

# CDC Data Science Certificate Program

## Basics of R Programming



# words of welcome

**You are here because** you want to work better with data. Becoming comfortable with using a programming language for your data manipulations is a gigantic step forward. This way you will be able to automate, reproduce, and communicate your manipulations faster and better.

**R is (perhaps) the best language to start with:** it is an open-source language that is heavily tailored for handling data: from data wrangling through data visualization to machine learning. R's functionality is huge and, kind of like our universe, R's universe is always expanding.

**Packages** are the main source of functionality in R: they comprise different families of functions intended for a specific manipulation or field. Many R's packages are stored on the CRAN project website <https://cran.r-project.org/>, but there are CRAN-external sources, too.



R is a programming language, particularly in the world of data analysis and data science. This is an introductory course to get you started with RStudio with hands-on examples. In this course, you will learn:

- How to use RStudio, a free and open-source development environment for R,
- Learn the fundamentals of R syntax,
- How to assign and manipulate variables,
- Learn the data types,
- Use R for math: variable types, vectors, calling functions, and more
- Learn the data structures; vectors, matrices, lists, arrays, and data frames,
- Exploit data structures, including data.frames, matrices, and lists
- Read many different types of data

## DATA PREPARATION

### DATA CLEANING

INCONSISTENT DATATYPES  
MISSPELLED ATTRIBUTES  
MISSING AND DUPLICATE VALUES

### TRANSFORMATION



## EXPLORATORY DATA ANALYSIS



DEFINES AND REFINES  
THE SELECTION OF FEATURE  
VARIABLES THAT WILL BE USED  
IN THE MODEL DEVELOPMENT

## DATA MODELING

simplilearn

KNN



NAIVE BAYES

DECISION TREE

## VISUALIZATION AND COMMUNICATION



# WHAT IS DATA SCIENCE?

### DATA ACQUISITION

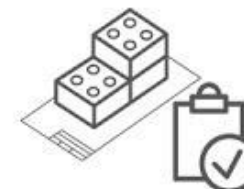
- WEB SERVERS
- LOGS
- DATABASES
- APIS
- ONLINE REPOSITORIES



WHY?...WHY?...WHY?....



## DEPLOYS AND



# Setting up your machine

## RStudio Desktop 2021.09.0+351 - [Release Notes](#)

1. Install R. RStudio requires [R 3.0.1+](#).
2. Download RStudio Desktop. Recommended for your system:



**DOWNLOAD RSTUDIO FOR WINDOWS**

2021.09.0+351 | 156.88MB

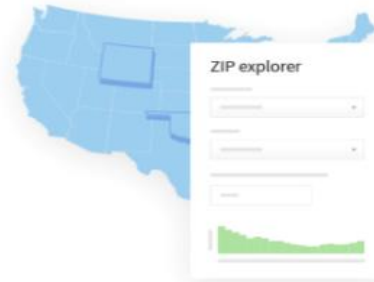
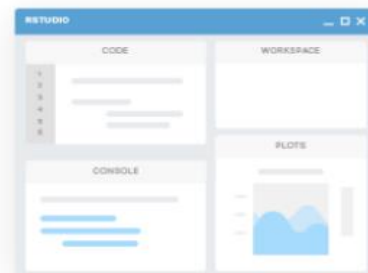
Requires Windows 10 (64-bit)

## Software Links:

- <https://www.r-project.org/>
- <https://rstudio.com/products/rstudio/download/#download>
- <https://cran.r-project.org/mirrors.html>

# RStudio

Open source and enterprise-ready  
professional software for R

[Download RStudio](#)[Discover Shiny](#)[shinyapps.io Login](#)[Discover RStudio Connect](#)

<https://cran.r-project.org/doc/manuals/R-intro.pdf>

# Getting Started with RStudio

Using RStudio for data analysis and programming in R provides many advantages. Here are a few examples of what RStudio provides:

- An intuitive interface that lets us keep track of saved objects, scripts, and figures
- A text editor with features like color-coded syntax that helps us write clean scripts
- Auto complete features save time
- Tools for creating documents containing a project's code, notes, and visuals
- Dedicated Project folders to keep everything in one place
- RStudio can also be used to program in other languages including SQL, Python, and Bash, to name a few.



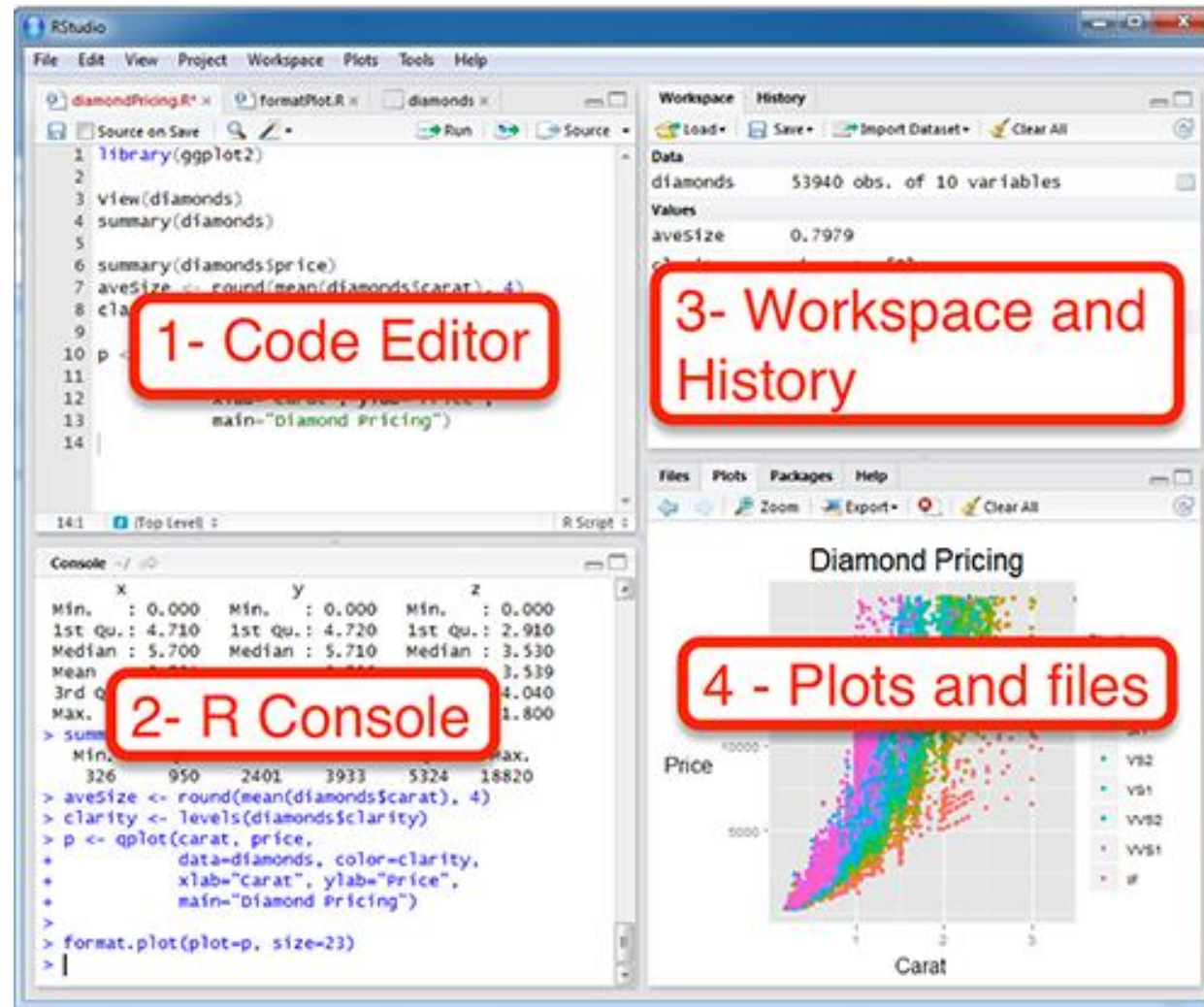
RStudio is a four pane work-space for 1) creating file containing R script, 2) typing R commands, 3) viewing command histories, 4) viewing plots and more.

### 1.Top-left panel:

- Code editor allowing you to create and open a file containing R script.
- The R script is where you keep a record of your work. R script can be created as follow: File → New → R Script

### 2.Bottom-left panel:

- R console for typing R commands



### 3.Top-right panel:

- Workspace tab: shows the list of R objects you created during your R session
- History tab: shows the history of all previous commands

### 4.Bottom-right panel:

- Files tab: show files in your working directory
- Plots tab: show the history of plots you created. From this tab, you can export a plot to a PDF or an image files
- Packages tab: show external R packages available on your system. If checked, the package is loaded in R.



# Shortcut Keys

---

Console		
Description	Windows & Linux	Mac
Move cursor to Console	Ctrl+2	Ctrl+2
Clear console	Ctrl+L	Ctrl+L
Move cursor to beginning of line	Home	Cmd+Left
Move cursor to end of line	End	Cmd+Right
Navigate command history	Up/Down	Up/Down
Popup command history	Ctrl+Up	Cmd+Up
Interrupt currently executing command	Esc	Esc
Change working directory	Ctrl+Shift+H	Ctrl+Shift+H

---

# Shortcut Keys

Editing (Console and Source)		
Description	Windows & Linux	Mac
Undo	Ctrl+Z	Cmd+Z
Redo	Ctrl+Shift+Z	Cmd+Shift+Z
Cut	Ctrl+X	Cmd+X
Copy	Ctrl+C	Cmd+C
Paste	Ctrl+V	Cmd+V
Select All	Ctrl+A	Cmd+A
Jump to Word	Ctrl+Left/Right	Option+Left/Right
Jump to Start/End	Ctrl+Home/End or Ctrl+Up/Down	Cmd+Home/End or Cmd+Up/Down
Delete Line	Ctrl+D	Cmd+D
Select	Shift+[Arrow]	Shift+[Arrow]
Select Word	Ctrl+Shift+Left/Right	Option+Shift+Left/Right
Select to Line Start	Alt+Shift+Left	Cmd+Shift+Left
Select to Line End	Alt+Shift+Right	Cmd+Shift+Right
Select Page Up/Down	Shift+PageUp/PageDown	Shift+PageUp/Down

