R PROGRAMMING FOR DATA ANALYTICS

Introduction to R:

R is a programming language and software environment for statistical analysis, graphics representation and reporting. R was created by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand, and is currently developed by the R Development Core Team. R is freely available under the GNU General Public License, and pre-compiled binary versions are provided for various operating systems like Linux, Windows and Mac. This programming language was named R, based on the first letter of first name of the two R authors (Robert Gentleman and Ross Ihaka), and partly a play on the name of the Bell Labs Language S.

What is R?

Statistics for relatively advanced users: R has thousands of packages, designed, maintained, and widely used by statisticians.

Statistical graphics: try doing some of our plots in Stata and you won't have

much fun.

Flexible code: R has a rather liberal syntax, and variables don't need to be

declared as they would in (for example) C++, which makes it very easy to

code in. This also has disadvantages in terms of how safe the code is.

Vectorization: R is designed to make it very easy to write functions which

are applied pointwise to every element of a vector. This is extremely useful in statistics.

R is powerful: if a command doesn’t exist already, you can code it yourself.

Why R?

I don’t know if I have a solid reason to convince you, but let me share what got me started. I have no prior coding experience. Actually, I never had computer science in my subjects. I came to know that to learn data science, one must learn either R or Python as a starter. I chose the former. Here are some benefits I found after using R:

The style of coding is quite easy.

It’s open source. No need to pay any subscription charges.

Availability of instant access to over 7800 packages customized for various computation tasks.

The community support is overwhelming. There are numerous forums to help you out.

Get high performance computing experience (require packages)

One of highly sought skill by analytics and data science companies.

There are many more benefits. But, these are the ones which have kept me going. If you think they are exciting, stick around and move to next section. And, if you aren’t convinced, you may like Complete Python Tutorial from Scratch.

Advantages of R over Other Programming Languages:

Academics and statisticians have developed R over two decades. R has now one of the richest ecosystems to perform data analysis. There are around 12000 packages available in CRAN (open-source repository). It is possible to find a library for whatever the analysis you want to perform. The rich variety of library makes R the first choice for statistical analysis, especially for specialized analytical work.

The cutting-edge difference between R and the other statistical products is the output. R has fantastic tools to communicate the results. Rstudio comes with the library knitr. Xie Yihui wrote this package. He made reporting trivial and elegant. Communicating the findings with a presentation or a document is easy.

Python can pretty much do the same tasks as R: data wrangling, engineering, feature selection web scrapping, app and so on. Python is a tool to deploy and implement machine learning at a large-scale. Python codes are easier to maintain and more robust than R. Years ago; Python didn't have many data analysis and machine learning libraries. Recently, Python is catching up and provides cutting-edge API for machine learning or Artificial Intelligence. Most of the data science job can be done with five Python libraries: Numpy, Pandas, Scipy, Scikit-learn and Seaborn.

Python, on the other hand, makes replicability and accessibility easier than R. In fact, if you need to use the results of your analysis in an application or website, Python is the best choice.

R Studio:

RStudio is an Integrated Development Environment (IDE) for R, a programming language for statistical computing and graphics. It is available in two formats: RStudio Desktop is a regular desktop application while RStudio Server runs on a remote server and allows accessing RStudio using a web browser.

R command Prompt:

Once you have R environment setup, then it's easy to start your R command prompt by just typing the following command at your command prompt –

```
$ R
```

This will launch R interpreter and you will get a prompt

```
> where you can start typing your program as follows –
> myString
<- "Hello, World!"
> print ( myString)
 [1] "Hello, World!"
```

Here first statement defines a string variable myString, where we assign a string "Hello, World!" and then next statement print() is being used to print the value stored in variable myString.

R script file:

Usually, you will do your programming by writing your programs in script files and then you execute those scripts at your command prompt with the help of R interpreter called Rscript. So let's start with writing following code in a text file called test.R as under −

Live Demo

```
# My first program in R Programming
myString <- "Hello, World!"
print ( myString)
```

Save the above code in a file test.R and execute it at Linux command prompt as given below. Even if you are using Windows or other system, syntax will remain same.

```
$ Rscript test.R
```

When we run the above program, it produces the following result.

```
[1] "Hello, World!"
```

comments:

Comments are like helping text in your R program and they are ignored by the interpreter while executing your actual program. Single comment is written using # in the beginning of the statement as follows −

```
# My first program in R Programming
```

R does not support multi-line comments but you can perform a trick which is something as follows −

Live Demo

```
if(FALSE)
{
   "This is a demo for multi-line comments and it should be put inside either a
      single OR double quote"
}
myString <- "Hello, World!"
print ( myString)
[1] "Hello, World!"
```

Though above comments will be executed by R interpreter, they will not interfere with your actual program. You should put such comments inside, either single or double quote.

Handling Packages in R:

The package is an appropriate way to organize the work and share it with others. Typically, a package will include code (not only R code!), documentation for the package and the functions inside, some tests to check everything works as it should, and data sets. Packages in R language are a set of R functions, compiled code, and sample data.

These are stored under a directory called “library” within the R environment. By default, R installs a group of packages during installation. Once we start the R console, only the default packages are available by default. Other packages that are already installed need to be loaded explicitly to be utilized by the R program that’s getting to use them.

Installing a R Package:

Packages can be installed with the install.packages() function in R

To install a single package, pass the name of the lecture to the install.packages() function as the first argument

The following the code installs the slidify package from CRAN

```
install.packages("slidify")
```

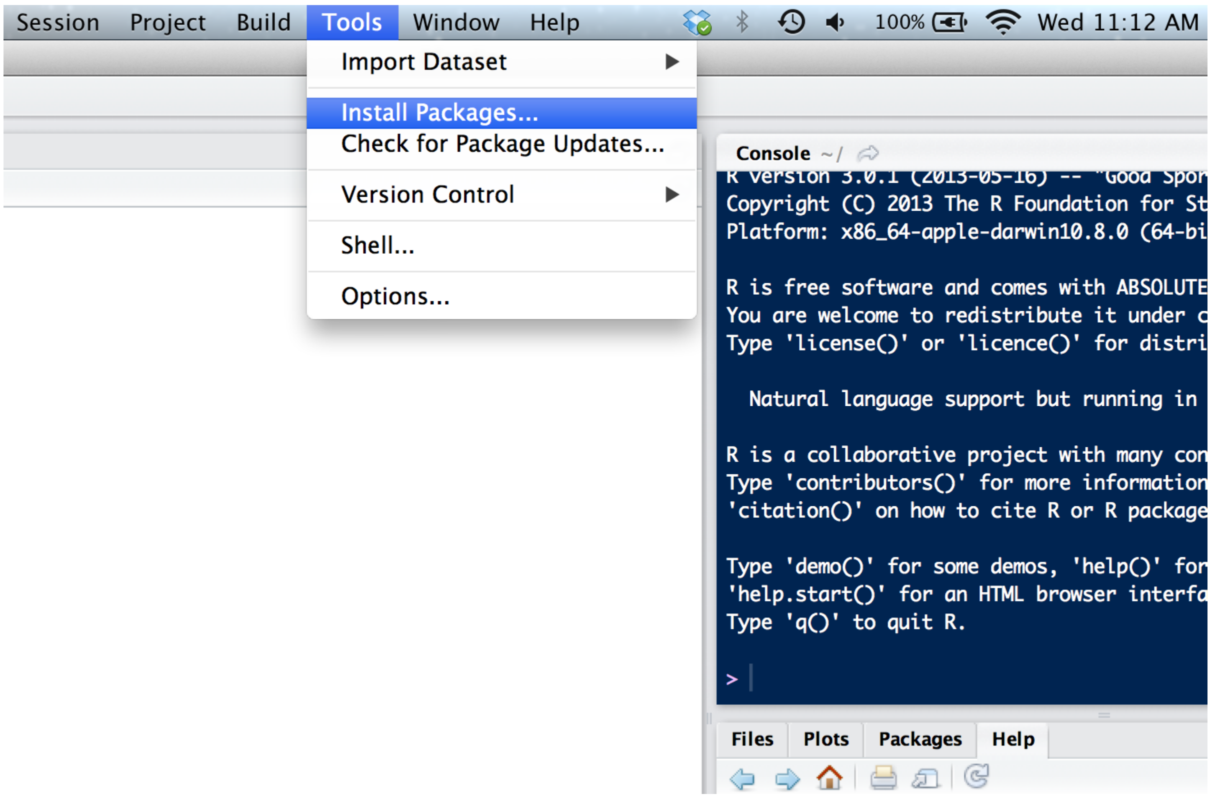
This command downloads the slidify package from CRAN and installs it on your computer

