



NYC DATA SCIENCE
ACADEMY

Introduction to R

Part I

Outline of today's class

- ❖ Introduction to R
- ❖ Introduction to RStudio
- ❖ R objects
 - Function calls and variables
 - Primitive data types
 - Vectors
 - Matrices and arrays
 - Data Frames
 - Lists
- ❖ Functional programming: apply

What is R?

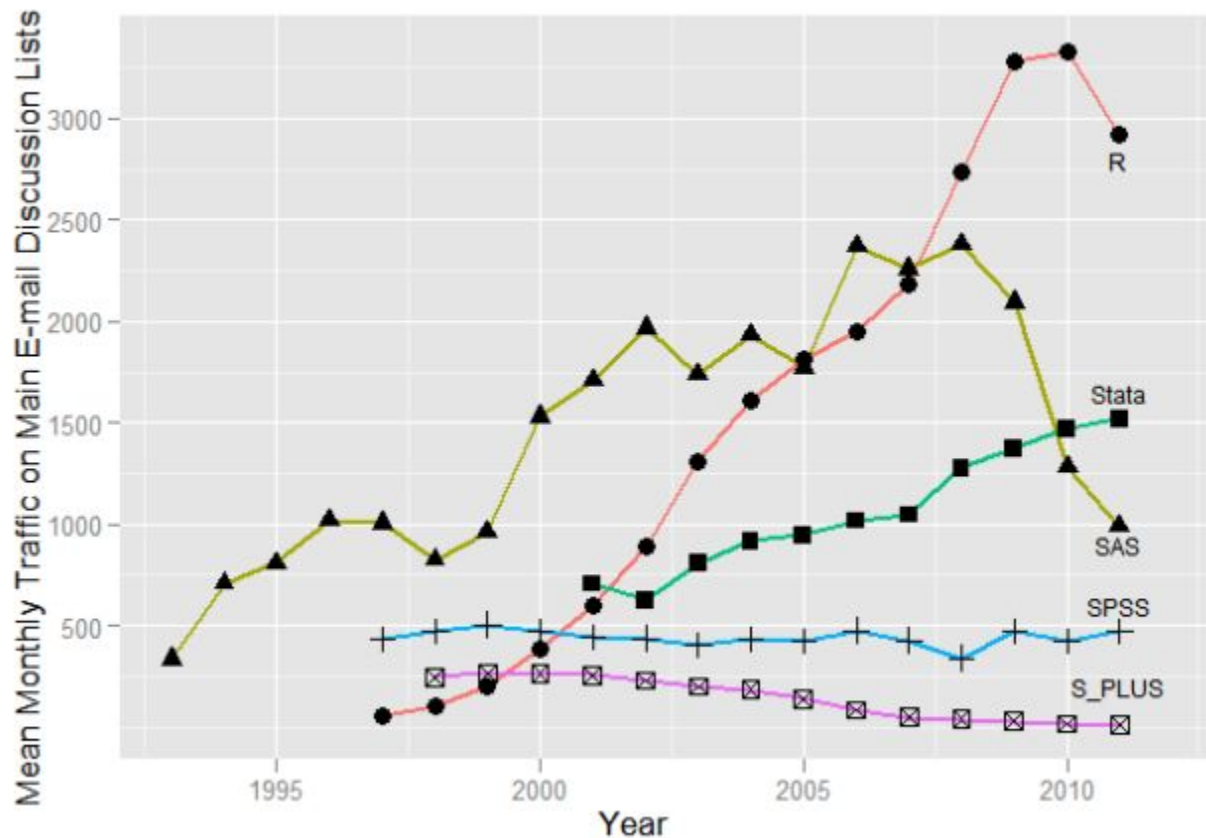
- ❖ R is a programming language for statistics and data analysis that is popular in both industry and academia.
 - The goal in its development was to aid in quickly and accurately turning analytical concepts into usable tools.
- ❖ R was inspired by the programming language S.
 - S was developed at Bell Labs in 1976.
 - R can be viewed as an open-source implementation of S, though the two are ultimately distinct.

What is R?

- ❖ 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?

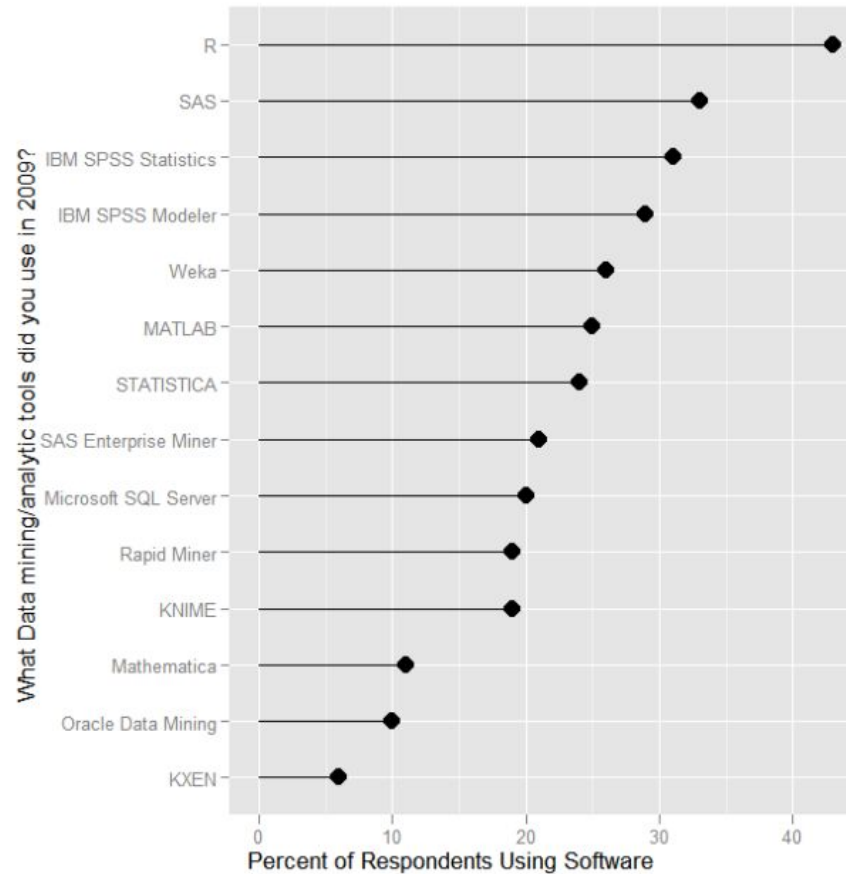
- ❖ R has been gaining importance and popularity:



Source: Arthur Charpentier, *Freakonometrics*

Why R?

- ❖ R has been gaining importance and popularity:



Source: Arthur Charpentier, *Freakonometrics*

Key Features of R

- ❖ Open-source and free.
- ❖ Powerful and increasingly scalable.
- ❖ Has the ability to interact with other software.
- ❖ On the cutting-edge for data management, modeling, and graphics.
- ❖ Provides reproducible analyses.
- ❖ 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:

- R is an interpreted language.
- All data is read into memory.
- R is single threaded, limiting speed and efficiency.

❖ Performance Solutions:

- Convert to bytecode; Integrate C/C++/Fortran with R.
- Utilize cloud computing (Amazon Elastic Computing Cloud).
- Implement parallel computing.

Outline of today's class

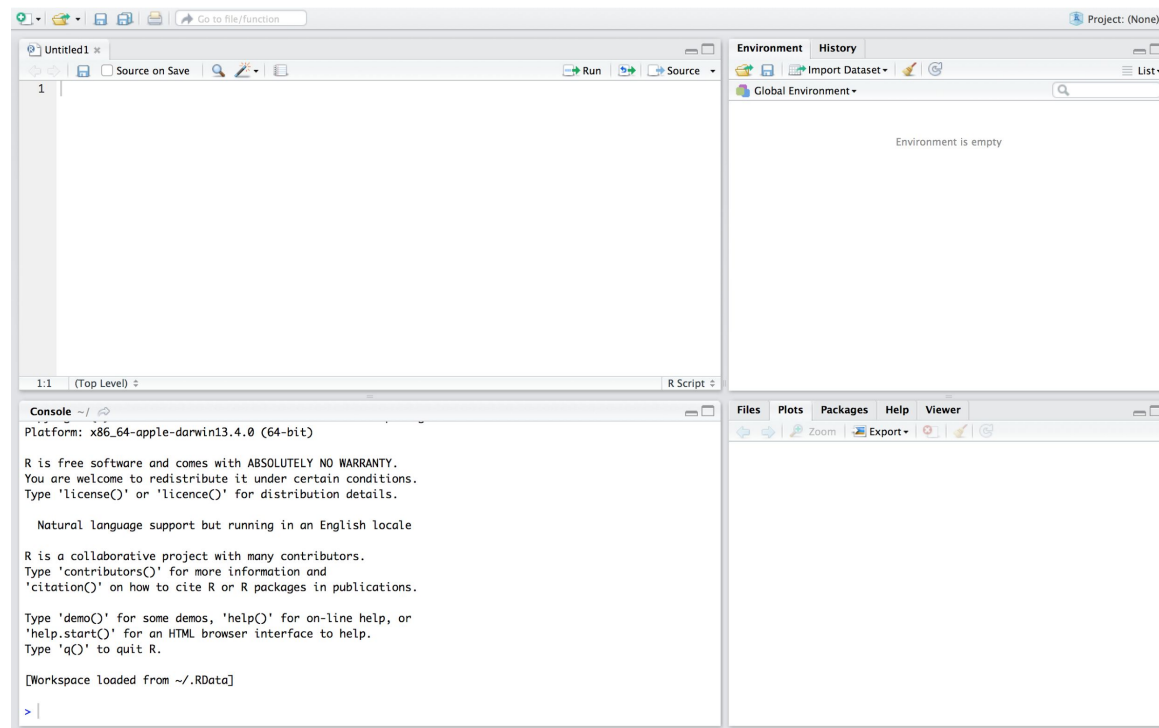
- ◆ Introduction to R
- ◆ Introduction to RStudio
- ◆ R objects
 - Function calls and variables
 - Primitive data types
 - Vectors
 - Matrices and arrays
 - Data Frames
 - Lists
- ◆ Functional programming: apply

Downloading and Installing R

- ❖ Official website of [The R Project](#)
- ❖ [RStudio](#) IDE (Integrated Development Environment)
 - 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

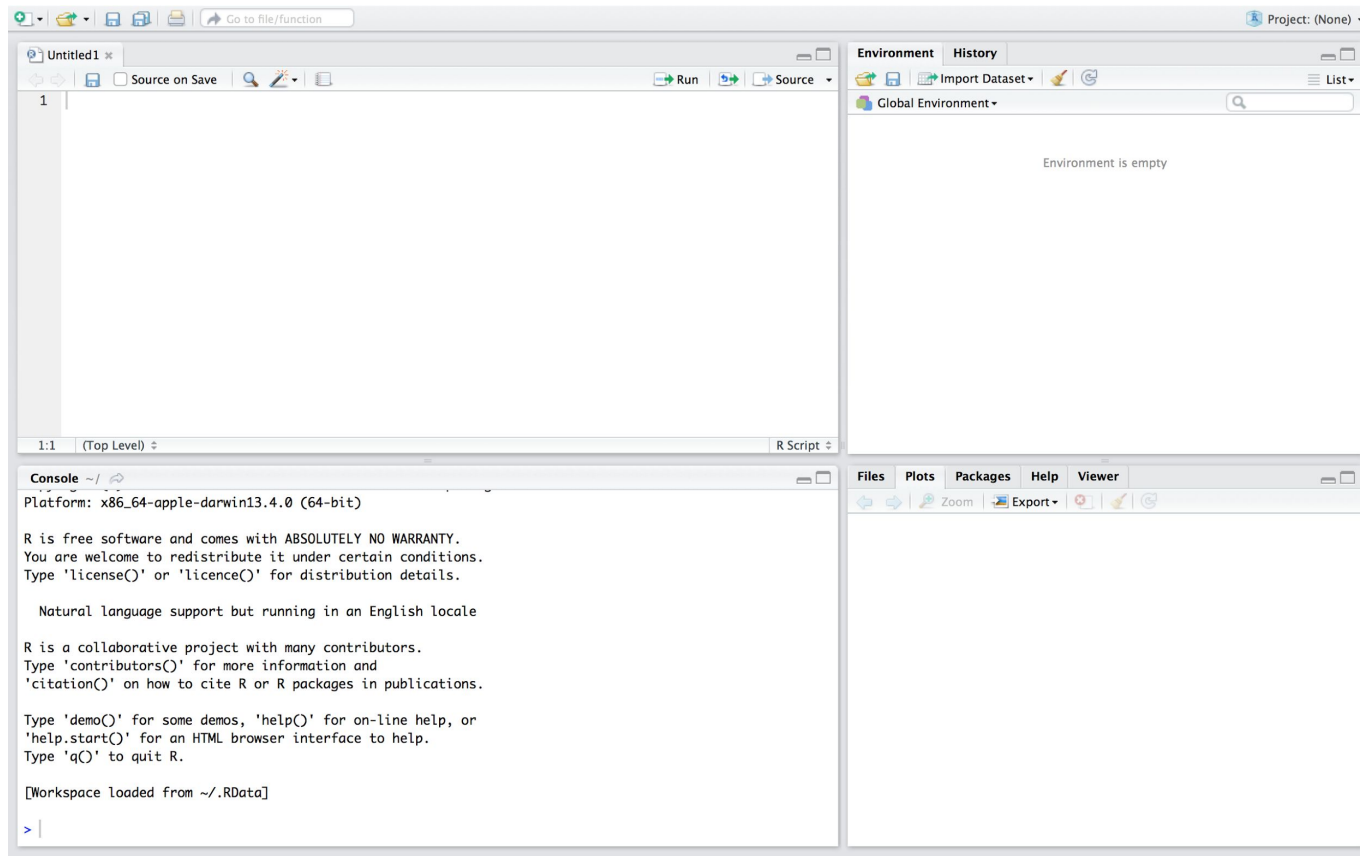
RStudio

- ❖ RStudio is an Integrated Development Environment (IDE) for R.
- ❖ It can be downloaded for free from the official [RStudio](https://www.rstudio.com/) website.
- ❖ The RStudio interface:

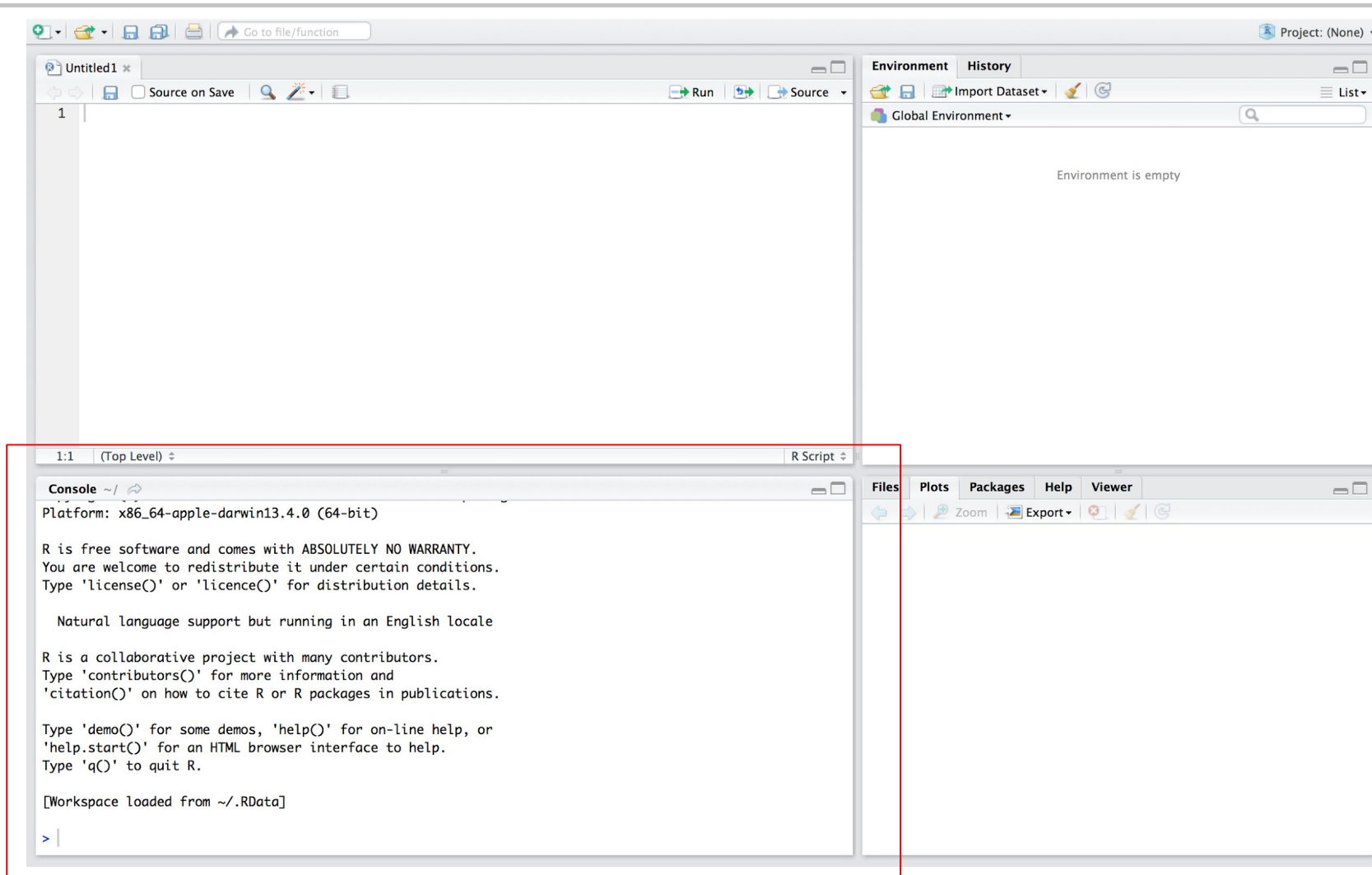


RStudio

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

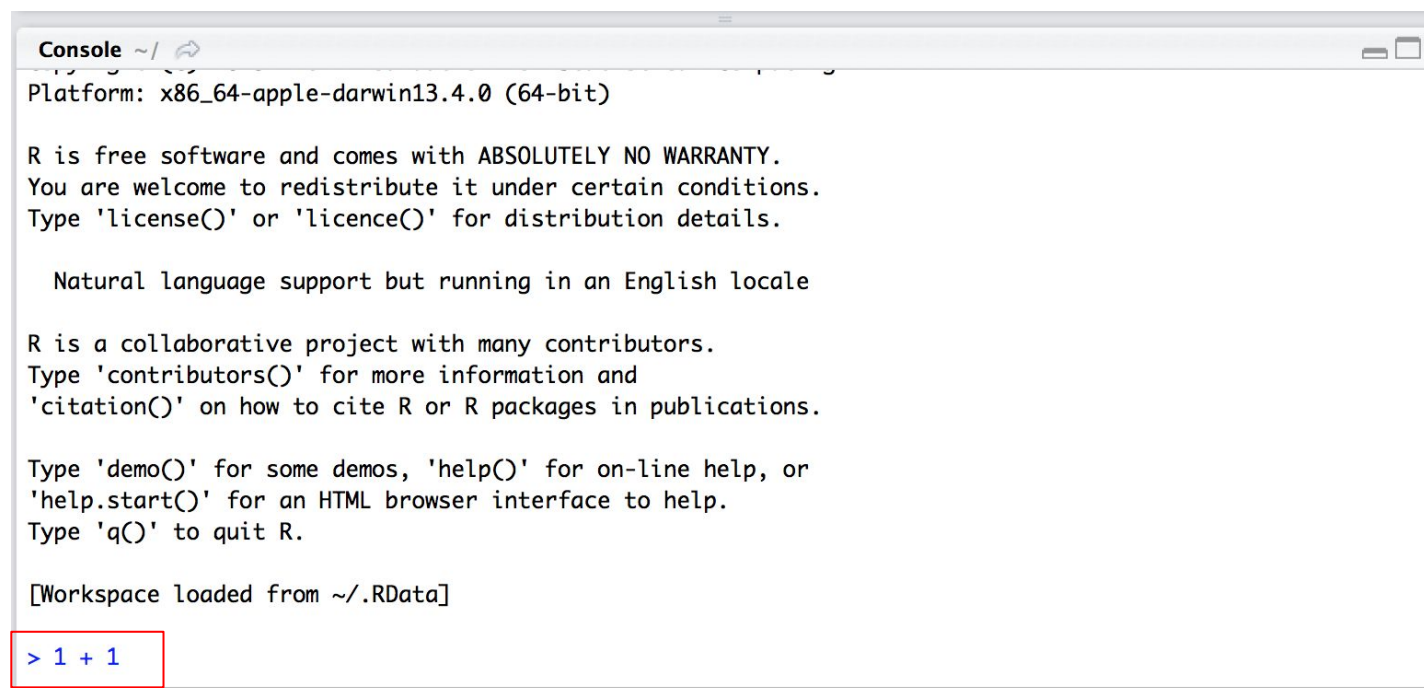


RStudio Console



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.

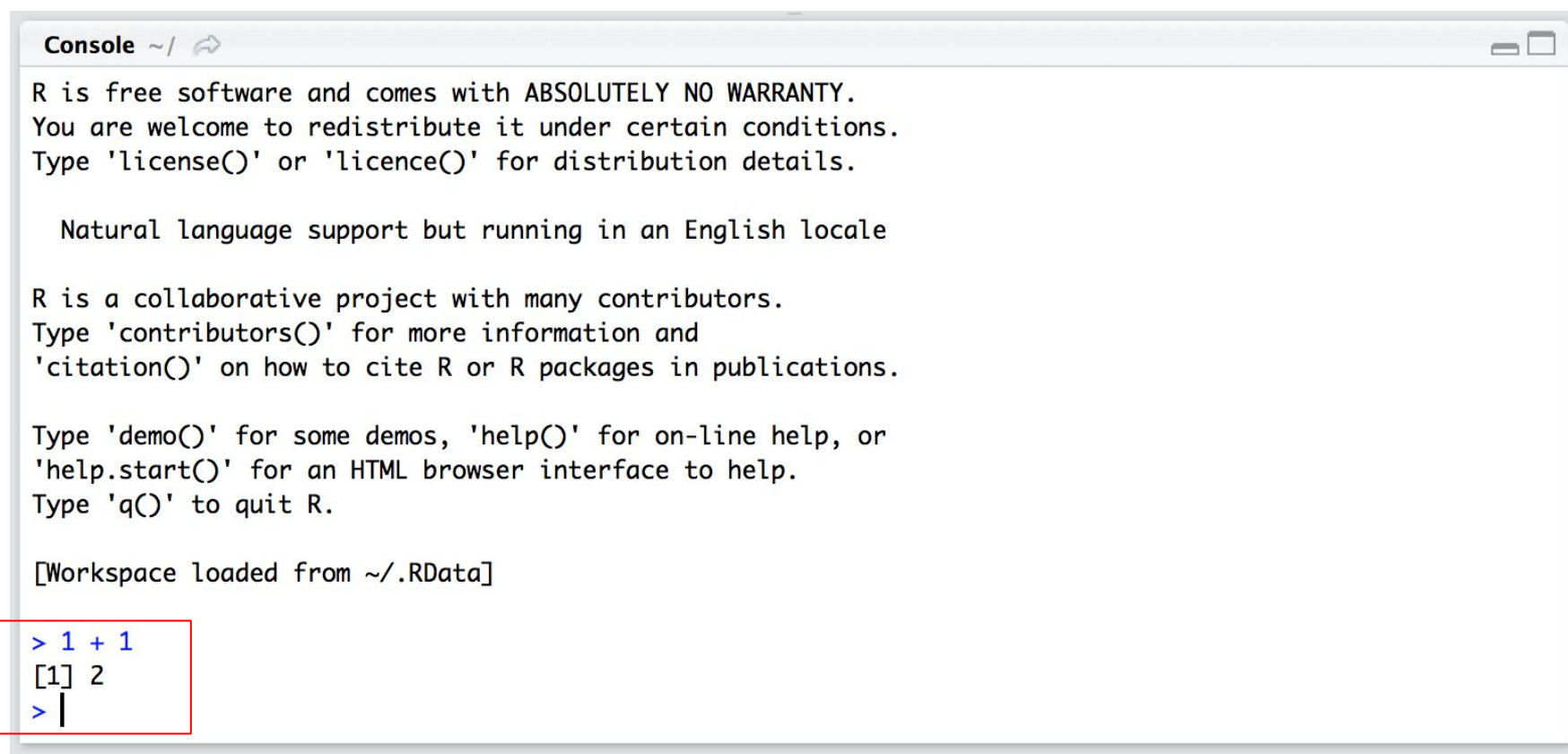
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.


[Workspace loaded from ~/.RData]