# Shortcut Keys

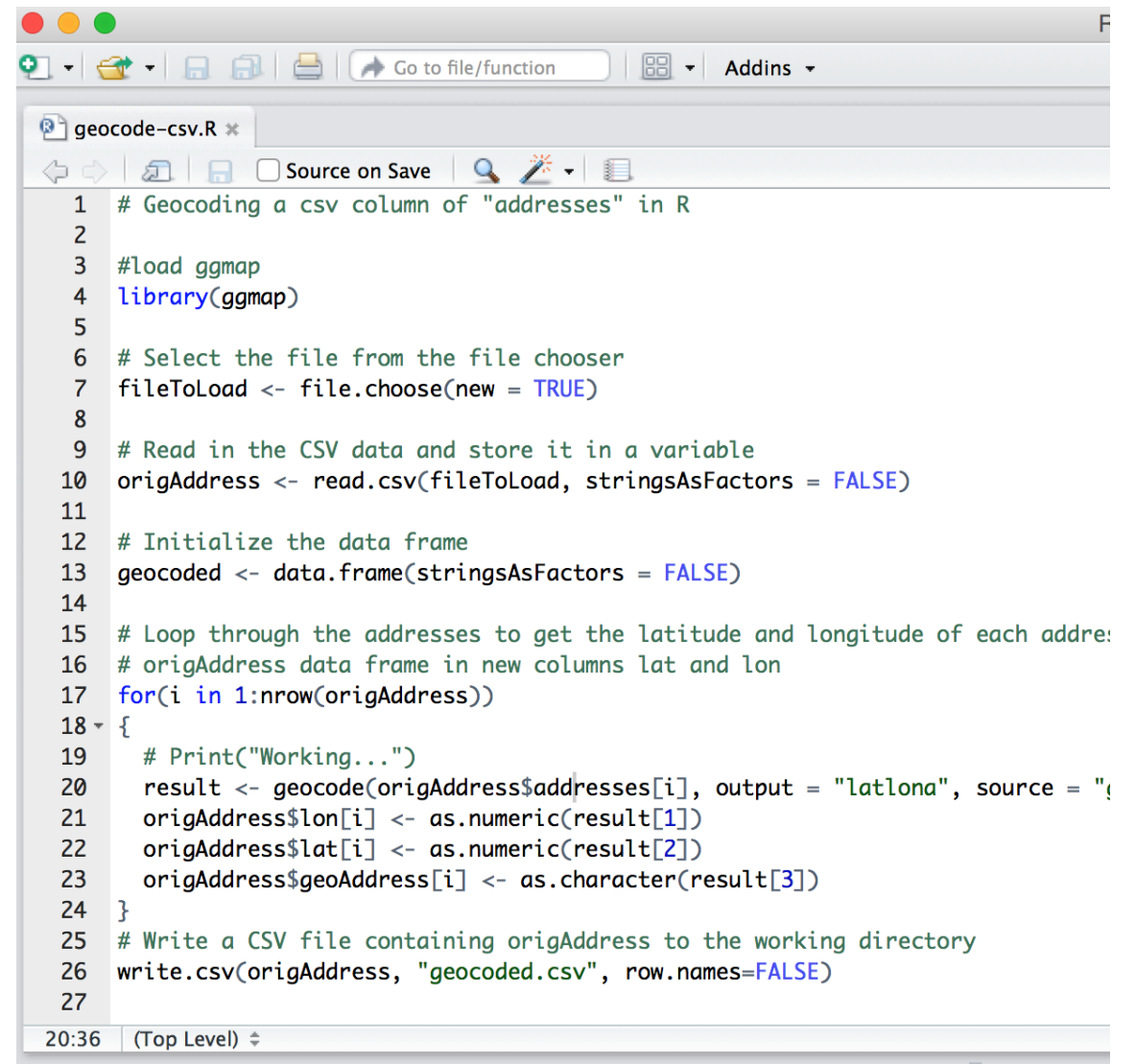
Description	Editing (Console and Source)	
	Windows & Linux	Mac
Select to Start/End	Ctrl+Shift+Home/End or Shift+Alt+Up/Down	Cmd+Shift+Up/Down
Delete Word Left	Ctrl+Backspace	Option+Backspace or Ctrl+Option+Backspace
Delete Word Right	No shortcut	Option+Delete
Delete to Line End	No shortcut	Ctrl+K
Delete to Line Start	No shortcut	Option+Backspace
Indent	Tab (at beginning of line)	Tab (at beginning of line)
Outdent	Shift+Tab	Shift+Tab
Yank line up to cursor	Ctrl+U	Ctrl+U
Yank line after cursor	Ctrl+K	Ctrl+K
Insert currently yanked text	Ctrl+Y	Ctrl+Y
Insert assignment operator	Alt+-	Option+-
Show help for function at cursor	F1	F1
Insert code section	Ctrl+Shift+R	Cmd+Shift+R
Run current line/selection	Ctrl+Enter	Cmd+Return
Run current line/selection (retain cursor position)	Alt+Enter	Option+Return

Source		
Description	Windows & Linux	Mac
Move cursor to Source Editor	Ctrl+1	Ctrl+1
Open document	Ctrl+O	Cmd+O
Save active document	Ctrl+S	Cmd+S
Save all documents	Ctrl+Alt+S	Cmd+Option+S
Close active document (except on Chrome)	Ctrl+W	Cmd+W
Close active document (Chrome only)	Ctrl+Alt+W	Cmd+Option+W
Close all open documents	Ctrl+Shift+W	Cmd+Shift+W
Compile Notebook	Ctrl+Shift+K	Cmd+Shift+K
Insert code section	Ctrl+Shift+R	Cmd+Shift+R
Run current line/selection	Ctrl+Enter	Cmd+Return
Run current line/selection (retain cursor position)	Alt+Enter	Option+Return
Re-run previous region	Ctrl+Alt+P	Cmd+Alt+P
Run current document	Ctrl+Alt+R	Cmd+Option+R
Run from document beginning to current line	Ctrl+Alt+B	Cmd+Option+B
Run from current line to document end	Ctrl+Alt+E	Cmd+Option+E
Run the current function definition	Ctrl+Alt+F	Cmd+Option+F
Run the current code section	Ctrl+Alt+T	Cmd+Option+T
Send current line/selection to terminal	Ctrl+Alt+Enter	Cmd+Option+Return
Edit lines from start	Ctrl+Alt+Shift+A	Ctrl+Shift+Option+A

# Shortcut Keys

# R Script Files

Often you will do your programming by writing your programs in script files and then you execute those scripts at your command prompt with the help of R interpreter called Rscript.



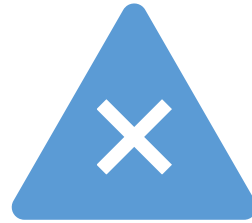
```
1 # Geocoding a csv column of "addresses" in R
2
3 #load ggmap
4 library(ggmap)
5
6 # Select the file from the file chooser
7 fileToLoad <- file.choose(new = TRUE)
8
9 # Read in the CSV data and store it in a variable
10 origAddress <- read.csv(fileToLoad, stringsAsFactors = FALSE)
11
12 # Initialize the data frame
13 geocoded <- data.frame(stringsAsFactors = FALSE)
14
15 # Loop through the addresses to get the latitude and longitude of each address:
16 # origAddress data frame in new columns lat and lon
17 for(i in 1:nrow(origAddress))
18 {
19   # Print("Working...")
20   result <- geocode(origAddress$addresses[i], output = "latlon", source = "ggmap")
21   origAddress$lon[i] <- as.numeric(result[1])
22   origAddress$lat[i] <- as.numeric(result[2])
23   origAddress$geoAddress[i] <- as.character(result[3])
24 }
25 # Write a CSV file containing origAddress to the working directory
26 write.csv(origAddress, "geocoded.csv", row.names=FALSE)
27
```

20:36 (Top Level) ↕

# Comments



Comments are like helping text in your R program



These statements are ignored by the interpreter while executing your actual program.



Single comment is written using `#` in the beginning of the statement.

`#It is considered good code to have your scripts commented.`



# Quotation Marks “”

```
> print("""hi""")
```

```
Error: unexpected symbol in  
"print("""hi"
```

```
> print("hi")
```

```
[1] "hi"
```

```
> print("hi")
```

```
[1] "hi"
```

- Use quotes to tell R to interpret something as a string. You should prefer double quotes (") to single quotes (').
- Both double quotes (") and single (') quotes work, but there are some guidelines for which to use.
- Unfortunately, if your string has " inside it, R will interpret the double quote as "this is the end of the string", not as "this is the character " ".
- Cases where you need both ' and " inside the string.
  - In this case, fall back to the first guideline and use " to define the string, but you'll have to escape any double quotes inside the string using a backslash (i.e., \").

# Getting Help



The command **help.start()** or choosing HTML Help from the Help menu will yield a table of contents that points to help files, manuals, frequently asked questions and the like.



If you don't know the name of a command or operator, use **help.search("your search string")** to search the built-in help files



To get help on an operator, enclose it in quotes as in **help("<-")** for the assignment operator.



To get help for a certain function such as summary, use **help(summary)** or prefix the topic with a question mark: **?summary**.



To get help for a quick reminder of the function arguments use **args("functionname")**.



To view examples of using a function use **example("functionname")**.



Search the R site using **RSiteSearch("your search string")** or go to <http://www.r-project.org/> and click search.

# The Working Directory

- The working directory is the default location for all input and output
  - Reading and writing data files
  - Opening and saving script files
  - Saving workspace image
- From the command line,
  - `getwd()` – reports the working directory
  - `setwd()` – changes the working directory
    - `setwd("C:/myfolder/data")`
    - `setwd("C :\\myfolder\\data")`



# Terms

- A variable is a quantity, quality, or property that you can measure.
- A value is the state of a variable when you measure it. The value of a variable may change from measurement to measurement.
- An observation is a set of measurements made under similar conditions (you usually make all the measurements in an observation at the same time and on the same object). An observation will contain several values, each associated with a different variable. I'll sometimes refer to an observation as a data point.
- Tabular data is a set of values, each associated with a variable and an observation. Tabular data is tidy if each value is placed in its own “cell”, each variable in its own column, and each observation in its own row.

# Variables

	A	B	C	D	E	F
1	name	100m	Long jump	Shot_put	High jump	400m
2	SEBRLE	11.04	7.58	14.8	2.07	49.81
3	CLAY	10.76	7.4	14.26	1.86	49.37
4	KARPOV			14.77		
5	BERNARD			14.25		
6	YURKOV	11.34	7.09			
7	WARNERS	11.11	7.6		1.98	48.68
8	ZSIVOCZKY	11.13	7.3	13.48	2.01	48.62
9	McMULLEN	10.83	7.31	13.76	2.13	49.91
10	MARTINEAU		6.81	14.57	1.95	50.14
11	HERNU	11.37	7.56	14.41		51.1
12	BARRAS		6.97	14.09		49.48
13	NOOL	11.33			1.98	49.2
14	BOURGUIGNO	11.36			1.86	51.16

- Variables store values and are an important component in programming. A variable can store a number, an object, a statistical result, vector, dataset, a model prediction basically anything R outputs
- A valid variable name consists of letters, numbers and the dot or underline characters.
- The variable name starts with a letter, or the dot should not follow by a number.

## Variable Name

var\_name2.

## Validity

valid

## Reason

Has letters, numbers, dot and underscore

var\_name%

Invalid

Has the character "%". Only dot(.) and underscore allowed.

2var\_name

invalid

Starts with a number

.var\_name, var.name

valid

Can start with a dot(.) but the dot(.) should not be followed by a number.

.2var\_name

invalid

The starting dot is followed by a number making it invalid.

\_var\_name

invalid

Starts with \_ which is not valid

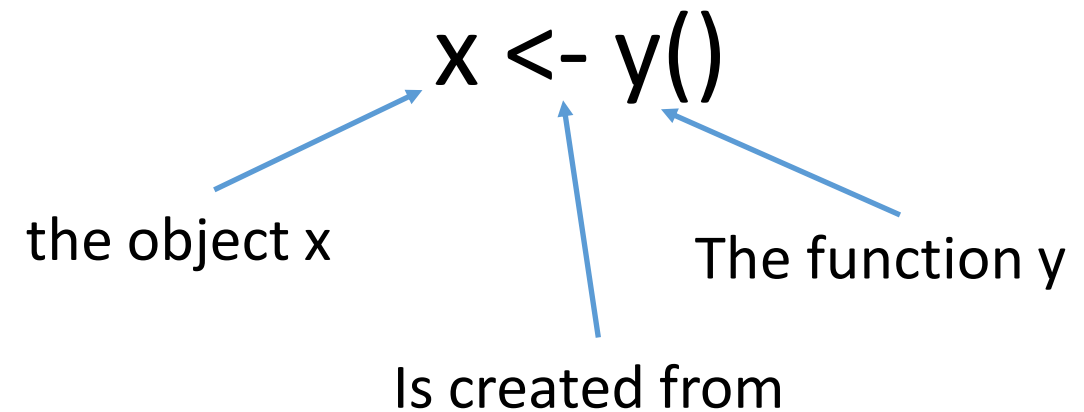
# Variable Assignment

- Objects are named data structures inside of which you store data. An object can be a single digit, a character, a Boolean value, a sentence, a data frame, a list of data frames, a multi-dimensional structure, and so on. You use functions to create objects.
- To declare a variable, we need to assign a variable name. The name should not have space. We can use `_` to connect to words.
- To add a value to the variable, use `<-` or `=`.