Any packages on which this package depends will also be downloaded and installed

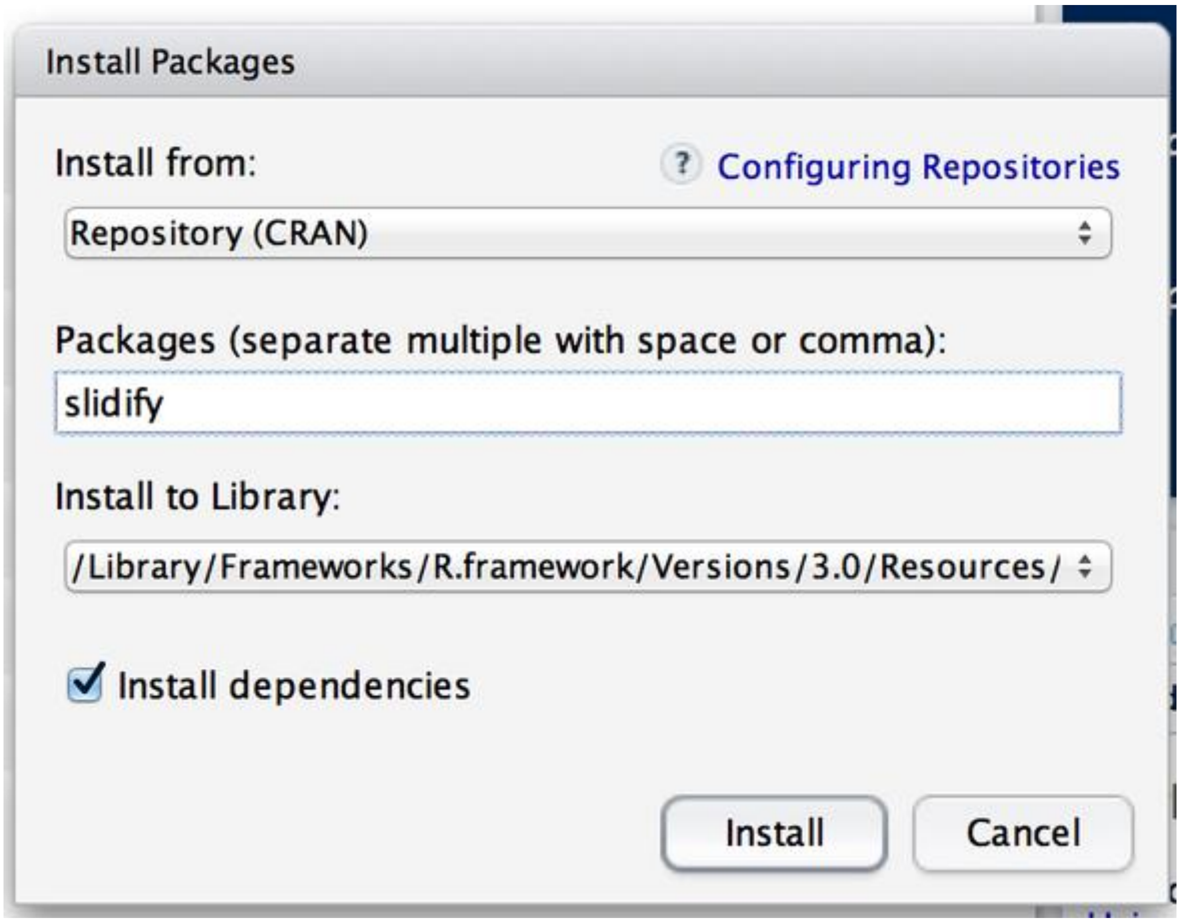
You can install multiple R packages at once with a single call to install.packages()

Place the names of the R packages in a character vector

```
install.packages(c("slidify", "ggplot2", "devtools"))
```



```
R version 3.0.1 (2013-05-16) -- "Good Spor
Copyright (C) 2013 The R Foundation for St
Platform: x86_64-apple-darwin10.8.0 (64-bi

R is free software and comes with ABSOLUTE
You are welcome to redistribute it under c
Type 'license()' or 'licence()' for distri

  Natural language support but running in

R is a collaborative project with many con
Type 'contributors()' for more information
'citation()' on how to cite R or R package

Type 'demo()' for some demos, 'help()' for
'help.start()' for an HTML browser interfa
Type 'q()' to quit R.

>
```



Install from: Repository (CRAN)

Packages (separate multiple with space or comma): slidify

Install to Library: /Library/Frameworks/R.framework/Versions/3.0/Resources/

☑ Install dependencies

Installing an R Package from Bioconductor:

- To get the basic installer and basic set of R packages (warning, will install multiple packages)

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

- Place the names of the R packages in a character vector

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

http://www.bioconductor.org/install/

Loading R Packages

Installing a package does not make it immediately available to you in R; you must load the package

The library() function is used to load packages into R

The following code is used to load the ggplot2 package into R

```
library(ggplot2)
```

Any packages that need to be loaded as dependencies will be loaded first, before the named package is loaded

NOTE: Do not put the package name in quotes!

Some packages produce messages when they are loaded (but some don't)

After loading a package, the functions exported by that package will be attached to the top of the search() list (after the workspace)

```
library(ggplot2)
search()
```

Few commands to get started:

R packages are a collection of R functions, complied code and sample data. They are stored under a directory called "library" in the R environment. By default, R installs a set of packages during installation. More packages are added later, when they are needed for some specific purpose. When we start the R console, only the default packages are available by default. Other packages which are already installed have to be loaded explicitly to be used by the R program that is going to use them.

All the packages available in R language are listed at R Packages.

Below is a list of commands to be used to check, verify and use the R packages.

Check Available R Packages

Get library locations containing R packages

Live Demo

```
.libPaths()
```

When we execute the above code, it produces the following result. It may vary depending on the local settings of your pc.

```
[2] "C:/Program Files/R/R-3.2.2/library"
```

Get the list of all the packages installed

Live Demo

library()

When we execute the above code, it produces the following result. It may vary depending on the local settings of your pc.

Packages in library ‘C:/Program Files/R/R-3.2.2/library’:

base	The R Base Package
boot	Bootstrap Functions (Originally by Angelo Canty
	for S)
class	Functions for Classification
cluster	"Finding Groups in Data": Cluster Analysis
	Extended Rousseeuw et al.
codetools	Code Analysis Tools for R
compiler	The R Compiler Package
datasets	The R Datasets Package
foreign	Read Data Stored by 'Minitab', 'S', 'SAS',
	'SPSS', 'Stata', 'Systat', 'Weka', 'dBase', ...
graphics	The R Graphics Package
grDevices	The R Graphics Devices and Support for Colours
	and Fonts
grid	The Grid Graphics Package
KernSmooth	Functions for Kernel Smoothing Supporting Wand
	& Jones (1995)
lattice	Trellis Graphics for R
MASS	Support Functions and Datasets for Venables and
	Ripley's MASS
Matrix	Sparse and Dense Matrix Classes and Methods
methods	Formal Methods and Classes
mgcv	Mixed GAM Computation Vehicle with GCV/AIC/REML
	Smoothness Estimation
nlme	Linear and Nonlinear Mixed Effects Models
nnet	Feed-Forward Neural Networks and Multinomial
	Log-Linear Models
parallel	Support for Parallel computation in R
rpart	Recursive Partitioning and Regression Trees
spatial	Functions for Kriging and Point Pattern

	Analysis
splines	Regression Spline Functions and Classes
stats	The R Stats Package
stats4	Statistical Functions using S4 Classes
survival	Survival Analysis
tcltk	Tcl/Tk Interface
tools	Tools for Package Development
utils	The R Utils Package

Get all packages currently loaded in the R environment

Live Demo

```
search()
```

When we execute the above code, it produces the following result. It may vary depending on the local settings of your pc.

```
[1] ".GlobalEnv"       "package:stats"     "package:graphics"
[4] "package:grDevices" "package:utils"     "package:datasets"
[7] "package:methods"   "Autoloads"        "package:base"
```

Install a New Package

There are two ways to add new R packages. One is installing directly from the CRAN directory and another is downloading the package to your local system and installing it manually.

