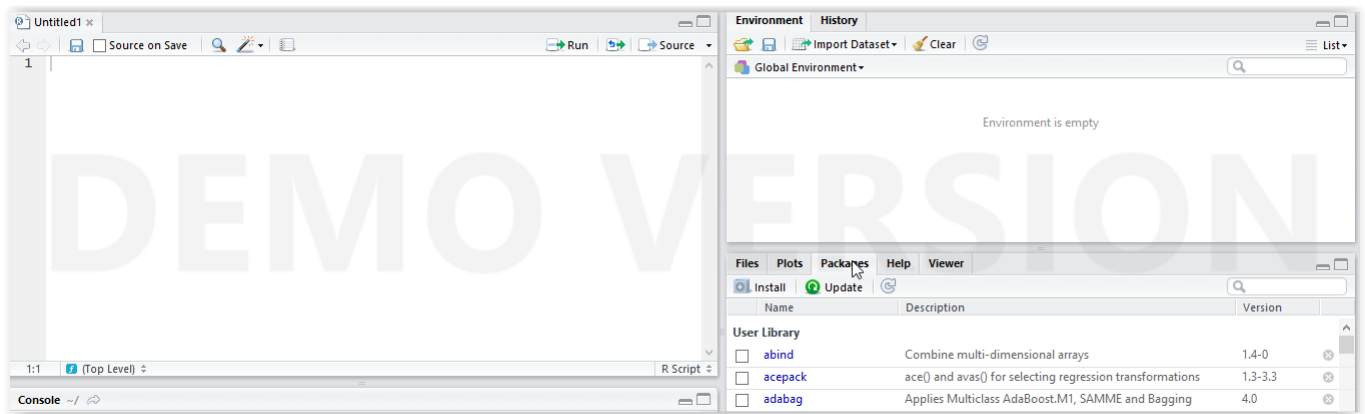


Figure 22: Click on the Packages tab

2. Click on the **Install** button

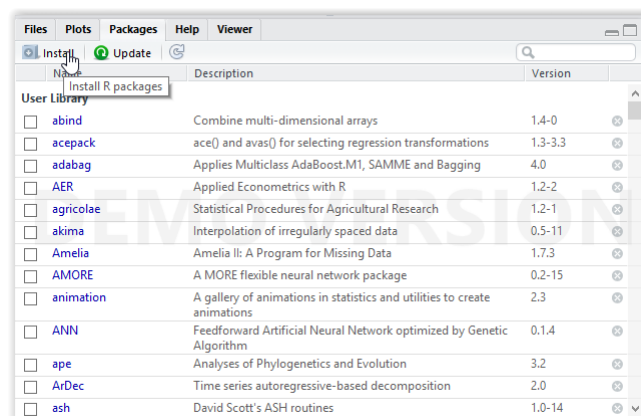


Figure 23: Press Install Button

3. A pop-up menu will appear, prompting you to enter the name of the package you wish to install.

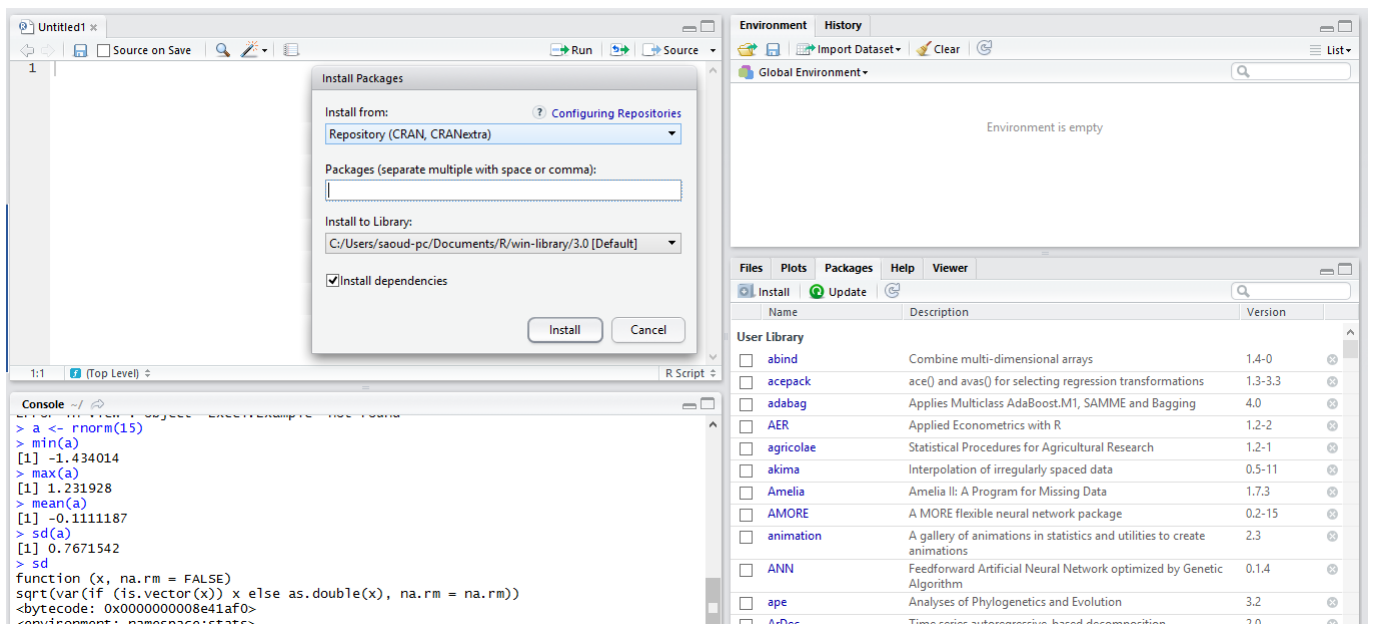


Figure 24: Install Pop-Up Menu

In **Packages** field, type the name of the package (forecast). You will notice that as you type along an auto-fill of the available package names will appear. Select the forecast and click on **Install**.

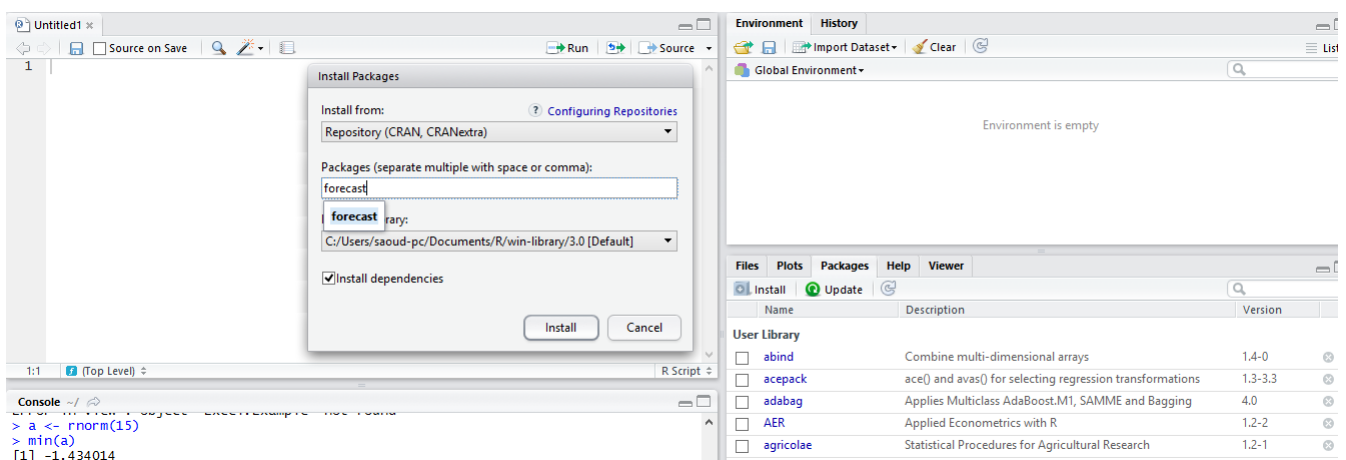


Figure 25: Typing the package name

Now the package will install, as you will see statements in red and black in the console. Those statements are displaying the progress of the installation. On the top right edge of the console box, you will see a **STOP** sign. This means that the installation is still taking place. Once this sign disappears that means that the installation is done and you can start working.

```
Console ~/\n\nThere is a binary version available (and will be installed) but the\nsource version is later:\nbinary source\nforecast 5.9 6.2\n\ntrying URL 'http://cran.rstudio.com/bin/windows/contrib/3.0/forecast_5.9.zip'\nContent type 'application/zip' length 1211623 bytes (1.2 Mb)\nopened URL\ndownloaded 1.2 Mb\n\npackage 'forecast' successfully unpacked and MD5 sums checked\n\nThe downloaded binary packages are in\nC:\\Users\\saoud-pc\\AppData\\Local\\Temp\\RtmpozzBmI\\downloaded_packages
```

Figure 26: Stop Button

11.2 Loading a Package

Here you will see how to load a package. So far, you have installed the *forecast* package. Installing it does not mean that you have loaded it! You have to load the package every time you launch R. In order to do so:

1. Click on the **Packages** tab, just like when you were installing the software.

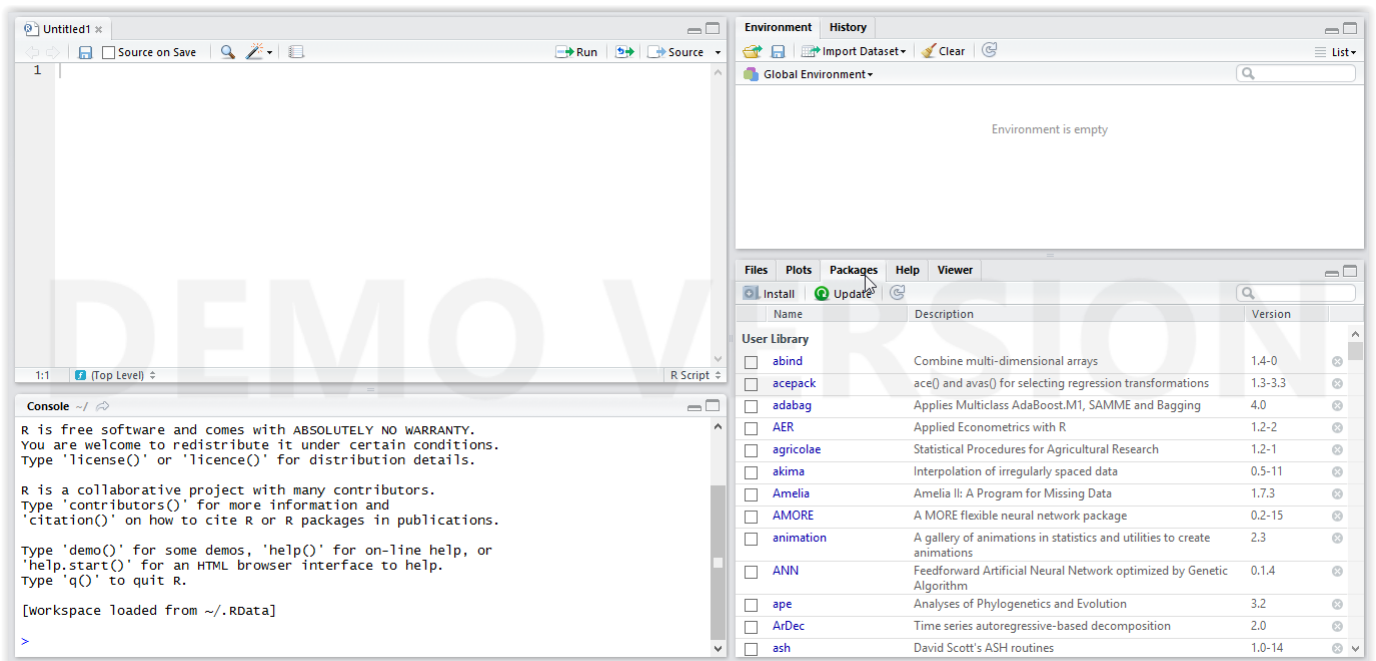


Figure 27: Click on the Packages Tab

2. Click on the search bar at the right.

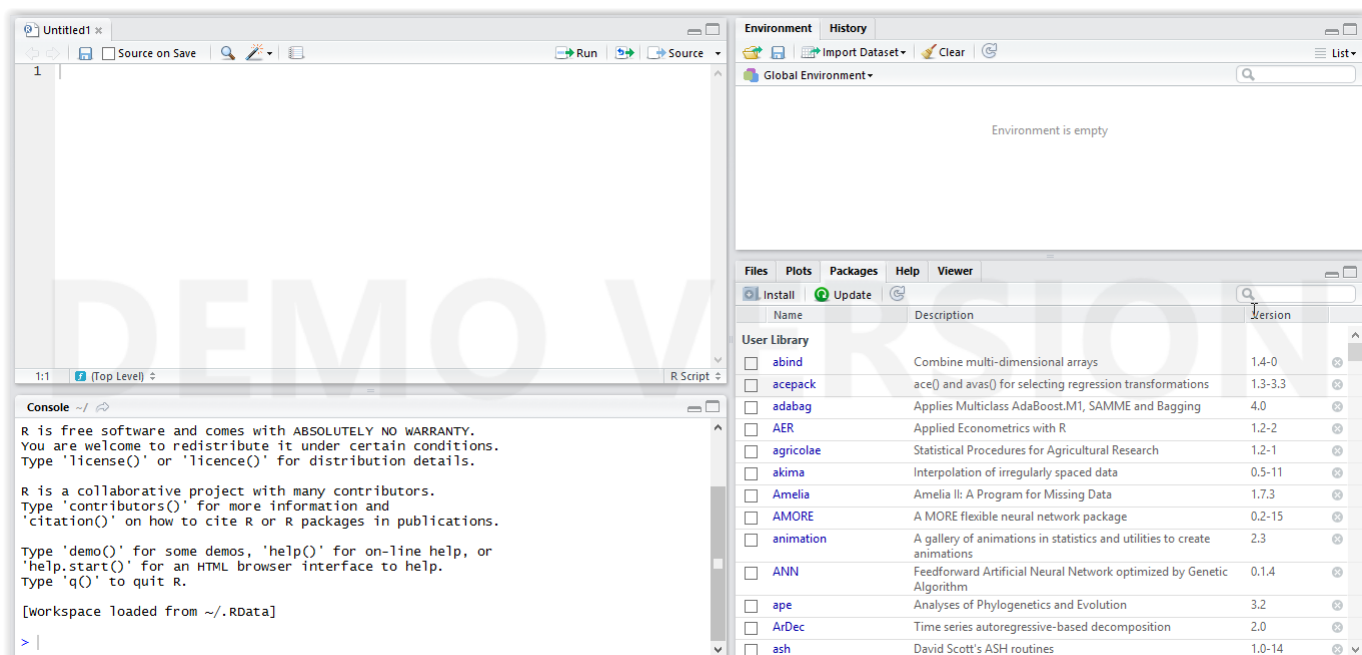


Figure 28: Click on the search bar

3. Start typing the name of the package (i.e. forecast). The list of packages will narrow down.

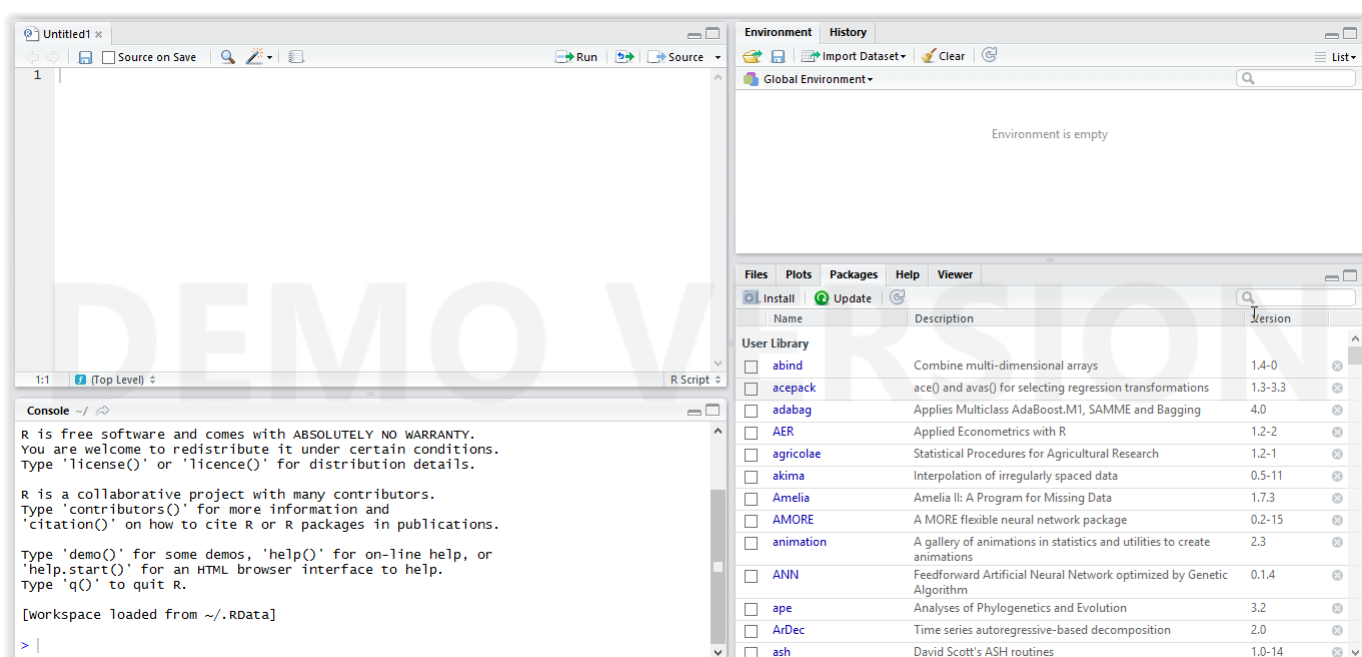


Figure 29: Type the package name

4. Once the *forecast* package appears, tick the box next to it. This will load the package.

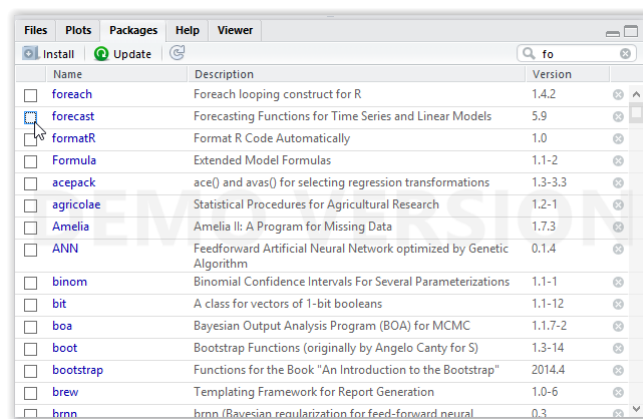


Figure 30: Tick the forecast box

And now the package is installed. Alternatively, you could launch the package by executing a simple statement in the console or in the script. To do so, just type:

```
#Loading the forecast library
library("forecast")
```

NOTE: Every time you start an R session, you have to load the packages you want. The package that you have loaded will be unloaded once you exit R, and so you have to do it **EVERY** time you run R.