> 1 + 1
```

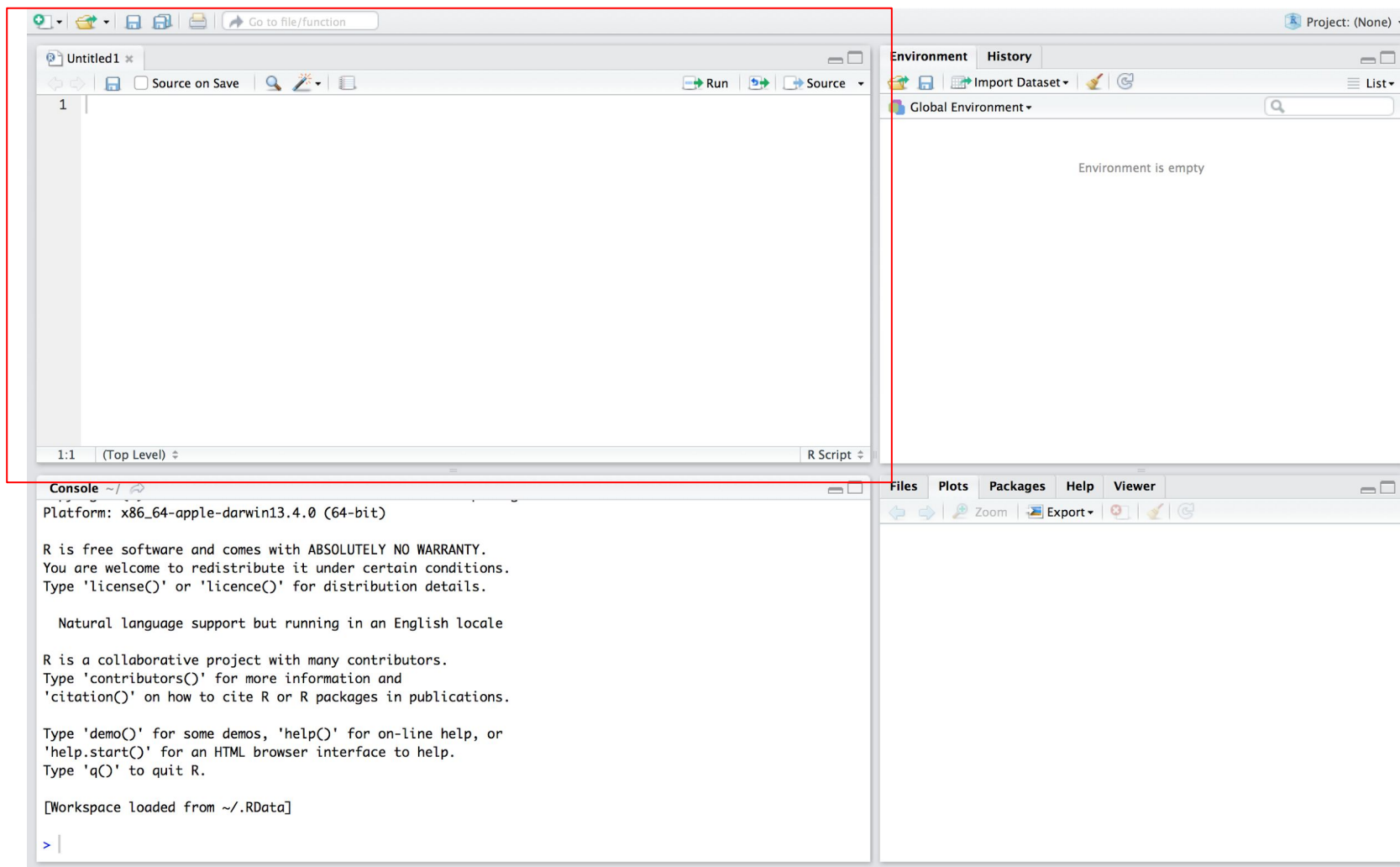
RStudio Console

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



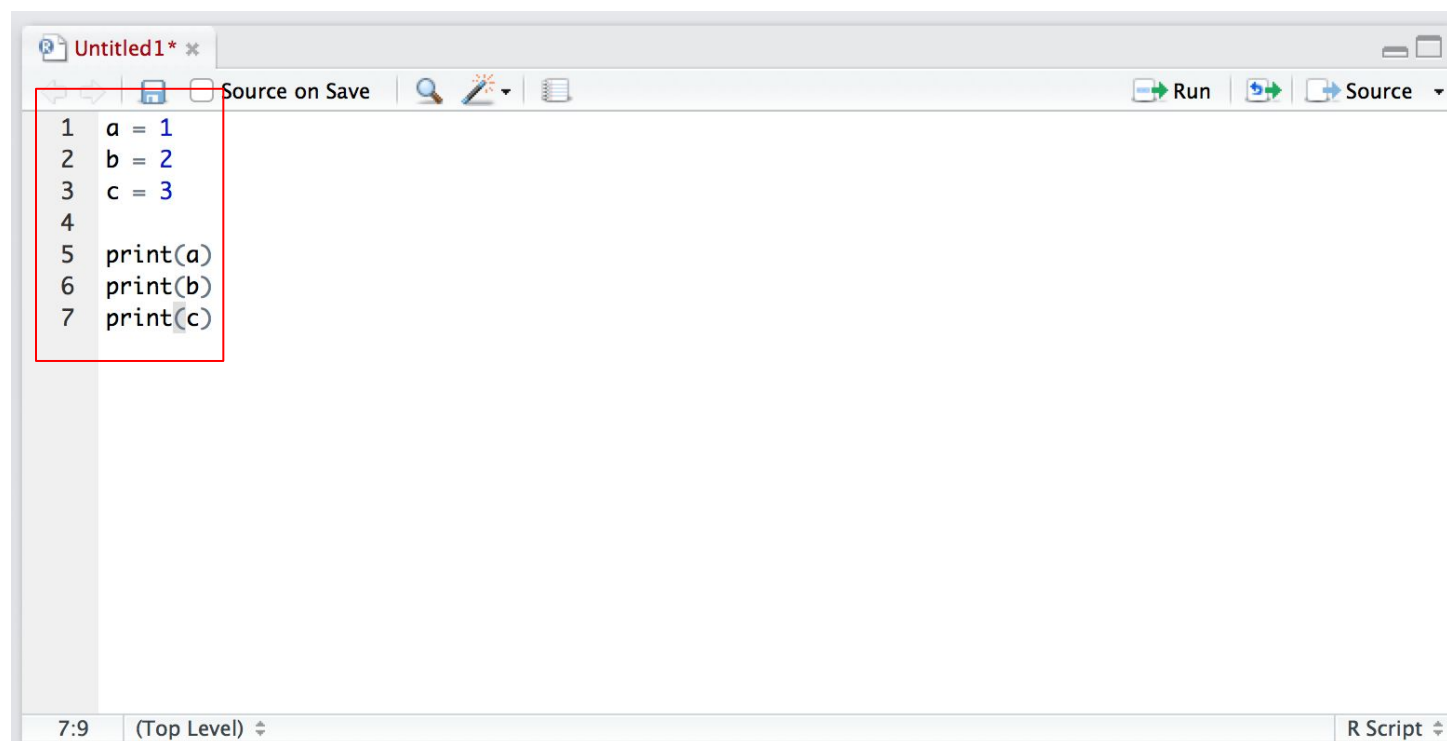
```
Console ~/   
R is free software and comes with ABSOLUTELY NO WARRANTY.  
You are welcome to redistribute it under certain conditions.  
Type 'license()' or 'licence()' for distribution details.  
  
Natural language support but running in an English locale  
  
R is a collaborative project with 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



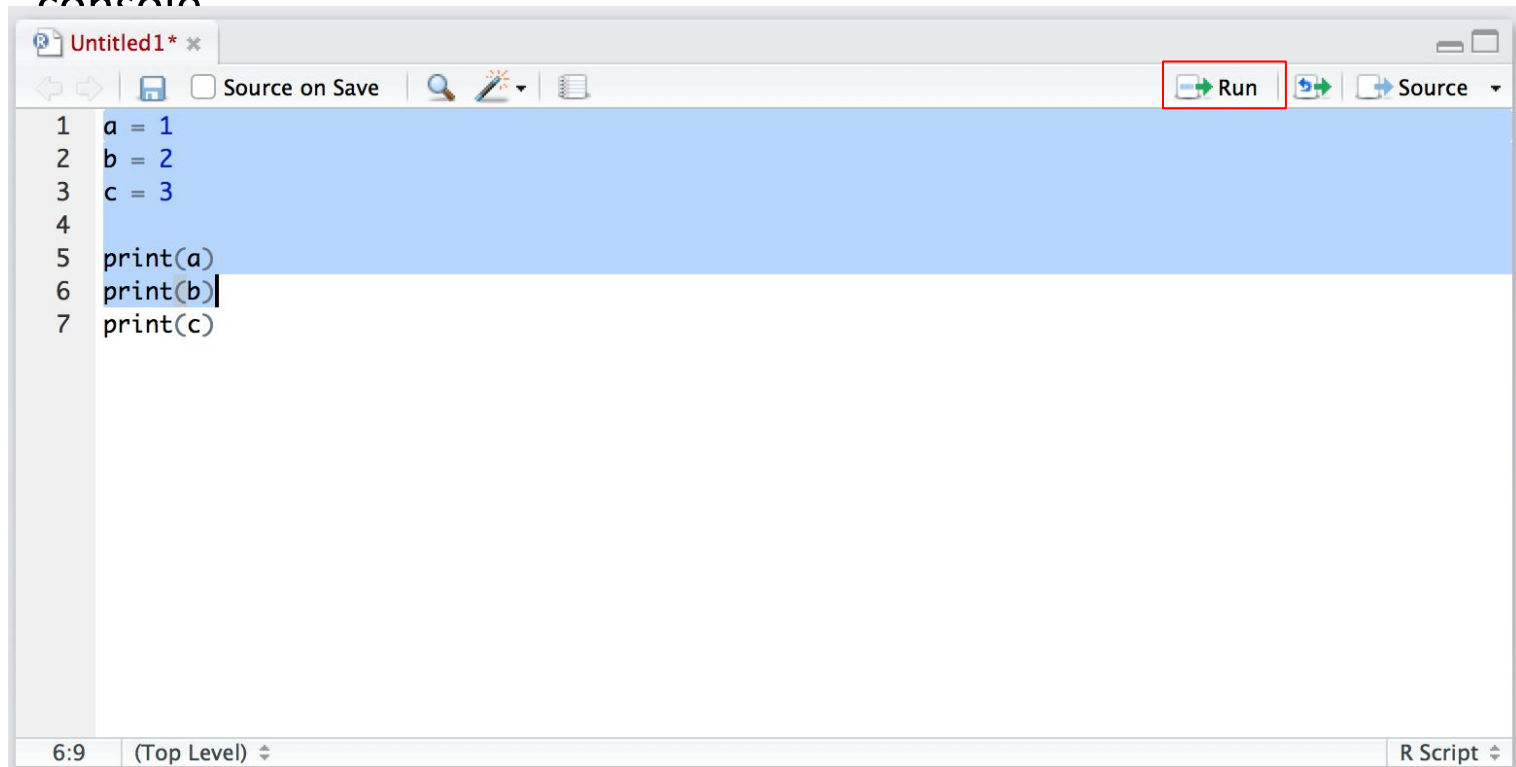
RStudio Source Editor

- ❖ 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).



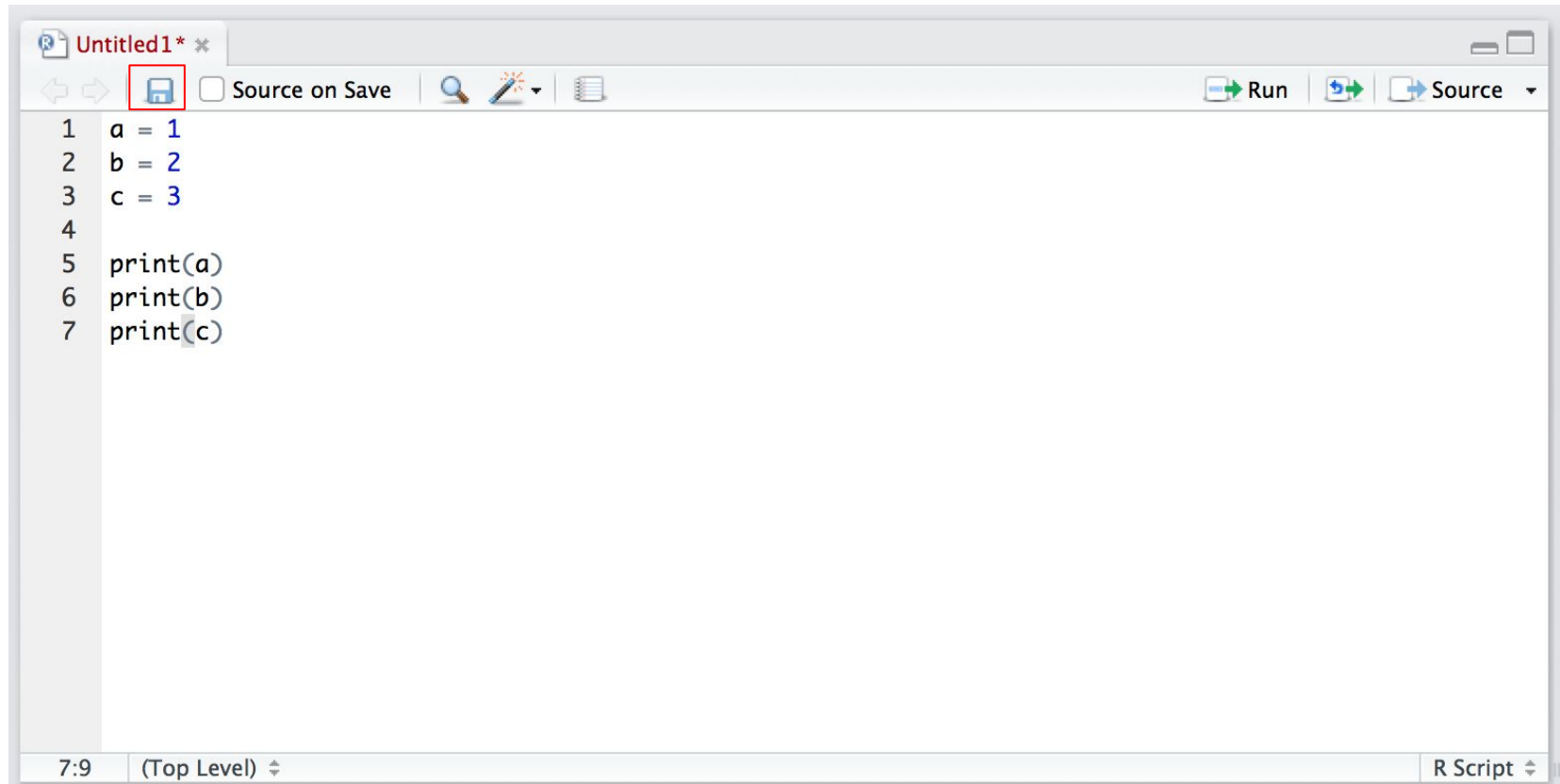
RStudio Source Editor

- ❖ 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.