Here is the syntax:

```
# First way to declare a variable: use the `<-`  
name_of_variable <- value
```

```
# Second way to declare a variable: use the `=`  
name_of_variable = value
```





# Setting Variables

- When you define a variable at the command prompt, the variable is held in your workspace.
- The workspace is held in the computer's main memory but can be saved to disk when you exit from R.
- The variable definition remains in the workspace until removed.
- Once you store data inside an object, you can use the object name to call that data and do operations with it. An operation will be carried out on each element of the data structure, systematically.
- R is a dynamically typed language, which means that we can change a variable's data type at will.
- We could set `a` to be numeric, and then turn around and immediately overwrite that with (by example) a vector of character strings. eg;  
    `> a <- 4`  
    `> a <- c("sun", "moon", "rain", "clouds")`

# Printing using the print() function

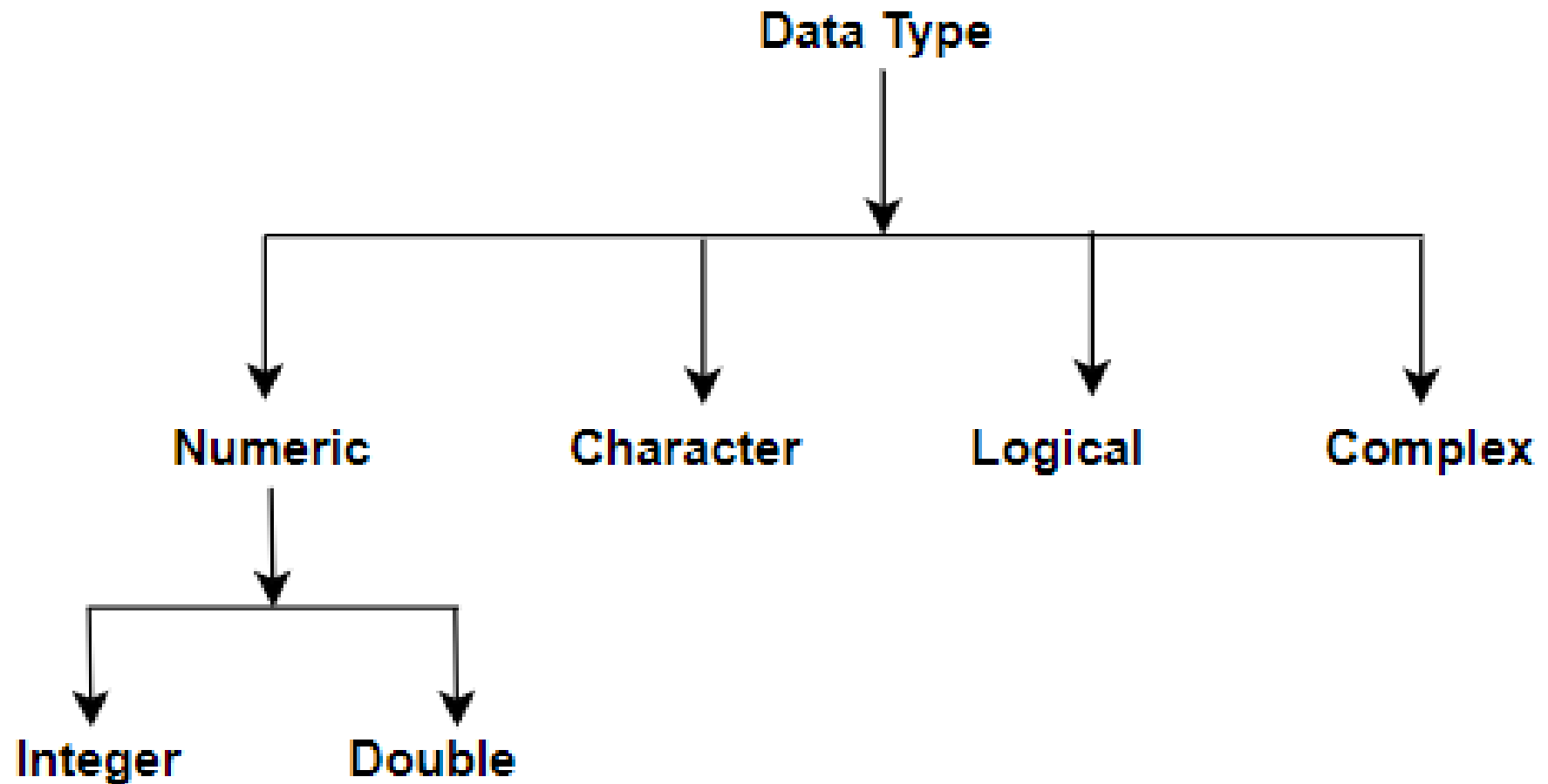
- One can use the print() function which displays same result;  
    > **print(pi)**  
    Output: [1] 3.141593  
    > **print(sqrt(2))**  
    Output: [1] 1.414214  
    > **print(x)**  
    Output: [1] 3
- The print function has a significant limitation, it prints only one object at a time.
- The only way to print multiple items is to print them one at a time.  
    > **print("The zero occurs at"); print(2\*pi); print("radians")**  
    Output: [1] "The zero occurs at"  
    Output: [1] 6.283185  
    Output: [1] "radians"

# “print” function vs. printing using the cat() function

## Definitions:

- The print R function returns a data object to the R (or RStudio) console.
- The cat R function returns a character string in a readable format.
- If we want to return the below character string to the RStudio console, we can either use the print function....
  - > print('The location occurs at', 2\*pi, "radians") # Apply print to character string  
Output: [1] "The location occurs at", 2\*pi, \"radians"
- Or the cat() function, an alternative to print and lets you combine multiple items;
  - > cat("The location occurs at", 2\*pi, "radians.", "\n") # "\n" passed as a separator  
Output: [1] The location occurs at 6.283185 radians.

# Data Types in R



## Integer

- An integer is a whole number; any number that doesn't need anything after the decimal point is an integer.
- R doesn't usually jump to creating an integer vector when you pass numbers. Often, you need to explicitly tell it to do so.
- You do that by passing the letter L after each number in the integer object you're creating

## Double

- Doubles store regular numbers: they can be large, small, positive, negative, with digits after the decimal or without.
- Because R is heavily used for data analysis and most operations typically either involve or result in a double, this is R's default way of saving numerical data.

## Character

- Character vectors store text data. A character element can be a single letter, number, or a symbol, or a longer string, like a sentence or even a paragraph.
- To create a character string, you must pass the value you want stored as a string in quotation marks.

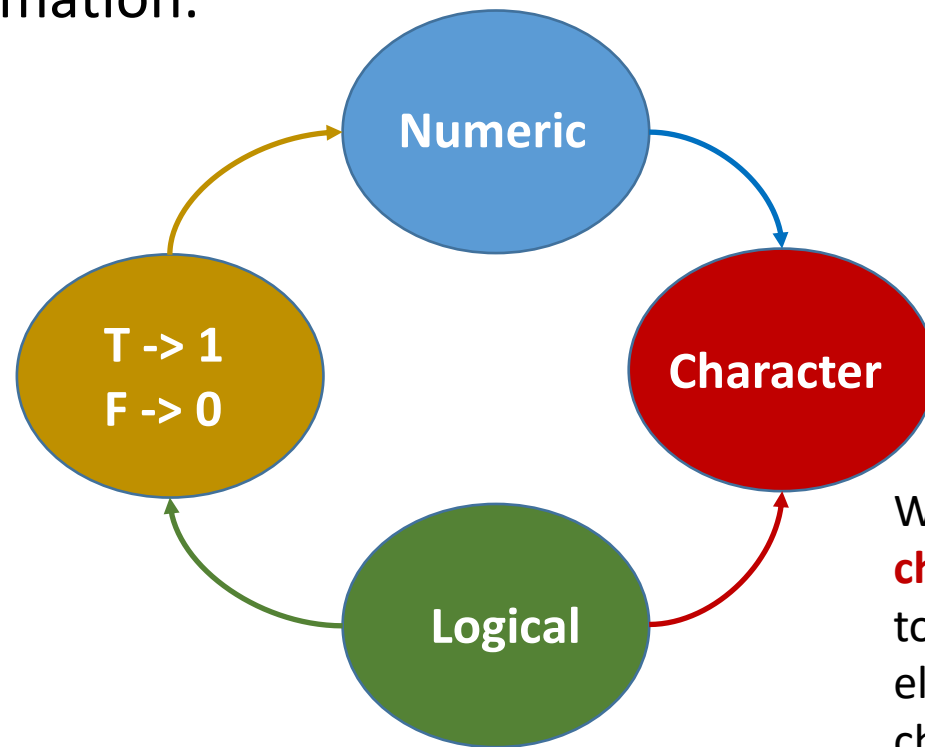
## Logical

- Logical vectors store Boolean data; these are TRUE and FALSE values.
- TRUE and FALSE and T and F can be used interchangeably as R recognizes both. It is better, however, to use TRUE and FALSE because these are the reserved words which cannot be overwritten by the user.
- TRUE and FALSE values must be inputted in capital letters, and without quotation marks.

# Coercion rules

R has ways to prevent certain mistakes from happening. For example, if you are trying to create an object, and you are passing the wrong type of data as an argument, then R will convert the value to the correct type, so you can end up creating your object. The correct type is typically the simplest type necessary to represent all the information.

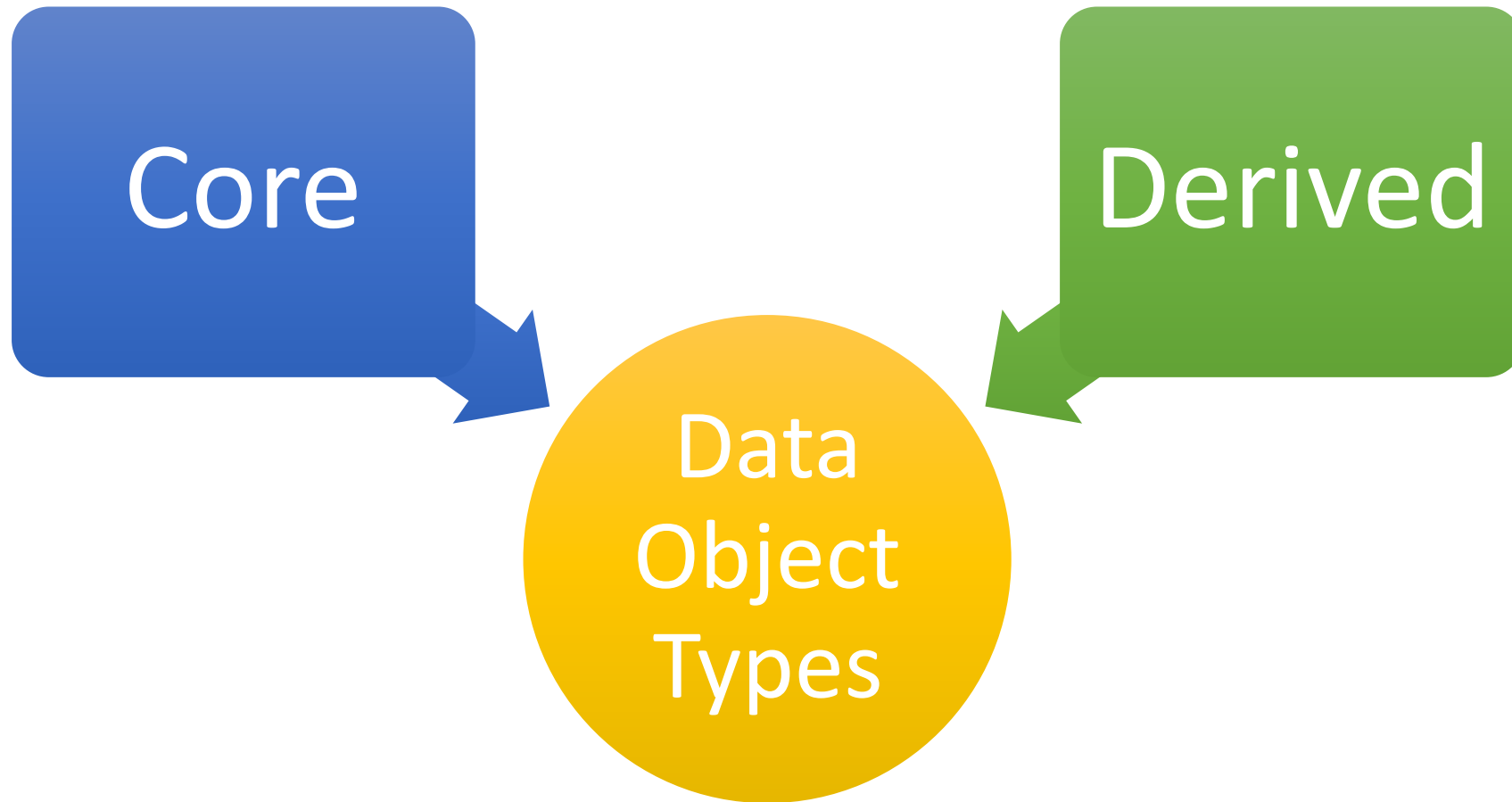
When a **numeric** and a **logical** element are present together, the logical value is converted to a numerical one.  
**TRUE is encoded as 1**  
**FALSE is encoded as 0**



When a **numeric** and a **character** value are present together, the numeric value is converted to a character

When a **logical** and a **character** value are present together, the logical element is coerced to a character.



