NYC DATA SCIENCE ACADEMY

Basic Programming with R (Data Analytics)

Outline For Today

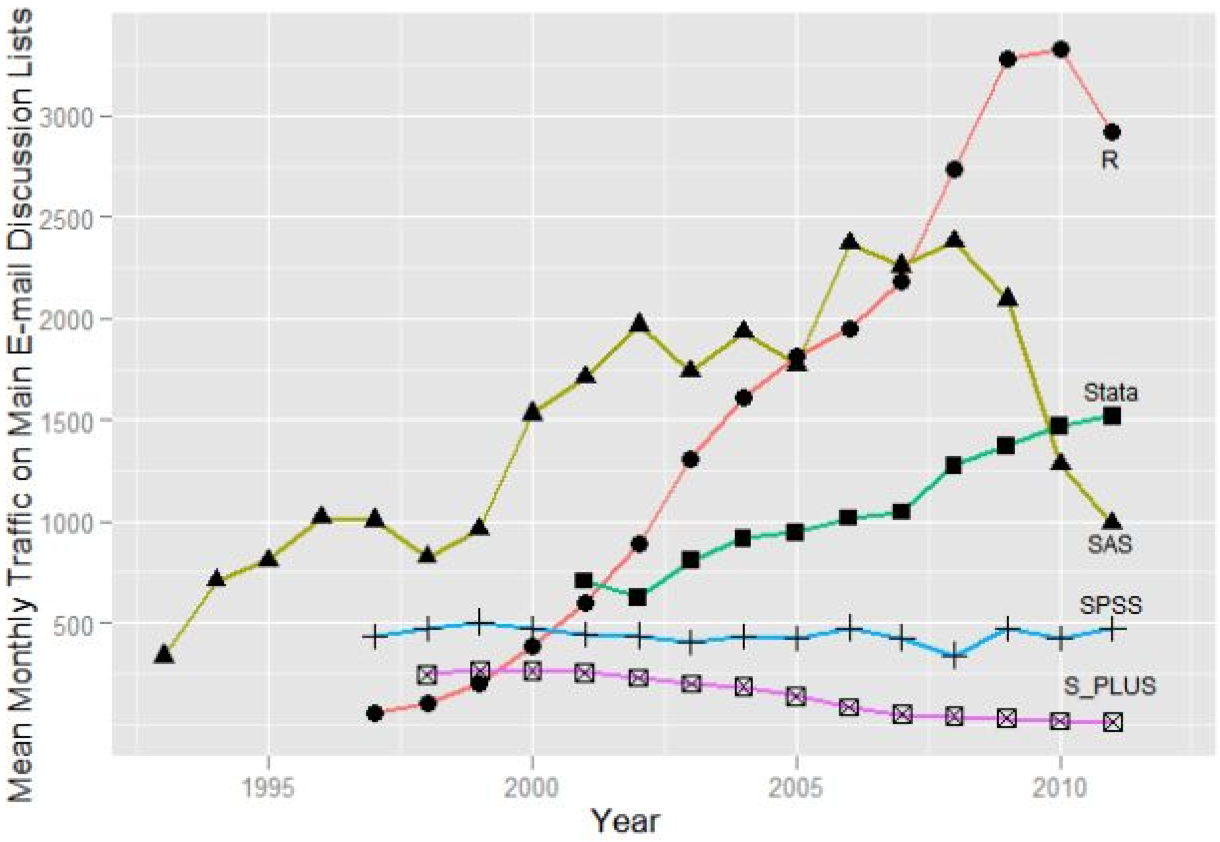
- Introduction to R
    - What is R?
    - Why R?
    - How to learn R
    - RStudio, packages, and the workspace
- Basic R language elements
    - Survey of data object types
    - Local data import/export
    - Introducing functions and control statements
- In-depth study of data objects
- Functions
- Functional Programming

Introduction to R

What is R?

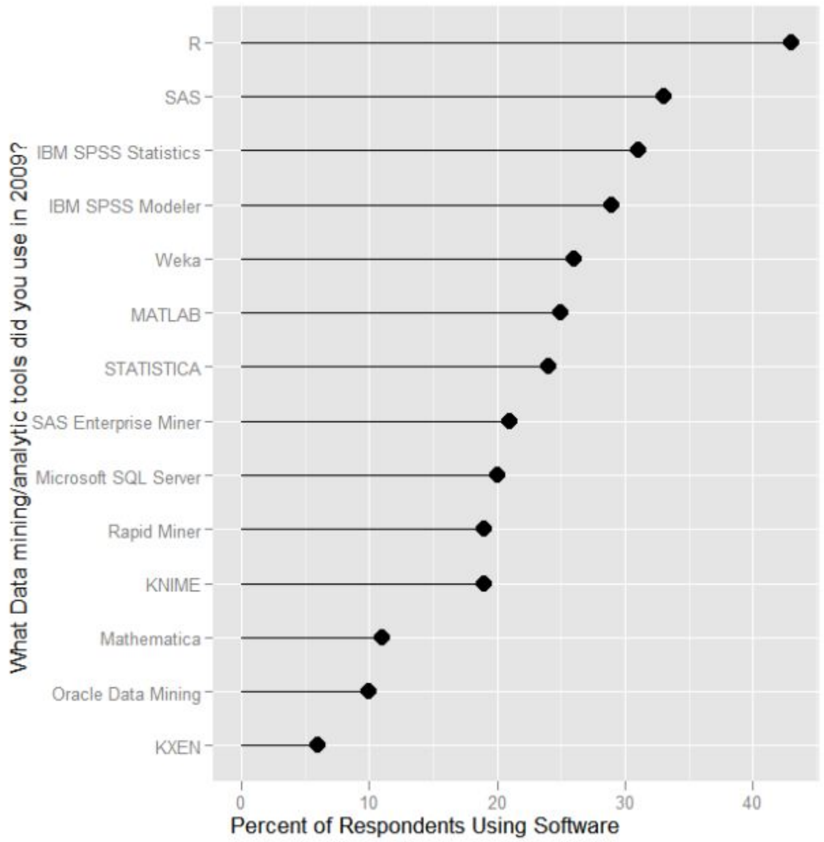
- R is a programming language for statistics and data analysis that is popular in both industry and academia.
  - The goal in its development is to quickly and accurately turn analytical concepts into usable tools.
- R was inspired by S.
  - The S language was developed at Bell Labs in 1976.
  - R can be viewed as an open-source implementation of the S language, though the two are ultimately distinct.
- Important dates:
  - 1997: R officially becomes a member of the GNU Project (“GNU’s Not Unix”).
  - 2008: R eclipses SAS, SPSS, and Stata in online programming forums like StackExchange.

Why R?



Source: Arthur Charpentier, *Freakonometrics*

Why R?



Source: Arthur Charpentier, *Freakonometrics*

Key Features of R

- Open-source and free
- Powerful and increasingly scalable
- Ability to interact with other software
- Cutting-edge data management, modeling, and graphics
- Reproducible analysis
- Lightweight and multi-platform

Extensibility of R

- Can connect to databases (e.g. Oracle, MySQL)
- Can call C, C++, and Fortran code
- Can be used as an embedded computing engine (Rserve)
- Can be deployed in interactive applications on the web (Shiny)

Performance

Weaknesses

1. R is an interpreted language
2. All data is read into memory
3. R is *single threaded*, limiting speed and efficiency

Performance Solutions

1. Compile R code
    - Convert to bytecode
    - Integrate C/C++/Fortran with R
2. Utilize cloud computing (EC2)
3. Implement *parallel computing*
    - parallel
    - RHadoop

R Resources

- R Project homepage
- The R Journal
- R-bloggers: A collection of 500+ blogs on R
- Quick-R: Straightforward resource on essential R functions
- R documentation: Online help
- StackOverflow: Q&A posts on R
- Twitter: #rstats

Learning R: Videos

- Google Developers' Intro to R
- Twotorials: 2 minute tutorials on R
- Coursera: Statistics One

Learning R: Books

Beginner Level

- *R in Action* - Robert Kabacoff (2011)
- *The Art of R Programming* - Norman Matloff (2011)

Advanced Statistics

- *Modern Applied Statistics With S* - W. N. Venables and B. D. Ripley (2002)

Data Mining

- *An Introduction to Statistical Learning: With Applications in R* - James et al. (2013)
- *The Elements of Statistical Learning: Data Mining, Inference, and Prediction* - Hastie et al. (2009)

Learning R: Books

Data Visualization

- *ggplot2: Elegant Graphics for Data Analysis* - Hadley Wickham (2009)

Reference Manual

- *R Cookbook* - Paul Teetor (2011)
- *R in a Nutshell: A Desktop Quick Reference* - Joseph Adler (2010)
- *The R Book* - Michael Crawley (2007)

Advanced Programming

- *Software for Data Analysis: Programming with R* - John Chambers (2008)
- *Practical Data Science with R* - Nina Zumel and John Mount (2014)

Downloading and Installing R

- Official website of The R Project
- RStudio IDE
- Install and use packages in R

```
#install.packages("ggplot2")
library(ggplot2)
installed.packages()[5:10, 3:5] #Your output is likely to be different.
```

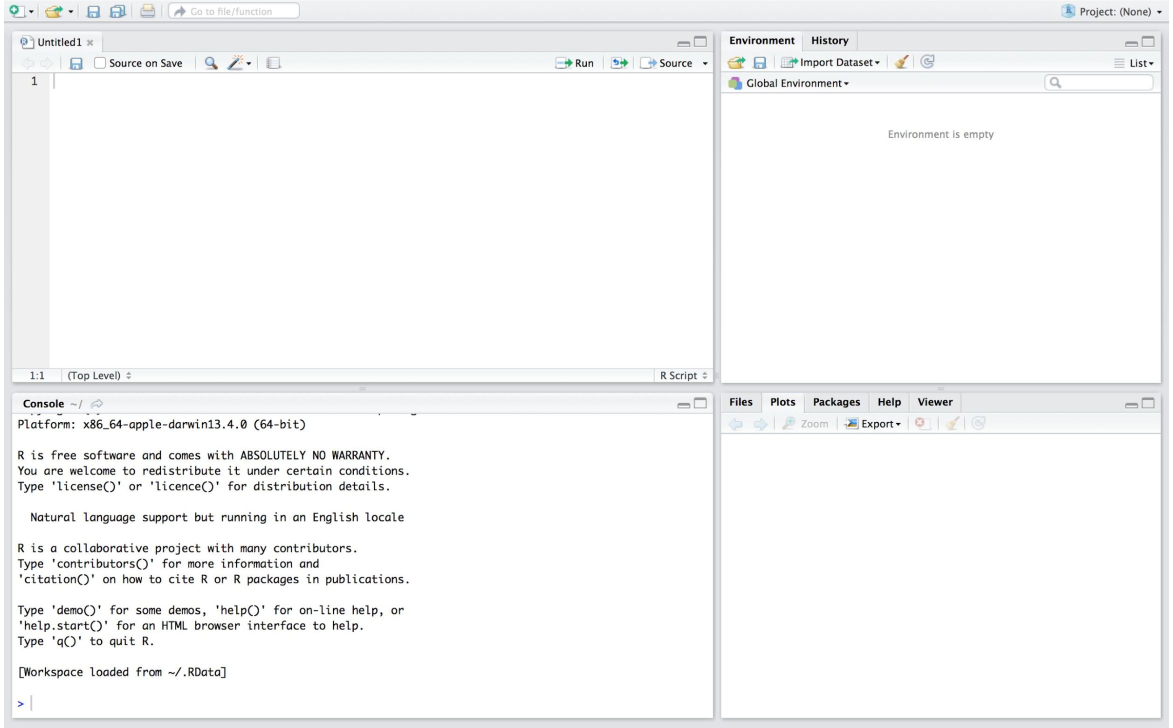
```
          Version  Priority      Depends
boot      "1.3-16" "recommended" "R (>= 3.0.0), graphics, stats"
brew      "1.0-6"  NA            NA
caTools   "1.17.1" NA            "R (>= 2.2.0)"
class     "7.3-12" "recommended" "R (>= 3.0.0), stats, utils"
cluster   "2.0.1"  "recommended" "R (>= 2.15.0), stats, utils"
codetools "0.2-11" "recommended" "R (>= 2.1)"
```

Introducing RStudio

- The RStudio interface is intuitive and simple.
- The scripting interface offers syntax highlighting and code completion.
- Many further utilities are provided, like a documentation browser and object listing.
- Supports mixed code in the document, including HTML, CSS, and LaTeX.