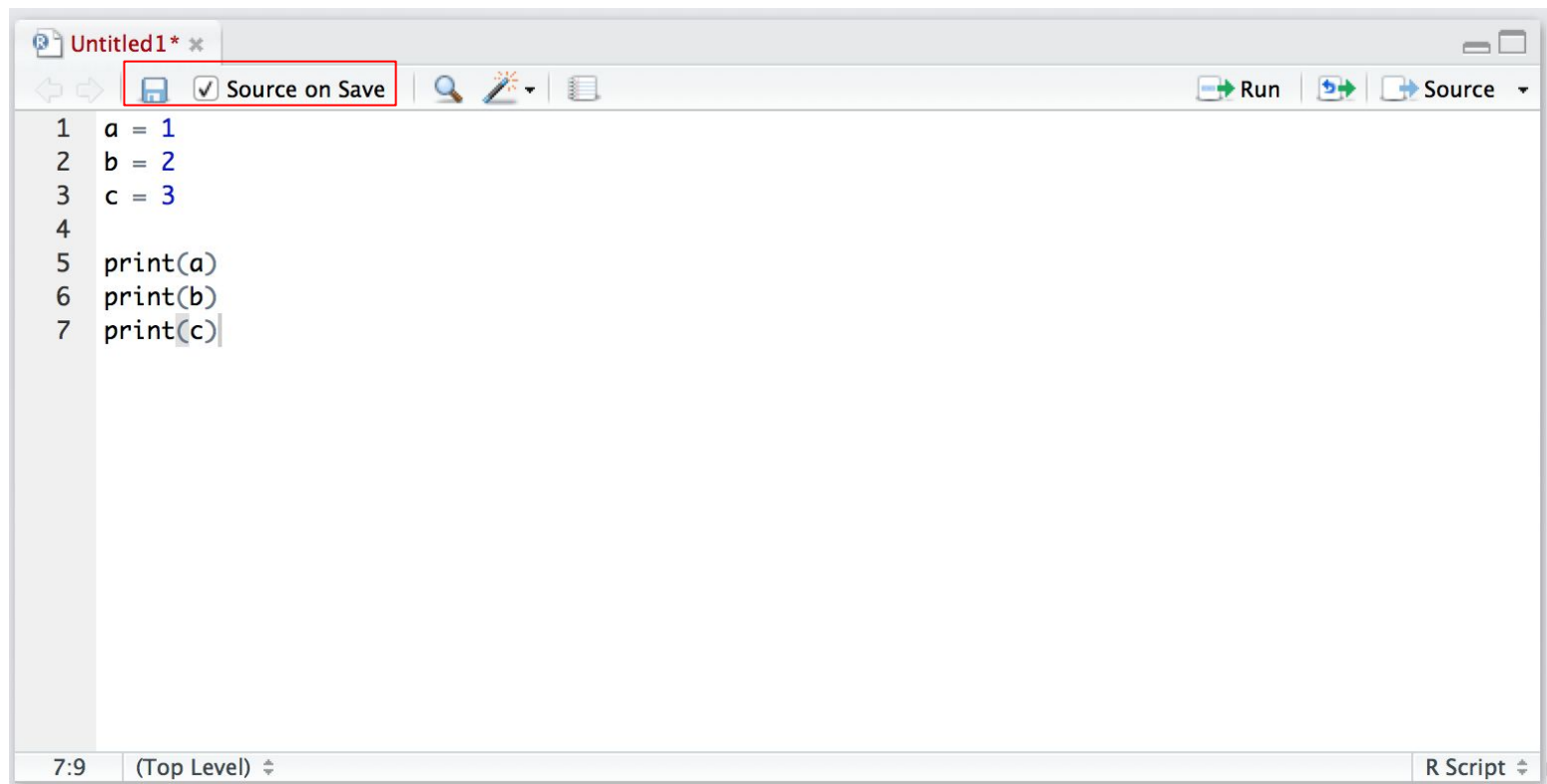
RStudio Source Editor

- ❖ Hitting the “save” icon will allow you to save the script.

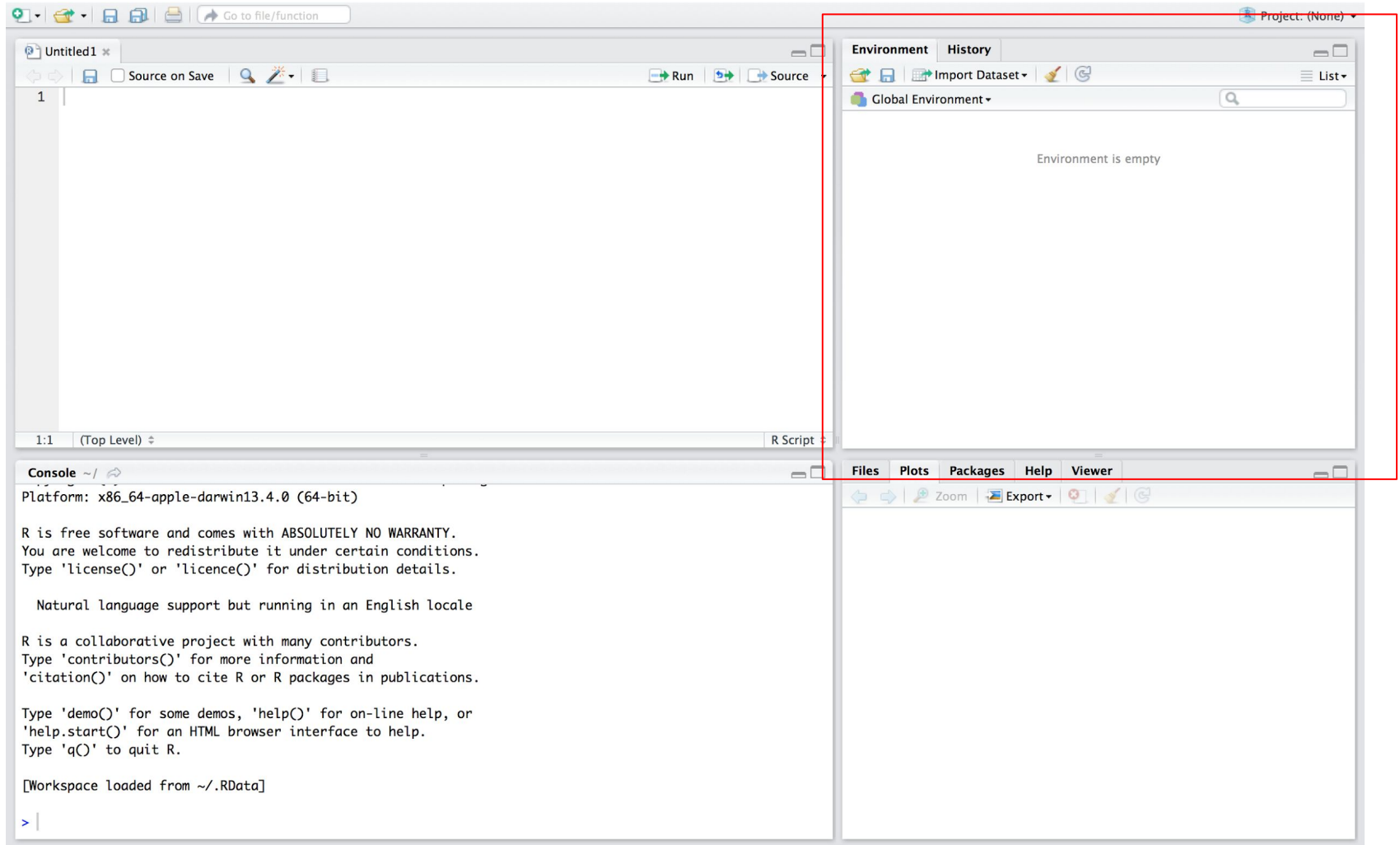


RStudio Source Editor

- ❖ 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.