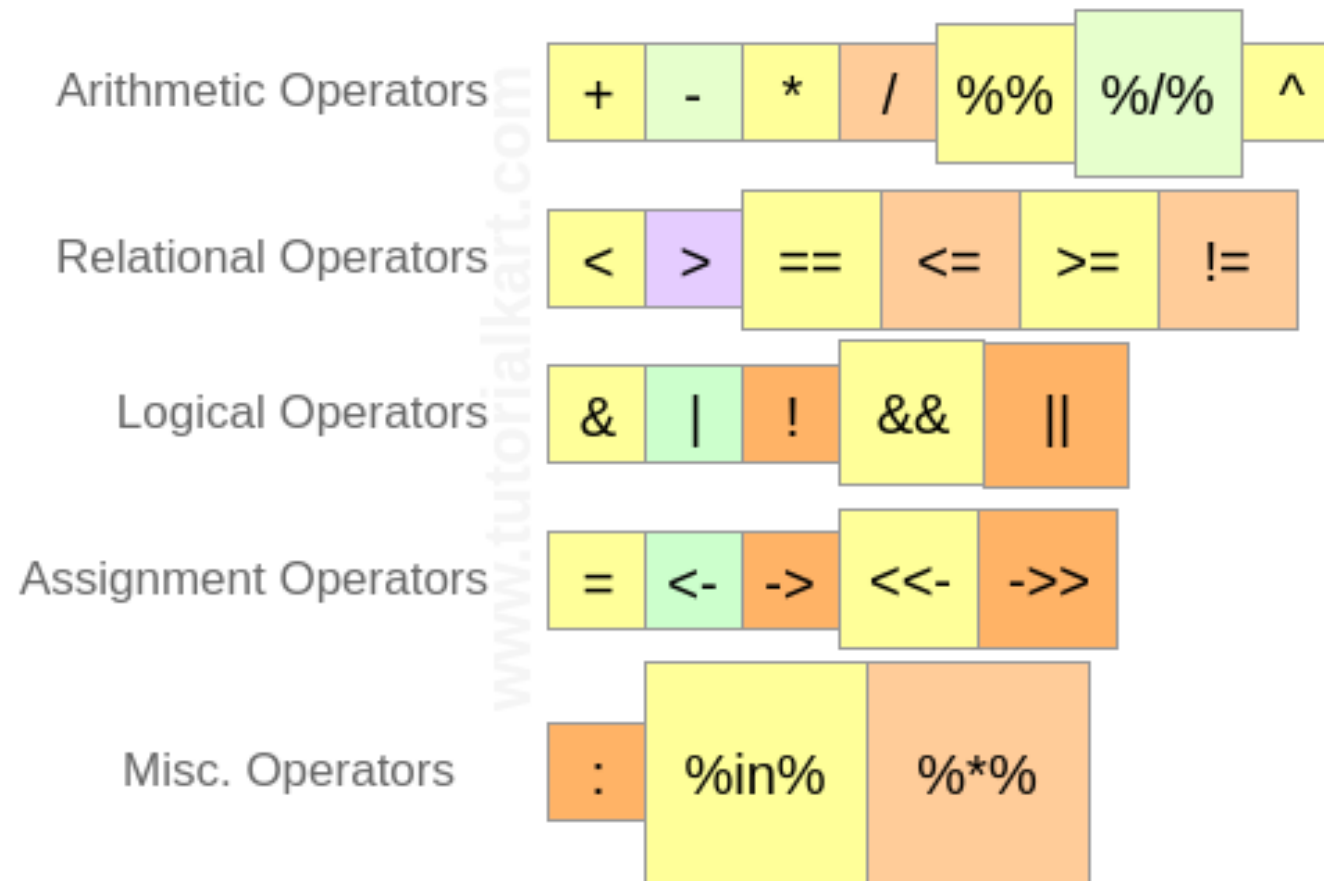


Data Type	Example	Code
Logical	TRUE, FALSE	v <- TRUE print(class(v)) [1] "logical"
Numeric	16.3, 5, 999	v <- 16.3 print(class(v)) [1] "numeric"
Integer	3L, 34L, 0L	v <- 3L print(class(v)) [1] "integer"
Complex	3 + 2i	v <- 3 + 2i print(class(v)) [1] "complex"
Character	Friday is a' , '"good day", "TRUE", '23.4'	v <- "TRUE" print(class(v)) [1] "character"
Raw	"Hello" is stored as 48 65 6c 6c 6f	v <- charToRaw("Hello") print(class(v)) [1] "raw"

# Logical data types

- These are applicable only to vectors of type logical, numeric or complex.
- All numbers greater than 1 are considered as logical value TRUE.
- Each element of the first vector is compared with the corresponding element of the second vector.
- The result of comparison is a Boolean value.

R Operators: There are four main categories of Operators in R programming language. They are shown in the following picture :



# Basic Arithmetic Operators

Arithmetic Operators are used to accomplish arithmetic operations. They can be operated on the basic data types Numericals, Integers, Complex Numbers. Vectors with these basic data types can also participate in arithmetic operations, during which the operation is performed on one-to-one element basis.

Operator	Description	Meaning in Formula	Usage
+	Addition	Add term	$a+b$
-	Subtraction	Remove or exclude term	$a-b$
*	Multiplication	Main effect and interactions	$a*b$
/	Division	Main effect and nesting	$a/b$
^ or **	Exponentiation	Limit depth of interactions	$a^b$
%%	Remainder	Amount left that does not entirely go into the divisor	$a\%b$
%/	Quotient	quotient is the number of times a division is completed fully	$a\%/b$

## R Arithmetic Operators Example for integers

```
a <- 5.5
```

```
b <- 3
```

```
print ( a+b )    #addition
```

```
print ( a-b )    #subtraction
```

```
print ( a*b )    #multiplication
```

```
print ( a/b )    #Division
```

```
print ( a%%b )   #Remainder
```

```
print ( a%/%b )  #Quotient
```

```
print ( a^b )    #Power of
```



## Types

Converting between common data types in R. Can always go from a higher value in the table to a lower value.

as.logical	TRUE, FALSE, TRUE	Boolean values (TRUE or FALSE).
as.numeric	1, 0, 1	Integers or floating point numbers.
as.character	'1', '0', '1'	Character strings. Generally preferred to factors.
as.factor	'1', '0', '1', Levels: '1', '0'	Character strings with preset levels. Needed for some statistical models.

## Maths Functions

log(x)	Natural log.	sum(x)	Sum.
exp(x)	Exponential.	mean(x)	Mean.
max(x)	Largest element.	median(x)	Median.
min(x)	Smallest element.	quantile(x)	Percentage quantiles.
round(x, n)	Round to n decimal places.	rank(x)	Rank of elements.
signif(x, n)	Round to n significant figures.	var(x)	The variance.
cor(x, y)	Correlation.	sd(x)	The standard deviation.

## Variable Assignment

```
> a <- 'apple'
> a
[1] 'apple'
```




## The Environment

ls()	List all variables in the environment.
rm(x)	Remove x from the environment.
rm(list = ls())	Remove all variables from the environment.

You can use the environment panel in RStudio to browse variables in your environment.

## Matrices

```
m <- matrix(x, nrow = 3, ncol = 3)
# Create a matrix from x.
```

 m[2, ]	- Select a row	t(m)	Transpose
 m[, 1]	- Select a column	m %*% n	Matrix Multiplication
 m[2, 3]	- Select an element	solve(m, n)	Find x in: m * x = n

## Lists

```
l <- list(x = 1:5, y = c('a', 'b'))
# A list is a collection of elements which can be of different types.
```

l[[2]]	l[[1]]	l\$x	l['y']
Second element of l.	New list with only the first element.	Element named x.	New list with only element named y.




Also see the [dplyr](#) package.

## Data Frames



```
df <- data.frame(x = 1:3, y = c('a', 'b', 'c'))
# A special case of a list where all elements are the same length.
```

x	y
1	a
2	b
3	c

### Matrix subsetting

df[, 2]	
df[2, ]	
df[2, 2]	

### List subsetting

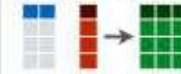
df\$x		df[[2]]	
Understanding a data frame			
View(df)	See the full data frame.		
head(df)	See the first 6 rows.		

nrow(df)  
Number of rows.

ncol(df)  
Number of columns.

dim(df)  
Number of columns and rows.

cbind - Bind columns.



rbind - Bind rows.



## Strings

Also see the [stringr](#) package.

paste(x, y, sep = ' ')	Join multiple vectors together.
paste(x, collapse = ' ')	Join elements of a vector together.
grep(pattern, x)	Find regular expression matches in x.
gsub(pattern, replace, x)	Replace matches in x with a string.
toupper(x)	Convert to uppercase.
tolower(x)	Convert to lowercase.
nchar(x)	Number of characters in a string.

## Factors

factor(x)	Turn a vector into a factor. Can set the levels of the factor and the order.
cut(x, breaks = 4)	Turn a numeric vector into a factor by 'cutting' into sections.

## Statistics

lm(y ~ x, data=df)	Linear model.	t.test(x, y)	Test for a difference between means.	prop.test	Test for a difference between proportions.
glm(y ~ x, data=df)	Generalised linear model.	pairwise.t.test	Perform a t-test for paired data.	aov	Analysis of variance.
summary	Get more detailed information out a model.				

## Distributions

	Random Variates	Density Function	Cumulative Distribution	Quantile
Normal	rnorm	dnorm	pnorm	qnorm
Poisson	rpois	dpois	ppois	qpois
Binomial	rbinom	dbinom	pbinom	qbinom
Uniform	runif	dunif	punif	qunif

## Plotting

Also see the [ggplot2](#) package.

 plot(x)	Values of x in order.	 plot(x, y)	Values of x against y.	 hist(x)	Histogram of x.
---	-----------------------	--	------------------------	---	-----------------

## Dates

See the [lubridate](#) package.

# Data Structures in



VECTORS

1

MATRIX

2

ARRAY

3

LIST

4

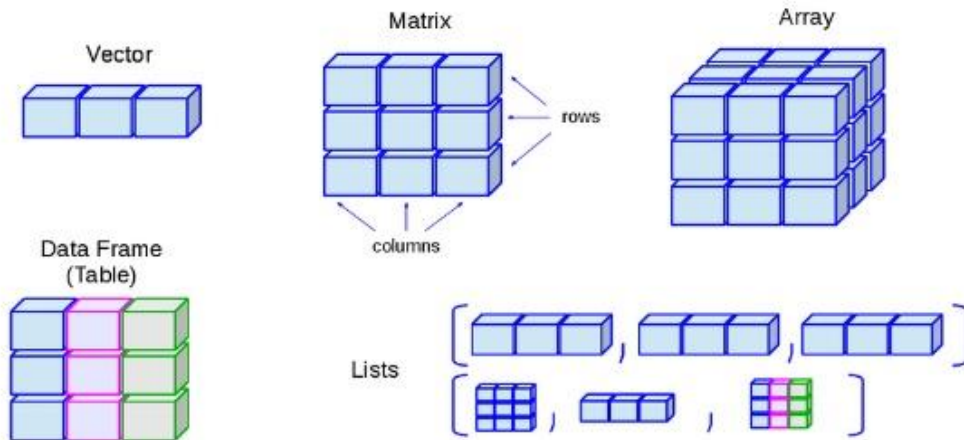
DATA FRAME

5

# Learning objectives of this module:

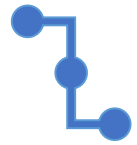
1. What are the three properties of a vector, other than its contents?
2.
  - a. What are the four common types of atomic vectors?
  - b. What are the two rare types?
3.
  - a. How is a list different from an atomic vector?
  - b. How is a matrix different from a data frame?