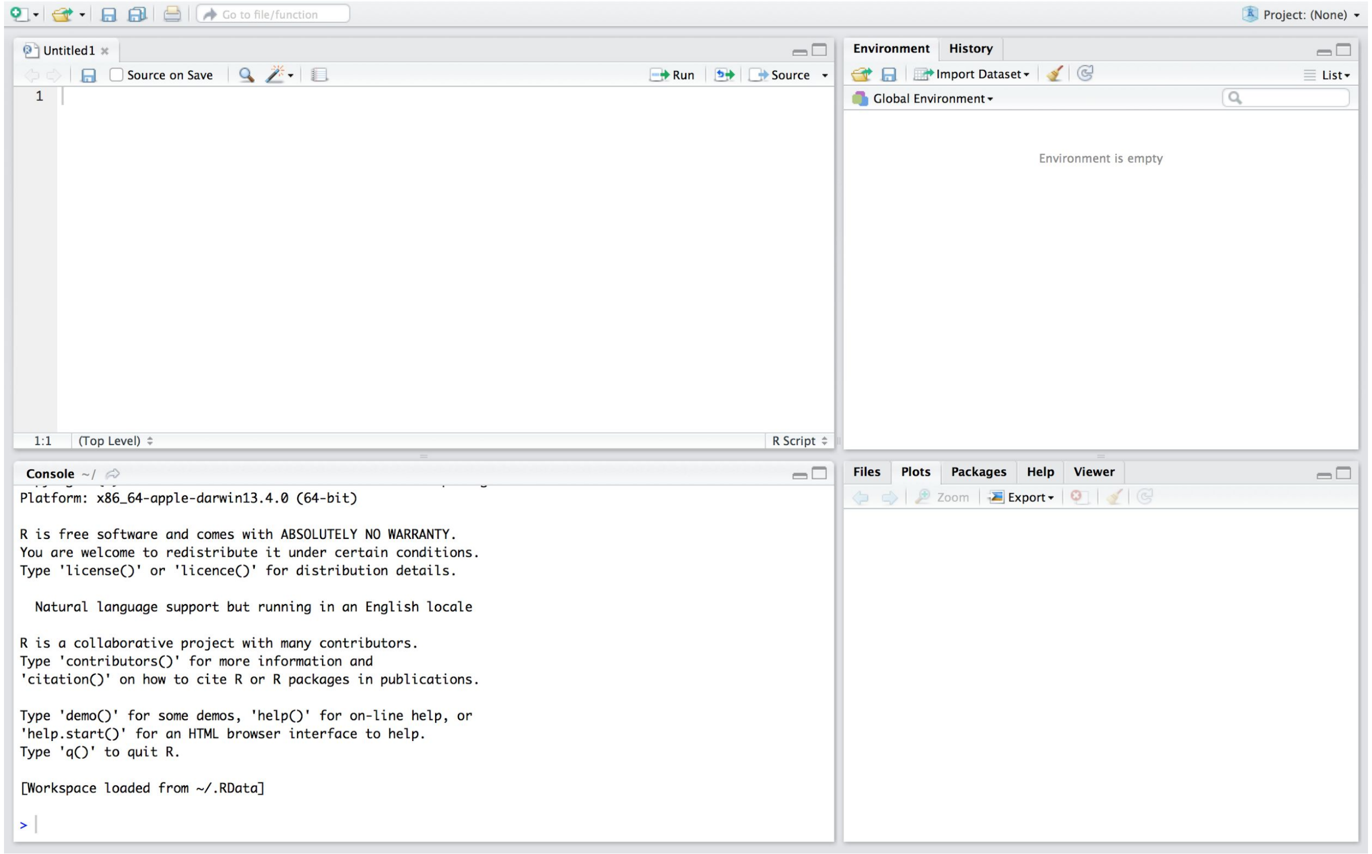
RStudio

- RStudio is an Integrated Development Environment (IDE) for R.
- It can be downloaded for free from the official RStudio website.
- The RStudio interface:

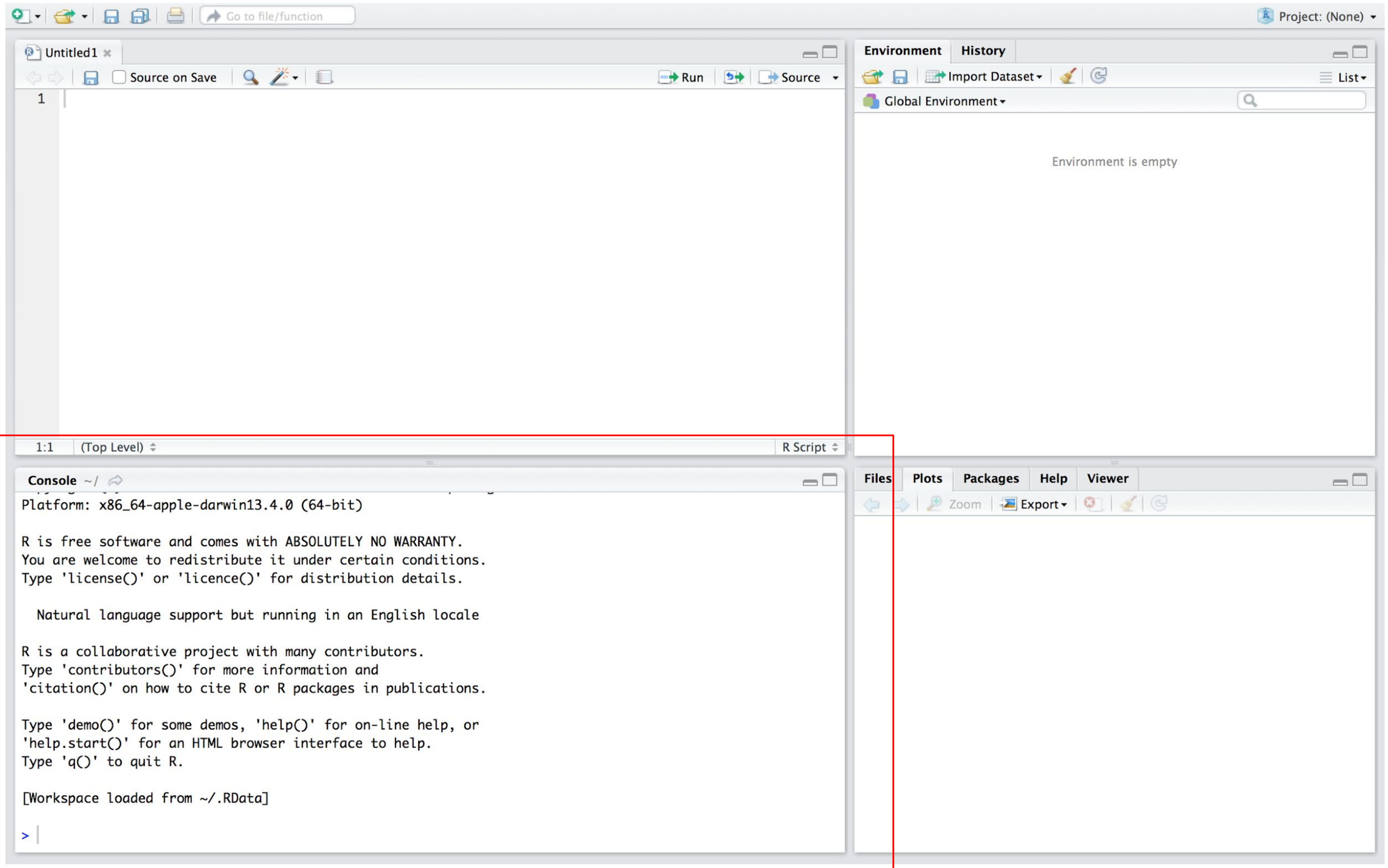


RStudio

- Let's explore each of the four panes of the interface.



RStudio Console



RStudio Console

- The console is like a calculator. You type in an R command next to a prompt “>,” and then press enter.

```
Console ~/
Platform: x86_64-apple-darwin13.4.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

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]

> 1 + 1
```

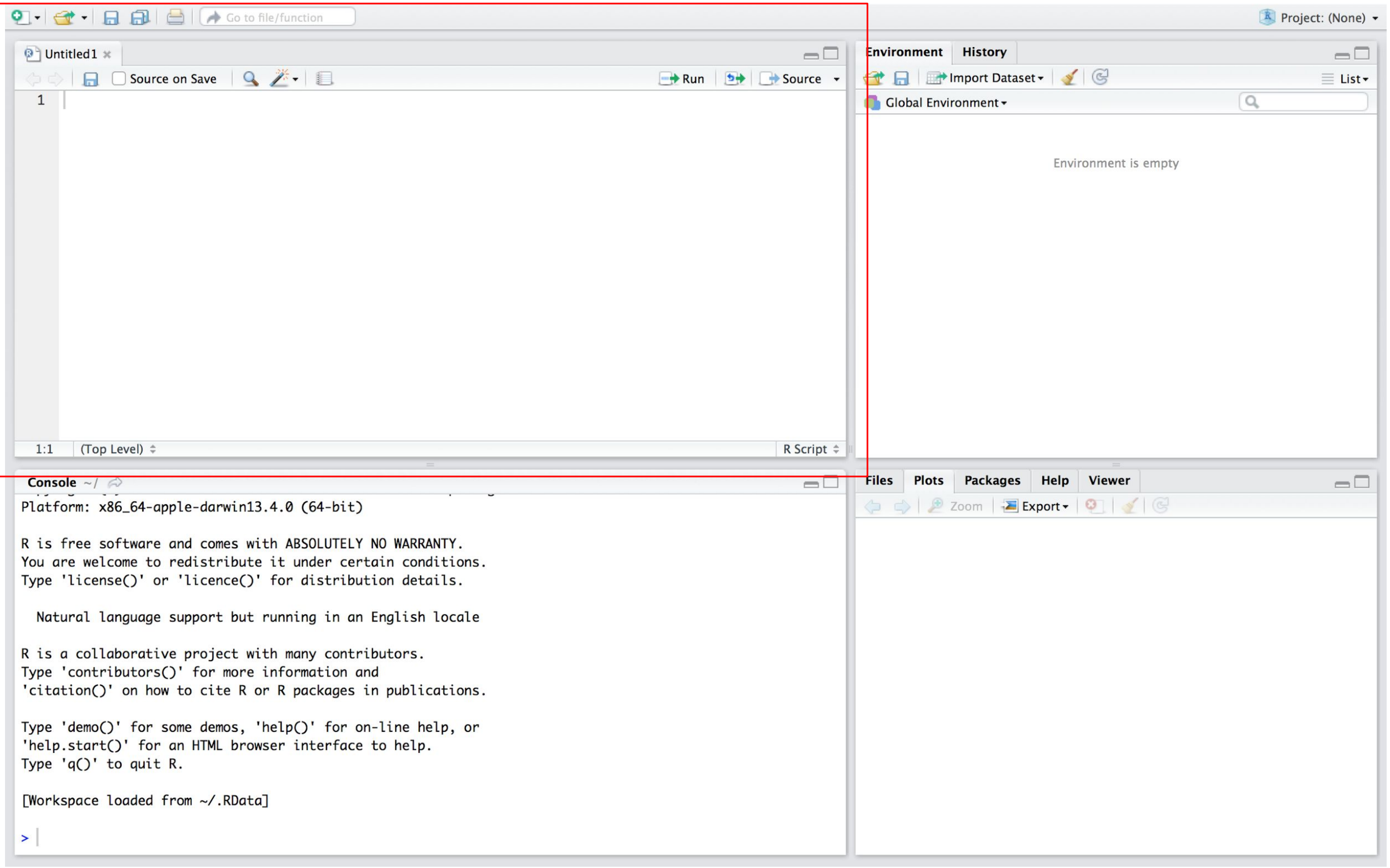
RStudio Console

- RStudio reports the result, and then prompts for the next command.