11.3 Suggested Packages

Throughout the module, several packages will be used, each containing specific functions needed for certain tasks. The packages are:

- forecast
- MASS
- Leaps
- tsintermittent
- hts
- MComp
- fpp
- devtools

Install these packages so that you can use them. Some packages can't be installed directly through R, as they only exist in Github, which is a code repository. This is the case of the "TStools" package, which will be used in future workshops. In order to install it, simply enter the following statement in the console:

```
devtools::install_github("trnnick/TStools")
```

This package is one of the two packages that will be used in the next workshop.

NOTE: During the installation, you might encounter an error message that notifies you of the failure of the installation. A simple trick that circumvents this issue is to reinstall the "RCP" package. So in case you are faced with this problem, just go to *Packages* → *Install* and type "RCP". Furthermore, for Mac Users, follow this link:

<http://www.thecoatlessprofessor.com/programming/rcpp-rcpparmadillo-and-os-x-mavericks>

12 Importing Data

So far, you have seen how to create variables that will store data. However, the data can be long and it can be quite tedious to manually enter every single point. Instead, the data can be loaded to R, and you can then store it in a variable. Attached to the instructions are three files of different types containing the same data. One file is a *txt* file, another is a *csv* file and the third is an Excel file. You will see how to import from these three types of data. Each type of data in the attached files has two files: one that contains a header and another that doesn't.

In order to import your data, first make sure you copy it to your working directory. If you are not sure where your working directory is, follow the instructions from Section 4 and set the directory to where you want it. Once this has been done,

12.1 .TXT files

First start with the data that does not contain any headers. In order to load it and store it into a variable called *data*, type the following:

```
data <- read.table("Example (no header).txt")
```

Notice that the name of the file, as well as the format are included between double quotes. Now you can see that a new variable called *data* has been created.

If your data contained a header (a title), such as in the second ".txt" file, the following statement should handle it:

```
data <- read.table("Example (with header).txt",header=TRUE)
```

12.2 .CSV Files

CSV stands for Comma Separated Values file, which are plain-text files. This type of file enables data to be stored in a table structured format. This extension can be opened from many programs, such as Notepad or Excel, and is often used to import data into any spreadsheet or database.

Similarly for a ".csv" file, type the following:

```
data <- read.csv("Example (no header).csv")
```

Notice that the name of the file, as well as the format are included between double quotes. Now you can see that a new variable called *data* has been created.

If your data contained a header (a title), such as in the second csv file, similarly to before amend the statement so it looks like this:

```
data <- read.csv("Example (no header).csv",header=TRUE)
```

12.3 Excel Files

Many times the data might be in an Excel format. For Excel versions 2007 and newer ones, the file extension would typically be ".xlsx". When confronted with this format of data, there are two ways to handle it. The first way would consist of going to Excel and saving the document as a ".txt" or ".csv" and repeating the above procedure. This is done by opening the Excel file, and clicking *SaveAs* and selecting the desired format.

The second way would be to use a function from the package *openxlsx*. The first step is to install the package. Go to the **Packages** tab on the right, click on the **Install** button on the left, and type *openxlsx*. After the installation is complete, you can now use the *read.xlsx* function as follows

```
data <- read.xlsx("Example (no header).xlsx")
```

For more complicated excel files, such as many sheets or many rows and columns, refer to the **Help** page of the *read.xlsx* function, as explained in section 10.

12.4 Other types of files

Of course these three types are not the only types of data available. The data you can encounter might be an SPSS, SAS, Eviews file, or it can any other type. If you are interested in more data types, follow this link: <http://www.statmethods.net/input/importingdata.html>

13 Exporting Data

Suppose you have finished your session on R, and decided to export your data from R. One reason for this would be to exploit the better graph quality found on SPSS or Excel, where you could produce more colorful plots. Just like in the case of importing data, you need to specify what type of file do you want the actual data to be exported to. In order to do that, let's first create a vector of numbers.

```
#Create a vector so we can export it  
v <- 1:10
```

Now that the variable has been created, it can be exported to the type of file desired

13.1 .TXT files

For .txt files, the command to be executed is the following:

```
write.table(v,file="example.txt")
```

This command will export your data to the .txt file, as well as the names of the rows and columns. The .txt file will be saved on your working directory by default. Notice that the extension of the file (.txt) is specified at the end. If these names are not wanted, then the above statement should be amended to the following:

```
write.table(v,file="example.txt",row.names = FALSE, col.names = FALSE)
```

Here you specify that the names should be omitted, and now the .txt file just contains the data itself.