# Data structure types



15

- **Vector**: A sequence of numbers or characters, or higher-dimensional arrays like matrices.
- **Matrix**: The basic two-dimensional data structure with rows and columns,
- **Array**: If a higher dimension vector is desired, then use the function to generate the n-dimensional object.
- **List**: An ordered set of components stored in a 1D vector.
- **Data.Frame**: A table-like structure (experimental results often collected in this form).
- **Factor**: A sequence assigning a category to each index.



# Atomic Data Elements: Vectors

- In R the “base” type is a vector, not a scalar.
- A vector is an indexed set of values that are all of the same type. The type of the entries determines the class of the vector.
- An integer is a subclass of numeric.
- Cannot combine vectors of different modes



# Creating Vectors: combine (c) and colon (:)

- Vectors can only contain entries of the same type: numeric or character; you can't mix them.
- The most basic way to create a vector is with `c(x1, . . . , xn)`, and it works for characters and numbers alike. Note that characters should be surrounded by `" "`.

```
> x <- c(1,2,3)          #numeric
```

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

```
> x <- c("a", "b", "c")  #character
```

```
[1] "a" "b" "c"
```

```
> typeof(x)
```

```
[1] "character"
```

```
> length(x)
```

```
[1] 3
```

- Use the `:` operator to create and define the vector variable.

```
> v <- (10:15)
```

```
> print(v)
```

```
[1] 10 11 12 13 14 15
```

# Creating Vectors: sequence and replicate

You can also generate regular sequences:

**seq**(from = #, to = #, by = #): allows you to create a sequence from a starting number to an ending number.

```
> seq(from = 0, to = 6, by = 2)
```

```
[1] 0, 2, 4, 6
```

```
> seq(0, 1, length.out = 11) #desired length of the final sequence (only use if you don't specify by)
```

```
[1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

**rep**(x, times= #, each = #): function allows you to repeat a scalar (or vector) a specified number of times, or to a desired length.

```
> rep(c(7, 8), times = 2, each = 2)
```

```
[1] 7, 7, 8, 8, 7, 7, 8, 8
```

```
> rep(x = 3, times = 10)
```

```
[1] 3 3 3 3 3 3 3 3 3 3
```

```
> rep(x = 1:3, length.out = 10)
```

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

```
> rep("spades", 2)           #Rep can be used in replicating character string
```

```
[1] "spades" "spades"
```



# Vector Arithmetic

Numeric vectors can be used in arithmetic expressions, in which case the operations are performed element by element to produce another vector.

```
> x <- (1:20)
```

```
> print (x)
```

```
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

```
> x + 1
```

```
[1] 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
```

```
> y <- rnorm(20)
```

```
> x*y
```

```
> c(1, 2, 3, 4, 5) + c(5, 4, 3, 2, 1)
```

```
[1] 6 6 6 6 6
```



## More Vector Arithmetic Statistical operations on numeric vectors

- In studying data, you will make frequent use of `sum`, which gives the sum of the entries, `max`, `min`, `mean`.

Function	Example	Result
<code>sum(x)</code> , <code>product(x)</code>	<code>sum(1:20)</code>	210
<code>min(x)</code> , <code>max(x)</code>	<code>min(1:20)</code>	1
<code>mean(x)</code> , <code>median(x)</code>	<code>mean(1:20)</code>	10.5
<code>sd(x)</code> , <code>var(x)</code> , <code>range(x)</code>	<code>sd(1:20)</code>	5.91608
<code>quantile(x, probs)</code>	<code>quantile(1:20, probs = .2)</code>	20%, 4.8
<code>summary(x)</code>	<code>summary(1:20)</code>	Min = 1.00. 1st Qu. = 5.75, Median = 10.50, Mean = 10.50, 3rd Qu. = 15.25, Max = 20.0

(i.e.,  $\text{sum}((x - \text{mean}(x))^2)/(\text{length}(x) - 1)$ )

A useful function for quickly getting properties of a vector: `summary(x)`

# More vector arithmetic statistical operations on continuous vectors

Function	Description	Example	Result
<code>round(x, digits)</code>	Round elements in x to digits digits	<code>round(c(3.712, 3.1415), digits = 1)</code>	3.7, 3.1
<code>ceiling(x), floor(x)</code>	Round elements x to the next highest (or lowest) integer	<code>ceiling(c(2.6, 8.1))</code>	3, 9
<code>x %% y</code>	Modular arithmetic (ie. $x \bmod y$ )	<code>8 %% 4</code>	0

# Vector arithmetic statistical operations on discrete vectors

Function	Description	Example	Result						
unique(x)	Returns a vector of all unique values.	unique(c(3, 3, 4,5,12))	3, 4, 5, 12						
table(x, exclude)	Returns a table showing all the unique values as well as a count of each occurrence. To include a count of NA values, include the argument exclude = NULL	table(c("x", "x", "y", "z"))	<table><tr><td>x</td><td>y</td><td>z</td></tr><tr><td>2</td><td>1</td><td>1</td></tr></table>	x	y	z	2	1	1
x	y	z							
2	1	1							

# Missing Data

Missing data in R appears as NA. NA is not a string or a numeric value, but an indicator of missingness. We can create vectors with missing values.

```
> x1 <- c(1, 5, 9, NA, 7)
```

```
> x2 <- c("y", "D", NA, "NA")
```

NA is the one of the few non-numbers that we could include in **x1** without generating an error (and the other exceptions are letters representing numbers or numeric ideas like infinity).

In **x2**, the third value is missing while the fourth value is the character string "NA".

To see which values in each of these vectors R recognizes as missing, we can use the **is.na** function. It will return a TRUE/FALSE vector with as many elements as the vector we provide.

```
is.na(x1)
```

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

```
is.na(x2)
```

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

Our missing value cannot be compared to 0 and none of our values can be compared to NA because NA is not assigned a value—it simply is or it isn't.

- *NA is used for all kinds of missing data:* In other packages, missing strings and missing numbers might be represented differently—empty quotations for strings, periods for numbers. In R, NA represents all types of missing data. We saw a small example of this in **x1** and **x2**. **x1** is a “numeric” object and **x2** is a “character” object.
- *Non-NA values cannot be interpreted as missing:* Other packages allow you to designate values as “system missing” so that these values will be interpreted in the analysis as missing. In R, you would need to explicitly change these values to NA. The **is.na** function can also be used to make such a change:

```
is.na(x1) <- which(x1 == 7)
```

```
x1
```

```
## [1] 1 5 9 NA NA
```

## NA options in R

- We have introduced `is.na` as a tool for both finding and creating missing values. It is one of several functions built around NA. Most of the other functions for NA are options for `na.action`.
- Just as there are default settings for functions, there are similar underlying defaults for R as a software. You can view these current settings with `options()`. One of these is the “`na.action`” that describes how missing values should be treated. The possible `na.action` settings within R include:
- `na.omit` and `na.exclude`: returns the object with observations removed if they contain any missing values; differences between omitting and excluding NAs can be seen in some prediction and residual functions
- `na.pass`: returns the object unchanged
- `na.fail`: returns the object only if it contains no missing values

# Defining a Function

- Create a function by using the function keyword followed by a list of parameters and the function body. A one-liner looks like this:
  - `function(param1, ..., paramN) expr`
- The function body can be a series of expressions, in which case curly braces should be used around the function body:

```
function(param1, ..., paramN) {  
  expr1  
  .  
  .  
  .  
  exprM  
}
```

# Defining a Function

- e.g., function for calculating the coefficient of variation.

```
cv <- function(x) {mean(x)+2}  
cv(1:4)  
[1] 4.5
```

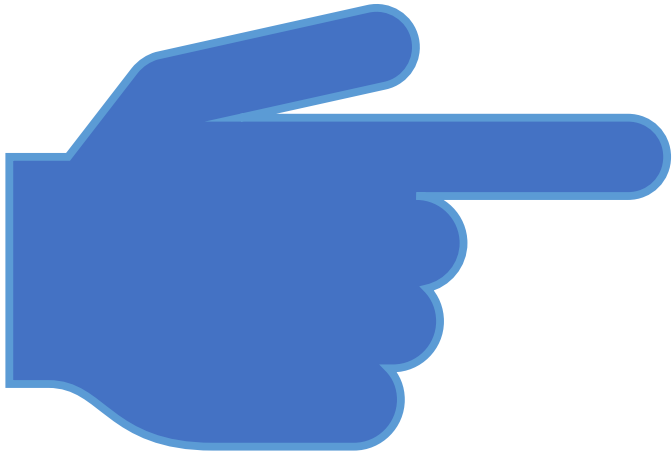
- The first line creates a function and assigns it to cv.
- The second line invokes the function, using 1:4 for the value of parameter x.

```
> cv <- function(x) {sd(x)/mean(x)}  
> cv(1:18)  
[1] 0.5619515
```

- The first line creates a function and assigns it to cv.
- The second line invokes the function, using 1:18 for the value of parameter x.



# Selecting Vector Elements



You can select the indexing technique appropriate:

- Use square brackets [ ] to select vector elements by their position  
     $v[2]$  – second element of vector  $v$
- Use negative indexes to exclude elements
- Use a vector of indexes to select multiple values.
- Use a logical vector to select elements based on a condition
- Use names to access named elements.

Relational Operators return values inside the vector based on logical conditions. Each element of the first vector is compared with the corresponding element of the second vector. The result of comparison is a Boolean value

You can add many conditional statements, but we need to include them in a parenthesis. Follow this structure to create a conditional statement:

Operator	Description	Meaning in Description	Usage
<	Less than	Is first operand less than second operand	a < b
>	Greater than	Is first operand greater than second operand	a > b
==	Exactly equal to	Is first operand equal to second operand	a == b
<=	Less than or equal to	Is first operand less than or equal to second operand	a <= b
>=	Greater than or equal to	Is first operand greater than or equal to second operand	a > = b
!=	Not equal to	Is first operand not equal to second operand	a!=b
!x	Not a	Is first operand not equal to itself	!a
a&b	a AND b	Is first operand and second operand	a&b
isTRUE(a)	Test if a is TRUE	Is first operand is TRUE	isTRUE(a)

# Logical Operators

- These are applicable only to vectors of type logical, numeric or complex.
- All numbers greater than 1 are considered as logical value TRUE.
- Each element of the first vector is compared with the corresponding element of the second vector.
- The result of comparison is a Boolean value.

&	It is called Element-wise Logical AND operator. It combines each element of the first vector with the corresponding element of the second vector and gives a output TRUE if both the elements are TRUE.
	It is called Element-wise Logical OR operator. It combines each element of the first vector with the corresponding element of the second vector and gives a output TRUE if nay one the elements are TRUE.
!	It is called Logical NOT operator. Takes each element of the vector and gives the opposite logical value.

- The logical operator && and || considers only the first element of the vectors and gives a vector of single element as output.

# Example:

# Create a vector from 1 to 8

```
logical_vector <- c(1:8)
```

```
logical_vector > 6
```

Output: ## [1] FALSE FALSE FALSE FALSE FALSE FALSE TRUE TRUE

# Example:

```
logical_vector <- c(1:8)
```

```
logical_vector[(logical_vector>3) & (logical_vector<5)]
```

Output: ## [1] 4

- #See the “indexing vectors.R” in the folder.
- #Run the script to create the five vectors (columns shown in the table below).
- #Write and execute your script to answer the questions on the next page (see R script. )

<u>baby.name</u>	<u>baby.city</u>	<u>baby.ages</u>	<u>baby.weight</u>	<u>baby.eyecolor</u>
amy	macon	13	21	brown
brittany	athens	21	22	brown
carol	pink	32	41	green
donna	savannah	6	16	blue
erin	savannah	12	18	blue
fran	atlanta	11	19.4	grey
gigi	atlanta	18	26	brown
helen	athens	16	23	green
irene	macon	17	22	brown
jackie	macon	34	36	brown

#Exercise continued:

#Access the vectors to select elements to answer the below questions.