Console ~/

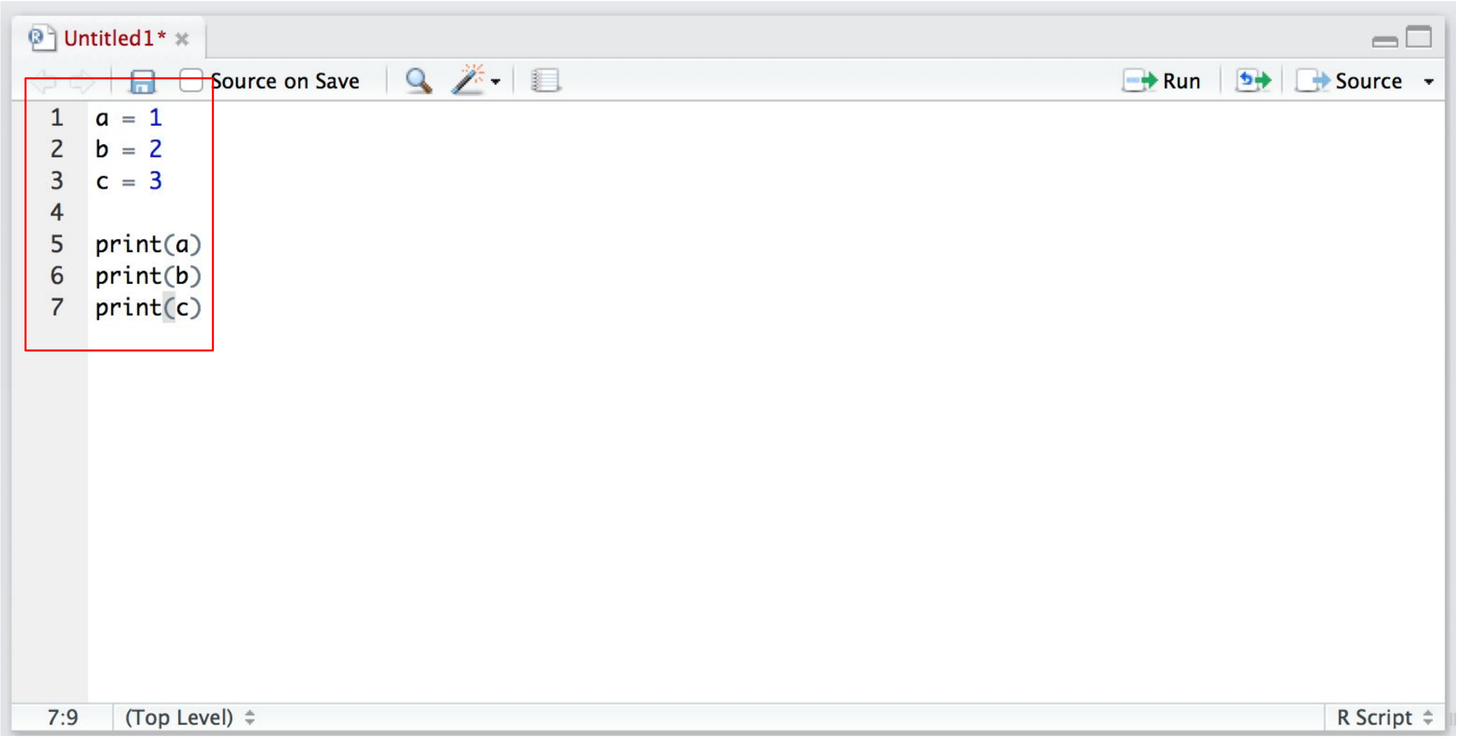
```
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

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]

> 1 + 1
[1] 2
>
```

RStudio Source Editor



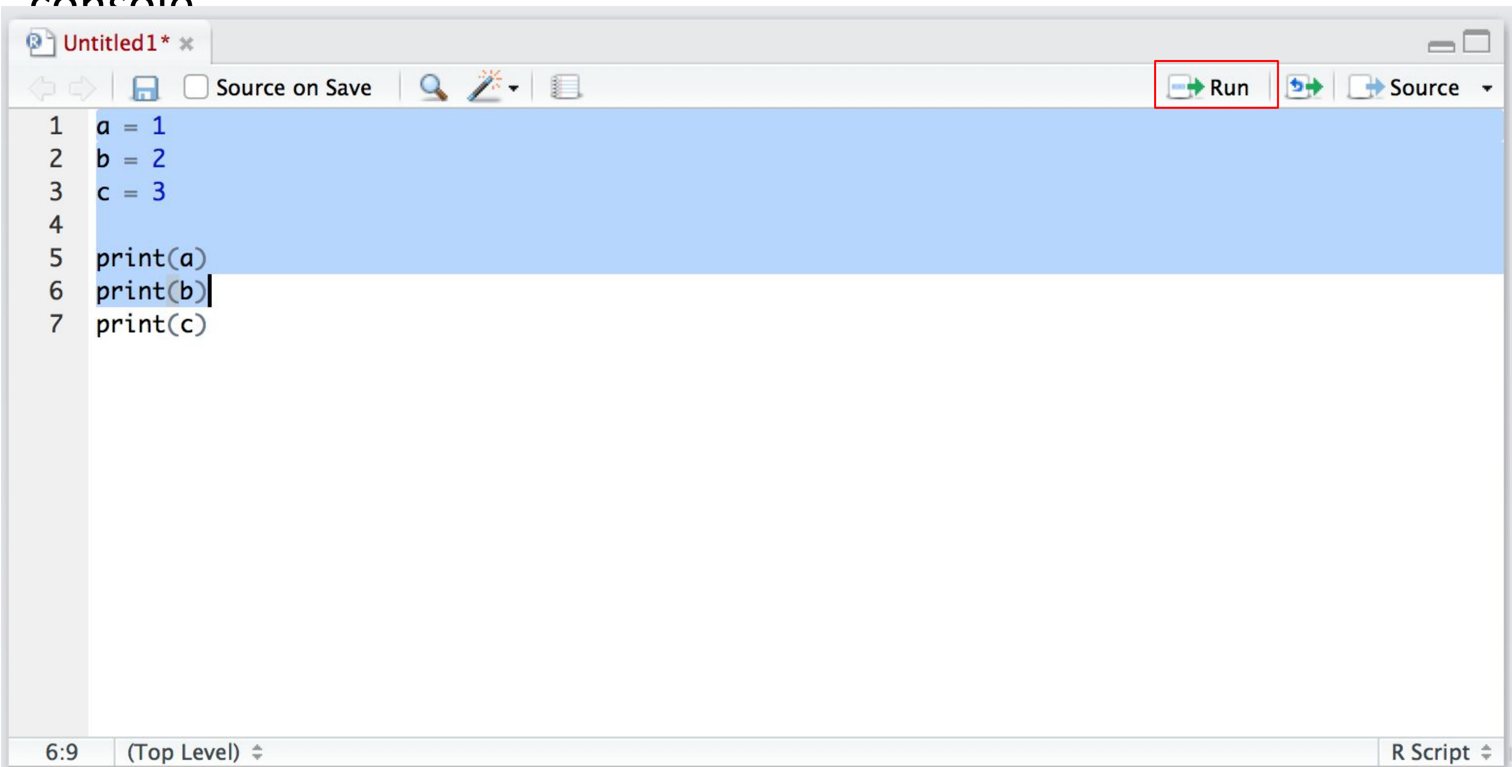
RStudio Source Editor

❖ The source editor is where you type lines of commands to form scripts; it is like a text editor. Hitting enter in the editor will add a new line instead of executing the code (like in the console).



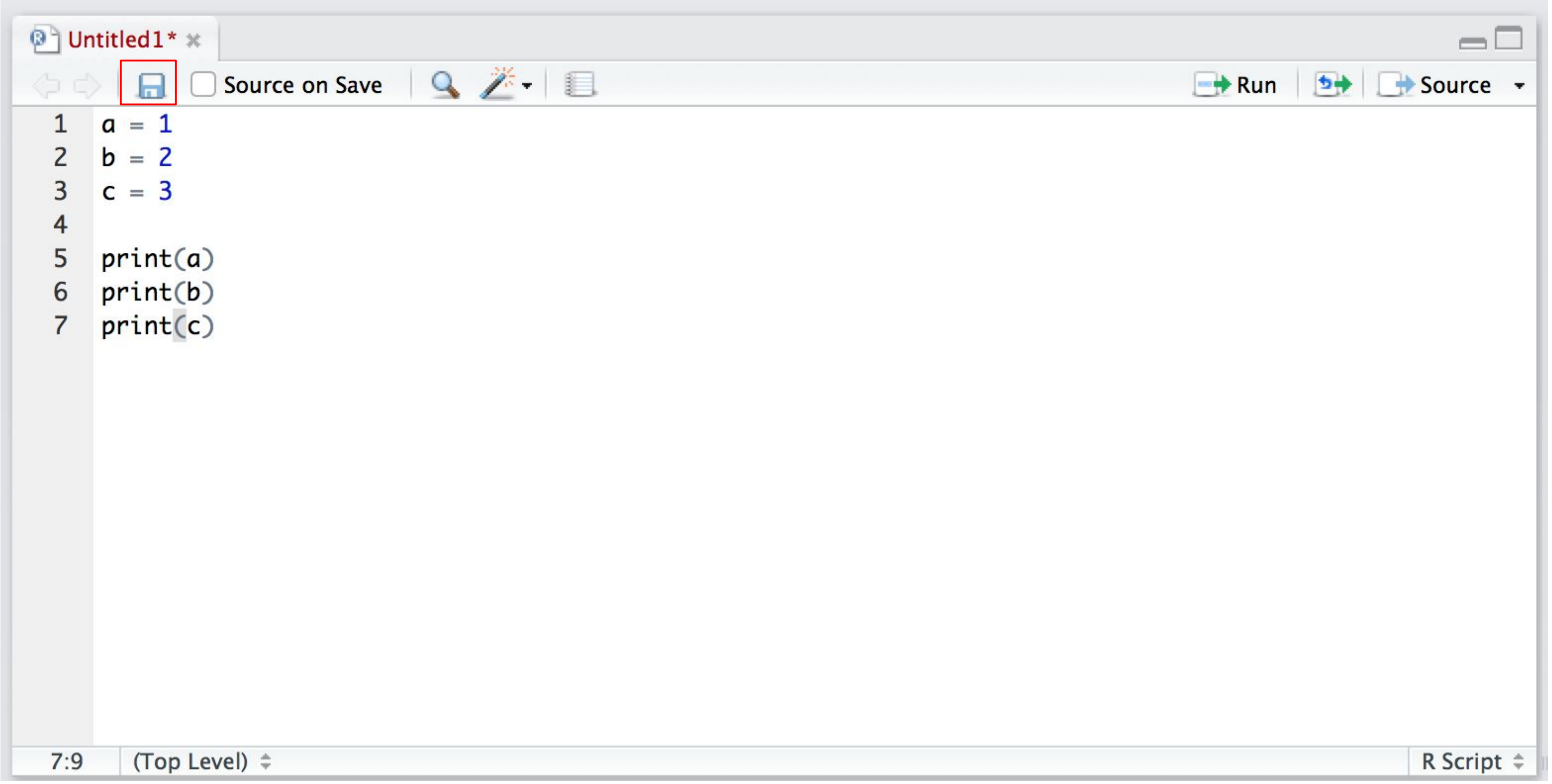
RStudio Source Editor

- You can, however, execute code from the source editor. First select the lines to execute, then hit "Run." You can also use the keyboard shortcut *command + enter/return.* The result will be shown in the console.



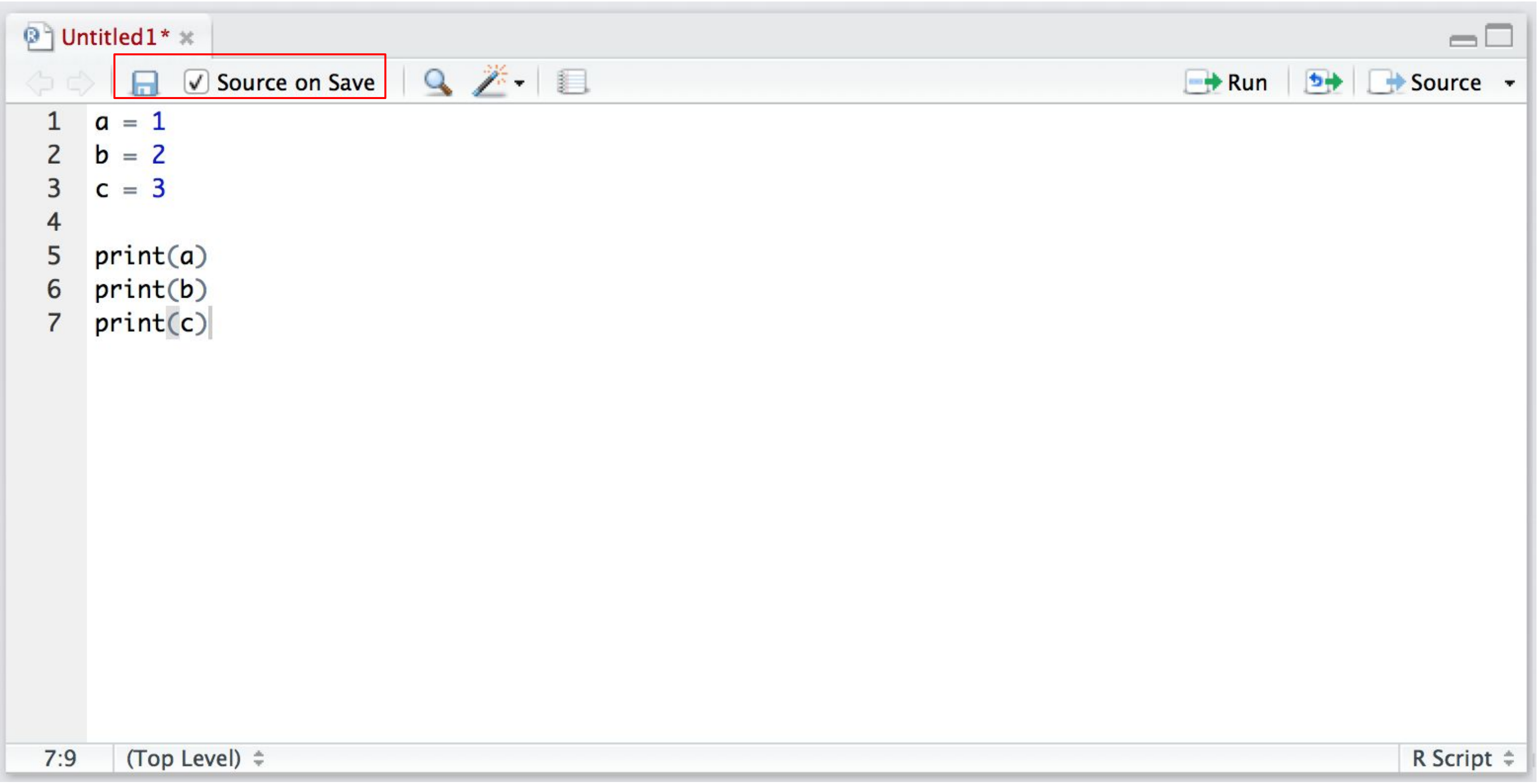
RStudio Source Editor

❖ Hitting the "save" icon will allow you to save the script.