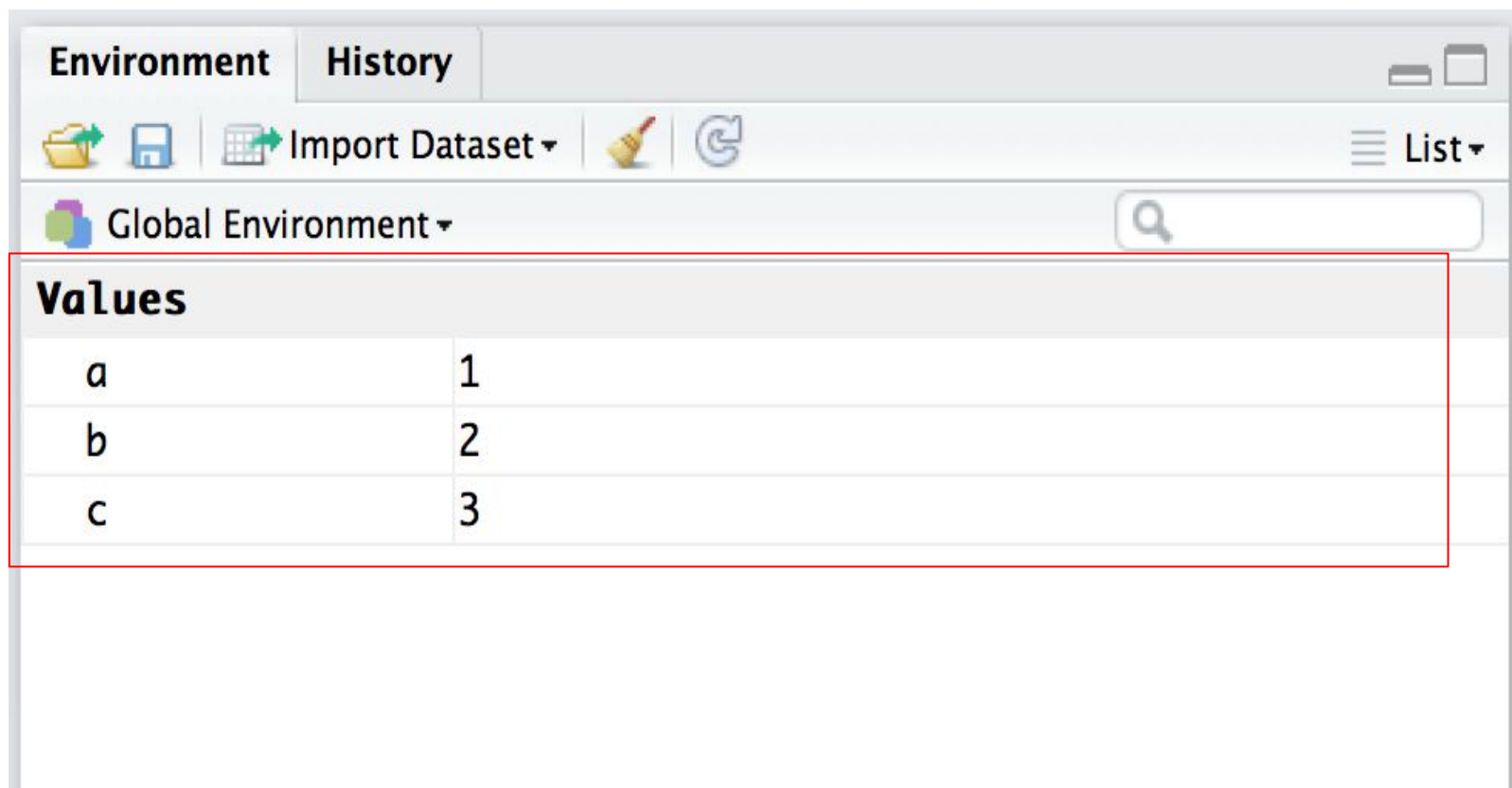


RStudio Environment



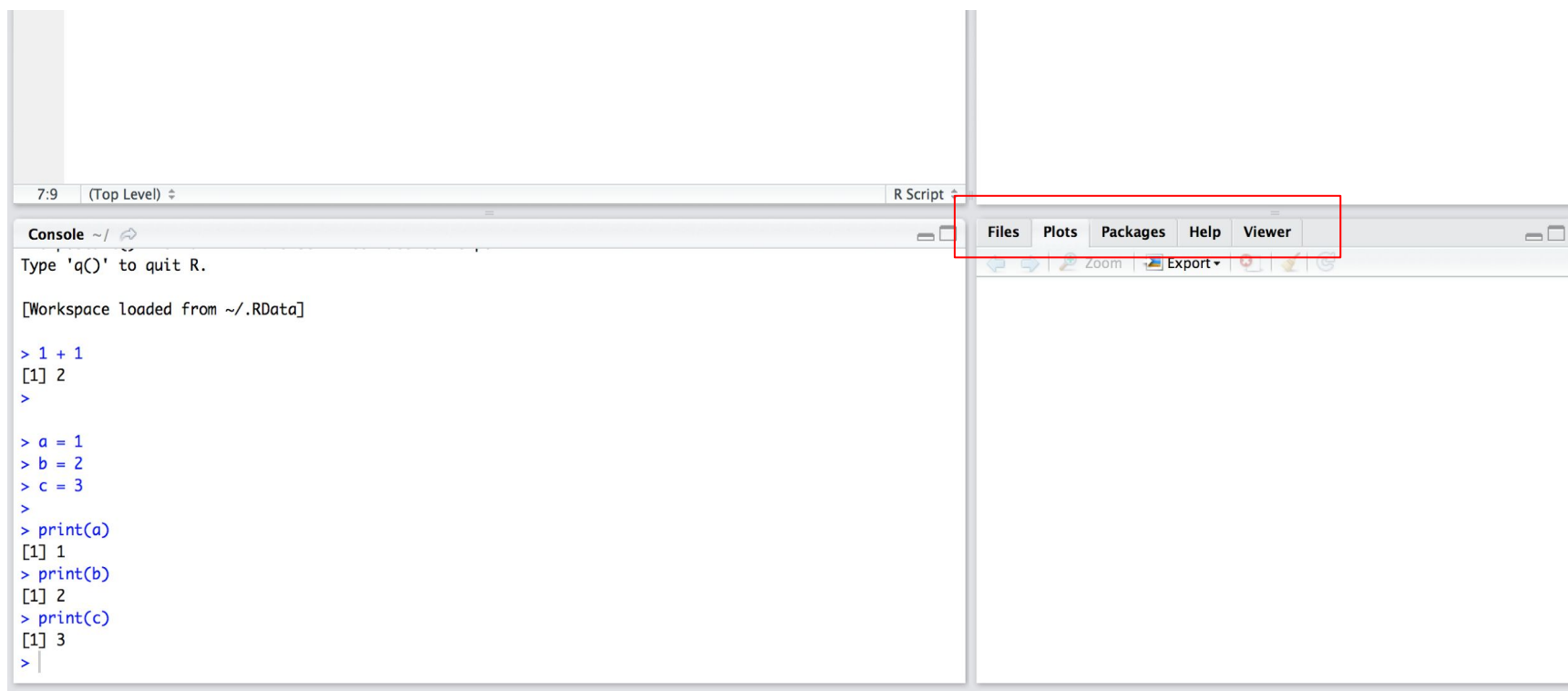
RStudio Environment

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



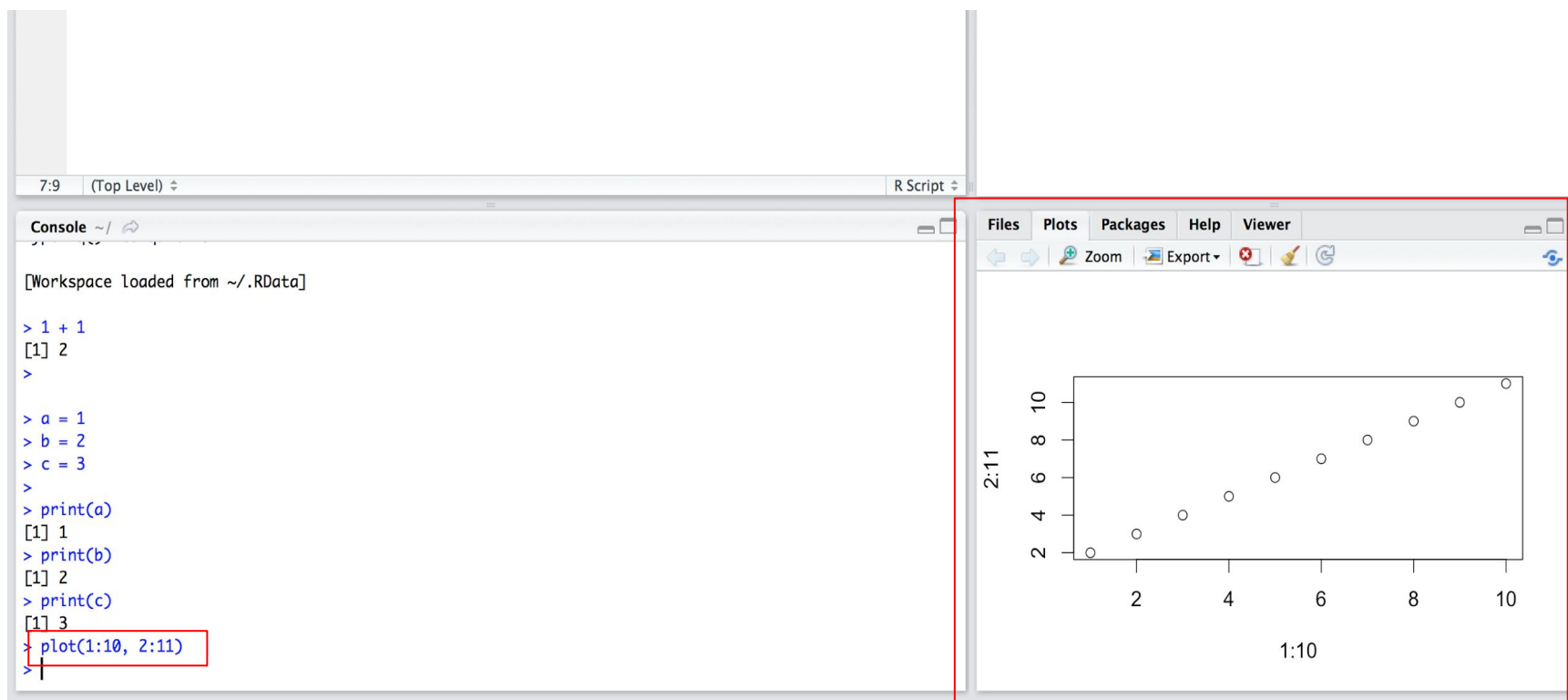
RStudio Plots and Help

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



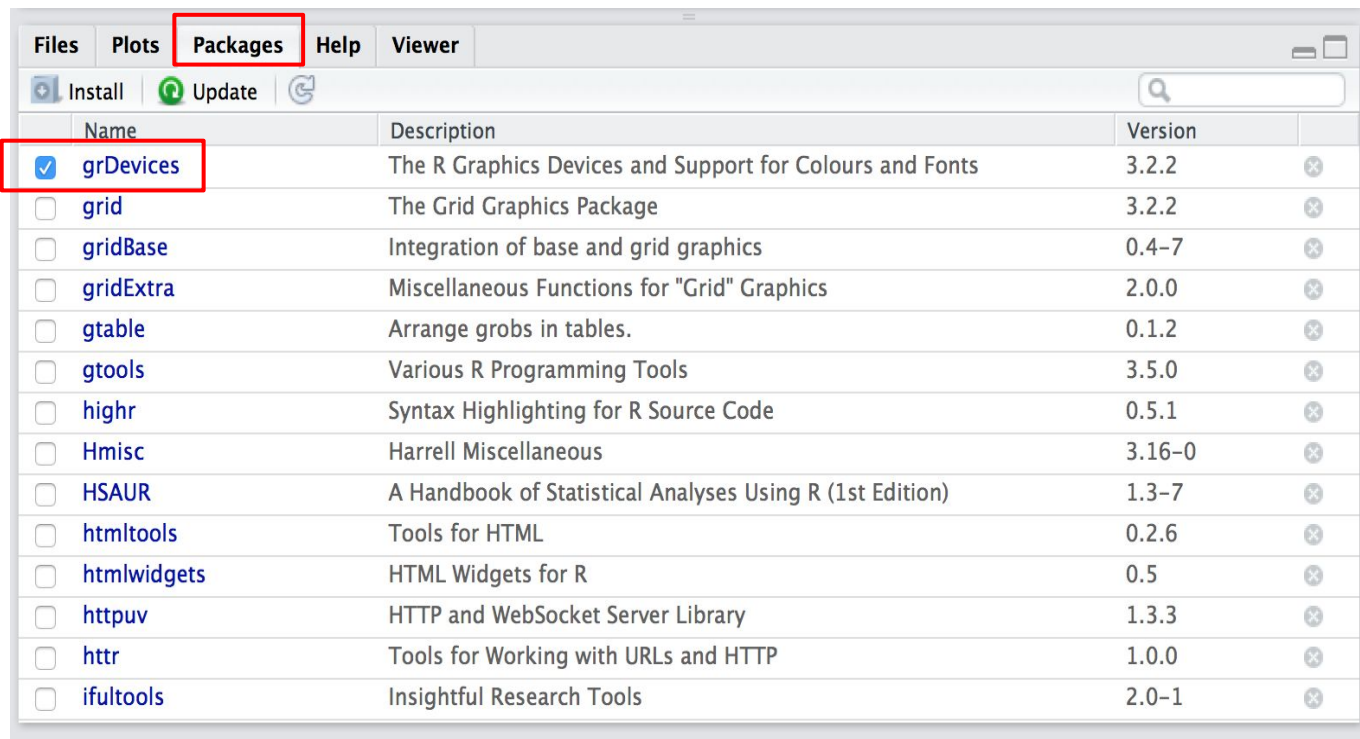
RStudio Plots

- ❖ 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 are available for use.

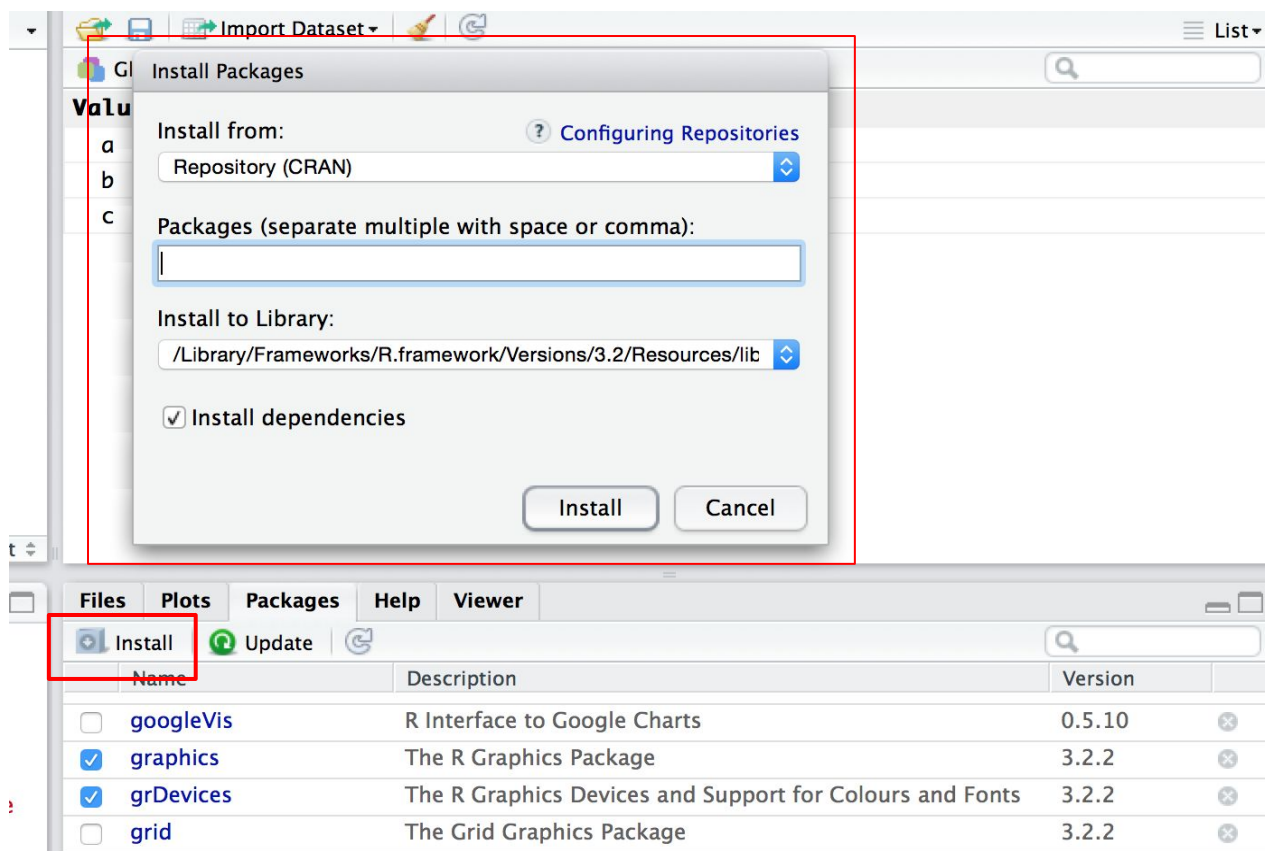


The screenshot shows the RStudio interface with the 'Packages' tab selected. The 'grDevices' package is highlighted with a red box, and its checkbox is checked. The table below lists the packages shown in the interface.

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

Outline of today's class

- ❖ Introduction to R
- ❖ Introduction to RStudio
- ❖ R objects
 - **Function calls and variables**
 - Primitive data types
 - Vectors
 - Matrices and arrays
 - Data Frames
 - Lists
- ❖ Functional programming: apply

Basic R Language Elements

- ❖ In today's class, we will introduce just two very basic language elements - variables and functions - and the major R data types.
- ❖ R has the same features as most programming languages - e.g. loops - but unlike most languages, the operations on data types are so powerful that you can do a lot without knowing a lot of programming.
- ❖ R also encourages a style of programming called functional programming. We will give a taste of functional programming in the last part of today's lecture.

Variables

- ❖ A variable is a symbolic name of a storage location which saves a value. Generally speaking, you can use a variable anywhere you can use a value, and it will be as if you used the value itself.
- ❖ Either an “=” sign or an arrow (“<-”) can be used to assign a value to a variable. For example:

```
x = 2
x
[1] 2
x+1
[1] 3
y <- 4
x+y+7
[1] 13
```

Variable Names

- ❖ R has rules about what can be used as a variable name.
 - A syntactically valid name consists of letters, numbers, periods, and the underscore character (`_`), and starts with a letter or period; if it starts with a period, the second character cannot be a number.
- ❖ Warning: Note that a period is a legal character in variables, and it is often used to help organize variable names (e.g. `person.age`, `person.name`). This can be confusing because in most languages (including Python), the period is used in a way that *looks similar*, but has a special meaning. In R variable names, the period is treated as a character.

Function Calls

- ❖ As in other languages - Python, Excel, SQL, etc. - functions take arguments, given in parentheses, and return values. If there are multiple arguments, they are separated by commas.
- ❖ Some functions are built into R. When you call a function, just as when you write any other expression, RStudio prints the result.

```
sin(0)
[1] 0
log(100, 10) # What is going on here?
[1] 2
```

- ❖ *Syntactic note:* # starts a comment in R. The comment extends to the end of the line. We often show comments in italics to make it easier to distinguish them from actual code.

Function Calls

- ❖ We can use any expression as the argument to a function:

```
x <- pi/2    # pi is a built-in constant
sin(x)
[1] 1
cos(x)
[1] 6.123234e-17
sin(sin(x) + x)
[1] 0.5403023
```

Function Calls

- ❖ Functions can also be called using the *names* of their arguments. This is handy when a function has a lot of arguments. You can give a subset of them - in any order; the ones you don't give may take on default values.
- ❖ E.g. `log()` has a first argument, `x`, with no default value, and a second argument, `base`, that defaults to the natural logarithm (2.718282):

```
log(8)           # use default value of base
[1] 2.079442
log(8, 2)        # give value of base positionally
[1] 3
log(8, base=2)   # give value of base by name
[1] 3
log(base=2)
Error: argument "x" is missing, with no default
```

Outline of today's class

- ❖ Introduction to R
- ❖ Introduction to RStudio
- ❖ R objects
 - Function calls and variables
 - **Primitive data types**
 - Vectors
 - Matrices and arrays
 - Data Frames
 - Lists
- ❖ Functional programming: apply

Primitive Data Types

- ❖ The following are the primitive data types in R. Today we will go over the first three in detail, showing how to create them and some of the operations applicable to them:
 - numeric: The data type for real numbers. A number is assumed by R to be of type “numeric” even if it is an integer.
 - character: The data type for strings.
 - logical: Values TRUE and FALSE.
 - factor: Categorical variables that can be either numeric or string variables. These represent groups.
 - integer: There is an integer type - e.g. 7L is the integer 7, while 7 is the real number 7.0. However, its usage is rare.