Install directly from CRAN

The following command gets the packages directly from CRAN webpage and installs the package in the R environment. You may be prompted to choose a nearest mirror. Choose the one appropriate to your location.

```
install.packages("Package Name")
```

```
# Install the package named "XML".
install.packages("XML")
```

Install package manually

Go to the link R Packages to download the package needed. Save the package as a .zip file in a suitable location in the local system.

Now you can run the following command to install this package in the R environment.

```
install.packages(file_name_with_path, repos = NULL, type = 
"source")
```

```
# Install the package named "XML"
install.packages("E:/XML_3.98-1.3.zip", repos = NULL, type = 
"source")
```

Load Package to Library

Before a package can be used in the code, it must be loaded to the current R environment. You also need to load a package that is already installed previously but not available in the current environment.

A package is loaded using the following command −

```
library("package Name", lib.loc = "path to library")
```

```
# Load the package named "XML"
install.packages("E:/XML_3.98-1.3.zip", repos = NULL, type =
"source")
```

Input and Output:

To read the data from the keyboard, we use three different functions; scan(), readline(), print().

- scan()

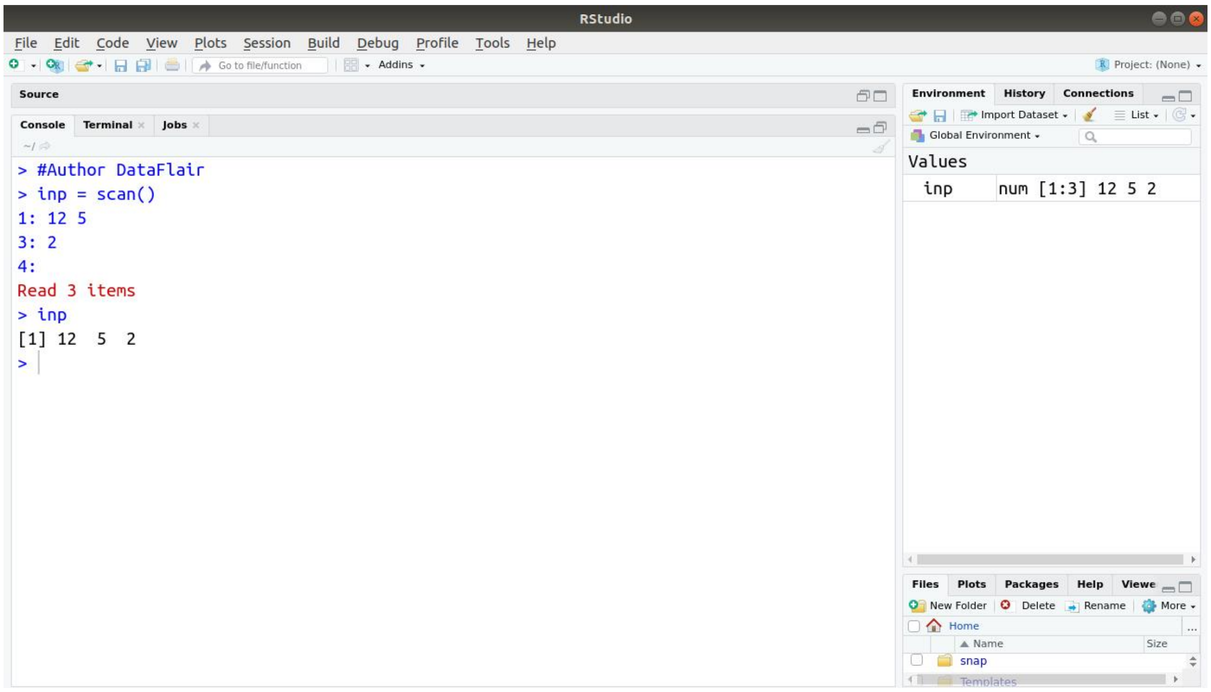
Read Data Values: This is used for reading data into the input vector or an input list from the environment console or file.

Keywords: File, connection.

For example:

```
> #Author DataFlair
> inp = scan()
> inp
```

Output:



```
> #Author DataFlair
> inp = scan()
1: 12 5
3: 2
4:
Read 3 items
> inp
[1] 12  5  2
>
```

```
Values
inp    num [1:3] 12 5 2
```

Wait! Have you completed the Principles and Functions of R Debugging

- **readline()**

With readline(), we read multiple lines from a connection.

Keywords: File, connection.

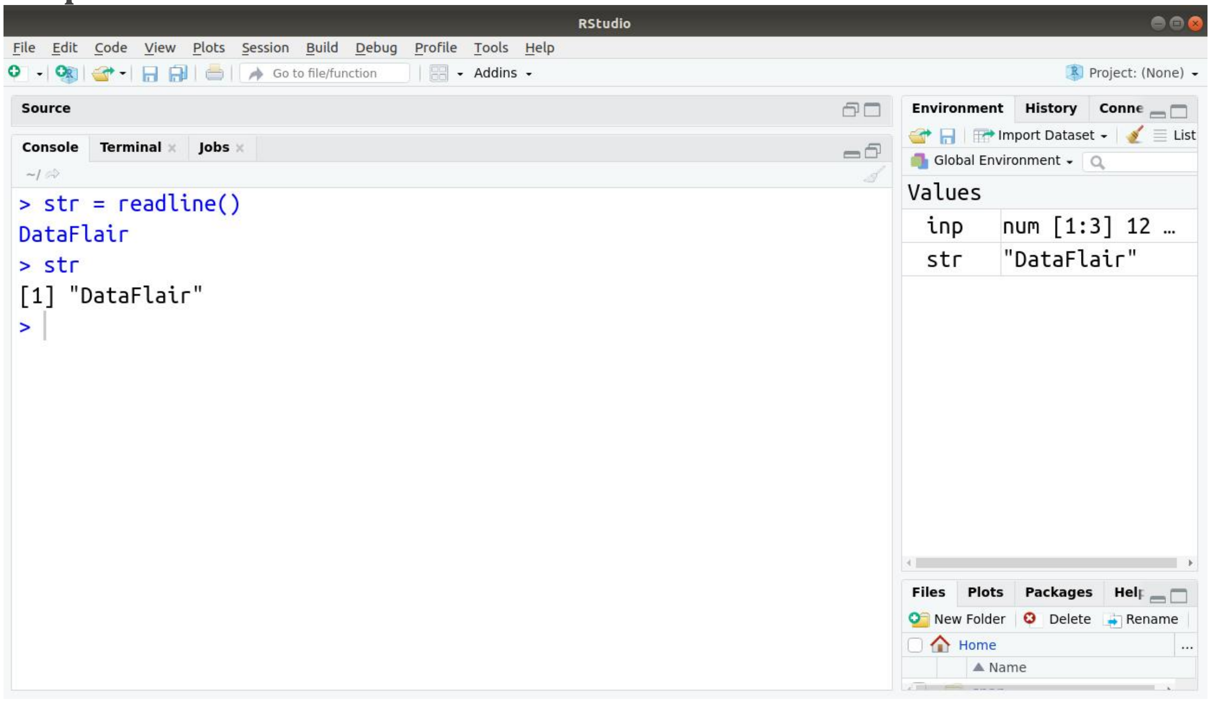
We can use readline() for inputing a line from the keyboard in the form of a string:

For example:

> str = readline()

> str

Output:



```
> str = readline()
DataFlair
> str
[1] "DataFlair"
>
```

```
Values
  inp    num [1:3] 12 …
  str    "DataFlair"
```

print()

A print function simply displays the contents of its argument object. New printing methods can be easily added for new classes through this generic function.