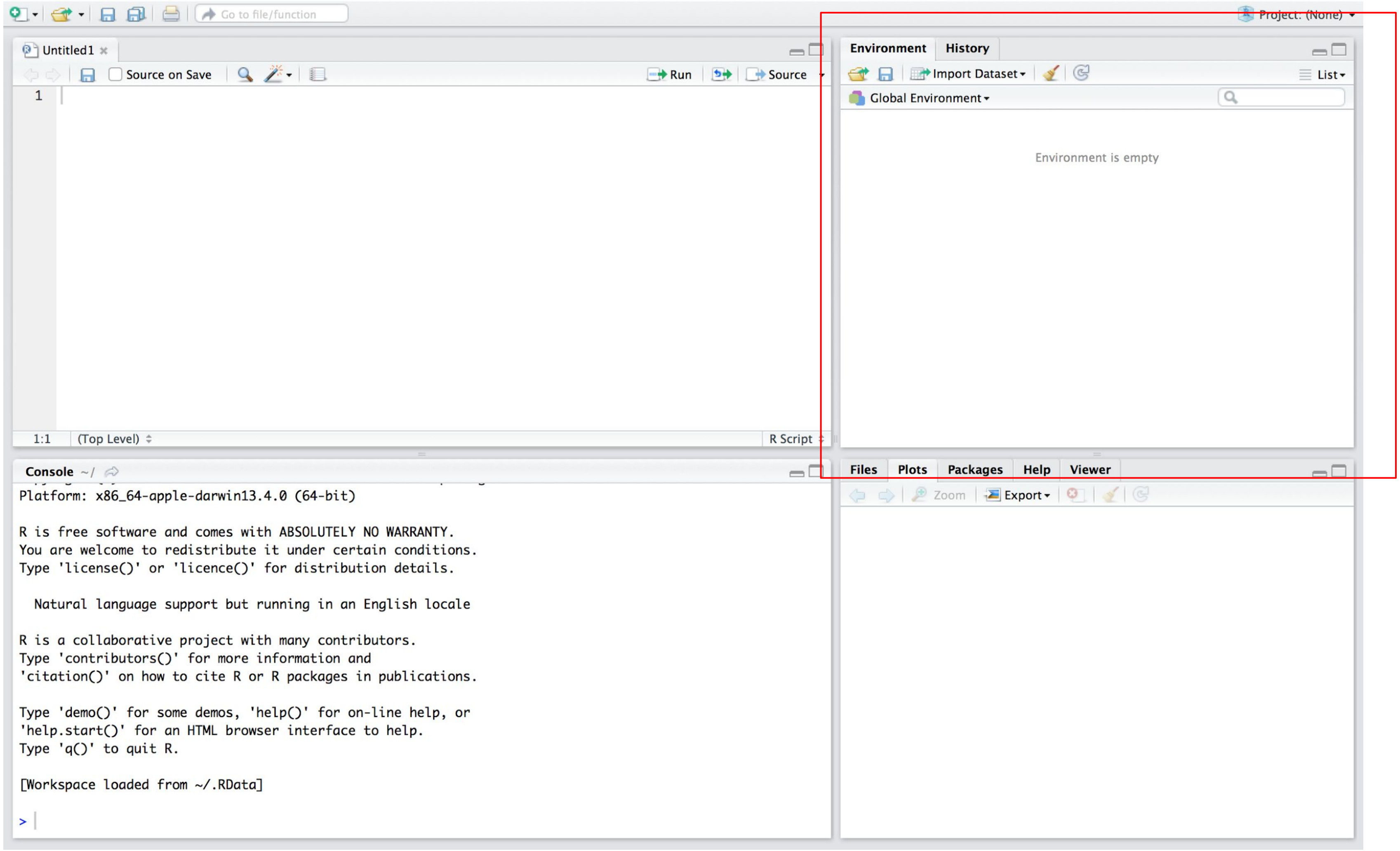


RStudio Source Editor

- Hitting the “save” icon with the box “Source on Save” checked will not only save, but also execute ALL lines within the source editor. Once again, the result will appear in the console.

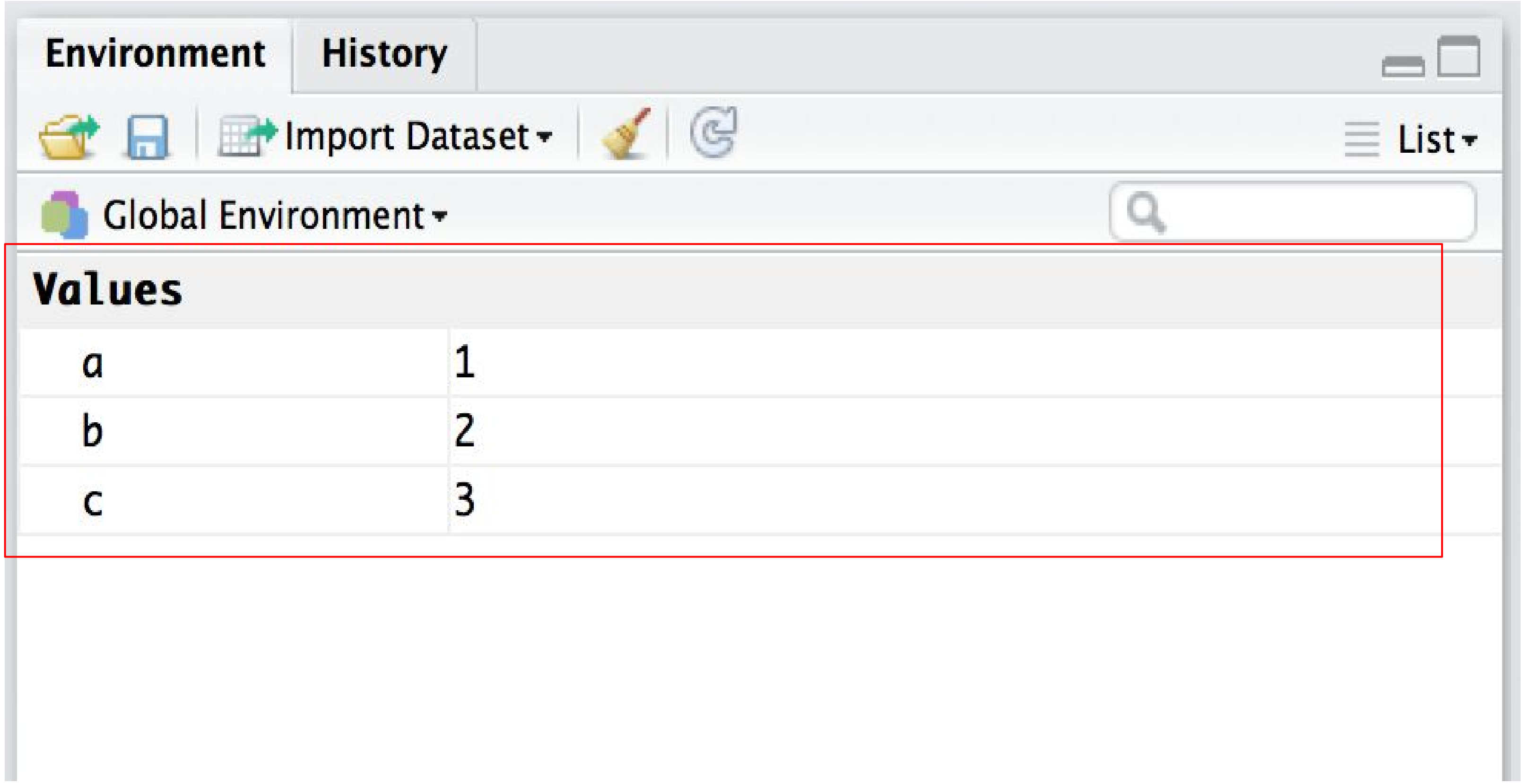


RStudio Environment



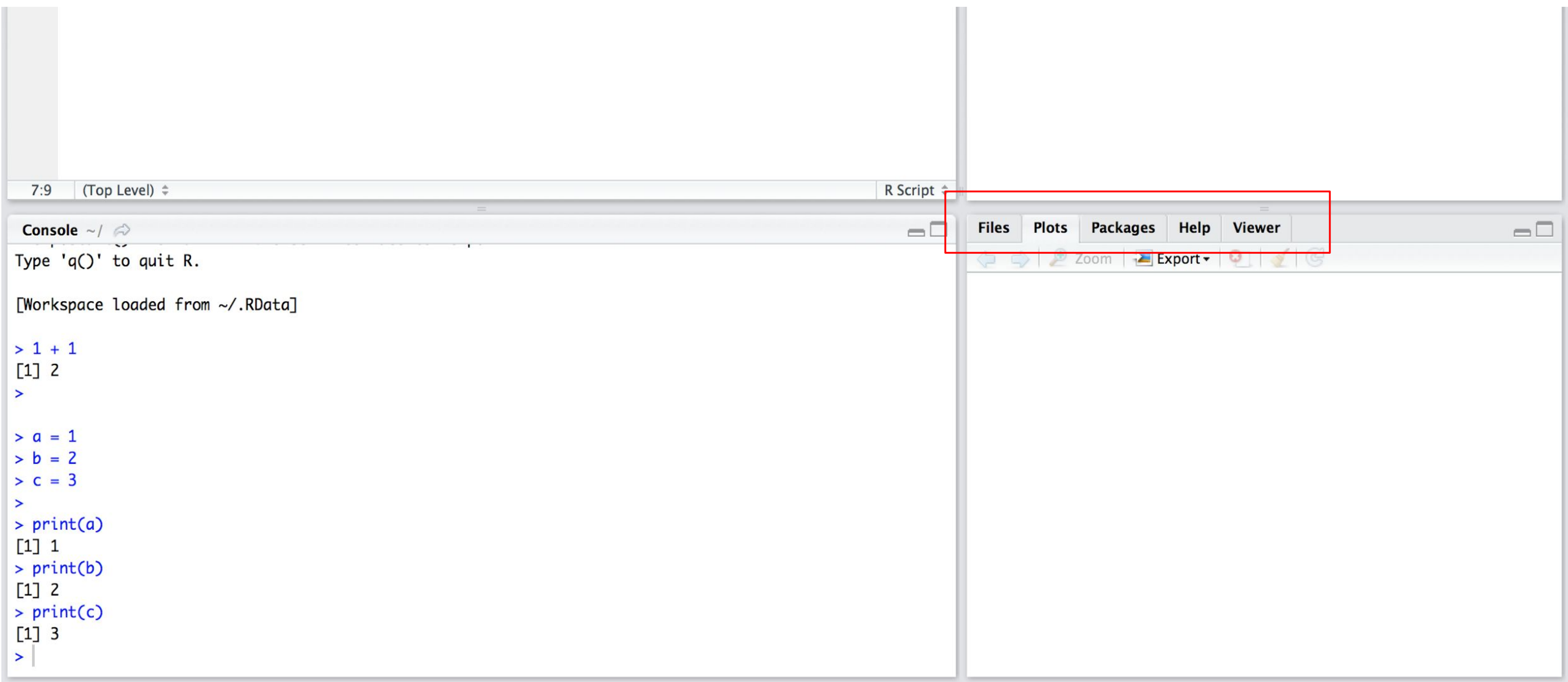
RStudio Environment

- The environment is like a catalog; it will list all variables you have created during the session.