13.2 .CSV files

For .csv files, similarly to its .txt counterpart, the command to be executed is the following:

```
write.table(v,file="example.csv")
```

Again the extension is specified at the end. Similarly, to keep the names out, change the statement to:

```
write.table(v,file="example.csv",row.names = FALSE, col.names = FALSE, sep=",")
```


13.3 Excel files

In order to export the data directly to an Excel sheet, you need to load to use the *openxlsx* package. The first step is to install the package. Go to the **Packages** tab on the right, click on the **Install** button on the left, and type *openxlsx*. After the installation is complete, you can now use the *write.xlsx* function as follows

```
data <- write.xlsx(v,file="example.xlsx")
```

14 Workspace

So far, you have seen how to import and export data to and from R. But sometimes, you might be running a big session, where you have executed several lines of statements. Your **Environment/Workspace** will be filled with variables. Now you might want to stop for the day, and pick it up on a next session. Obviously you can rerun your whole script again. But another alternative does exist, which is to export the **Workspace**.

The **Workspace** is where all the variables are stored. Exporting it would mean that you could reload it later on, and spare yourself the trouble of waiting for the code to run itself again. In order to see how this can be done, first create some random variables for example:

```
x <- 1000  
y <- 20  
c <- 1:6
```

Now you have some variables that fill up your **Workspace**. In order to export it, take your mouse first to the **Environment/Workspace** box, then press on the Save icon that resembles a floppy disk. A menu will open, asking you where you want to save your **Workspace**, and what do you want to call the file. In this example, the **Workspace** is stored in the working directory, and is called "Workspace". Click *Save*, and the workspace is now saved. If you go to the destination folder, you will see a file called "Workspace", that has a big blue R logo. This is the file you just saved.