#1. What was the weight of the first baby?

#2. What were the ages of the first 5 babies?

#3. What were the names of the babies born with green eyes?

#4. What were the weights of either blue or grey eyed babies?

#5. Change the age of baby "irene" to 18.

#6. How many babies born in canton are in the data?

# Factors in R: Categorical & Continuous Variables

Factors are variables in R which take on a limited number of different values; such variables are often referred to as categorical variables.

In a dataset, we can distinguish two types of variables:

**categorical** and **continuous**

- In a categorical variable, the value is limited and usually based on a particular finite group. For example, a categorical variable can be countries, year, gender, occupation.
- A continuous variable, however, can take any values, from integer to decimal. For example, we can have the revenue, price of a share, etc..

# Categorical Variables

R stores categorical variables into a factor. Let's check the code below to convert a character variable into a factor variable. Characters are not supported in a machine learning algorithm, and the only way is to convert a string to an integer.

## Syntax

```
factor(x = character(), levels, labels = levels, ordered = is.ordered(x))
```

## Arguments:

- **x**: A vector of data. Need to be a string or integer, not decimal.
- **Levels**: A vector of possible values taken by x. This argument is optional. The default value is the unique list of items of the vector x.
- **Labels**: Add a label to the x data. For example, 1 can take the label `male` while 0, the label `female`.
- **ordered**: Determine if the levels should be ordered.



# Nominal Categorical Variable

A categorical variable has several values, but the order does not matter. For instance, male or female categorical variable do not have ordering.

```
# Create a color vector
```

```
color_vector <- c('turquoise', 'red', 'green', 'ivory', 'black', 'yellow')
```

```
# Convert the vector to factor
```

```
factor_color <- factor(color_vector)
```

```
factor_color
```

**Output:**

```
## [1] turquoise red green ivory black yellow
```

```
## Levels: black turquoise green red ivory yellow
```

# Ordinal Categorical Variable

Ordinal categorical variables do have a natural ordering. We can specify the order, from the lowest to the highest with `order = TRUE` and highest to lowest with `order = FALSE`.

**Example:** We can use `summary` to count the values for each factor.

```
# Create Ordinal categorical vector
day_vector <- c('evening', 'morning', 'afternoon', 'midday', 'midnight', 'evening')
# Convert `day_vector` to a factor with ordered level
factor_day <- factor(day_vector, order = TRUE, levels = c('morning', 'midday', 'afternoon', 'evening', 'midnight'))
# Print the new variable
factor_day
```

**Output:**

```
## [1] evening morning afternoon midday midnight evening
```

**Example:**

```
## Levels: morning < midday < afternoon < evening < midnight
# Append the line to above code
# Count the number of occurrence of each level summary(factor_day)
```

**Output:**

```
## morning midday afternoon evening midnight
## 1 1 1 2 1
```

# Accessing the elements of a factor

#Accessing the 4th element

```
> factor_day[4]
```

#Accessing the 2nd & 3rd element

```
> factor_day[c(2,3)]
```

#Accessing everything except 1st element

```
> factor_day[-1]
```

#Accessing using Logical Vector

```
> factor_day[c(TRUE, FALSE, TRUE, TRUE, FALSE, FALSE)]
```

## Lists in R:

- A list is a generic object consisting of an ordered collection of objects.
- Lists are heterogeneous data structures.
- A list is a one-dimensional data structure.
- A list can be a list of vectors, list of matrices, a list of characters and a list of functions and so on.
- Lists are different from atomic vectors because their elements can be of any type, including lists.
- You construct lists by using `list()` instead of `c()`:

```
x <- list(1:5, "d", c(FALSE, FALSE, TRUE), c(3.5, 7.2))  
str(x)
```

## # Example - Illustrate a List

# The first attributes is a numeric vector  
# containing the employee IDs which is  
# created using the 'c' command here  
empld = c(1, 2, 3, 4)

# The second attribute is the employee's name  
# which is created using this line of code here  
# which is the character vector  
empName = c("Ann", "Sanjan", "Ellison", "Evette")

# The third attribute is the number of employees  
# which is a single numeric variable.  
numberOfEmp = 4

# We can combine all these three different  
# data types into a list  
# containing the details of employees  
# which can be done using a list command  
empList = list(empld, empName, numberOfEmp)

print(empList)

# Matrix – What is a Matrix?

- A matrix is a 2-dimensional array that has m number of rows and n number of columns. In other words, matrix is a combination of two or more vectors with the same data type.

$$\begin{array}{ccc} \begin{bmatrix} 1 & 5 \\ -3 & 6 \end{bmatrix} & \begin{bmatrix} -1 & 5 \\ 4 & 7 \\ -8 & 2 \end{bmatrix} & \begin{bmatrix} 3 \\ 10 \\ -1 \end{bmatrix} & [-2 \ 4 \ 7 \ -6] \\ (2 \times 2) & (3 \times 2) & (3 \times 1) & (1 \times 4) \end{array}$$

- You can create a matrix with the function `matrix()`. This function takes three arguments:

`matrix(data, nrow, ncol, byrow = FALSE)`

Arguments:

- `data`: The collection of elements that R will arrange into the rows and columns of the matrix.
- `nrow`: Number of rows
- `ncol`: Number of columns
- `byrow`: When `byrow='TRUE'` the rows are filled from the left to the right. We use ``byrow = FALSE``, if we want the matrix to be filled down the columns i.e., the values are filled top to bottom.

## #Print dimension of the matrix with dim()

Defining names of columns and rows in a matrix

```
# Print dimension of the matrix with dim()  
dim(matrix_a)
```

In order to define rows and column names, you can create two vectors of different names, one for row and other for a column. Then, using the Dimnames attribute, you can name them appropriately:

```
rows = c("row1", "row2", "row3", "row4")    #Creating our character vector of row names  
  
cols = c("coln1", "coln2", "coln3")          #Creating our character vector of column names  
  
matrix_a <- matrix(c(1:12), nrow = 4, byrow = TRUE, dimnames = list(rows, cols) )  
#creating our matrix mat and assigning our vectors to dimnames
```

```
# Print matrix  
print(matrix_a)
```

# Add a Column to a Matrix with the cbind()

- You can add a column to a matrix with the cbind() command.
- cbind() means column binding. cbind() can concatenate as many matrix or columns as specified.

## Example:

```
# concatenate c(13:16) to the matrix_a  
matrix_a1 <- cbind(matrix_a, c(13:16))  
# Check the dimension dim(matrix_a1)
```

Output:

```
## [1] 4 4
```

## Example:

matrix\_a1

Output

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



**Syntax:**

We can also add more than one column.

(matrix name)<-matrix(c(n:m), byrow = FALSE, ncol = (number of desired columns)), where n and m are integers. Byrow=FALSE populates down the column.

**Example:**

Add a sequence of numbers to the matrix\_b matrix. The dimension of the new matrix will be 4x6 with number from 13 to 36.

```
matrix_b <- matrix(c(13:24))  
##      [,1] [,2] [,3]  
## [1,]  13  17  21  
## [2,]  14  18  22  
## [3,]  15  19  23  
## [4,]  16  20  24
```

**Example:**

```
matrix_c <- matrix(c(25:36), byrow = FALSE, ncol = 3)  
matrix_d <- cbind(matrix_b, matrix_c)  
dim(matrix_d)
```

Output:

```
## [1] 4 6
```

```
      [,1] [,2] [,3] [,4] [,5] [,6]  
[1,]  13  17  21  25  29  33  
[2,]  14  18  22  26  30  34  
[3,]  15  19  23  27  31  35  
[4,]  16  20  24  28  32  36
```

# Slice a Matrix: Accessing individual components

We can select elements one or many elements from a matrix by using the square brackets `[]`. This is where slicing comes into the picture.

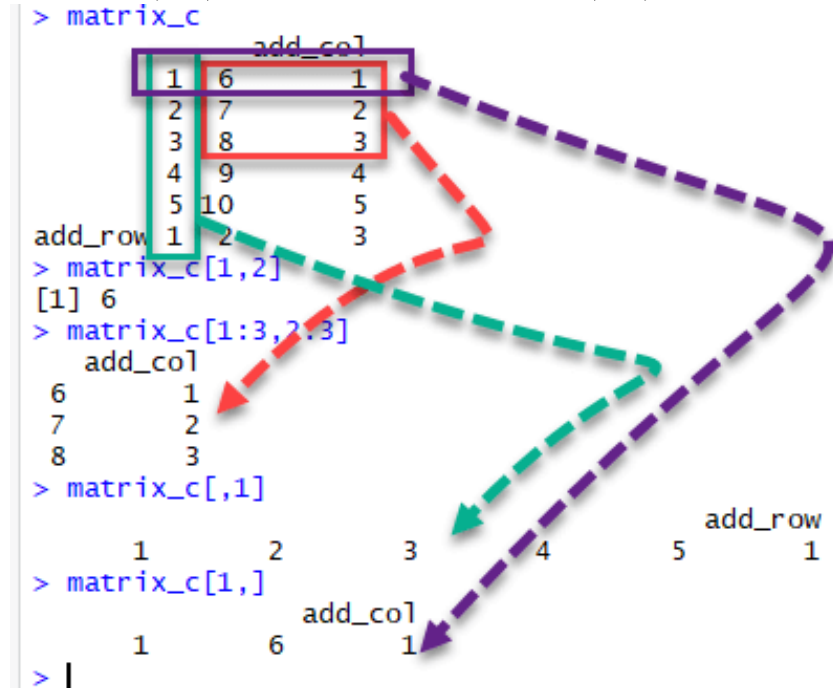
For example:

`matrix_c[1,2]` selects the element at the first row and second column.

`matrix_c[1:3,2:3]` results in a matrix with the data on the rows 1, 2, 3 and columns 2, 3,

`matrix_c[,1]` selects all elements of the first column.

`matrix_c[1,]` selects all elements of the first row.



### # Example:

```
> A = matrix(c(2, 4, 3, 1, 5, 7),  
             nrow=2,  
             ncol=3,  
             byrow = TRUE)
```

```
# create a matrix and add the data elements  
# number of rows  
# number of columns  
# fill matrix by rows
```

```
> A
```

```
> A[2, 3]
```

```
# element at 2nd row, 3rd column
```

```
> A[2, ]
```

```
# the 2nd row
```

```
> A[,c(1,3)]
```

```
# the 1st and 3rd columns
```

```
> B<- t(A)
```

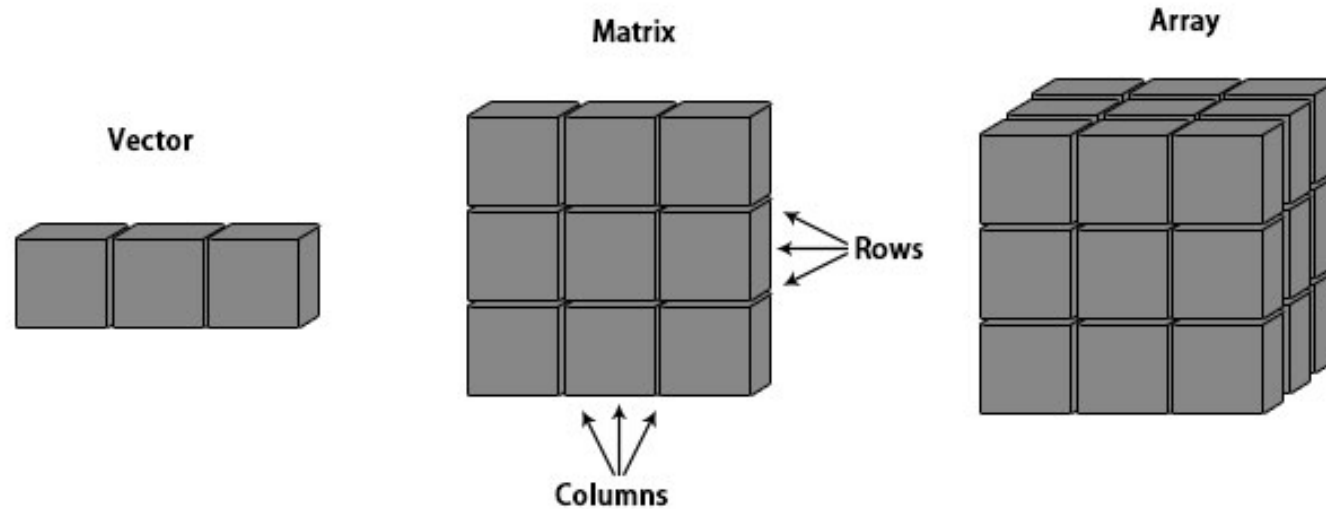
```
# Use the t function to transpose of B
```

```
> c(B)
```

```
# Use the c function to deconstruct a matrix
```

# Introduction to Arrays in R

- In arrays, data is stored in the form of matrices, rows, and columns.