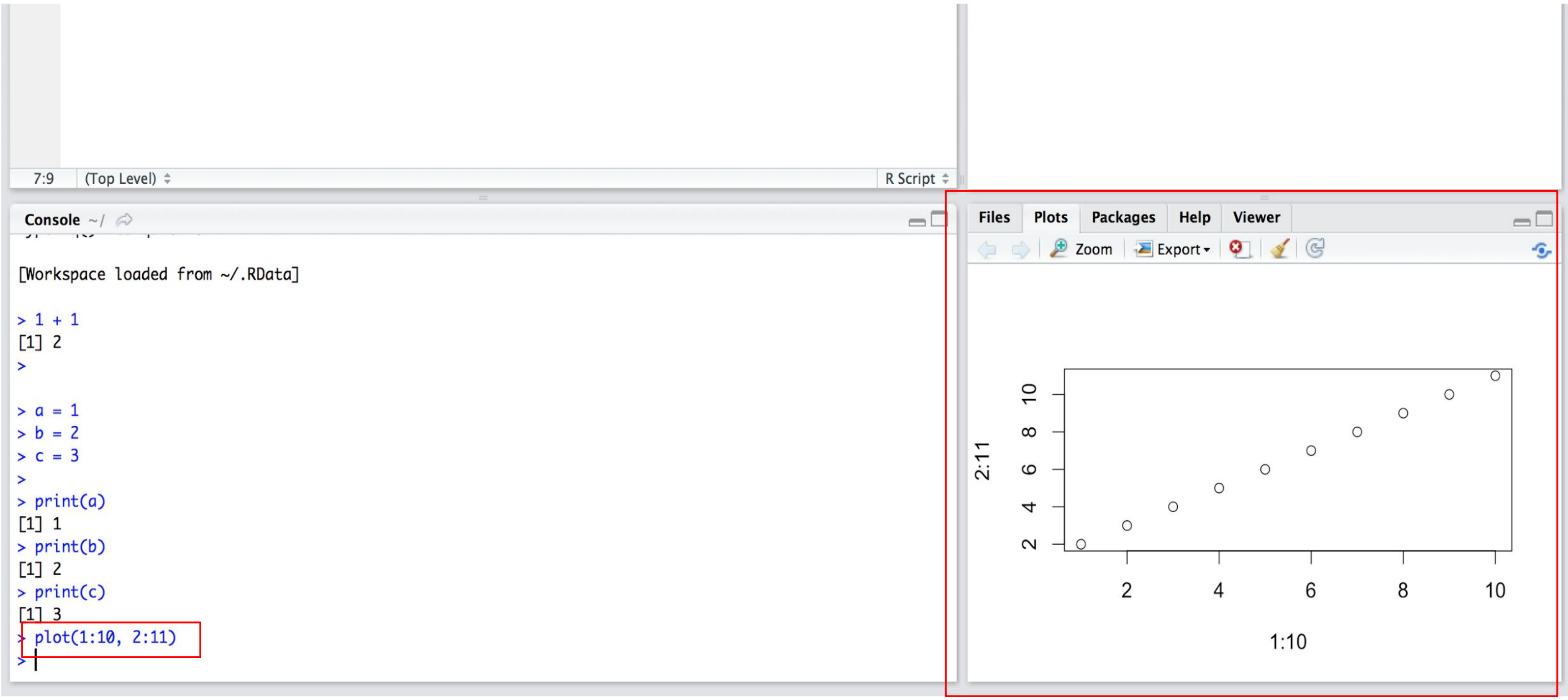
RStudio Plots and Help

- We see a lot going on in the lower right corner of RStudio interface. We will discuss only “Plots” and “Packages” here.



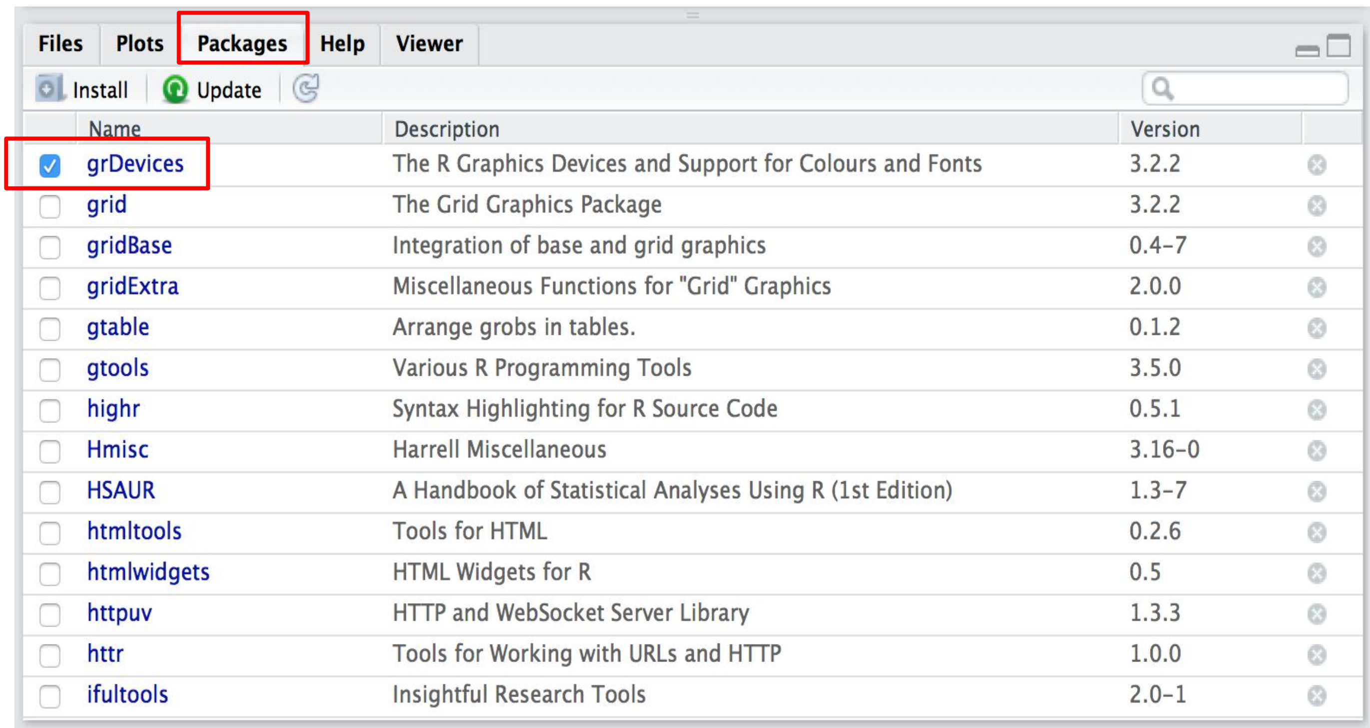
RStudio Plots

- If you enter a plotting command in the console (or execute it from the source editor), the plot shows up in the lower right corner.



RStudio Packages

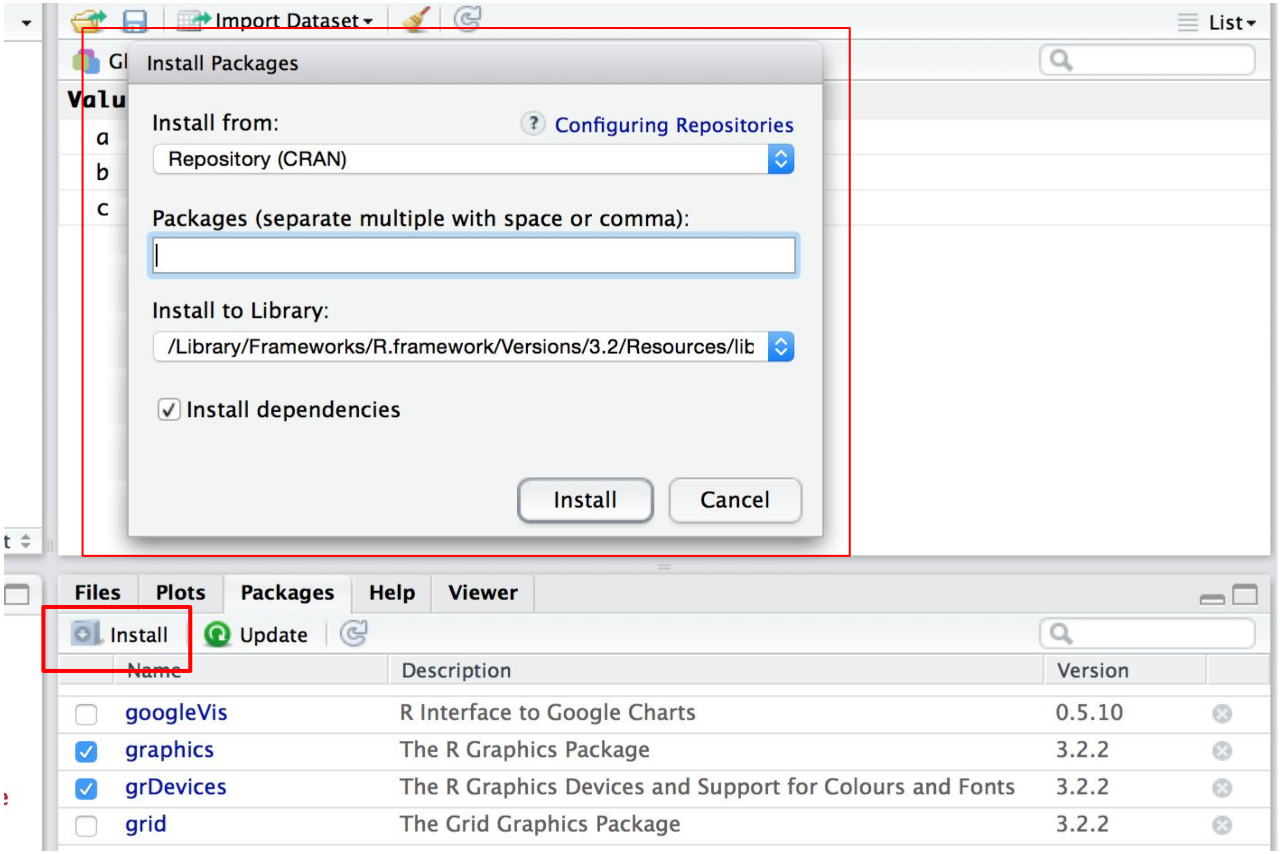
- Packages are like additional toolkits for R. The “Packages” tab shows summary information for installed packages. Those that are checked have been imported and that are available for use.



	Name	Description	Version
☑	grDevices	The R Graphics Devices and Support for Colours and Fonts	3.2.2
☐	grid	The Grid Graphics Package	3.2.2
☐	gridBase	Integration of base and grid graphics	0.4-7
☐	gridExtra	Miscellaneous Functions for "Grid" Graphics	2.0.0
☐	gtable	Arrange grobs in tables.	0.1.2
☐	gtools	Various R Programming Tools	3.5.0
☐	highr	Syntax Highlighting for R Source Code	0.5.1
☐	Hmisc	Harrell Miscellaneous	3.16-0
☐	HSAUR	A Handbook of Statistical Analyses Using R (1st Edition)	1.3-7
☐	htmltools	Tools for HTML	0.2.6
☐	htmlwidgets	HTML Widgets for R	0.5
☐	httpuv	HTTP and WebSocket Server Library	1.3.3
☐	httr	Tools for Working with URLs and HTTP	1.0.0
☐	ifultools	Insightful Research Tools	2.0-1

RStudio Packages

- If you hit the "Install" icon, a window will pop up in which you can search for and install new packages.



RStudio Packages

- *Installing* packages and *importing* them are different processes. One can think of an installed package as "stored" in your local machine; however, the tools in a package are not ready to be used if they are only installed.
- The user must import a package into the workspace before its tools can be used.
- Installing is like purchasing a toolbox, whereas importing is like opening the toolbox and laying the tools on your desk.
- To import a package, click the checkbox next to the package name in the "Packages" tab, or execute the following command in the console:

```
library(packagename)
```

How to Get Help

- help.start(): Open the help documentation Home
- help("lm") or ?lm: See documentation on the function lm() (quotes may be omitted)
- help.search("lm") or ??lm: Search *lm* as a keyword in local documentation
- RSiteSearch("lm"): Search *lm* as a keyword in online documentation
- example("lm"): Explore lm usage examples (quotes may be omitted)
- apropos("lm", mode="function"): List the names of all available functions containing lm()
- data(): List all of the available sample datasets in the currently loaded package
- vignette(): List all currently available vignette documents in installed packages
- vignette("interactive"): Display the interactive vignette from the grid package

Working Directory and Workspace

- Working directory
  - getwd(): Display the current working directory
  - setwd(): Change the current working directory
- Workspace
  - ls(): List the objects in the current workspace
  - rm(objectlist): Remove one or more objects from the workspace
  - rm(list = ls()): Remove all objects in the current workspace
  - save.image("myfile"): Save the workspace to the file 'myfile' in working directory (default suffix = .RData)
  - save(object, file = "myfile"): Save the specified object to a file
  - load("myfile"): Read a workspace into the current session

Coding Practice Time:

Open your Rstudio!