Keywords: print

Printing to the screen: In interactive mode, one can print the value of the variable by just typing the variable name or expression. *print()* function can be used in the batch mode as:

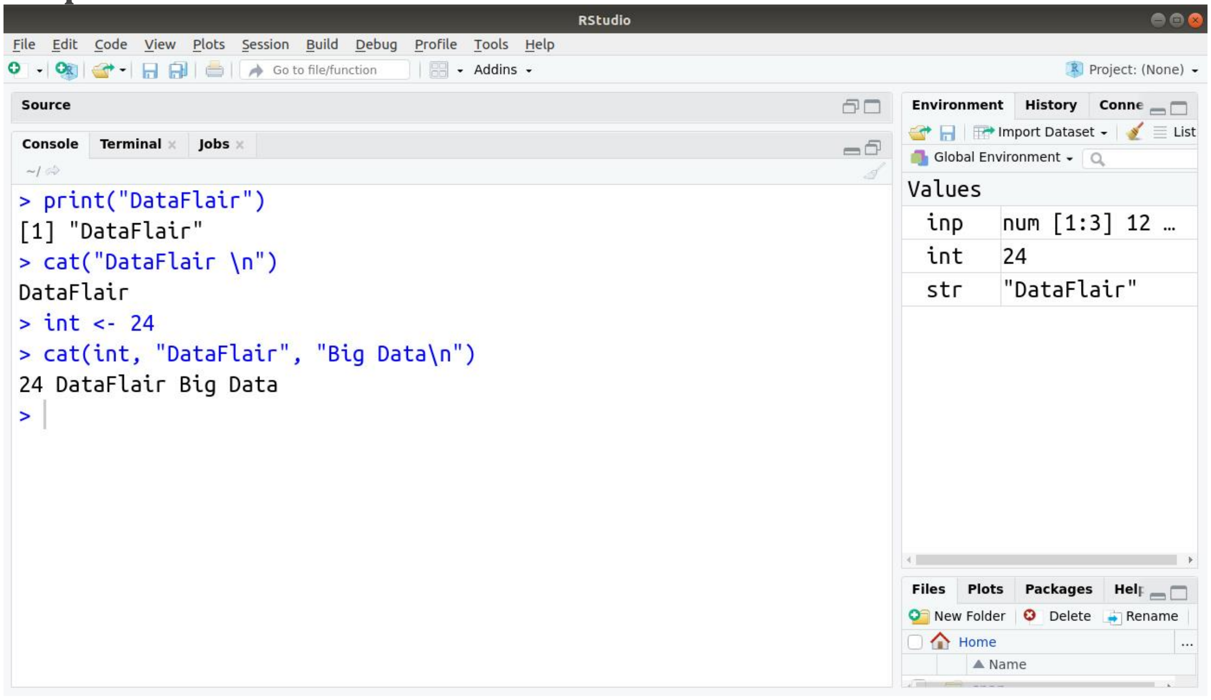
print(x)

The argument might be an object. So it is better to use *cat()* instead of *print()*, as the last one can print only one expression and its result is numbered, which may be a nuisance to us. Here is an example written below:

> print("DataFlair")

```
> cat("DataFlair \n")
DataFlair
> int <- 24
> cat(int, "DataFlair", "Big Data\n")
24 DataFlair Big Data
```

Output:



```
> print("DataFlair")
[1] "DataFlair"
> cat("DataFlair \n")
DataFlair
> int <- 24
> cat(int, "DataFlair", "Big Data\n")
24 DataFlair Big Data
>
```

```
Values
  inp    num [1:3] 12 …
  int    24
  str    "DataFlair"
```

There are many methods to read and write files in R programming:

Usually we use function *read.table()* to read data. A header has a default value of ‘FALSE’. Therefore, having no header pertains to no value. R factors are also called as character strings. In order to disable this feature, the argument can be stated *as.is = T* as a part of your call to read.table().

When you have a spreadsheet export file, i.e. having a *type.csv* where the fields are divided by commas in place of spaces, use *read.csv()* in place of *read.table()*. To read spreadsheet files, we can use read.xls.

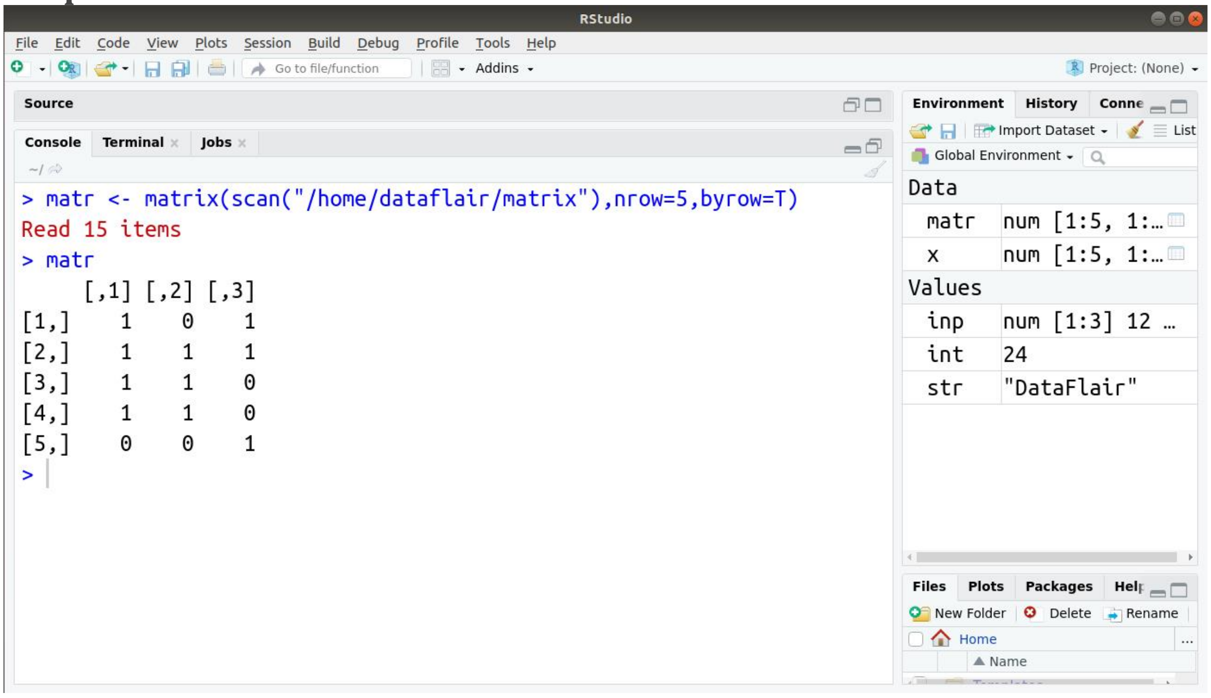
When you read in a matrix using read.table(), the resultant object will become a data frame, even when all the entries got to be numeric. A case exists which may followup call towards *as.matrix()* in a matrix.