## R Array Syntax

`Array_NAME <- array(data, dim = (row_Size, column_Size, matrices, dimnames)`

- **data** – Data is an input vector that is given to the array.
- **matrices** – Array in R consists of multi-dimensional matrices.
- **row\_Size** – row\_Size describes the number of row elements that an array can store.
- **column\_Size** – Number of column elements that can be stored in an array.
- **dimnames** – Used to change the default names of rows and columns to the user's preference.

## Arguments in Array

The array function in R can be written as:

`array(data = NA, dim = length(data), dimname = NULL)`

- **data** is a vector that provides data to fill the array.
- **dim** attribute provides maximum indices in each dimension
- **dimname** can be either NULL or can have a name for the array.

# How to Create an Array in R

# Example:

# Create two vectors of different lengths.

```
vector1 <- c(2,8,3)
```

```
vector2 <- c(10,14,17,12,11,15)
```

# Take these vectors as input to the array.

```
result <- array(c(vector1,vector2), dim = c(3,3,2))
```

```
print(result)
```

# Different Operations on Rows and Columns

## Naming Columns And Rows

```
# Example:
# Create two vectors of different lengths.
vector1 <- c(2,8,3)
vector2 <- c(10,14,17,12,11,15)
column.names <- c("COL1","COL2","COL3")
row.names <- c("ROW1","ROW2","ROW3")
matrix.names <- c("Matrix1","Matrix2")

# Take these vectors as input to the array.
result <- array(c(vector1,vector2),dim = c(3,3,2), dimnames = list(row.names,
column.names, matrix.names))

print(result)
```

# Print the third row of the first matrix of the array.

```
print(result[3,,1])
```

# Print the element in the 2nd row and 3rd column of the 2nd matrix.

```
print(result[2,3,2])
```

# Print the 2nd Matrix.

```
print(result[:,,2])
```



# Manipulating R Array Elements

# Example:

# Create two vectors of different lengths.

```
vector1 <- c(1,4,6)
```

```
vector2 <- c(2,3,5,6,7,9)
```

# Take these vectors as input to the array.

```
array1 <- array(c(vector1,vector2), dim = c(3,3,2))
```

# Create two vectors of different lengths.

```
vector3 <- c(6,4,1)
```

```
vector4 <- c(9,7,6,5,3,2)
```

```
array2 <- array(c(vector3,vector4), dim = c(3,3,2))
```

```
# create matrices from these arrays.
```

```
matrix1 <- array1[,2]
```

```
matrix2 <- array2[,2]
```

```
# Add the matrices.
```

```
result <- matrix1+matrix2
```

```
print(result)
```

# Calculations across R Array Elements

- `apply()` function for calculations in an array in R.
- Syntax `apply(x, margin, fun)`

Following is the description of the parameters used:

- `x` is an array.
- `margin` is the name of the dataset used.
- `fun` is the function to be applied to the elements of the array.
- For Example:
  - We use the `apply()` function below in different ways. To calculate the sum of the elements in the rows of an array across all the matrices.

# Example:

# We will create two vectors of different lengths.

```
vector1 <- c(1,4,6)
```

```
vector2 <- c(2,3,5,6,7,9)
```

# Now, we will take these vectors as input to the array.

```
new.array <- array(c(vector1,vector2),dim = c(3,3,2))
```

```
print(new.array)
```

# Use (apply) to calculate the sum of the rows across all the matrices.

```
result <- apply(new.array, c(1), sum)
```

```
print(result)
```

# R Data Frames:

- Data frames combine the behavior of lists and matrices to make a structure ideally suited for the needs of statistical data. The data frame is a list of vectors which are of equal length.
- A data-frame must have column names and every row should have a unique name.
  - `names()`, `colnames()`, and `rownames()`
- Each column must have the identical number of items.
- Each item in a single column must be of the same data type.
- Different columns may have different data types.
- A matrix contains only one type of data, while a data frame accepts different data types (numeric, character, factor, etc.). This makes it a 2-dimensional structure, so it shares properties of both the matrix and the list.

# How to Create a Data Frame

- We can create a data frame by passing the variable a,b,c,d into the `data.frame()` function. We can name the columns with `name()` and simply specify the name of the variables.
- `data.frame(df, stringsAsFactors = FALSE)`

## Arguments:

- `df`: It can be a matrix to convert as a data frame or a collection of variables to join
- `stringsAsFactors`: Convert string to character by default
- Note, versions prior to R 4.0.0, `stringsAsFactors` is set to default TRUE. See next slide

For versions of R prior to 4.0.0

```
df <- data.frame(x = 1:3, y = c("a", "b", "c"))  
str(df)
```

```
#> 'data.frame': 3 obs. of 2 variables:  
#> $ x: int 1 2 3  
#> $ y: Factor w/ 3 levels "a","b","c": 1 2 3
```

```
df <- data.frame(  
  x = 1:3,  
  y = c("a", "b", "c"),  
  stringsAsFactors = FALSE)  
str(df)
```

To import strings as characters, use in  
the script, stringsAsFactors=FALSE

```
#> 'data.frame': 3 obs. of 2 variables:  
#> $ x: int 1 2 3  
#> $ y: chr "a" "b" "c"
```

Because a data.frame is an S3 class, its type reflects the underlying vector used to build it: the list. To check if an object is a data frame, use `class()` or test explicitly with `is.data.frame()`:

```
typeof(df)
#> [1] "list"
class(df)
#> [1] "data.frame"
is.data.frame(df)
#> [1] TRUE
```

You can coerce an object to a data frame with `as.data.frame()`:

- A vector will create a one-column data frame.
- A list will create one column for each element; it's an error if they're not all the same length.
- A matrix will create a data frame with the same number of columns and rows as the matrix.



## Functions for viewing matrices and dataframes and returning information about them.

Function	Description
head(x), tail(x)	Print the first few rows (or last few rows)
View(x)	Open the entire object in a new window
nrow(x), ncol(x), dim(x)	Count the number of rows and columns
rownames(), colnames(), names()	Show the row (or column) names
str(x), summary(x)	Show the structure of the data frame (ie., dimensions and classes) and summary statistics

```
# Example script to illustrate data frame

# A vector which is a character vector
Name = c("Auriel", "Ray", "Asia")

# A vector which is a character vector
Type = c("O+", "B-", "A-")

# A vector which is a numeric vector
Age = c(36, 23, 62)

# To create data frame use data.frame command
# and then pass each of the vectors
# we have created as arguments
# to the function data.frame()

dfPatient = data.frame(Name, Type, Age)

print(dfPatient)
```

## Adding new columns through simple list-like assignments.

```
> dfPatient$State <- c("KY", "PA", "CA")
```

```
> dfPatient
```

<i>SN</i>	<i>Name</i>	<i>Type</i>	<i>Age</i>	<i>State</i>
1	Auriel	O+	36	KY
2	Ray	B-	23	PA
3	Asia	A-	62	CA

Data frame columns can be **deleted** by assigning NULL to it.

```
> dfPatient$State <- NULL
```

```
> dfPatient
```

<i>SN</i>	<i>Name</i>	<i>Type</i>	<i>Age</i>
1	Auriel	O+	36
2	Ray	B-	23
3	Asia	A-	62

Similarly, rows can be **deleted** through assignments.

```
> dfPatient<- dfPatient [-1,]
```

```
> dfPatient
```

<i>SN</i>	<i>Name</i>	<i>Type</i>	<i>Age</i>
1	Ray	B-	23
2	Asia	A-	62

# How to access components of a data frame?

Assessing like a list: We can also access the components of the list of data frames using indices. To access the top-level components of a list of data frames we have to use a double slicing operator "[[ ]]" which is two square brackets and if we want to access the lower or inner level components of a list, we have to use another square bracket "[ ]" along with the double slicing operator "[[ ]]"..

Using the list, empList, example created earlier:

```
> empList
```

```
[[1]]
```

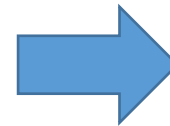
```
[1] 1 2 3 4
```

```
[[2]]
```

```
[1] "An" "Be" "Cm" "Di"
```

```
[[3]]
```

```
[1] 4
```



```
> empList[[2]]
```

```
[1] "An" "Be" "Cm" "Di"
```

```
> empList[[2]][2]
```

```
[1] "Be"
```

# How to access components of a data frame?

Assessing like a matrix: Data frames can be accessed like a matrix by providing index for row and column. We can use either `[`, `[[` or `$` operator to access columns of data frame.

```
#Extract the first row
```

```
> dfPatient[,1]
```

```
Name
```

```
1 Auriel
```

```
2 Ray
```

```
3 Asia
```

```
#Extract the second row
```

```
> dfPatient[2,]
```

```
  Name Type Age
```

```
2 Ray  B-  23
```

```
#Extract first two rows and then all columns
```

```
> dfPatient[1:2,]
```

It's a common mistake to try and create a data frame by `cbind()`ing vectors together. This doesn't work because `cbind()` will create a matrix unless one of the arguments is already a data frame. Instead use `data.frame()` directly:

```
bad <- data.frame(cbind(a = 1:2, b = c("a", "b")))
```

```
str(bad)
```

```
'data.frame': 2 obs. of 2 variables:
```

```
$ a: chr "1" "2"
```

```
$ b: chr "a" "b"
```

```
good <- data.frame(a = 1:2, b = c("a", "b"))
```

```
str(good)
```

```
'data.frame': 2 obs. of 2 variables:
```

```
$ a: int 1 2
```

```
$ b: chr "a" "b"
```

# Merging: Combine data frames using cbind() and rbind()

```
cbind(df, data.frame(z = 3:1))  
#>      x y z  
#> 1 1 a 3  
#> 2 2 b 2  
#> 3 3 c 1  
  
rbind(df, data.frame(x = 10, y = "z"))  
#>      x y  
#> 1 1 a  
#> 2 2 b  
#> 3 3 c  
#> 4 10 z
```

- When combining column-wise, the number of rows must match, but row names are ignored.
- When combining row-wise, both the number and names of columns must match.
- Use `plyr::rbind.fill()` to combine data frames that don't have the same columns.

Example:

Let's create a factor data frame.

```
# Create gender vector
```

```
gender_vector <- c("Male", "Female", "Female", "Male", "Male")
```

```
class(gender_vector)
```

```
# Convert gender_vector to a factor
```

```
factor_gender_vector <- factor(gender_vector)
```

```
class(factor_gender_vector)
```

Output:

```
## [1] "character"
```

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



# Homework Exercises

---

See matrix and dataframes.R for examples

# Examine a Data Frame in R with 7 Basic Functions

- [dim\(\)](#): shows the dimensions of the data frame by row and column
- [str\(\)](#): shows the structure of the data frame
- [summary\(\)](#): provides summary statistics on the columns of the data frame
- [colnames\(\)](#): shows the name of each column in the data frame
- [head\(\)](#): shows the first 6 rows of the data frame
- [tail\(\)](#): shows the last 6 rows of the data frame
- [View\(\)](#): shows a spreadsheet-like display of the entire data frame

# Fixing a broken data frame

Some useful functions:

- `?read.csv`
- `head()`
- `str()`
- `class()`
- `unique()`
- `levels()`
- `which()`
- `droplevels()`

Note: For these functions you have to put the name of the data object in the parentheses (i.e., `head(CO2)`).