Run some code with me

Basic R Language Elements

Objects

Vector Basics

```
#basic arithmetic
1 + 1 * 3

#numerical and string vectors
c(0, 1, 1, 2, 3, 9)
c("Hello, World!", "I am an R user")
1:6

#vector addition
c(1, 2, 3, 4) + c(3, 4, 5, 6)
c(1, 2, 3, 4) + c(1, 2)
```

Objects

Comparison operators can be used on vectors with expected Boolean results:

```
c(1, 2, 3, 4) > c(1, 2, 1, 2)
```

```
[1] FALSE FALSE  TRUE  TRUE
```

```
c(1, 2, 3, 4) <= c(1, 5)
```

```
[1]  TRUE  TRUE FALSE  TRUE
```

Variables

R has a few ways to assign a value to an object. The default assignment operator is a pointing arrow: `<-`, but you can also use the equal sign: `=`, as in most other languages.

```
x = c(1, 2, 3, 4)
x
```

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

Elements and a range of elements in a vector can be accessed by using the index operator (the square brackets):

```
x[2]; x[2:4]
x[-4]
x[x > 2]
```

Logical Operators

- Equality: `x == y`
- Less than or equal to: `x <= y`
- Greater than or equal to: `x >= y`
- Logical AND operation: `x & y`
- Logical OR operation: `x | y`
- Logical NOT: `!x`
- Logical values TRUE and FALSE can be abbreviated as T and F (both must be uppercase). In arithmetic expressions, they will be converted to 1 and 0.

Arithmetic Operators

- Addition: `x + y`
- Subtraction: `x - y`
- Multiplication: `x * y`
- Division: `x / y`
- Exponentiation: `x ^ y`
- Modulo (remainder): `x %% y`
- Integer Division: `x %/% y`

Mathematical Functions

R has many built-in mathematical functions, which for the most part behave the same as user-defined functions.

```
exp(1)
```

```
[1] 2.718282
```

```
exp(c(1, 2, 3, 4))
```

```
[1]  2.718282  7.389056 20.085537 54.598150
```

Data Frames

A data frame is a spreadsheet-like data structure in which the data type of each column can be different, but the data length must be the same.

```
city = c('New York', 'San Francisco', 'Chicago', 'Houston', 'Los Angeles')
age = c(23, 43, 51, 32, 60)
sex = c('F', 'M', 'F', 'F', 'M')
people = data.frame(city, age, sex)
people
```

```
  city           age  sex
1 New York       23   F
2 San Francisco  43   M
3 Chicago        51   F
4 Houston        32   F
5 Los Angeles    60   M
```

Data Frames

We can use square brackets to extract data frame elements as we did for arrays, but another method is to use the $ symbol to extract a column.

```
people$age; people$sex
```

```
[1] 23 43 51 32 60
```

```
[1] F M F F M
Levels: F M
```

```
people$age > 30 #Conditioned samples extracted from column
people$city[people$age > 30] #Conditioning across variables
```

Importing Local Data into a Data Frame

Of course, most of the time we'll be working with data in external files. Importing text-format spreadsheet data can be done with the general read.table() function or with format-specific functions like read.csv().

```
inspections = read.csv('data/BrooklynInspectionResults.csv', header=TRUE)
inspections[c(66, 70, 71, 72), -2]
```

```
   DBA        CUISINE.DESCRIPTION  INSPECTION.DATE  VIOLATION.CODE  CRITICAL.FLAG
66 SACHIKO    Japanese             2014/01/02       10F             Not Critical
70 ALCHEMY    American             2014/01/02       02H             Critical
71 CHIMU      Peruvian             2014/01/02       04L             Critical
72 CHIMU      Peruvian             2014/01/02       06D             Critical
```

```
class(inspections)
```

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

Importing Local Data into a Data Frame

Now we have a data frame structure that can be manipulated just as we did with our manually created people data set above.

```
#Extract the restaurants surveyed
restaurants = inspections$DBA

#Count the number of unique restaurants in the data set
restaurant_set = unique(restaurants)
length(restaurant_set)
```

```
[1] 4651
```

```
#Limit the data to only those entries with critical violations
inspections = inspections[inspections$CRITICAL.FLAG == "Critical", ]
```

Importing Local Data into a Data Frame

While the basic import process in R is through text files accessed with the `read.table()` function, R has a number of packages which allow you to work with different file types and data formats.

```
#install.packages("openxlsx")
library(openxlsx)
excel_data = read.xlsx("data/excel.xlsx", sheet=1)  #read first sheet

#install.packages("foreign")
library(foreign)
stata_data = read.dta("data/statafile.dta")
spss_data = read.spss("data/spssfile.sav")
sas_data = read.xport("data/sasfile.xpt")
```

Exporting R Data to a Local File

The data export process is very similar to data import, only here we pass our data object to the `write.table()` or `write.csv()` function, adding the file name we want the data to be saved as.

```
write.table(people, file='write/people.csv', sep=',')
write.csv(people, file='write/people.csv') #Equivalent statement
```

Lists

Lists are the most flexible data structure; their elements can be of different types and different lengths.

```
people.list = list(AgeOfIndividual=age, Location=city, Gender=sex)
people.list
```

```
$AgeOfIndividual
[1] 23 43 51 32 60

$Location
[1] "New York"      "San Francisco" "Chicago"       "Houston"       "Los Angeles"

$Gender
[1] "F" "M" "F" "F" "M"
```

Lists

New elements can be added just like you would add a column to a data frame.

```
people.list$tabular.data = people
people.list$tabular.data
```

	city	age	sex
1	New York	23	F
2	San Francisco	43	M
3	Chicago	51	F
4	Houston	32	F
5	Los Angeles	60	M

Lists

We can use a double index operator to extract elements of a list. For example, to extract the last data element, you can do the following:

```
people.list[[length(people.list)]]
```

	city	age	sex
1	New York	23	F
2	San Francisco	43	M
3	Chicago	51	F
4	Houston	32	F
5	Los Angeles	60	M

Using Lists

- Lists are very useful structures, though this is not readily apparent at first.
- We will return to specific uses of lists later, but even before you start creating your own lists you will encounter them elsewhere in R.
- The most common early experience with lists will come in the form of values returned from statistical functions.

Object Features

For any one object, we can use the class() function to print its class(es). Additionally, you can use the attributes() function to print its properties.

```
class(people)
attributes(people)
```

str() is another useful function , which can be used to understand an object's class, attributes, and sample data.

```
str(people)
```

```
'data.frame':   5 obs. of  3 variables:
 $ city: Factor w/ 5 levels "Chicago","Houston",..: 4 5 1 2 3
 $ age : num  23 43 51 32 60
 $ sex : Factor w/ 2 levels "F","M": 1 2 1 1 2
```

Models and Formulas

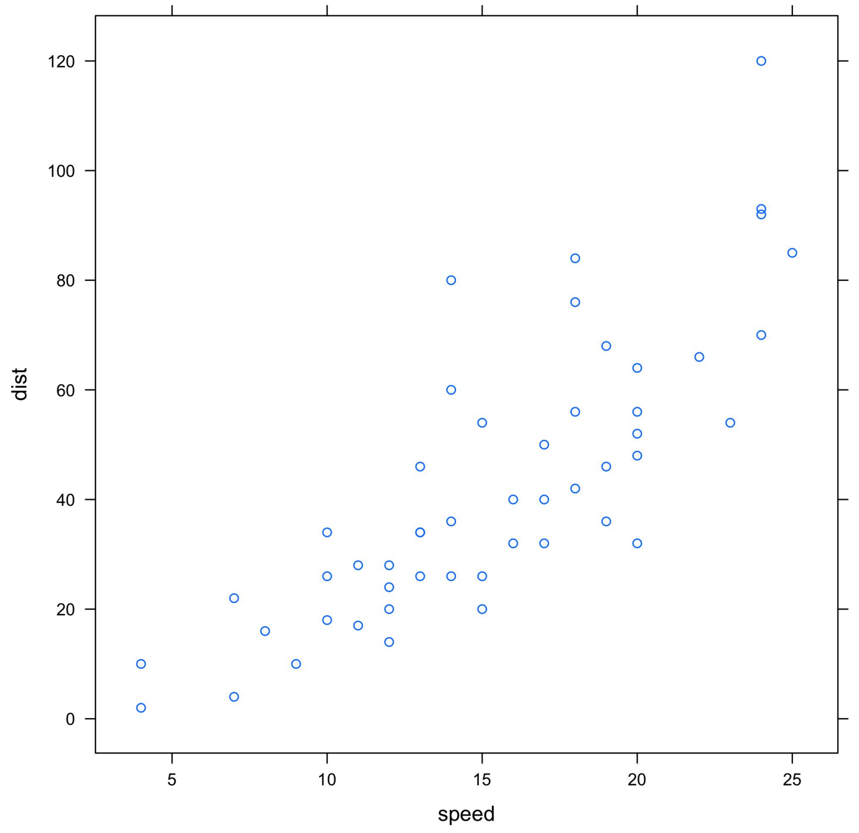
For statisticians, a model is a simple way to describe the data. In R, a formula is used as an expression of the model. Formulas are commonly used in graphical and statistical functions in R.

```
#A sample model y is a function of variables x1 to xn
y ~ x1 + x2 + x3 + ... + xn
```

For example, we can plot the relationship between distance and speed in the cars data set with the following function:

```
#install.packages("lattice")
library(lattice)
xyplot(dist ~ speed, data=cars)
```

Models and Formulas



Models and Formulas

The R function for linear regression is lm. Here we run the regression and store the results in the variable model:

```
model = lm(dist ~ speed, data=cars)
```

Models and Formulas

The summary function provides useful regression results, including variable coefficients and their corresponding *p* values, residuals, standard error and so on. You can also use the previously mentioned class, attributes, and str functions to understand the model object.

```
class(model)
```

```
[1] "lm"
```

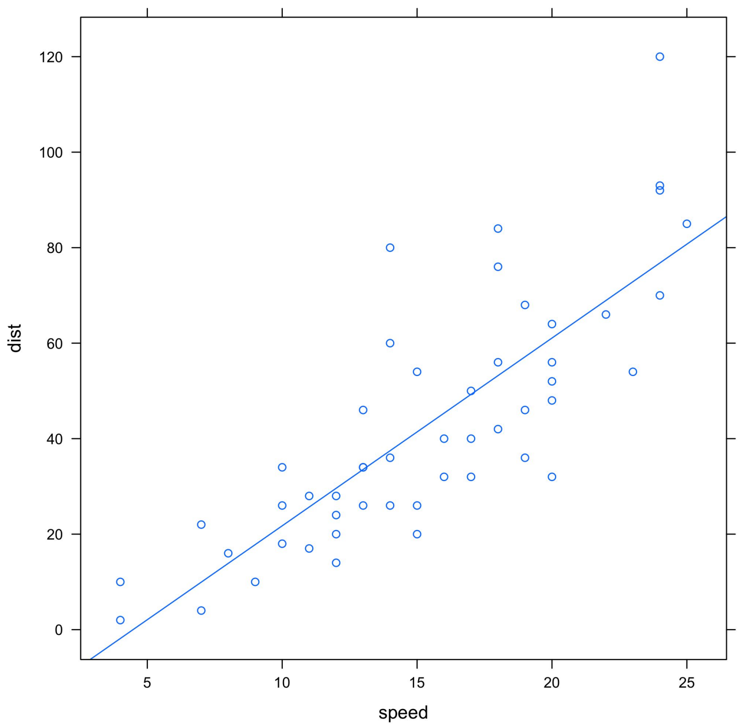
Models and Formulas

```
summary(model)
```

```
Call:
lm(formula = dist ~ speed, data = cars)

Residuals:
   Min     1Q Median     3Q    Max
-29.07  -9.53  -2.27   9.21  43.20

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  -17.579      6.758   -2.60    0.012 *
speed          3.932      0.416    9.46  1.5e-12 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 15.4 on 48 degrees of freedom
Multiple R-squared:  0.651, Adjusted R-squared:  0.644
F-statistic: 89.6 on 1 and 48 DF,  p-value: 1.49e-12
```

Models and Formulas

```
#Putting the two together
xyplot(dist ~ speed, data=cars, type = c("p","r"))
```



Review

- These fundamental elements of vectors, data frames, lists, and formulas are the building blocks of most of R's statistical and graphical operations.
- Now that you know the landscape of how to work with data in R, we will take some time to go in depth into these different data objects.
- Later we will introduce the essential programming concepts of writing functions and control statements to automate operations on these objects.

Coding Practice Time:

Open your Rstudio!