For example:

We store this matrix in a file. We will then use the scan() function to read the contents of this file.

```
> matr <- matrix(scan("/home/dataflair/matrix"),nrow=5,byrow=T)
```

Output:



```
> matr <- matrix(scan("/home/dataflair/matrix"),nrow=5,byrow=T)
Read 15 items
> matr
     [,1] [,2] [,3]
[1,]    1    0    1
[2,]    1    1    1
[3,]    1    1    0
[4,]    1    1    0
[5,]    0    0    1
> 
```

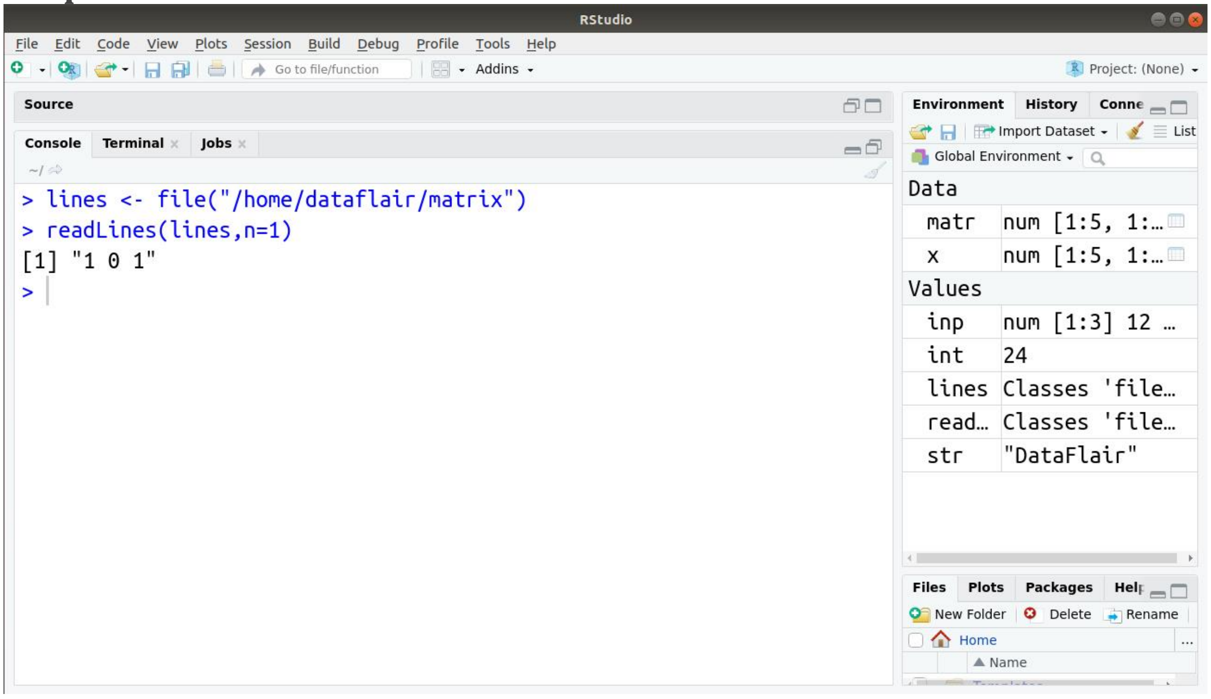
Data	
matr	num [1:5, 1:…
x	num [1:5, 1:…
Values	
inp	num [1:3] 12 …
int	24
str	"DataFlair"

We can use *readLines()* for this, but we need to produce a connection first, by calling the file().

For example:

```
> lines <- file("/home/dataflair/matrix")
> readLines(lines,n=1)
```

Output:



```
> lines <- file("/home/dataflair/matrix")
> readLines(lines,n=1)
[1] "1 0 1"
>
```

```
Data
  matr   num [1:5, 1:…
  x      num [1:5, 1:…
Values
  inp    num [1:3] 12 …
  int    24
  lines  Classes 'file…
  read…  Classes 'file…
  str    "DataFlair"
```

In R, we use *write.table()* function to write a data-frame in the form of a table. It is same as *read.table()* which writes a data frame instead of reading one. Let us first create our table as follows:

data <- read.table(header=TRUE, text='

subject sex size

1 M 7

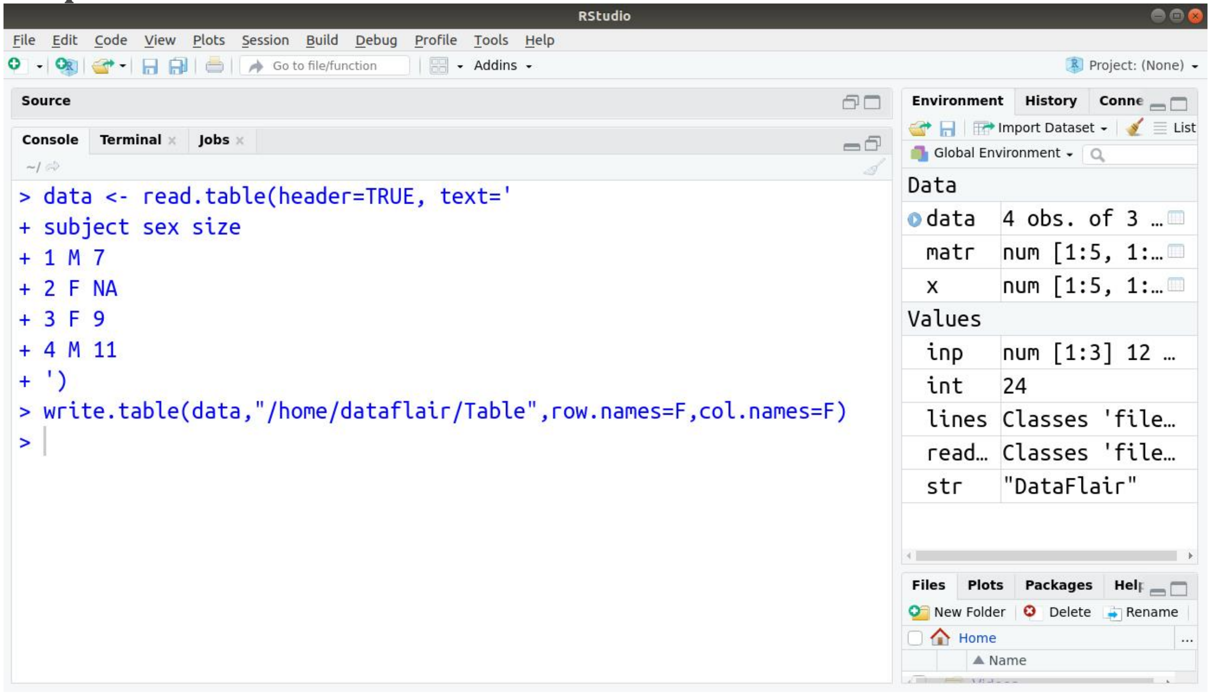
2 F NA

3 F 9

4 M 11

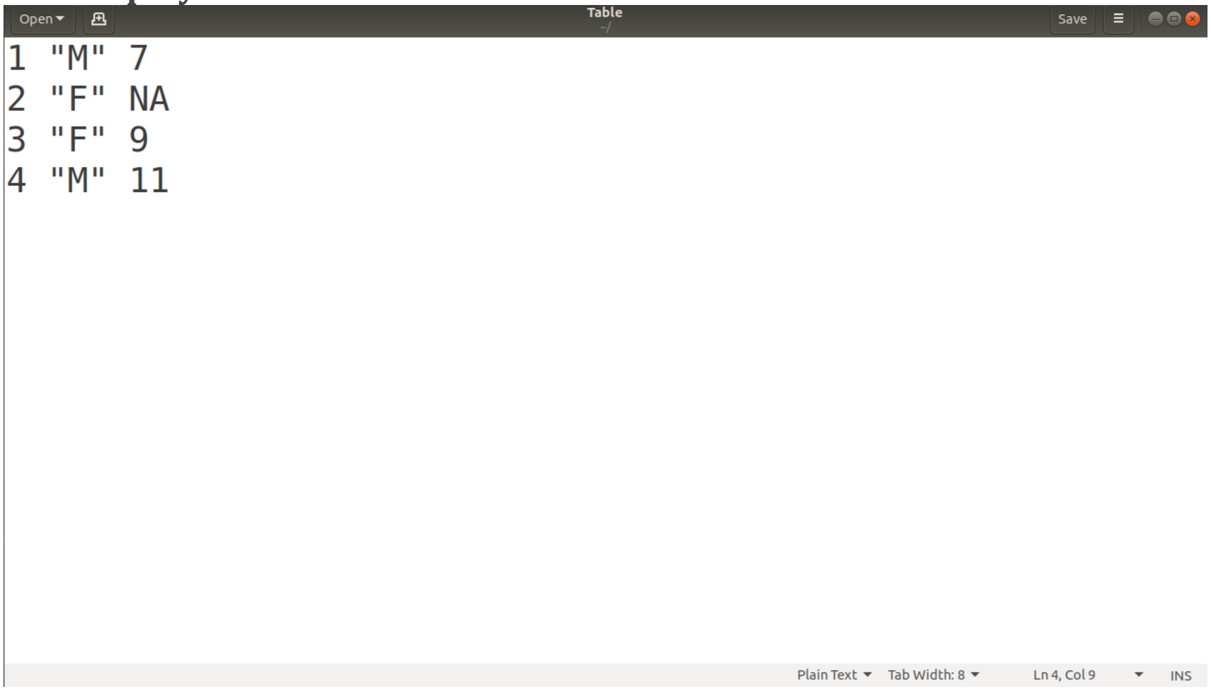
')