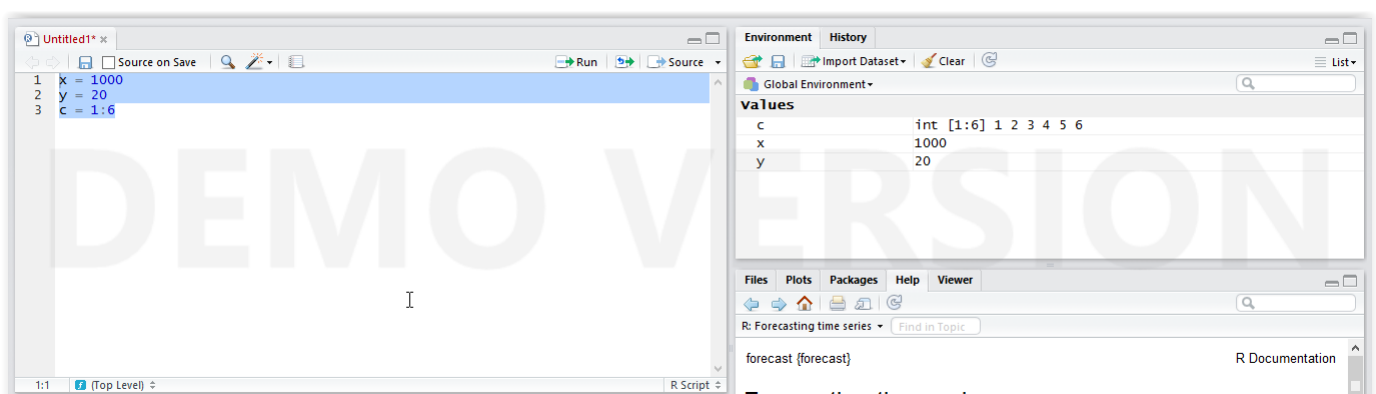


Figure 31: Go to the environment box

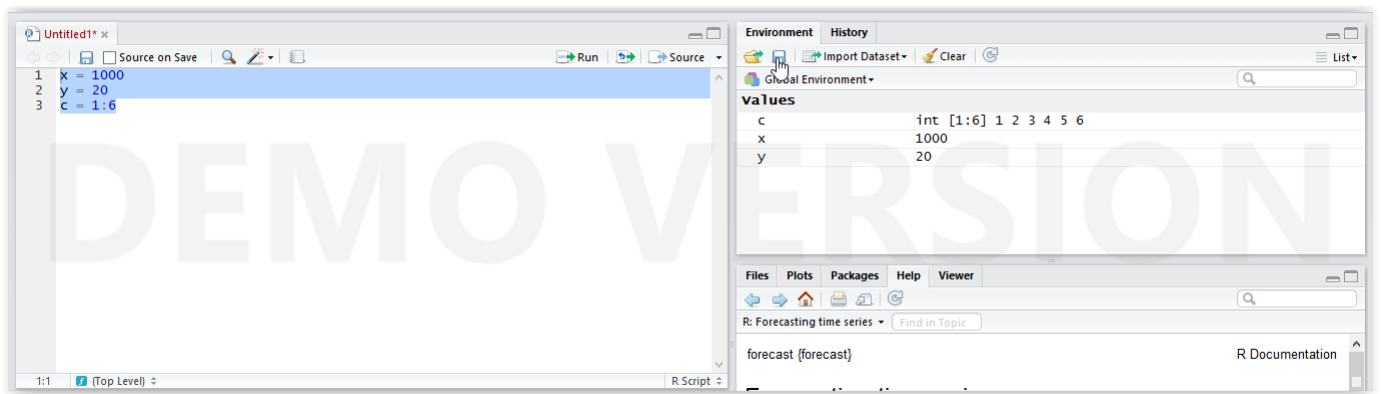


Figure 32: Click on the Save button

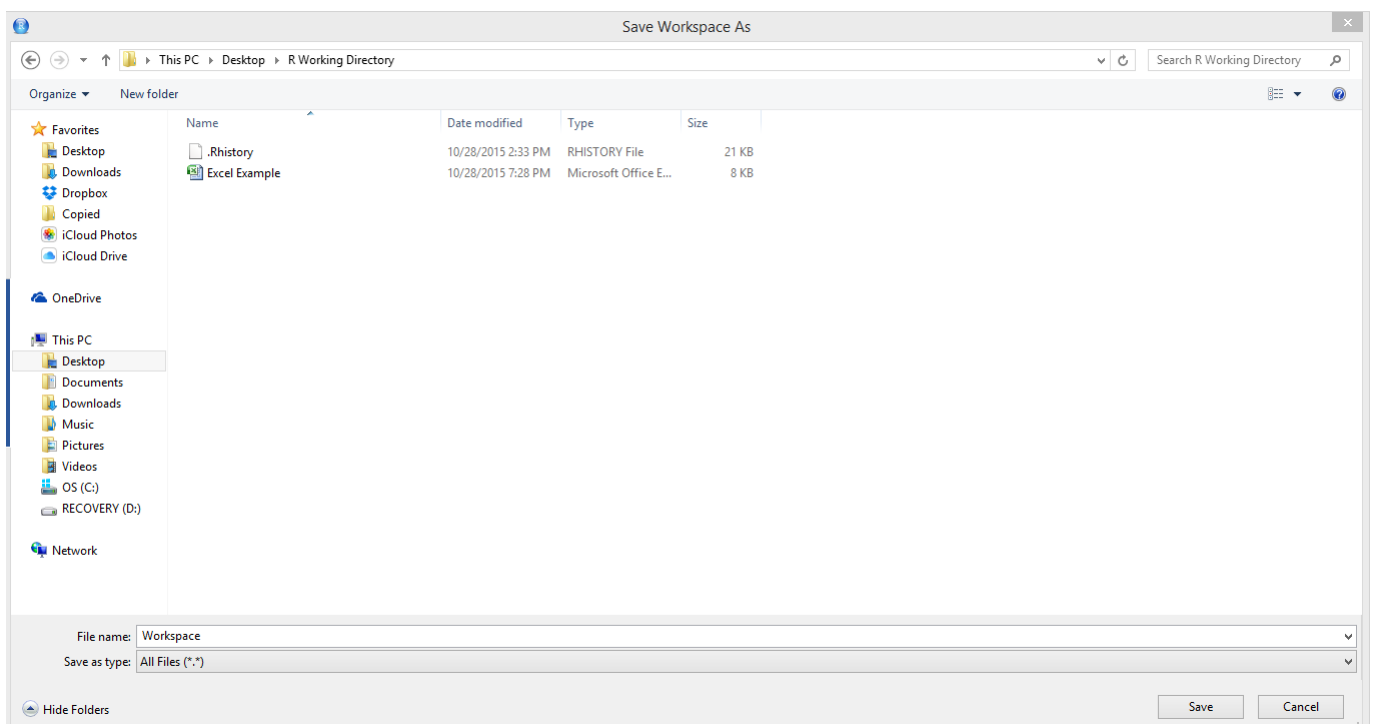


Figure 33: Choose the destination and type the name

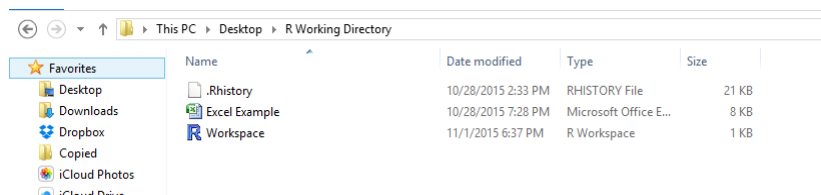


Figure 34: The Workspace file

Now in order to import it, first clear the **Environment/Workspace**. Then go to the **Environment/Workspace** box, and click on the Load button, which is to the right of the

Save button. A menu will open to ask where to load from the workspace, choose the one saved previously and click on *Open*. You will now see that the variables have been loaded again.