Run some code with me

In-depth Study of Data Objects

Atomic Object: Vectors

- A single value (scalar) is a special case of a vector.
- Elements of the vector must belong to a mode, which can be of integer, numeric, character, logical, or complex type.
- Recycling: In certain circumstances, a vector can be automatically extended.
- Use seq() to create a vector of sequential values.
- Use rep() to create a vector replicating from a pattern.

```
vector1 = seq(2, 10, by=2)
vector2 = 1:10 + 2
vector3 = 1:(10 + 2)
```

Example: Numerical Integration

Use the seq() function and numerical integration to approximate the area under the sine function from 0 to pi.

```
n = 100
w = pi/n
x = seq(from = 0, to = pi, length = n)
rect = sin(x) * w
sum(rect)
```

```
[1] 1.979834
```

Vector Generation

Creating categorical vectors with the rep() function:

```
group1 = rep(1:3, times=c(8, 10, 9))
group2 = factor(group1)
class(group1)
```

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

```
class(group2)
```

```
[1] "factor"
```

Vector Generation

Logical, *character*, and *numeric* vectors:

```
set.seed(0)
vec_logic = c(TRUE, TRUE, TRUE, FALSE)
vec_char = c('A', 'B', 'C', 'D')
vec_num1 = runif(5)
vec_char2 = sample(c('A', 'B'), size=10, replace=TRUE)
vec_num2 = numeric(10) #10-item zero vector
vec_logic; vec_char2; vec_num2
```

```
[1]  TRUE  TRUE  TRUE FALSE
```

```
[1] "A" "B" "B" "B" "B" "A" "A" "A" "B" "A"
```

```
[1] 0 0 0 0 0 0 0 0 0 0
```

Vector Computation

Suppose we want to remove the maximum and minimum values and then take the mean manually.

```
set.seed(0)
vector = rnorm(10)
vec_max = max(vector)
vec_min = min(vector)
vector_trimmed = vector[vector < vec_max & vector > vec_min]
vec_mean = mean(vector_trimmed)
vec_mean
```

```
[1] 0.340567
```

Matrices

One way to create a matrix is by wrapping a vector around multiple rows/columns.

```
vector = 1:12
my_matrix = matrix(vector, nrow = 3, ncol = 4, byrow = F) #Default.
my_matrix
dim(vector) = c(4, 3)
vector
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
```

Matrices

The function cbind() can be used to create a matrix by stacking column vectors.

```
set.seed(0)
vector1 = vector2 = vector3 = rnorm(3)
my_matrix = cbind(vector1, vector2, vector3)
vector1
my_matrix
```

```
[1]  1.2629543 -0.3262334  1.3297993


        vector1    vector2    vector3
[1,]  1.2629543  1.2629543  1.2629543
[2,] -0.3262334 -0.3262334 -0.3262334
[3,]  1.3297993  1.3297993  1.3297993
```

Matrix Degradation

Sometimes we want to extract a subset of a matrix. We can do this using the same index operator we used for vectors.

```
my_mat = matrix(1:9,3,3)
my_mat
```

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

```
my_mat[1:2, ]
```

```
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
```

Arrays

An array is a multidimensional vector. We create arrays as follows:

```
a = array(1:8, dim = c(2, 2, 2))
a
```

```
, , 1

     [,1] [,2]
[1,]    1    3
[2,]    2    4

, , 2

     [,1] [,2]
[1,]    5    7
[2,]    6    8
```

Data Frames

Creating a data frame of temperatures in different cities:

```
city = c('Seattle', 'Chicago', 'Boston', 'Houston')
temp = c(78, 74, 50, 104)
data = data.frame(city, temp)
```

Here are three different ways we can extract the city column:

```
data[ ,1]
data[ ,'city']
data$city
```

```
[1] Seattle Chicago Boston  Houston
Levels: Boston Chicago Houston Seattle
```

Data Frames

Converting variables to factors:

```
data = data.frame(city, temp, stringsAsFactors = F)
data$city = factor(data$city)
```

To find the cities with higher than average temperature:

```
ave = mean(data$temp)
data[data$temp > ave, ]
```

```
     city     temp
1    Seattle  78
4    Houston  104
```

Data Frames

The following functions are useful to understand the data structure you're working with.

```
data = data.frame(city, temp)
summary(data)
```

```
 city          temp
 Boston :1    Min.   : 50.0
 Chicago:1    1st Qu.: 68.0
 Houston:1    Median : 76.0
 Seattle:1    Mean   : 76.5
              3rd Qu.: 84.5
              Max.   :104.0
```

```
dim(data)
```

```
[1] 4 2
```

Data Frames

The following functions are useful to understand the data structure you're working with.

```
head(data)
```

```
     city       temp
1    Seattle    78
2    Chicago    74
3    Boston     50
4    Houston    104
```

```
str(data)
```

```
'data.frame':   4 obs. of  2 variables:
 $ city: Factor w/ 4 levels "Boston","Chicago",..: 4 2 1 3
 $ temp: num  78 74 50 104
```

Data Frames

Sorting data frames:

```
order(data$temp)
data[order(data$temp), ]
```

```
[1] 3 2 1 4
     city temp
3  Boston   50
2 Chicago   74
1 Seattle   78
4 Houston  104
```

```
data[order(data$temp, decreasing=TRUE), ][1:2, ]
```

```
      city      temp
4     Houston   104
1     Seattle   78
```

Missing and Null Values

When doing data analysis, you will often encounter data loss situations. Missing data in R is generally expressed as NA.

```
temp = c(27, 29, 23, 14, NA)
mean(temp)
```

```
[1] NA
```

```
mean(temp, na.rm=TRUE)
```

```
[1] 23.25
```

Missing and Null Values

A missing value means that the data exists but we don't know the value. This is different from the value NULL.

```
temp = c(27, 29, 23, 14, NULL)
length(temp)
```

```
[1] 4
```

NULL can sometimes be used to facilitate the removal of an element of a complex object, such as deleting an element from the previously defined people.list object.

```
people.list$tabular.data = NULL
```

Applying Functions to Data

The sapply() function can take a data frame as an input (or selected samples from a data frame), and apply a function as specified in the second argument.

```
sapply(iris[ ,1:4], function(x) sd(x)/mean(x))
```

```
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
   0.1417113    0.1425642    0.4697441    0.6355511
```

Applying Functions to Data

In addition to vectors and data frames, sapply can also operate on lists:

```
mylist = as.list(iris[ ,1:4])
sapply(mylist, mean)
```

```
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
    5.843333     3.057333     3.758000     1.199333
```

Applying Functions to Data

The lapply() function is similar to sapply but it returns the values in list form.

```
lapply(mylist, mean)
```

```
$Sepal.Length
[1] 5.843333

$Sepal.Width
[1] 3.057333

$Petal.Length
[1] 3.758

$Petal.Width
[1] 1.199333
```

Applying Functions to Data

Sometimes we want to convert the results of lapply to a more convenient format.

```r
myfunc = function(x) {
    ret = c(mean(x), sd(x))
    return(ret)
}
result = lapply(mylist, myfunc)
result
```

```
$Sepal.Length
[1] 5.8433333 0.8280661

$Sepal.Width
[1] 3.0573333 0.4358663

$Petal.Length
[1] 3.758000 1.765298

$Petal.Width
[1] 1.1993333 0.7622377
```

Applying Functions to Data

We can convert the result list to a matrix as follows:

```
result.matrix = t(as.data.frame(result))
colnames(result.matrix) = c("mean", "sd")
result.matrix
```

```
                 mean        sd
Sepal.Length 5.843333 0.8280661
Sepal.Width  3.057333 0.4358663
Petal.Length 3.758000 1.7652982
Petal.Width  1.199333 0.7622377
```

Applying Functions to Data

apply() allows for convenient manipulation of matrices.

```
set.seed(1)
vec = round(runif(12) * 100)
mat = matrix(vec, 3, 4)
apply(mat, 1, sum) #Applying across the rows.
```

```
[1] 218 144 228
```

```
apply(mat, 2, function(x) max(x)-min(x)) #Applying across the columns.
```

```
[1] 30 71 31 15
```

Applying Functions to Data

tapply() is useful when you want to apply a function to different subgroups.

```
tapply(X = iris$Sepal.Length, INDEX=list(iris$Species), FUN=mean)
```

```
    setosa versicolor  virginica
     5.006      5.936      6.588
```

For simplicity, the following method can also be used:

```
with(iris, tapply(X = Sepal.Length, INDEX=list(Species), FUN=mean))
```

Applying Functions to Data

There are other ways to do the same operation. aggregate is one such option.

```
with(iris, aggregate(Sepal.Length, by = list(Species), mean))
```

```
    Group.1     x
1     setosa 5.006
2 versicolor 5.936
3  virginica 6.588
```

Coding Practice Time:

Open your Rstudio!

Run some code with me

Control Statements

Conditionals

Typically, R code execution is performed sequentially by rows of text, but in order to perform more complex tasks we can execute code conditionally:

```
num = 5
if (num %% 2 != 0) {
  cat(num, 'is odd')
}
```

```
5 is odd
```

Conditionals

A simple if is sufficient if we only need to check one logical; however, if our control flow is a more complex tree, we must incorporate else statements for multiple branches.

```
num = 4
if (num %% 2 != 0) {
  cat(num, 'is odd')
} else {
  cat(num, 'is even')
}
```

```
4 is even
```

Conditionals

For more than two conditional branches, multiple if-else statements are necessary.

```
if (num %% 2 != 0) {
  cat(num, 'is odd')
} else if (num == 0) {
  cat(num, 'is even, although many people do not realize it.')
} else {
  cat(num, 'is even')
}
```

```
4 is even
```

Multiple Conditionals (ifelse)

If we want to run a conditional on a vector, we can use the function ifelse(). Returning to our example of the even/odd evaluator, we have:

```
num = 1:6
ifelse(num %% 2 == 0, yes='even', no='odd')
```

```
[1] "odd"  "even" "odd"  "even" "odd"  "even"
```

This ifelse structure can also be nested to evaluate multiple conditions:

```
set.seed(0)
age = sample(0:100, 20, replace=TRUE)
res = ifelse(age > 70, 'old', ifelse(age <= 30, 'young', 'mid'))
res
```

```
 [1] "old"   "young" "mid"   "mid"   "old"   "young" "old"   "old"   "mid"   "mid"
[11] "young" "young" "young" "mid"   "mid"   "old"   "mid"   "old"   "old"   "mid"
```

Multiple Conditionals (switch)

If you have many conditions, you might want to consider the switch() function. Demonstrating first with the age evaluator we just saw:

```
age[1]
```

```
[1] 90
```

```
age_group = cut(age, breaks=c(0,30,70,100), labels=FALSE) #Returns integers.
age_group
switch(age_group[1], 'young', 'middle', 'old')
```

```
 [1] 3 1 2 2 3 1 3 3 2 2 1 1 1 2 2 3 2 3 3 2
[1] "old"
```

Later we'll see how to make this process operate over the whole age set.

Multiple Conditionals (switch)

When the first parameter to switch is a string and not an integer, the function returns the value assigned to the matched string in the arguments that follow. For example, the following is the inverse of what we just did by assigning categories to ages.

```
age_type = 'middle'
switch(age_type,
      young = age[age <= 30],
      middle = age[age <= 70 & age > 30]  ,
      old = age[age > 70]
)
```

```
[1] 37 57 66 63 69 38 50 38
```

Multiple Conditionals (switch)

Let's take a practical case where we might want to divide a set of values into different categories and then query those assignments. Below we use a data set containing campaign contribution details for candidates running for local office in New York City in 2013.

```
campaign_data = read.csv('data/campaign_contributions.csv', header=TRUE)
campaign_data = campaign_data[campaign_data$AMNT > 0, ]
summary(campaign_data$AMNT)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   0.71   50.00  250.00  704.20  500.00 8250.00
```

```
str(campaign_data$CANDID)
```

```
Factor w/ 32 levels "1075","1078",..: 18 21 22 20 21 18 31 31 31 31 ...
```

Multiple Conditionals (switch)

We'll base our categories on the overall quantiles and then write a simple switch statement to tell us the percentage of a candidate's contributions that fit into a specified category.

```
id = "BB"  #Bloomberg.
size = "platinum"
data = campaign_data[campaign_data$CANDID == id, ]
switch(size,
      "bronze" = nrow(data[data$AMNT <= 50, ])/nrow(data),
      "silver" = nrow(data[data$AMNT <= 250 & data$AMNT > 50, ])/nrow(data),
      "gold" = nrow(data[data$AMNT <= 500 & data$AMNT > 250, ])/nrow(data),
      "platinum" = nrow(data[data$AMNT > 500, ])/nrow(data)
)
```

```
[1] 0.5285219
```

Loops

Loops can be used to repeatedly run a piece of code. For example, we can loop through the elements in a vector and identify the missing values as follows:

```
sign_data = read.csv('data/TimesSquareSignage.csv', header=TRUE)
obs = nrow(sign_data)
for (i in 1:obs) {
    if (is.na(sign_data$Width[i])) {
        cat('WARNING: Missing width for sign no.', i, '\n')
    }
}
```

Loops

```
WARNING: Missing width for sign no. 2
WARNING: Missing width for sign no. 11
WARNING: Missing width for sign no. 14
WARNING: Missing width for sign no. 22
WARNING: Missing width for sign no. 26
WARNING: Missing width for sign no. 36
WARNING: Missing width for sign no. 45
WARNING: Missing width for sign no. 54
WARNING: Missing width for sign no. 133
```