Output:


```
> data <- read.table(header=TRUE, text='
+ subject sex size
+ 1 M 7
+ 2 F NA
+ 3 F 9
+ 4 M 11
+ ')
> write.table(data,"/home/dataflair/Table",row.names=F,col.names=F)
>
```

Data	
data	4 obs. of 3 …
matr	num [1:5, 1:…
x	num [1:5, 1:…
Values	
inp	num [1:3] 12 …
int	24
lines	Classes 'file…
read…	Classes 'file…
str	"DataFlair"

We then write a table to the file as follows:
> write.table(data,"/home/dataflair/Table",row.names=F,col.names=F)
File Display:


```
1 "M" 7
2 "F" NA
3 "F" 9
4 "M" 11
```

Opening the "Table" file at the saved location, we obtain:

A must learn concept that you can't miss – R Data Frame

There are functions to Manipulate Connections (Files, URLs,…). Functions to create, open and close connections.

For example: URLs, pipes and other types of generalised files.

Keywords: file, connection

Extended Example: Reading PUMS sample files

There is a collection of records for every sample of housing units that store information about the characteristics of each unit.

There is greater accessibility to the inexpensive data, mostly for research purposes. Therefore, for students, this is highly beneficial because they are searching for higher accessibility to inexpensive data. Social scientists often use the PUMS for regression analysis and modeling applications.

Statistical software is a tool used to work with PUMS files.

Do you know about Linear Regression in R Programming

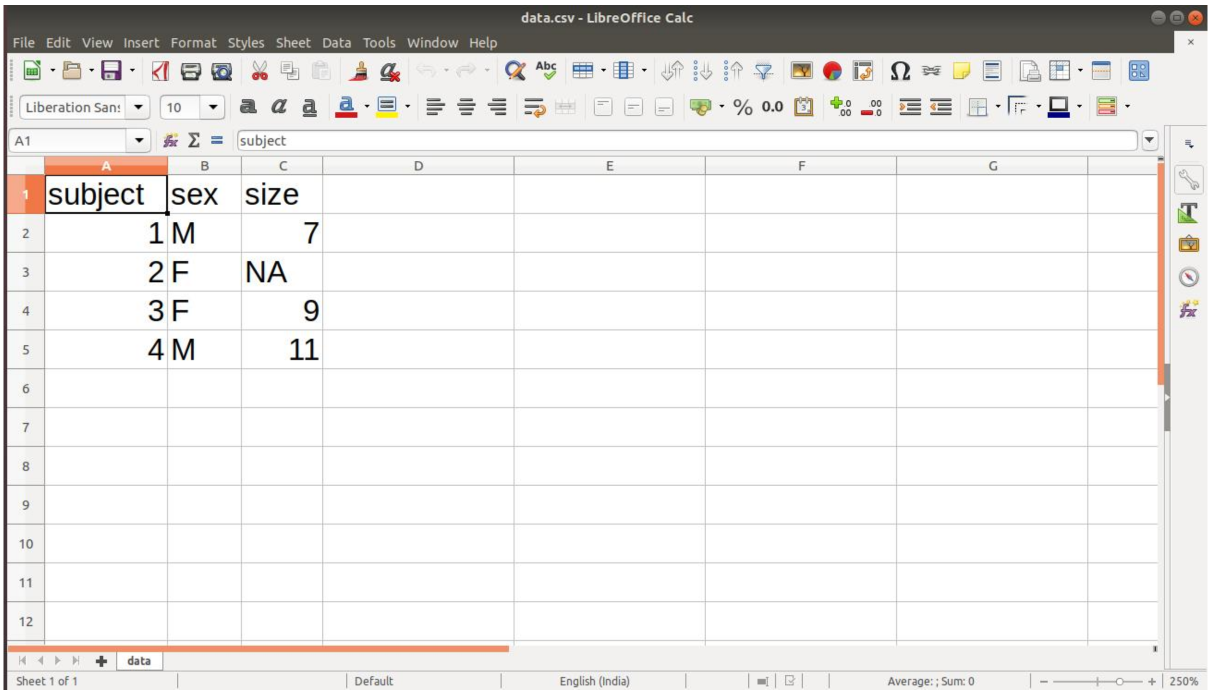
We use write.csv() to write files. By default, write.csv() includes row names.

```
# Author DataFrame

data <- read.table(header=TRUE, text='

subject sex size

1 M 7

2 F NA

3 F 9

4 M 11

')
```

Write to a file, suppress row names

write.csv(data, "/home/dataflair/data.csv", row.names=FALSE)

We obtain the following output in our data.csv file:

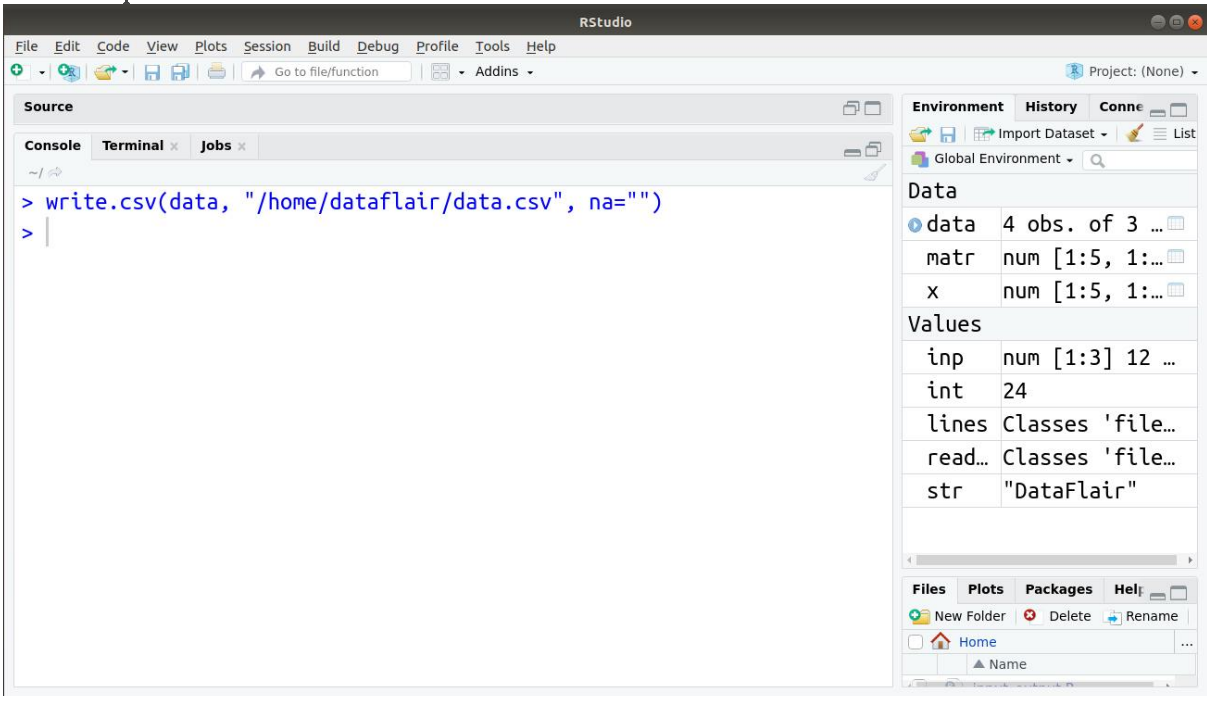


	A	B	C
1	subject	sex	size
2	1	M	7
3	2	F	NA
4	3	F	9
5	4	M	11

In the above output of our file, we have a missing value denoted by NA. We can replace this NA with "" using the following line of code:

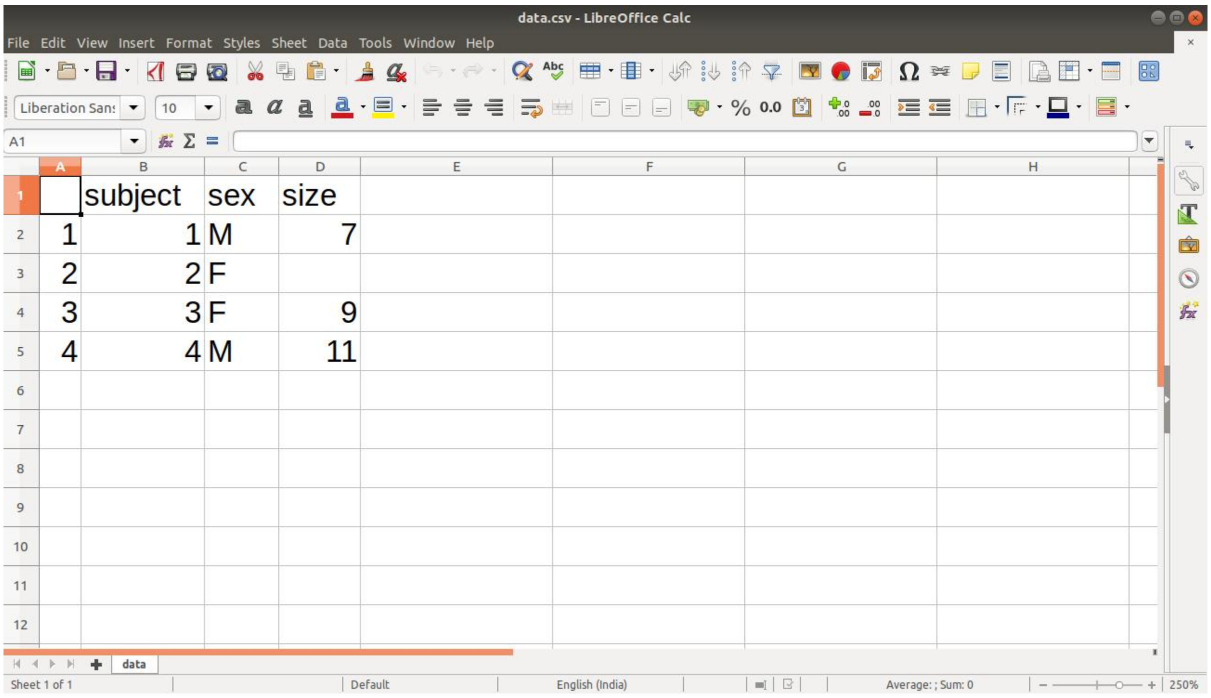
```
> write.csv(data, "/home/dataflair/data.csv", na="")
```

The output is saved in our file data.csv as follows:



```
> write.csv(data, "/home/dataflair/data.csv", na="")
> 
```

The output is saved in our file data.csv as follows:

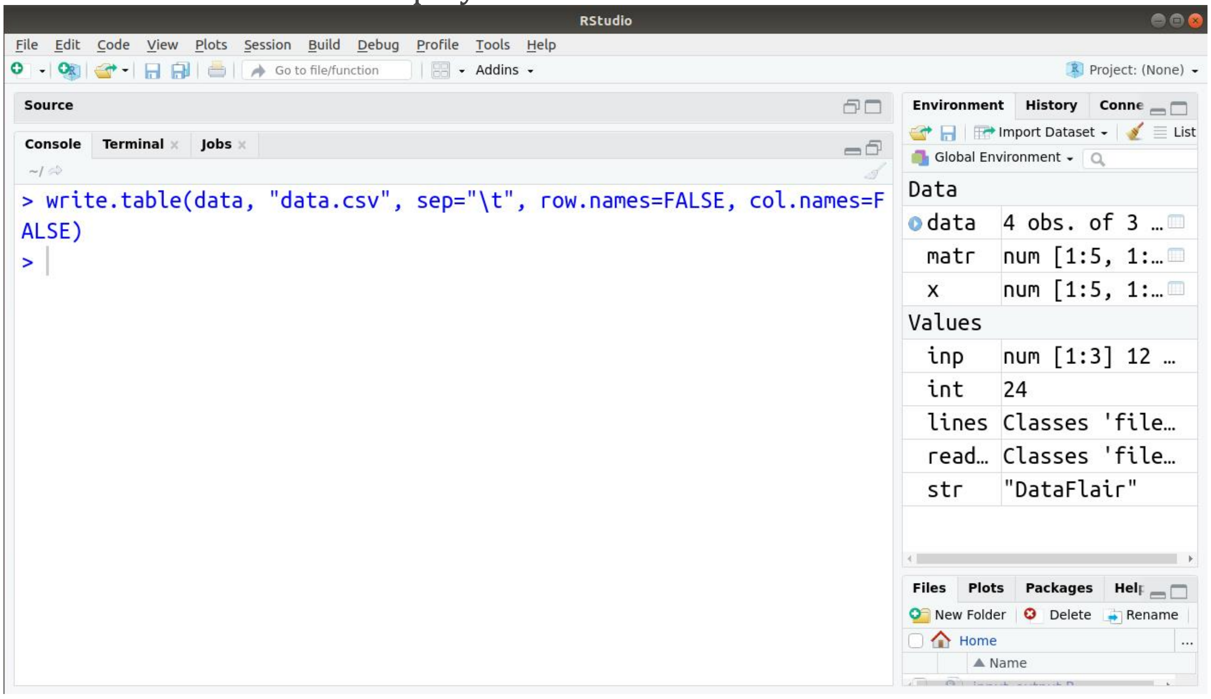


	subject	sex	size
1	1	M	7
2	2	F	
3	3	F	9
4	4	M	11

We can also use tabs, suppress row and column names using the following line of code:

> write.table(data, "data.csv", sep="\t", row.names=FALSE, col.names=FALSE)

Our file data.csv is now displayed as follows:



```
> write.table(data, "data.csv", sep="\t", row.names=FALSE, col.names=F
ALSE)
>
```

```
Data
data   4 obs. of 3 …
matr   num [1:5, 1:…
x      num [1:5, 1:…
Values
inp    num [1:3] 12 …
int    24
lines  Classes 'file…
read…  Classes 'file…
str    "DataFlair"
```

Merge all files in a directory using R into a single data frame.

Set the directory:

setwd("/home/dataflair/DataFlair")

Getting a list of files in a directory:

```
file_list <- list.files()
```

If we want to list the files in a different directory, specify the path to list.files.

For example:

If we want the files in the folder C:/foo/, we can use the following code:

```
file_list <- list.files()
```

The final step is to iterate through the list of files in the current working directory and put them together to form a data frame. When the script encounters the first file in the file_list, it creates the main data frame to merge everything into (called dataset here). This is done using the *!exists conditional*:

If a dataset already exists, then a temporary data frame, called *temp_dataset* is created and added to the dataset. The temporary data frame is removed when we're done with it using the *rm(temp_dataset)* command.

If the dataset doesn't exist (!exists is true), then we have to create it.