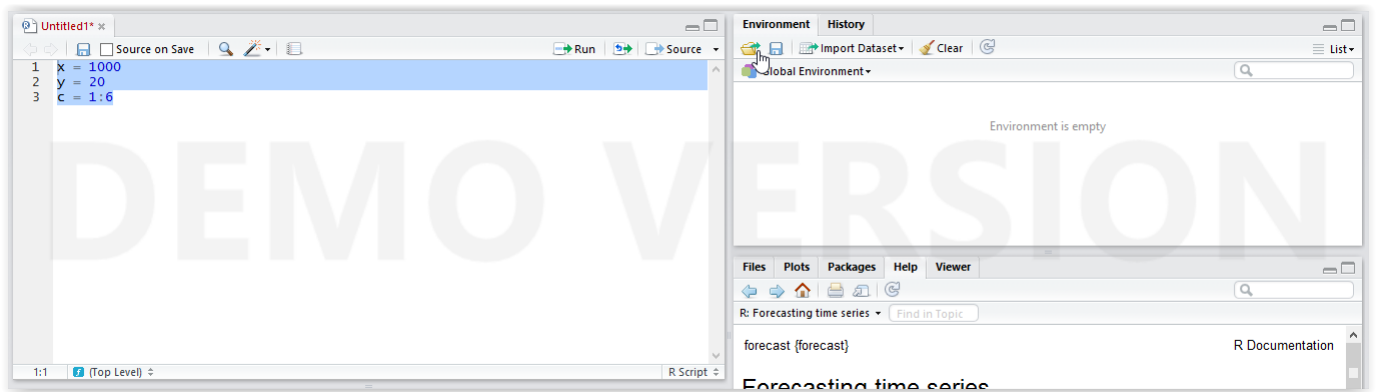


Figure 35: Click the Load button

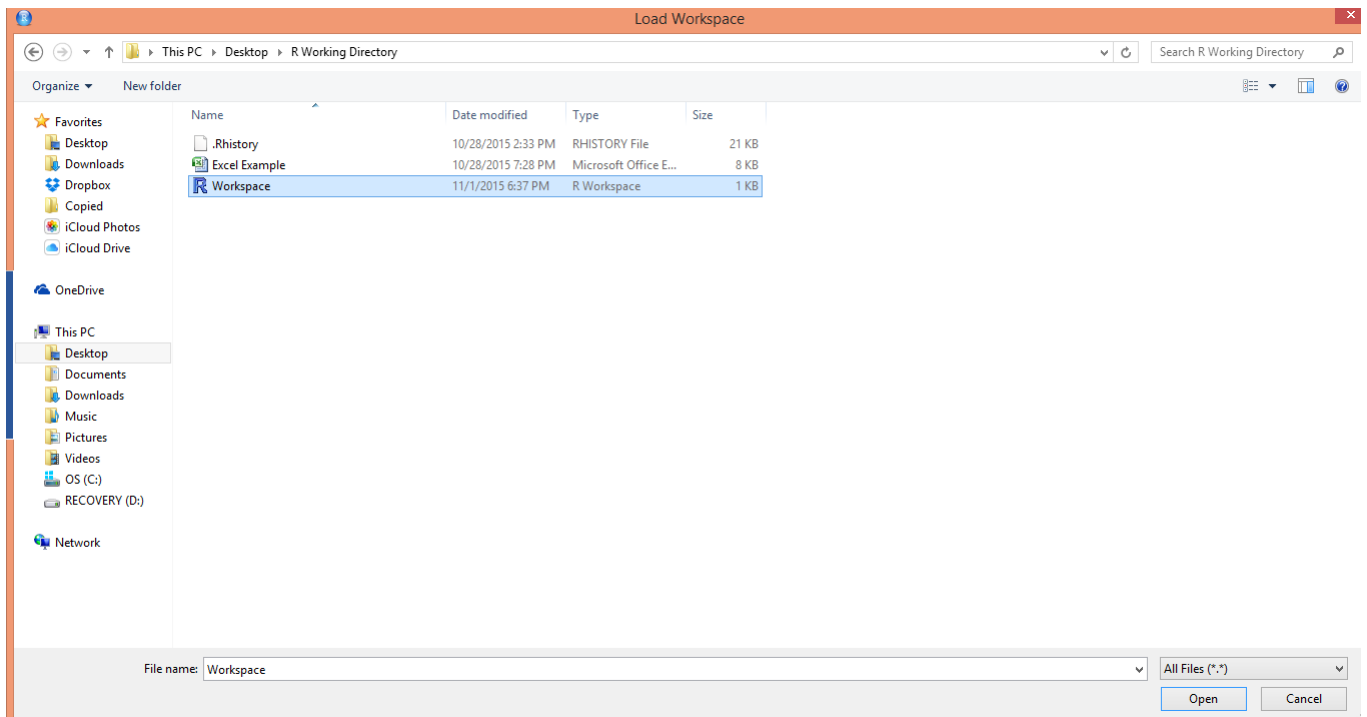


Figure 36: Select the Workspace file

NOTE: The packages that you will use will not be automatically reloaded if you import the **Workspace**. You have to load again the packages you were using, if you do actually need them.