Loops

A while loop can accomplish the same task as follows:

```
i = 1
while (i <= obs) {
    if (is.na(sign_data$Width[i])) {
        cat('WARNING: Missing width for sign no.', i, '\n')
    }
    i = i + 1
}
```

Loops

```
WARNING: Missing width for sign no. 2
WARNING: Missing width for sign no. 11
WARNING: Missing width for sign no. 14
WARNING: Missing width for sign no. 22
WARNING: Missing width for sign no. 26
WARNING: Missing width for sign no. 36
WARNING: Missing width for sign no. 45
WARNING: Missing width for sign no. 54
WARNING: Missing width for sign no. 133
```

Loops

The advantage of the while loop vs. the for loop is that in some situations the number of times you want to run the loop is conditional on some other value. For example, maybe we only want to print out the first few warnings.

```
i = 1
nas = which(is.na(sign_data$Width))
while (i < 6) {
    cat('WARNING: Missing width for sign no.', nas[i], '\n')
    i = i + 1
    if (i > 5) {
        cat('WARNING: Turned up more than 5 missing values')
    }
}
```

Loops

```
WARNING: Missing width for sign no. 2
WARNING: Missing width for sign no. 11
WARNING: Missing width for sign no. 14
WARNING: Missing width for sign no. 22
WARNING: Missing width for sign no. 26
WARNING: Turned up more than 5 missing values
```

Loops

Another loop, less commonly used but actually more valuable for our missing value program, is the repeat loop, which will run until you break out of it.

```
i = 1
j = 1
repeat {
    if (is.na(sign_data$Width[i])) {
        cat('WARNING: Missing width for sign no.', i, '\n')
        j = j + 1
    }
    if (j > 5) {
        cat('WARNING: Encountered more than 5 missing values')
        break
    }
    i = i + 1
    if (i > nrow(sign_data)) {
        break
    }
}
```

Loops

```
WARNING: Missing width for sign no. 2
WARNING: Missing width for sign no. 11
WARNING: Missing width for sign no. 14
WARNING: Missing width for sign no. 22
WARNING: Missing width for sign no. 26
WARNING: Encountered more than 5 missing values
```

Loop Efficiency

In R, growing a vector internally will result in recopying the vector; this consumes memory and computation time. We use an example of determining whether a number is a prime number to compare the speeds of different methods.

```
findprime = function(x) {
    if (x %in% c(2, 3, 5, 7)) return(TRUE)
    if (x %% 2 == 0 | x == 1) return(FALSE)
    xsqrt = round(sqrt(x))
    xseq = seq(from = 3, to = xsqrt, by = 2)
    if (all(x %% xseq != 0)) return(TRUE)
    else return(FALSE)
}
```

Loop Efficiency

We now compare the speeds of three methods using the system.time function:

```
system.time({
    x1 = c()
    for (i in 1:1e4) {
        y = findprime(i)
        x1[i] = y
    }
})
```

```
user      system    elapsed
0.240     0.036     0.277   #Your output is likely to be different.
```

Loop Efficiency

```
system.time({
    x2 = logical(1e4)
    for (i in 1:1e4) {
        y = findprime(i)
        x2[i] = y
    }
})
```

```
user      system    elapsed
0.177     0.003     0.181   #Your output is likely to be different.
```

Loop Efficiency

```
system.time({
  sapply(1:1e4, findprime)
})
```

```
user    system   elapsed
0.173   0.001    0.174  #Your output is likely to be different.
```

When to Use (Explicit) Loops

When loop iterations depend on previous iterations, it is more difficult to avoid the explicit loop structure. Here we use an example of finding Fibonacci numbers below 1000.

```
i = 2
x = 1:2
while (x[i] < 1e3) {
  x [i+1] = x[i-1] + x[i]
  i = i + 1
}
x = x[-i]
print(x)
```

```
[1]   1   2   3   5   8  13  21  34  55  89 144 233 377 610 987
```

Functions

Custom Functions

A function to calculate the area of a circle with radius r:

```
calc_area = function(r) {
   area = pi*r^2
   return(area)
}
calc_area(4)
```

```
[1] 50.26548
```

Custom Functions

Here we have a custom function to convert degrees to radians.

```
DegreesToRadians = function(d) {
    valueInRadians = d * pi / 180
    return(valueInRadians)
}
DegreesToRadians(145)
```

```
[1] 2.530727
```

Custom Functions

For more complicated functions we can introduce multiple parameters. For example, the following function calculates the volume of a cone based on two input arguments: its *radius* and its *height*.

```
ConeVolume = function(r, h) {
    volume = pi * r^2 * (h / 3)
    return(volume)
}
ConeVolume(2, 5)
```

```
[1] 20.94395
```

Functions

R functions also allow you to specify default parameters:

```
SDcalc = function(x, type='sample') {
    n = length(x)
    mu = mean(x)
    if (type == 'sample') {
        stdev = sqrt(sum((x-mu)^2)/(n-1))
    }
    if (type == 'population') {
        stdev = sqrt(sum((x-mu)^2)/(n))
    }
    return(stdev)
}
SDcalc(1:10); SDcalc(1:10, type='population')
```

```
[1] 3.02765
```

```
[1] 2.872281
```

Functions

There is no built-in *safety* for function arguments. You must do the checking manually:

```
SDcalc = function(x, type = 'sample') {
  stopifnot(is.numeric(x), length(x) > 0,
            type %in% c('sample', 'population'))
  x = x[!is.na(x)]
  n = length(x)
  mu = mean(x)
  if (type == 'sample') {
    stdev = sqrt(sum((x-mu)^2)/(n-1))
  }
  if (type == 'population') {
    stdev = sqrt(sum((x-mu)^2)/(n))
  }
  return(stdev)
}
```

Functions

What if our input has NAs, or we need to transform it in some way prior to calculating the standard deviation? There is a special parameter '...', which gives us access to embedded function parameters.

```
SDcalc = function(x, type = 'sample', ...) {
  stopifnot(is.numeric(x), length(x) > 0,
            type %in% c('sample', 'population'))
  n = length(x)
  mu = mean(x, ...)
  if (type == 'sample') {
    stdev = sqrt(sum((x-mu)^2, ...)/(n-1))
  }
  if (type == 'population') {
    stdev = sqrt(sum((x-mu)^2, ...)/(n))
  }
  return(stdev)
}
```

Functions

```
test = c(1:10, NA)
SDcalc(test, type='sample')
```

```
[1] NA
```

```
SDcalc(test, type='sample', na.rm=TRUE)
```

```
[1] 2.872281
```

Recursion

Recursion is a technique by which a function calls itself. An intuitive way to define a function that calculates a factorial is the following:

```
Fac1 = function(n) {
  if (n == 0) return(1)
  return(n * Fac1(n-1))
}
Fac1(10)
```

```
[1] 3628800
```

Recursion

```
#Compare the recursive definition with this one:
Fac2 = function(n) {
  if (n == 0) {
    return(1)
  } else {
  res = n
    while (n > 1) {
      res = res * (n - 1)
      n = n - 1
    }
  }
  return(res)
}
Fac2(10)
```

```
[1] 3628800
```

Function Input

Recall our conditional which placed ages into the categories: old, mid, and young.
Let's first write a similar function which divides people into generations by birth year.

```
which_generation = function(birth_year) {
    if (birth_year > 2000) {
        category = 'Gen Z'
    } else if (birth_year > 1985) {
        category = 'Gen Y'
    } else if (birth_year > 1965) {
        category = 'Gen X'
    } else {
        category = 'Baby Boomer'
    }
    return(category)
}
```

Function Input

```
#First test on a single birth year
which_generation(1989)
```

```
[1] "Gen Y"
```

```
#Now with a set of birth years
years = c(1950, 1973, 1990, 2005)
which_generation(years)
```

```
[1] "Baby Boomer"
Warning messages:
1: In if (birth_year > 2000) { :
  the condition has length > 1 and only the first element will be used
2: In if (birth_year > 1985) { :
  the condition has length > 1 and only the first element will be used
...
```

Function Input

The type of object input to a function matters. In this case we want to handle vector input. Some functions can be *vectorized* to accomplish this.

```
which_generation = Vectorize(which_generation)
which_generation(years)
```

```
[1] "Baby Boomer" "Gen X"       "Gen Y"       "Gen Z"
```

Functions: Creating Custom Operators

You can create your own binary operators with the same function procedure. For example, we can define a set operator to find the intersection of two sets:

```
a <- c('NPR', 'New York Times', 'MSNBC')
b <- c('Wall Street Journal', 'NPR', 'Fox News')

'%int%' = function(x, y) {
  intersect(x, y)
}
a %int% b
```

```
[1] "NPR"
```

Functions: Creating Custom Operators

Binary operators only accept two arguments. You can either chain them together or use Reduce():

```
c = c('Salon', 'The Onion', 'NPR')
a %int% b %int% c
```

```
[1] "NPR"
```

```
news_list = list(a, b, c)
Reduce('%int%', news_list)
```

```
[1] "NPR"
```

Functional Programming

R is a functional programming language. It is commonly said that in R, functions are treated as *first class* objects.

- Some functions have names, but not all do
- A function can be returned from another function
- A function can also be used as an argument

Functional Programming

We'll demonstrate some of these features by first creating a list of several functions.

```
FuncList <- list(base = function(x) mean(x),
                 med = function(x) median(x),
                 manual = function(x) {
                   n <- length(x)
                   x <- sort(x)[c(-1,-n)]
                   mean(x)
                 })
set.seed(1)
x <- sample(100, 10)
FuncList$base(x)
```

```
[1] 53.9
```

Functional Programming

We can loop through the functions in two ways:

```
for (f in FuncList) {
  print(f(x))
}
```

```
[1] 53.9
[1] 57.5
[1] 54.5
```

```
sapply(FuncList, function(f) f(x))
```

```
base    med     manual
53.9    57.5    54.5
```

Functional Programming

Let's look at an example that we studied previously: calculating the standard deviation.

```
SdMean = function(x, type='sample') {
  stopifnot(is.numeric(x),
            length(x) > 0,
            type %in% c('sample', 'population'))
  x = x[!is.na(x)]
  n = length(x)
  mu = mean(x)
  if (type == 'sample') n = n-1
  stdev = sqrt(sum((x-mu)^2)/(n))
  return(stdev)
}
```

Functional Programming

Now we write a similar function but with deviations calculated about the *median*.

```
SdMedian = function(x, type='sample') {
  stopifnot(is.numeric(x),
            length(x) > 0,
            type %in% c('sample', 'population'))
  x = x[!is.na(x)]
  n = length(x)
  med = median(x)
  if (type == 'sample') n = n-1
  stdev = sqrt(sum((x-med)^2)/(n))
  return(stdev)
}
```

Functional Programming

We can combine the previous two functions into one:

```
SdFunc = function(x, func, type='sample') {
  stopifnot(is.function(func), is.numeric(x), length(x) > 0,
            type %in% c('sample', 'population'))
  x = x[!is.na(x)]
  n = length(x)
  m = func(x)
  if (type == 'sample') n = n-1
  stdev = sqrt(sum((x-m)^2)/(n))
  return(stdev)
}
```

Functional Programming

```
set.seed(1)
x = sample(100, 30)
SdFunc(x, type='sample') #Note the error. Missing func
```

```
Error in stopifnot(is.function(func), is.numeric(x), length(x) > 0, type %in%  :
  argument "func" is missing, with no default
```

```
SdFunc(x, func=median, type='sample')
```

```
[1] 29.00832
```

```
SdFunc(x, func=FuncList$manual, type='sample')
```

```
[1] 28.33786
```

Functional Programming

We can also use the option of returning a function to simplify the function definitions of SdMean and SdMedian. Why might we do this?

```
SdFunc = function(func, type) {
    stopifnot(is.function(func),
              type %in% c('sample', 'population'))
    function(x) {
        stopifnot(is.numeric(x), length(x) > 0)
        x = x[!is.na(x)]
        n = length(x)
        m = func(x)
        if (type == 'sample') n = n-1
        stdev = sqrt(sum((x-m)^2)/(n))
        return(stdev)
    }
}
```

Functional Programming

Now we can dynamically generate the SdMean and SdMedian functions from the SdFunc function!

```
SdMean = SdFunc(func=mean, type='sample')
SdMedian = SdFunc(func=median, type='sample')


set.seed(1)
x = sample(100, 30)
SdMean(x)
```

```
[1] 28.33707
```

```
SdMedian(x)
```

```
[1] 29.00832
```

Functional Programming

Now we can dynamically generate the SdMean and SdMedian functions from the SdFunc function!

```
SdMean = SdFunc(func=mean, type='sample')
SdMedian = SdFunc(func=median, type='sample')


set.seed(1)
x = sample(100, 30)
SdMean(x)
```

```
[1] 28.33707
```

```
SdMedian(x)
```

```
[1] 29.00832
```