Primitive Data Types: Numeric

- ❖ All numbers are given the numeric data type by default. The data type of an object can be seen by applying the “class()” function:

```
class(2)
[1] "numeric"

class(2.0)
[1] "numeric"
```

Primitive Data Types: Numeric

- ❖ R has the usual arithmetic operators:

```
1+1
[1] 2
2-1
[1] 1
2*3
[1] 6
3/2
[1] 1.5
3^2
[1] 9
5 %% 2      # remainder (aka modulus)
[1] 1
5 %/% 2     # integer division
[1] 2
```

Primitive Data Types: Character

- ❖ A character object is a string:

```
class("hello, world!")  
[1] "character"
```

- ❖ Strings are enclosed in quotation marks (either single or double). Notice that if a number is contained in quotation marks, it is considered a character:

```
class('1')  
[1] "character"  
  
class("2")  
[1] "character"
```

- ❖ We will cover more on character manipulation later.

Primitive Data Type: Logical

- ❖ Logical values are TRUE and FALSE (which can also be written as T and F).
Note that case matters here. Comparison operators return logical values.

```
1==1          # Use == for equality testing
[1] TRUE
1<=2
[1] TRUE
1>=2
[1] FALSE
'aardvark' < 'aaron' # alphabetical order
[1] TRUE
'a' != 'b'      # != for not equal
[1] TRUE
```

Logical Operators

- ❖ Expressions can be combined or altered by logical operators as well.

➤ Logical AND operation &:

```
1==1 & 2==3  
[1] FALSE
```

➤ Logical OR operation |:

```
1==1 | 2==3  
[1] TRUE
```

➤ Logical NOT !:

```
!(1==1)  
[1] FALSE
```

Type Conversions

- ❖ Values can sometimes be converted to different types:

```
as.integer(5.5)
[1] 5
as.character(5.5)
[1] "5.5"
```

- ❖ There are a number of conversion operators - all with names of the form `as.type()`. We will use them occasionally.
- ❖ Type conversions do not change the type of a value, but instead produce a new value with the new type:

```
x = 5.5
y = as.character(x)    # x keeps its original value
class(x)
[1] "numeric"
```

Variables and Types

- ❖ *Variables* don't truly have types; *values* have types. The type of a variable is just the type of whatever value it holds (which can change over time).

```
x = 'abc'
class(x)
[1] "character"
x = 3.3
class(x)
[1] "numeric"
x = as.character(x)
class(x)
[1] "character"
```

Outline of today's class

- ❖ Introduction to R
- ❖ Introduction to RStudio
- ❖ R objects
 - Function calls and variables
 - Primitive data types
 - **Vectors**
 - Matrices and arrays
 - Data Frames
 - Lists
- ❖ Functional programming: apply

Composite Data Objects

- ❖ Data objects can be collected inside larger, composite data objects. There are several composite types, which differ in how they are created and how their elements are accessed. We will discuss the major composite data types of R in detail:
 - *Vector*: Sequence of primitive data objects of the same type.
 - *Matrix*: Two-dimensional collection of primitive data objects, like the mathematical notion of matrix.
 - *Array*: Multi-dimensional collection of primitive data objects.
 - *List*: Collection of named values, possibly of different types.
 - *Dataframe*: Matrix with labelled columns and rows. Like a spreadsheet, or a table in a relational database.

Vectors

- ❖ A vector is a sequence of primitive data values of the same type.
- ❖ In R, a vector is an “atomic” object - the smallest object you can create. In fact, when you enter a primitive value, it is considered a vector of length 1.
- ❖ In this section, we will learn about:
 - How to create vectors.
 - Operations on vectors.
 - Accessing elements of a vector.

Creating Vectors

- ❖ A vector can be created from scalars with the function `c()`:

```
c(1, 2, 3, 4) # combine  
[1] 1 2 3 4
```

- ❖ Vectors can have any primitive type, but must consist of only one type (they are said to be “homogeneous”):

```
c('ab', 'cd', "ef")  
[1] "ab" "cd" "ef"  
c('ab', 3)          # 3 is converted to a character type  
[1] "ab" "3"
```


Creating Vectors

- ❖ `c()` is said to “flatten” its arguments to create one vector. Vectors do not contain sub-vectors. For example:

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

How R Displays Vectors

- ❖ Remember the “[1]”s that seem to appear when R prints a value? To understand these, consider the following:

```
c('aaaaaaaaaaaaa', 'bbbbbbbbbbbbbbb',  
  'ccccccccccccccc', 'ddddddddddd',  
  'eeeeeeeeeeeeee', 'fffffffffffff')  
[1] "aaaaaaaaaaaaa" "bbbbbbbbbbbbbbb" "ccccccccccccccc" "ddddddddddd"  
[5] "eeeeeeeeeeeeee" "fffffffffffff"
```

- ❖ Do you see what happened? Because it was a long vector, R could only print the first four elements on the first line. The “[5]” indicates that the second line starts with the fifth element.
- ❖ Thus, whenever R prints a vector, it starts with “[1]” to indicate that it is printing the first element. Since scalars are really one-element vectors, we also see the [1] when printing scalars.

Creating Vectors

- ❖ Another way to create a numeric vector is with **seq()**. This gives us a list of numbers between some first and last number:

```
seq(1, 10)      # can also be written: 1:10  
[1] 1 2 3 4 5 6 7 8 9 10
```

- ❖ We can also specify an increment:

```
seq(1, 10, by=2)  
[1] 1 3 5 7 9
```

- ❖ Or we can specify a desired length, and the increment will be calculated:

```
seq(1, 10, length=5)  
[1] 1.00 3.25 5.50 7.75 10.00
```

Creating Vectors

- ❖ Many functions in R return vectors. A few common ones are shown below:

- `rnorm` generates a sequence of random numbers drawn from a normal distribution; `runif` uses a uniform distribution:

```
rnorm(10) # default mean 0 and standard deviation 1
[1] 0.56868331 0.75967533 -0.06007403 -1.82629581 0.48237515
[6] -0.86437295 0.79634153 1.07775332 -0.89326448 2.24307115
runif(5) # default min 0 and max 1
[1] 0.7152145 0.9519796 0.7762757 0.5156712 0.4760483
```

- `sample` draws a sample randomly from a vector:

```
sample(c('A', 'B'), size=10, replace=TRUE)
[1] "B" "A" "A" "B" "A" "B" "A" "A" "A" "B"
```

Creating Vectors

- The **rep()** function repeats a value by a number of times given in the second argument:

```
rep(0, 10)  
[1] 0 0 0 0 0 0 0 0 0 0
```

- Its first argument can be a vector:

```
rep(c(1, 2, 3), 5)  
[1] 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
```

- and so can the second (it must have the same size as the first):

```
rep(c(1, 2, 3), c(3, 4, 2))  
[1] 1 1 1 2 2 2 2 3 3
```

Vector Slicing

- ❖ You can obtain a subvector (a “slice”) of a vector in many ways. The simplest slice is a single element. We can get this by subscripting (indexes start at 1):

```
x = c(1, 2, 3, 4)
x[1] # use square brackets
[1] 1
```

- ❖ You can get a slice by subscripting with a vector:

```
x[c(1, 3)]
[1] 1 3
```

- ❖ You can subscript with the output of a function too:

```
x[seq(1, 4, by=2)]
[1] 1 3
```

Vector Slicing

- ❖ You can also get a slice by subscripting with a logical vector:

```
x[c(T,F,T,T)]  
[1] 1 3 4
```

We will see shortly how to generate logical vectors by applying logical operations to an entire vector at once.

- ❖ Finally, you can extract all the elements except those specified by using negative indices:

```
x[c(-2, -4)] # all elements except 2 and 4  
[1] 1 3
```

Modifying a Vector

- ❖ The operations we have seen for slicing vectors always create new vectors, and never change in existing vector:

```
v = 10:15  
v  
[1] 10 11 12 13 14 15  
w = v[c(3, 2, 2)]  
w  
[1] 12 11 11  
v  
[1] 10 11 12 13 14 15
```

- ❖ Slicing is called a *non-mutating* operation. Most operations in R are non-mutating. However...

Modifying a Vector

- ❖ We can modify an element of a vector using direct assignment.

```
v[1] <- 20  
> v  
[1] 20 11 12 13 14 15
```

- ❖ You can also modify a set of elements by using slicing on the left-hand side of the assignment.

```
v[2:3] <- c(21, 22)  
> v  
[1] 20 21 22 13 14 15
```

Named Vector Elements

- ❖ We can add labels to elements of a vector; these will be printed above the values. Here, the right-hand side vector should be of the same length as the left-hand side vector:

```
v = 10:15
names(v) = c('a', 'b', 'c', 'd', 'e', 'f')
v
  a  b  c  d  e  f
10 11 12 13 14 15
v['a']      # v[1] also works
  a
10
```

Vector Computation

- ❖ Arithmetic operations apply to numeric vectors elementwise:

```
c(1,2,3,4) + c(5,6,7,8)
[1]  6  8 10 12
```

- ❖ When the length of the vectors is different, R repeats the shorter vector to match the length of the longer vector:

```
c(1,2,3,4) + 1
[1]  2  3  4  5
c(1,2,3,4) + c(1,2)  # same as + c(1,2,1,2)
[1]  2  4  4  6
```

Vector Computation

- ❖ Logical operations work the same way:

```
c(1,2,3,4) >= c(5,1,7,1)
[1] FALSE TRUE FALSE TRUE
c(1,2,3,4) >= 2
[1] FALSE TRUE TRUE TRUE
```

- ❖ We can also use logical values themselves:

```
c(F, T, F, T) & c(T, T, F, F)
[1] FALSE TRUE FALSE FALSE
```

- ❖ This allows us to select slices using complex conditions:

```
x[ x>=2 & x<4 ] # x = c(1,2,3,4)
[1] 2 3
```

Vector Computation

- ❖ Some functions apply to vectors elementwise:

```
x = c(1,2,3,4)
exp(x)
[1] 2.718282 7.389056 20.085537 54.598150
```

- ❖ Some functions do not:

```
class(x)
[1] "numeric"
```

- ❖ Aggregating functions do not apply pointwise:

```
mean(x)
[1] 2.5
```

Vector Computation

- ❖ More aggregating functions:
 - `mean`: compute the mean of the elements in a vector.
 - `max`: find the maximum of the elements in a vector.
 - `min`: find the minimum of the elements in a vector.
 - `sd`: find the standard deviation of the elements in a vector.
 - `length`: find the length of a vector.
 - `summary`: list the mean and quantiles of the elements in a numeric vector.

Example: Modified Mean

- ❖ *Problem:* compute the mean of all the elements of a vector except the maximum and minimum.

```
vector = c(1, 12, 17, 19, 100)
```

- ❖ Find the vector's maximum and minimum values:

```
vec_max = max(vector)  
vec_min = min(vector)
```

- ❖ Ignore the maximum and minimum, then compute the mean of the rest of the elements.

```
vector_trimmed = vector[vector < vec_max & vector > vec_min]  
mean(vector_trimmed)
```

```
[1] 16
```

Missing and Null Values

- ❖ When doing data analysis, you will often encounter data loss situations. Missing data in R is generally expressed as NA.
- ❖ If you are trying to calculate any statistics of data that contain NAs, you will also get an NA.

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

- ❖ We can set the `na.rm` parameter to `TRUE` to ignore the NAs.

```
mean(temp, na.rm=TRUE)
23.2
```


Summary

- ❖ Vectors are the most fundamental of R's data structures.
- ❖ Numerous operations take vectors as arguments and/or produce vectors as results.
- ❖ We will come across vectors many more times in this class.

Outline of today's class

- ◆ Introduction to R
- ◆ Introduction to RStudio
- ◆ R objects
 - Function calls and variables
 - Primitive data types
 - Vectors
 - **Matrices and arrays**
 - Data Frames
 - Lists
- ◆ Functional programming: apply

Matrices

- ❖ A matrix is a rectangular arrangement of elements, meant to model after mathematical matrices.
- ❖ Arrays are generalizations of matrices to n -dimensions.
- ❖ We will go over the creation, manipulation and slicing of matrices. The process for arrays is a generalization of the same methodology.

Creating Matrices

- ❖ To create a matrix, use the `matrix` function. The first argument is a vector containing the elements of the matrix (in default column-wise order); the second and third arguments are the dimensions (rows and columns):

```
my_mat = matrix(1:12, 4, 3)
```

```
my_mat
```

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

- ❖ If you can't remember whether rows or columns comes first, you can use named parameters instead:

```
my_mat = matrix(1:12, nrow=4, ncol=3)
```

Creating Matrices

- ❖ A matrix is actually just a vector with a “dim” attribute. We can create a matrix by setting this attribute directly:

```
v = 1:12
dim(v) = c(4, 3)
v
```

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

Creating Matrices

- ❖ The function **cbind()** is used to create a matrix by concatenating column vectors.

```
vector1 = 1:4
vector2 = 5:8
vector3 = 9:12
cbind(vector1, vector2, vector3)
```

	vector1	vector2	vector3
[1,]	1	5	9
[2,]	2	6	10
[3,]	3	7	11
[4,]	4	8	12

- ❖ **rbind()** works analogously for concatenating rows.

Matrix Slicing

- ❖ We can index and slice matrices much like we do with vectors. Indices of elements are now just pairs (row, column):

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

Matrix Slicing

- ❖ We can create submatrices by giving vectors for both dimensions:

```
my_mat[1, 1:3]  # first row, all columns
[1] 1 5 9
my_mat[1, ]     # same as above
[1] 1 5 9
my_mat[c(1,3), c(1, 2)]
      [,1] [,2]
[1,]    1    5
[2,]    3    7
my_mat[-1, ]    # omit first row; all columns
      [,1] [,2] [,3]
[1,]    2    6   10
[2,]    3    7   11
[3,]    4    8   12
```


Matrix Slicing

- ❖ Notice that if we get a one-dimensional result (single row or column), it becomes a vector:

```
my_mat[1, ]  
[1] 1 5 9  
my_mat[ , 2]  
[1] 5 6 7 8
```

- ❖ If we want to keep its matrix structure, we can specify **drop=F**:

```
my_mat[ , 2, drop=F]  
      [,1]  
[1,]    5  
[2,]    6  
[3,]    7  
[4,]    8
```

Modifying a Matrix

- ❖ As with vectors, most operations (e.g. `cbind`, `rbind`) are non-mutating; however, we can also modify a matrix by assignment:

```
my_mat = matrix(1:12, 4, 3)
my_mat[3, 2] = 15
my_mat[,3] = c(20,21,22,23)
my_mat
```

	[,1]	[,2]	[,3]
[1,]	1	5	20
[2,]	2	6	21
[3,]	3	15	22
[4,]	4	8	23

- ❖ Note that you cannot use slice assignment to add rows or columns to a matrix - you will get an “index out of bounds” error. To add a row or column, reassign to the entire matrix (`my_mat = cbind(my_mat, ...)`).