Here's the remainder of the code:

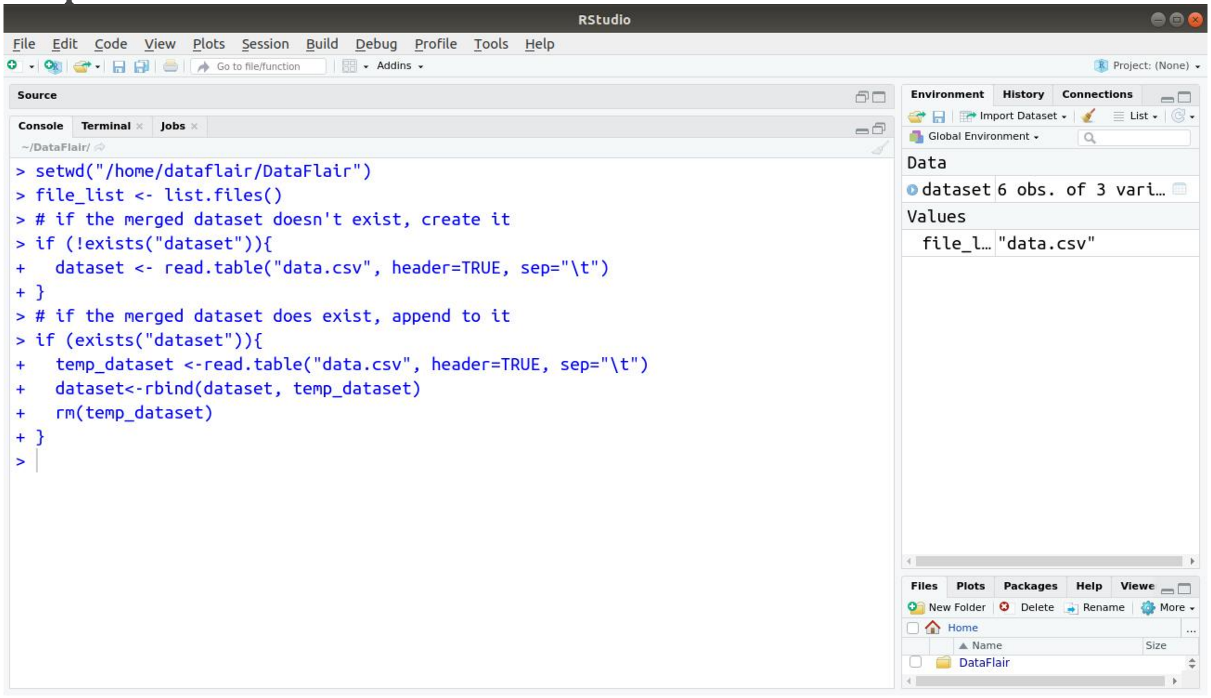
```
if (!exists("dataset")){

dataset <- read.table("data.csv", header=TRUE, sep="\t")

}

# if the merged dataset does exist, append to it

if (exists("dataset")){

temp_dataset <-read.table("data.csv", header=TRUE, sep="\t")

dataset<-rbind(dataset, temp_dataset)

rm(temp_dataset)
```

}

The full code:

```
setwd("/home/dataflair/DataFlair")

file_list <- list.files()

# if the merged dataset doesn't exist, create it

if (!exists("dataset")){

dataset <- read.table("data.csv", header=TRUE, sep="\t")

}

# if the merged dataset does exist, append to it

if (exists("dataset")){

temp_dataset <-read.table("data.csv", header=TRUE, sep="\t")

dataset<-rbind(dataset, temp_dataset)

rm(temp_dataset)

}
```

Output:



```
> setwd("/home/dataflair/DataFlair")
> file_list <- list.files()
> # if the merged dataset doesn't exist, create it
> if (!exists("dataset")){
+   dataset <- read.table("data.csv", header=TRUE, sep="\t")
+ }
> # if the merged dataset does exist, append to it
> if (exists("dataset")){
+   temp_dataset <-read.table("data.csv", header=TRUE, sep="\t")
+   dataset<-rbind(dataset, temp_dataset)
+   rm(temp_dataset)
+ }
> 
```

```
Data
dataset 6 obs. of 3 vari…
Values
file_l… "data.csv"
```

RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Project: (None)
Source
Console Terminal Jobs
~/DataFlair/
Environment History Connections
Import Dataset
List
Global Environment
Files Plots Packages Help Viewer
New Folder Delete Rename More
Home
Name Size
DataFlair

Entering Data from keyboard:

Usually you will obtain a data frame by importing it from SAS, SPSS, Excel, Stata, a database, or an ASCII file. To create it interactively, you can do something like the following.

```
# create a data frame from scratch

age <- c(25, 30, 56)

gender <- c("male", "female", "male")

weight <- c(160, 110, 220)

mydata <- data.frame(age,gender,weight)
```

You can also use R's built in spreadsheet to enter the data interactively, as in the following example.

```
# enter data using editor

mydata <- data.frame(age=numeric(0), gender=character(0),
weight=numeric(0))

mydata <- edit(mydata)

# note that without the assignment in the line above,

# the edits are not saved!
```

Printing fewer digits or more digits:

The reason it is only a suggestion is that you could quite easily write a print function that ignored the options value. The built-in printing and formatting functions do use the options value as a default.

As to the second question, since R uses finite precision arithmetic, your answers aren't accurate beyond 15 or 16 decimal places, so in general, more aren't required. The gmp and rcdd packages deal with multiple precision arithmetic (via an interace to the gmp library), but this is mostly related to big integers rather than more decimal places for your doubles.

Mathematica or Maple will allow you to give as many decimal places as your heart desires.

EDIT:

It might be useful to think about the difference between decimal places and significant figures. If you are doing statistical tests that rely on differences beyond the 15th significant figure, then your analysis is almost certainly junk.

On the other hand, if you are just dealing with very small numbers, that is less of a problem, since R can handle number as small as .Machine$double.xmin (usually 2e-308).

Compare these two analyses.

```
x1 <- rnorm(50, 1, 1e-15)
y1 <- rnorm(50, 1 + 1e-15, 1e-15)
t.test(x1, y1)  #Should throw an error

x2 <- rnorm(50, 0, 1e-15)
```

```
y2 <- rnorm(50, 1e-15, 1e-15)
t.test(x2, y2)  #ok
```

In the first case, differences between numbers only occur after many significant figures, so the data are "nearly constant". In the second case, Although the size of the differences between numbers are the same, compared to the magnitude of the numbers themselves they are large.

As mentioned by e3bo, you can use multiple-precision floating point numbers using the Rmpfr package.

```
mpfr("3.1415926535897932384626433832795028841971693993751058209749445923078164062862089986280348253")
```

These are slower and more memory intensive to use than regular (double precision) numeric vectors, but can be useful if you have a poorly conditioned problem or unstable algorithm.

If you are producing the entire output yourself, you can use sprintf(), e.g.

```
> sprintf("%.10f",0.25)
[1] "0.2500000000"
```

specifies that you want to format a floating point number with ten decimal points (in %.10f the f is for float and the .10 specifies ten decimal points).

I don't know of any way of forcing R's higher level functions to print an exact number of digits.

Displaying 100 digits does not make sense if you are printing R's usual numbers, since the best accuracy you can get using 64-bit doubles is around 16 decimal digits (look at .Machine$double.eps on your system)

Special Values:

There are a few special values that are used in R.

NA

In R, the NA values are used to represent missing values. (NA stands for “not available.”) You may encounter NA values in text loaded into R (to represent missing values) or in data loaded from databases (to replace NULL values).

If you expand the size of a vector (or matrix or array) beyond the size where values were defined, the new spaces will have the value NA:

```
> v <- c(1,2,3)
> v
[1] 1 2 3
> length(v) <- 4
> v
[1]  1  2  3 NA
```

Inf and -Inf

If a computation results in a number that is too big, R will return Inf for a positive number and -Inf for a negative number (meaning positive and negative infinity, respectively):

```
> 2 ^ 1024
[1] Inf
> - 2 ^ 1024
[1] -Inf
```

This is also the value returned when you divide by 0:

```
> 1 / 0
[1] Inf
```

NaN

Sometimes, a computation will produce a result that makes little sense. In these cases, R will often return NaN (meaning “not a number”):

```
> Inf - Inf
[1] NaN
> 0 / 0
[1] NaN
```

NULL

Additionally, there is a null object in R, represented by the symbol NULL. (The symbol NULL always points to the same object.) NULL is often used as an argument in functions to mean that no value was assigned to the argument. Additionally, some functions may return NULL. Note that NULL is not the same as NA, Inf, -Inf, or NaN.