Matrix Operations

- ❖ Operations on matrices apply elementwise:

```
my_mat + 1
      [,1] [,2] [,3]
[1,]    2    6   10
[2,]    3    7   11
[3,]    4    8   12
[4,]    5    9   13
(my_mat + 1) - my_mat
      [,1] [,2] [,3]
[1,]    1    1    1
[2,]    1    1    1
[3,]    1    1    1
[4,]    1    1    1
```

Matrix Operations

- ❖ Aggregating operations apply across the entire matrix:

```
max(my_mat)
[1] 12
mean(my_mat)
[1] 6.5
```

- ❖ We will see later on how to apply aggregating operations in other ways as well.

Arrays

- ❖ An array is a multidimensional vector. It is just like a matrix, but with any number of dimensions (not just two). We create arrays as follows:

```
a = array(1:24, dim = c(3, 4, 2))
```

```
a
```

```
, , 1
```

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

```
, , 2
```

	[,1]	[,2]	[,3]	[,4]
[1,]	13	16	19	22
[2,]	14	17	20	23
[3,]	15	18	21	24

Outline of today's class

- ❖ Introduction to R
- ❖ Introduction to RStudio
- ❖ R objects
 - Function calls and variables
 - Primitive data types
 - Vectors
 - Matrices and arrays
- **Data Frames**
 - Lists
- ❖ Functional programming: apply

Data Frames

- ❖ Data frames are like matrices, but they have labels on both columns and rows and can also have multiple variable types. Thus, they are similar to database tables or spreadsheets.
- ❖ Data frames are heavily used in R. Because they match up so well with standard data layouts, many functions have been defined to do data-processing and statistical computations on data frames.
- ❖ In today's class, we will concentrate on basics: creating and accessing data frames. We will also go over some common functions to extract information from data frames. We will be working with data frames heavily in the following classes.

Data Frames

- ❖ A data frame is a spreadsheet-like data structure in which the data type of different columns can be different. For example:

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

- ❖ Both columns and rows are labeled. Columns nearly always have descriptive names; rows often use the “default” labels, which are just sequential numbers or indices (as in this example).

Creating Data Frames

- ❖ To create a data frame as in the previous slide, note that it basically consists of two vectors placed side by side. Let's define those vectors:

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

- ❖ The function **data.frame()** combines vectors into data frames as separate columns. They must be of the same length, but can be of different types.

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

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

Creating Data Frames

- ❖ As a simplification, we can omit the labels; the variable names of the vectors will be used as the column labels.

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

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

Creating Data Frames

- ❖ You can obtain a vector of the column or row names of a data frame using the functions `colnames()` and `rownames()`.
- ❖ If you want different column names, you can change them by assignment:

```
colnames(data) = c('CITY', 'TEMP')  
data
```

```
      CITY TEMP  
1 Seattle   78  
2 Chicago   74  
3  Boston   50  
4 Houston  104
```

Creating Data Frames

- ❖ You can also assign names to rows in one of two ways:
 - In the call to `data.frame`, use the parameter `row.names`, or...
 - Use assignment with `rownames(df)`, as we just did with column names on the previous slide.

```
data = data.frame(city, temp,  
                  row.names=c('a', 'b', 'c', 'd'))
```

data

	city	temp
a	Seattle	78
b	Chicago	74
c	Boston	50
d	Houston	104

Data Frames: Subscripting and Slicing

- ❖ Data frames are subscripted and sliced much like matrices:

```
data = data.frame(city, temp)
data[1,1]
[1] Seattle
data[1,2]
[1] 78
data[3,]      # slice containing third row
      city temp
3 Boston   50
```

- ❖ Note that the *row* slice retains the data frame structure.

Data Frames: Selecting and Slicing

- ❖ A column can be selected similarly, but it loses the data frame structure and becomes a vector: (We will explain what *Levels* means shortly.)

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

- ❖ To retain the data frame structure, use `drop=F` (as for matrices):

```
data[, 1, drop=F]
      city
1 Seattle
2 Chicago
3  Boston
4  Houston
```

Data Frames: Selecting and Slicing

- ❖ Columns can be selected by their names with two different syntaxes:

```
data[ , 'city']  
data$city  
[1] Seattle Chicago Boston  Houston  
Levels: Boston Chicago Houston Seattle
```

- ❖ The column selected is then considered as a vector. More precisely, in this scenario it is a vector of *factors*, which means the feature represented by the column can take only finitely many values. In this particular case, the feature represented by the *city* column is restricted to be among *Boston*, *Chicago*, *Houston* and *Seattle*. From a statistical perspective, they are indeed different factors that might result in different distributions of a quantity of interest (income, age, etc.).

Aside: Factors

- ❖ A vector of factors facilitates analysis in R. For example, categorical variables enter into many statistical models in R differently than continuous variables. Thus, storing data as factors ensures that the modeling functions will treat such data correctly.

Data Frames: Selecting and Slicing

- ❖ As with vectors, data frames can be sliced by vectors of logical values. If we find the average temperature among all cities...

```
ave = mean(data$temp)
[1] 76.5
```

...we can select the cities with above average temperatures:

```
data[data$temp > ave, ]
  city    temp
1 Seattle    78
4  Houston  104
```

Data Frames: Selecting and Slicing

- ❖ Let's take a closer look. This code creates a logical vector:

```
data$temp > ave  
[1] TRUE FALSE FALSE TRUE
```

- ❖ Manually plugging the logical vector into the data frame, we obtain the same result.

```
data[c(T,F,F,T), ]  
   city    temp  
1  Seattle    78  
4  Houston   104
```

Functions on Data Frames

- ❖ The function **dim()** gives the dimensions of the data frame (this works for matrices also):

```
dim(data)
[1] 4 2
```

- ❖ To view the first n rows in a data frame, use the function **head(df, n)**. Without specifying n, it defaults to 6 rows. (Our data frame only has 4 rows, so all 4 are printed).

```
head(data)
  city temp
1 Seattle 78
2 Chicago 74
3 Boston 50
4 Houston 104
```

Functions on Data Frames

- ❖ The function **summary()** gives column-wise summary statistics. We see that the summary function treats a categorical feature and a numerical feature in different ways.

```
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
```

Functions on Data Frames

- ❖ The function **str()**, which stands for “structure,” compactly displays the internal structure of an R object.

```
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
```

Sorting Data Frames

- ❖ The function **order()** gives the order of the elements of a vector:

```
order(data$temp)
[1] 3 2 1 4
```

- ❖ The vector returned by the **order()** function represents the row indexes with their temp value in increasing order.

```
data[order(data$temp), ]
  city temp
3 Boston  50
2 Chicago 74
1 Seattle 78
4 Houston 104
```

Sorting Data Frames

- ❖ To order the data frame with temp value in decreasing order, we can specify the “decreasing” argument to be TRUE:

```
order(data$temp, decreasing=TRUE)
[1] 4 1 2 3
```

- ❖ The temp value is then in a decreasing order.

```
data[order(data$temp, decreasing=TRUE),]
  city temp
4 Houston 104
1 Seattle  78
2 Chicago  74
3  Boston  50
```

Exporting Data Frames to Files

- ❖ A data frame can be exported to a .txt file or a .csv file with **write.table()**. We need to specify the data frame, the file name, and the field separator string. By default, the indexes of the data frame will be the first column of the table written into the file. If that is not desired, set the argument **row.names = FALSE**:

```
write.table(data, file='my_data.csv', sep=',',  
            row.names = F)
```

- ❖ **write.csv()** can be used to export data frames as well. This is just write.table() with the field separator set to comma:

```
write.csv(data, file='my_data.csv', row.names = F)
```


Importing Data Frames from Files

- ❖ Similarly, we can import a .csv or a .txt file to a data frame in our workspace:

```
read.csv('data.csv')
```

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

- ❖ Again, we can use `read.table()`, but we have to specify that the separator is ',' and the .csv file has a header (the column names):

```
read.table('data.csv', header=T, sep=',')
```

Importing Data Frames from Files

- ❖ Packages `openxlsx` and `foreign` contain functions to import data of several formats:

```
# install.packages("openxlsx")
library(openxlsx)
excel_data = read.xlsx("file_name.xlsx", sheet=1)

# install.packages("foreign")
library(foreign)
stata_data = read.dta("file_name.dta")
spss_data = read.spss("file_name.sav")
sas_data = read.xport("file_name.xpt")
```

Outline of today's class

- ❖ Introduction to R
- ❖ Introduction to RStudio
- ❖ R objects
 - Function calls and variables
 - Primitive data types
 - Vectors
 - Matrices and arrays
 - Data Frames
 - **Lists**
- ❖ Functional programming: apply

Lists

- ❖ Lists are *heterogeneous* collections of *named* objects.
- ❖ The **list** is the most flexible data object in R. Elements in a list can be any object, from simple data types such as integers or characters, to very complicated objects like data frames or statistical models.
- ❖ As with other composite data types, we will discuss how to create lists and extract values from them.

Creating Lists

- ❖ Create lists using the `list` function:

```
employee = list(Name='John', Position='cashier',  
                Salary=9.50)
```

```
employee
```

```
$Name
```

```
[1] "John"
```

```
$Position
```

```
[1] "cashier"
```

```
$Salary
```

```
[1] 9.50
```

Creating Lists

- ❖ As noted above, lists can contain any type of value. This list contains a string and a vector:

```
student = list(Name="Peter", Classes=c("Bio101",  
                                         "Psych200", "CS125"))  
  
student  
$Name  
[1] "Peter"  
  
$Classes  
[1] "Bio101" "Psych200" "CS125"
```

Subscripting vs Slicing

- ❖ We can access the elements of a list in two ways: Using the names, or using the positions (given by the order in the original list creation step).
- ❖ There are two notions of “accessing”:
 - We can get a sub-list/slice containing a given name or position (i.e., *slicing* the list).
 - We can get the value associated with a name or position (i.e., *subscripting* the list).

Slicing Lists

- ❖ Slicing means obtaining a sub-list. Obtain the sublist containing just one item by using single square brackets; the subscript can be either a number or a name:

```
employee[1]  
$Name  
[1] "John"  
  
class(employee[1])  
[1] "list"  
  
employee['Name']  
$Name  
[1] "John"
```


Slicing Lists

- ❖ You can get a larger sub-list using previously discussed slicing methods, e.g.:

```
employee[-1]    # all elements except Name
$Position
[1] "cashier"

$Salary
[1] "9.50"
employee[c("Name", "Position")]
$Name
[1] "John"

$Position
[1] "cashier"
```

Subscripting Lists

- ❖ You can access an element of a list using the name:

```
employee$Name  
[1] "John"
```

- ❖ You can also use the position of this item in the list (Name is the first item). However, we need to use *double* square brackets because, as we saw before, single square brackets are used for slicing into sub-lists.

```
employee[[1]]  
[1] "John"
```

Modifying a List

- ❖ As we have seen with vectors and matrices, slicing is a *non-mutating* operation; however, we can modify lists by assignment as before:

```
employee$Position = 'manager'
employee[[3]] = 10.50
employee
$Name
[1] "John"

$Position
[1] "manager"

$Salary
[1] 10.50
```

Removing Elements from a List

- ❖ As another example of modifying a list, we can use NULL to completely remove an element. This differs from slicing with -1, which is a non-mutating operation:

```
employee$Name = NULL  
employee  
$Position  
[1] "cashier"  
  
$Salary  
[1] 10.5
```

List Operations *Do Not* Apply Elementwise

- ❖ Since list elements are heterogeneous, it doesn't make sense to apply operations elementwise. For example, we cannot apply an arithmetic operator to a sub-list:

```
employee = list(Name='John', Position='cashier',  
                Salary=9.50)
```

```
employee[3] + 1
```

```
Error in employee[3] + 1 : non-numeric argument to binary  
operator
```

```
employee[[3]] + 1
```

```
[1] 10.5
```

Outline of today's class

- ❖ Introduction to R
- ❖ Introduction to RStudio
- ❖ R objects
 - Function calls and variables
 - Primitive data types
 - Vectors
 - Matrices and arrays
 - Data Frames
 - Lists
- ❖ Functional programming: apply

Functional Programming: The apply Family

- ❖ *Functional programming* is a style of programming that avoids traditional programming using loops (“imperative programming”). It is often used with data frames in R.
- ❖ Conventionally, a data frame is arranged so that each row represents an observation (e.g., an individual person) and each column represents a variable (e.g., height, weight, gender, etc.). Some questions we may ask:
 - What are the mean and the standard deviation of the height and weight variables?
 - What are the mean and the standard deviation of the height and weight variables by different genders?
- ❖ The “apply” family of functions in R helps us to answer these questions easily without using loops or control statements.

Our Toy Data

- ❖ The Fisher/Anderson iris data is a famous data set that records various measurements on 150 iris flowers. There are 3 species of flower: *setosa*, *versicolor*, and *virginica*.
- ❖ This dataset is built into R as a sample data frame. Run the line below and the iris data set will show up in your workspace and be ready for use.

```
data(iris)
```


sapply

- ❖ If you want to implement a function f on each element of a vector v , write:

```
sapply(v, f)
```

- ❖ For example, to take the square root of each element in a vector:

```
v = 1:4  
sapply(v, sqrt)  
[1] 1.000000 1.414214 1.732051 2.000000
```

- ❖ We can also use `sapply` to implement a function to the columns of a data frame, or the elements of a list.

sapply

- ❖ Since a data frame is really a collection of vectors (its columns), we can use `sapply`. The result is a vector with named elements:

```
sapply(iris[,1:4], mean)
Sepal.Length Sepal.Width Petal.Length Petal.Width
    5.843333    3.057333    3.758000    1.199333
```

(We extracted the first four columns simply because the fifth column - Species - is not numeric, so taking an average would not make sense.)

- ❖ Using the `apply` family here helps us to avoid writing this verbose code:

```
c(mean(iris$Sepal.Length), mean(iris$Sepal.Width),
  mean(iris$Petal.Length), mean(iris$Petal.Width))
```

sapply

- ❖ In addition to vectors and data frames, sapply can operate on lists. The result is a vector with named elements:

```
mylist = as.list(iris[,1:4])
mylist
$Sepal.Length
 [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 ...
$Sepal.Width
 [1] 3.5 3.0 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
$Petal.Length ...
$Petal.Width ...
sapply(mylist, sd)
Sepal.Length Sepal.Width Petal.Length Petal.Width
 0.8280661    0.4358663    1.7652982    0.7622377
```

lapply

- ❖ The **lapply()** function is similar to `sapply` but it returns the values in list form instead.

```
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

- ❖ The main difference between **sapply()** and **lapply()** is that **sapply()** returns a vector while **lapply** returns a list.
- ❖ The reason this kind of syntax is referred to as *functional programming* is that a function (e.g., `mean`, `sd`, `sqrt`, etc.) is taken as an argument in another function (e.g., `sapply` or `lapply`).

Applying Functions to Data

- ❖ The function **apply()** allows for convenient manipulation of matrices. We first initialize a matrix as an example:

```
mat = matrix(1:6, nrow=2, ncol=3)
mat
```

	[,1]	[,2]	[,3]
[1,]	1	3	5
[2,]	2	4	6

- ❖ The function `apply()` implements a function along each column or row, depending on the second argument. The result is a vector:

```
apply(mat, 1, sum)    # across rows
[1]  9 12
apply(mat, 2, sum)    # across columns
[1]  3  7 11
```

Aggregating Data

- ❖ One of the most common operations in computing is aggregating a collection of data on a group value by computing summary statistics within each group. For example, this is the purpose of the “GROUP BY” clause in SQL statements.
- ❖ We present two methods of doing such aggregation in R, using the functions `tapply` and `aggregate`. Both take a function as an argument and implement the function within groups.

tapply

- ❖ The syntax of tapply is:

```
tapply(INDEX=group-by column,  
      X=column to aggregate,  
      FUN=aggregating function)
```

- ❖ The “index” contains a category across which to group, the “x” contains the data to be aggregated, and the “fun” is the function to implement.
- ❖ For the iris dataset, we can use tapply to find the average sepal length for each species:

```
tapply(iris$Species, X = iris$Sepal.Length,  
      FUN=mean)  
setosa  versicolor  virginica  
5.006    5.936      6.588
```


Applying Functions to Data

- ❖ The aggregate function is similar, but has syntactic differences. We must supply the data to be aggregated, the category across which to group (in list format), and the function to implement.

```
aggregate(iris$Sepal.Length, list(iris$Species),  
          mean)
```

	Group.1	x
1	setosa	5.0
2	versicolor	5.9
3	virginica	6.6

Appendix

How to Get Help

- ❖ There are many, many ways to get help in R:
 - **help.start()**: Open the R help documentation.
- ❖ How about if I don't know what a function does, like *lm()*?
 - **help("lm")** or **?lm**: See documentation on the *lm()* function (quotes may be omitted).
 - **help.search("lm")** or **??lm**: Search *lm* as a keyword in the local documentation.
 - **RSiteSearch("lm")**: Search *lm* as a keyword in the online documentation.
 - **example("lm")**: Explore usage examples of the *lm()* function (quotes may be omitted).
 - **...and more!**

Manipulating the Working Directory and Workspace

- ❖ For the working directory, we can:
 - **getwd()**: Display the current working directory.
 - **setwd()**: Change the current working directory.
- ❖ The workspace is the current working environment, including all the user-defined objects. For the workspace, we can:
 - **ls()**: List the objects in the current workspace.
 - **rm(objectlist)**: Remove objects from the workspace.
 - **save.image("myfile")**: Save the workspace to the file 'myfile' in working directory (default suffix = .RData).
 - **load("myfile")**: Read a workspace into the current session.

Learning R: Online 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](#)
- ❖ [Google Developers' Intro to R](#)
- ❖ [Twotorials: 2 minute tutorials on R](#)
- ❖ [Twitter: #rstats](#)

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)

Exercise 1: Using RStudio

1. Find your current working directory in RStudio.
2. Create a directory where you save all the files related to this class. Set the working directory to this directory.
3. Find the documentation for the function “lm”.
4. Run the function `ls()`, what does it print out?
5. We can remove everything in the workspace by specifying the parameter `list=ls()` within the function `rm()`. Try it!.

Exercise 1 Solution

1. `getwd()`
2. `setwd('~/.path.name/file.name')`
3. `?lm`
4. `ls()` # Prints all the objects in the workspace
5. `rm(list = ls())`

Exercise 2: Variables and Functions

1. Assign the text “Hello, I am” to a variable x.
2. Assign your name to a variable y.
3. The function `cat()` takes multiple character values, concatenates them, and returns the result. Run `cat(x, y)`.

Exercise 2 Solution

1.

```
x = "Hello, I am"
```
2.

```
y = "Vivian"
```
3.

```
cat(x,y)  
Hello, I am Vivian
```

Exercise 3: Numerics

- ❖ Do the following calculations in the console. When you enter these expressions, you may find that placing parentheses around parts of them gives different results:
 1. Calculate $5 / 2$
 2. Calculate $5 \% \% 2$ and compare it with the previous result.
 3. Calculate 3 to the power of $5 \% \% 2$.
 4. Calculate -3 to the power of $5 \% \% 2$.
 5. Calculate -3 to the power of $5 / 2$.
 6. Do any of the above NOT return a number? Why do you think that happens?

Exercise 3 Solution

1.

```
5/2
```

```
[1] 2.5
```

2.

```
5 %/% 2
```

```
[1] 2      # This returns integer division.
```

3.

```
3^(5 %/% 2)
```

```
[1] 9
```

4.

```
(-3)^(5 %/% 2)
```

```
[1] 9
```

Exercise 3 Solution

5.

```
(-3)^(5 / 2)
```

```
[1] NaN
```

6. Fractal powers involve roots. For example, raising to a $\frac{1}{2}$ power is the same as taking square root. Therefore, R forbids raising negative bases to fractal powers to avoid errors. That is why question 5. returns NaN.

Exercise 4: Primitive Types

- ❖ Do the following calculations in the console:
 1. Assign the value '8' (with the single quotes) to the variable x.
 2. Find the class of x.
 3. What happens when we compute $x + 1$?
 4. If we want to compute $x + 1$, we need to change the type of x to *numeric* first. How do we do that?
 5. Run the code `x = x + 1`, what is x now?

Exercise 4 Solution

1. `x = '8'`

2. `class(x)`
`[1] 'character'`

3. `x + 1`
`Error in x + 1 : non-numeric argument to binary operator`

4. `x = as.numeric(x)`
`x`
`[1] 8`

Exercise 4 Solution

5.

```
x = x + 1
```

```
x
```

```
[1] 9
```

Exercise 5: Vector Creations

1. Create a random vector of length 5 from draws of the standard normal distribution; call this vector `x_1`.
2. Create a vector consisting of integers from 1 to 10, and call it `x_2`.

Exercise 5 Solution

1.

```
x_1 = rnorm(5)
```

```
x_1
```

```
[1]  1.5117812  0.3898432 -0.6212406 -2.2146999  
1.124930      # Your result may be different
```

2.

```
x_2 = 1:10
```

```
x_2
```

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

Exercise 6: Vector Creation

1. Create a vector `x_3` which consists of the first 5 elements from `x_2` (the vector you created in the last exercise).

Exercise 6 Solution

```
1. x_3 = x_2[1:5]  
x_3  
[1] 1 2 3 4 5
```

Exercise 7: Vector Calculations

1. Find the sum of the elements of `x_1` and `x_3` and call it 'my_sum'.
2. What is the product of the third entry and the first entry in `x_1`?
3. What happens if you run the code `x_1 + c(1,2)`? Why do you think this happens?

Exercise 7 Solution

1.

```
my_sum = x_1 + x_3
```

```
[1] 2.511781 2.389843 2.378759 1.785300
```

```
[5] 6.124930
```

2.

```
x_1[1] * x_1[3]
```

3.

```
x_1 + c(1,2)
```

```
[1] 2.5117812 2.3898432 0.3787594 -0.2146999
```

```
2.1249309
```

```
Warning message:
```

```
In x + c(1, 2) :
```

```
longer object length is not a multiple of  
shorter object length
```

Exercise 7 Solution

3. (Continued) R added 1 to the first element of x , then 2 to the second element of x . Since $c(1,2)$ is shorter than x , it added 1 again to the third element of x . It repeats this process and warns that the length of the longer vector, x (which is 5 in this example) is not a multiple of the shorter one (which is 2 in this example).

Exercise 8: Creating Matrices

1. Arrange 1:6 into a 2 by 3 matrix (2 rows and 3 columns), then call it m_1.
2. Arrange 7:18 into a 4 by 3 matrix (4 rows and 3 columns), then call it m_2.
3. Stack m_1 on the top of m_2. You should end up with a 6 by 3 matrix.

Exercise 8 Solution

1. `m_1 = matrix(1:6, nrow=2, ncol=3)`

2. `m_2 = matrix(7:18, nrow=4, ncol=3)`

3. `rbind(m_1, m_2)`

	[,1]	[,2]	[,3]
[1,]	1	3	5
[2,]	2	4	6
[3,]	7	11	15
[4,]	8	12	16
[5,]	9	13	17
[6,]	10	14	18

Exercise 9: Matrix Computations

- ❖ Create `my_mat2` to be `matrix(1:9, 3, 3, byrow=T)`:
 1. Compared to `my_mat` constructed in the previous slides, what is the difference? Why?
 2. Compute the elementwise product of the first row and the second column of `my_mat2`.

Exercise 9: Solution:

❖ Create my_mat2 to be `matrix(1:9,3,3, byrow=T)`:

1. Compare to my_mat constructed earlier, what is the difference? Why?

```
my_mat2 = matrix(1:9,3,3, byrow=T)
```

```
my_mat2
```

```
      [,1] [,2] [,3]  
[1,]    1    2    3  
[2,]    4    5    6    # byrow=T indicates  
[3,]    7    8    9    # filling across rows
```

Exercise 9: Solution:

- ❖ Create my_mat2 to be `matrix(1:9,3,3, byrow=T)`:
- 2. Compute the product of the first row and the second column of my_mat2.

```
my_mat2[1,] * my_mat2[,2]
```

```
[1]  2 10 24
```

Exercise 10: Creating Data Frames

- ❖ Create a data frame *Pet* with two columns. The first column should be named *Species* and takes values `c('dog', 'cat', 'dog')`. The second should be named *weight* and takes values `c(20, 10, 40)`.

Exercise 10 Solution

1.

```
Pet = data.frame(Species = c('dog', 'cat',  
  'dog'),weight = c(20,10,40))
```

Pet

	Species	weight
1	dog	20
2	cat	10
3	dog	40

Exercise 11: Slicing Data Frames

- ❖ Select from *Pet* the rows where Species is dog.

Exercise 11 Solution

1. `Pet[Pet$Species=='dog',]`

Exercise 12: Data Frame Computations

- ❖ Use the data frame we called `data`, but this time order it based on the `city` column in decreasing order.
- ❖ Name the ordered data frame `data1` and check the structure of it using `str(data1)`.
- ❖ Compare `data1` and `data`.

Exercise 12: Solution

```
data1 = data[order(data$city, decreasing = T),]  
str(data)  
str(data1)
```

Exercise 13: Creating Lists

- ❖ Instead of a *Pet* data frame, create a *Pet* list. The first element should be named *Species* and take values `c('dog', 'cat', 'dog')`. The second should be named *weight* and take values `c(20, 10, 40)`.

Exercise 13 Solution

1.

```
Pet = list(Species = c('dog', 'cat', 'dog'),  
weight = c(20,10,40))
```

```
Pet
```

```
$Species
```

```
[1] "dog" "cat" "dog"
```

```
$weight
```

```
[1] 20 10 40
```

Exercise 14: List Calculations

- ❖ Find the mean of weight in *Pet*.
- ❖ Assume all the pets we record gain two pounds this year. Update your list.

Exercise 14: Solution

```
mean(Pet$weight)
Pet$weight = Pet$weight + 2
Pet[[2]] = Pet[[2]] + 2 # If you decide to use
                        # numeric indexes, make
                        # sure you use [[]].
```

Exercise 15: sapply and lapply

- ❖ For the iris data, find the standard deviation of each column and print out the result as a:
 - vector.
 - list.

Exercise 15: Solution

- ❖ Notice that we only apply the function `sd()` on numerical columns.
 - vector:

```
sapply(iris[,1:4], sd)
Sepal.Length Sepal.Width Petal.Length Petal.Width
0.8280661     0.4358663     1.7652982     0.7622377
```

Exercise 15: Solution

➤ list:

```
lapply(iris[,1:4], sd)
```

```
$Sepal.Length
```

```
[1] 0.8280661
```

```
$Sepal.Width
```

```
[1] 0.4358663
```

```
$Petal.Length
```

```
[1] 1.765298
```

```
$Petal.Width
```

```
[1] 0.7622377
```

Exercise 16: tapply

- ❖ For each species in the iris data set, find the maximum sepal length.

Exercise 16: Solution

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

setosa	versicolor	virginica
5.8	7.0	7.9