Also remember that you can use “?” to look up help for a function (i.e. `?str`).

# Knowledge Check:

1. What are the three properties of a vector, other than its contents?

The three properties of a vector are type, length, and attributes.

2a. What are the four common types of atomic vectors?

1. Logical
2. Integer
3. Double (sometimes called numeric)
4. Character

2b. What are the two rare types?

1. Complex
2. Raw

# Knowledge Check (cont.)

3a. How is a list different from an atomic vector?

The elements of a list can be any type (even a list); the elements of an atomic vector are all of the same type.

3b. How is a matrix different from a data frame?

Similarly, every element of a matrix must be the same type; in a data frame, the different columns can have different types

# Decision making

R provides the following types of decision-making statements.

Statement	Description
<i>if</i> statement	An <b>if</b> statement consists of a Boolean expression followed by one or more statements.
<i>if...else</i> statement	An if statement can be followed by an optional else statement, which executes when the Boolean expression is false.
switch statement	A switch statement allows a variable to be tested for equality against a list of values.

# Decision making

- **The if...else if...else ladder**
  - The basic syntax

```
if(boolean_expression 1) {  
    // Executes when the boolean expression 1 is true.  
}else if( boolean_expression 2) {  
    // Executes when the boolean expression 2 is true.  
}else if( boolean_expression 3) {  
    // Executes when the boolean expression 3 is true.  
}else {  
    // executes when none of the above condition is true.  
}
```

# Decision making

## Switch Statement

- Select One of a List of Alternatives
- switch evaluates EXPR and accordingly chooses one of the further arguments (in ...).
- The basic syntax

**switch(expression, case1, case2, case3....)**

- switch works in two distinct ways depending whether the first argument (EXPR) evaluates to a character string or a number
- If the value of expression is not a character string it is coerced to integer.
- If the value of the integer is between 1 and nargs()-1 (The max number of arguments) then the corresponding element of case condition is evaluated, and the result returned.
- If expression evaluates to a character string, then that string is matched (exactly) to the names of the elements.
- If there is more than one match, the first matching element is returned.
- In the case of no match, if there is an unnamed element of ... its value is returned.

```
x <- switch(5,"morning","midmorning","noon","midday","evening","night")
```

```
x
```

Output: evening



# Loops

R programming language provides the following kinds of loop statements

Statement	Description
repeat loop	Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
while loop	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
for loop	Like a while statement, except that it tests the condition at the end of the loop body.

## Control Statement & Description

break statement	Terminates the loop statement and transfers execution to the statement immediately following the loop.
-----------------	--

Next statement	The next statement simulates the behavior of R switch.
----------------	--

# Loops

## Repeat Loop

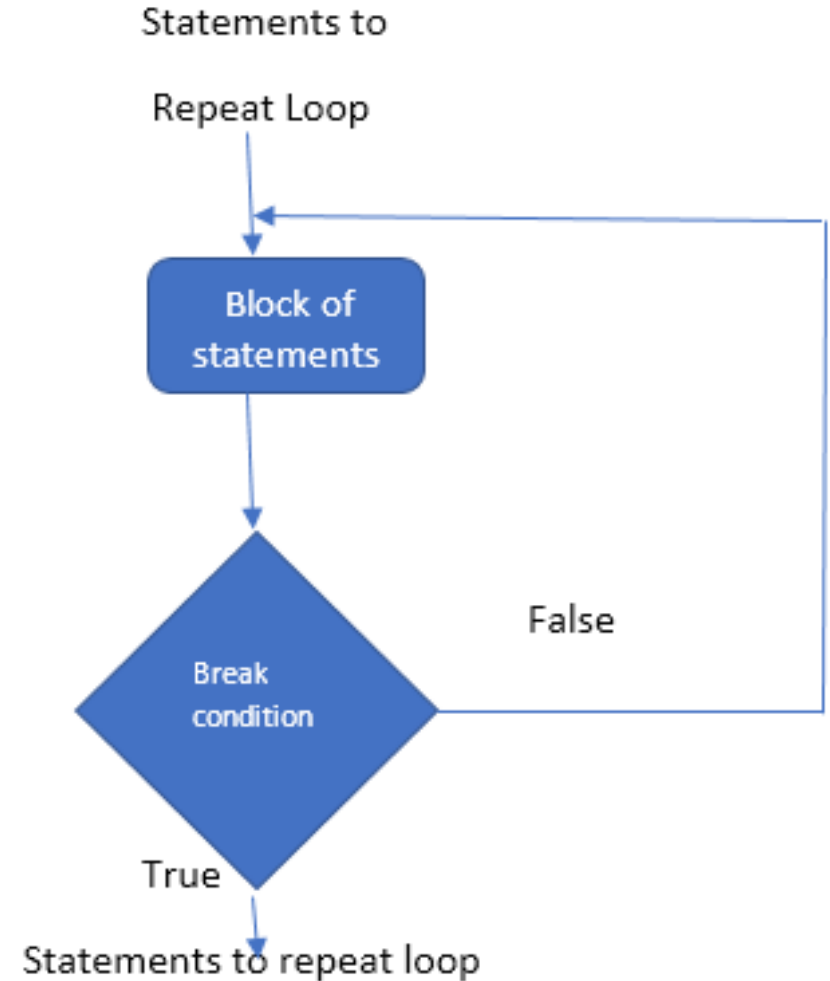
The Repeat loop executes the same code again and again until a stop condition is met.

The basic syntax

```
repeat {  
    commands  
    if(condition){  
        break  
    }  
}
```

Example;

```
s <- 1  
repeat  
{  
    s <- s+2;  
    print (s);  
    if (s>9)  
        break;  
}
```



# Loops

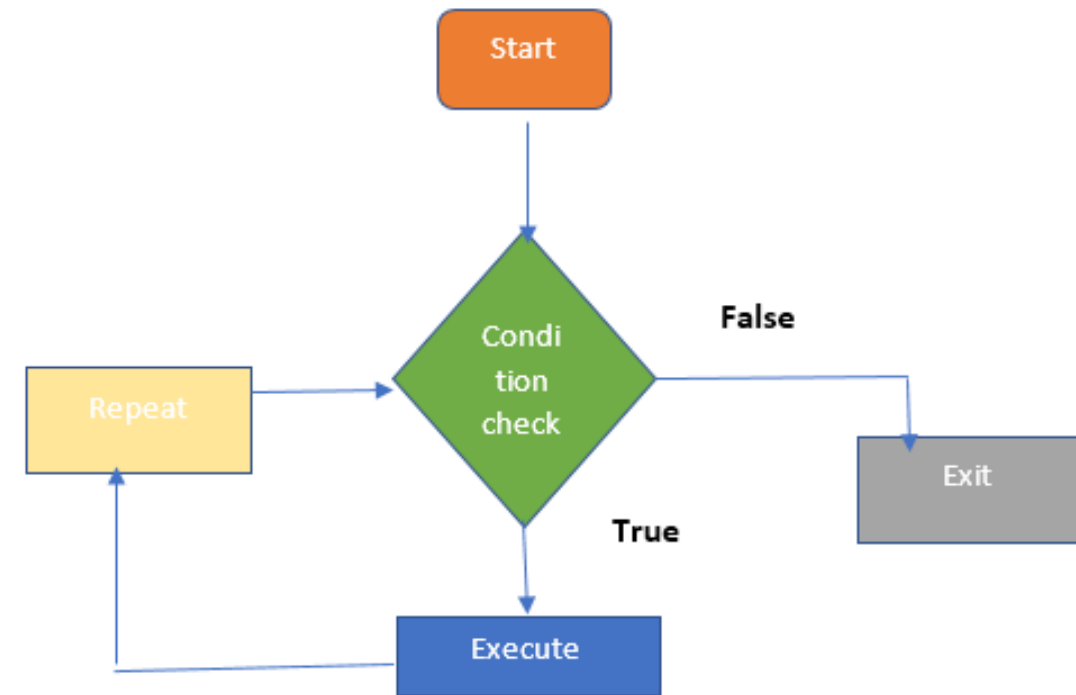
## While Loop

The While loop executes the same code again and again until a stop condition is met.

The basic syntax

```
while (test_expression) {  
    statement  
}
```

```
Example;  
v <- c("Wide","while loop")  
cnt <- 2  
while (cnt < 7){  
    print(v)  
    cnt = cnt + 1  
}
```



# Loops

## For Loop

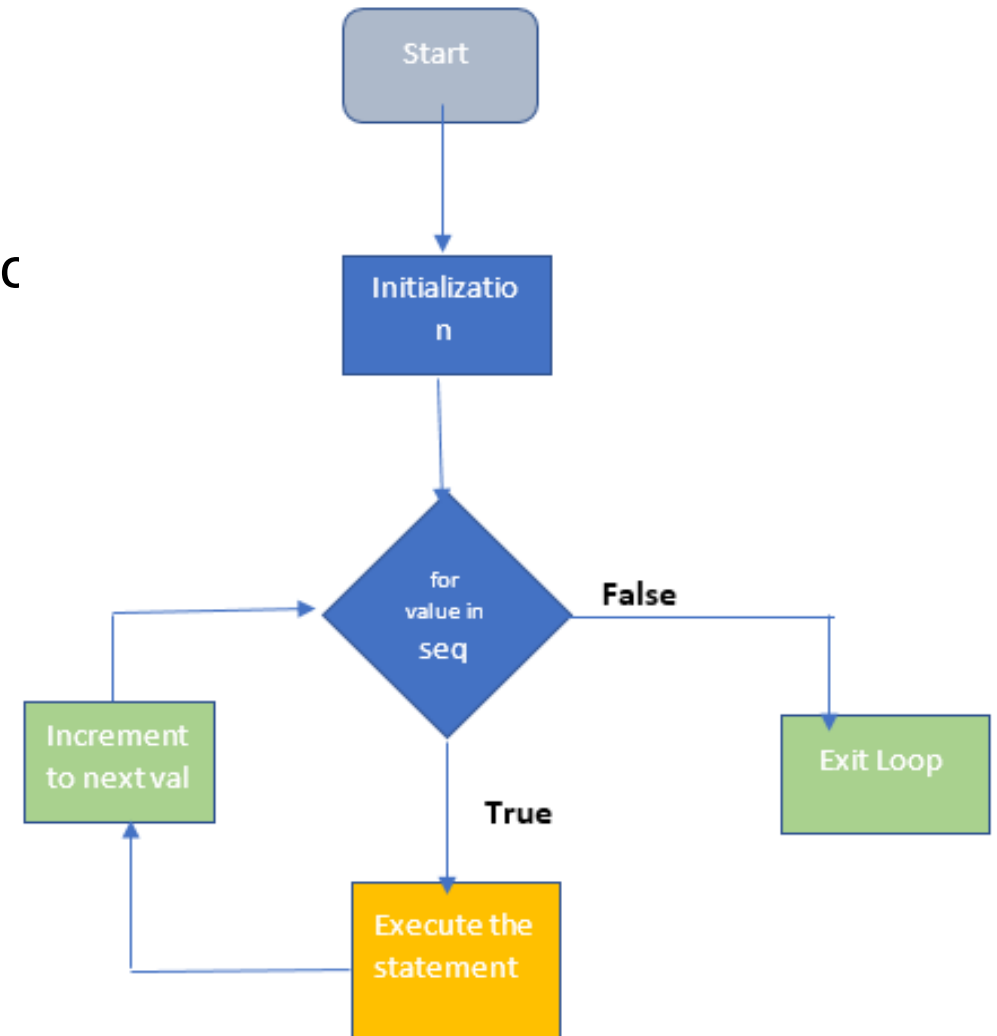
A for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

The basic syntax

```
for (value in vector) {  
    statements  
}
```

Example;

```
v <- LETTERS[1:4]  
for ( i in v) {  
    print(i)  
}
```



# Using packages

**1**

```
install.packages("readr")
```

Downloads files to computer

**1 x per computer**

**2**

```
library("readr")
```

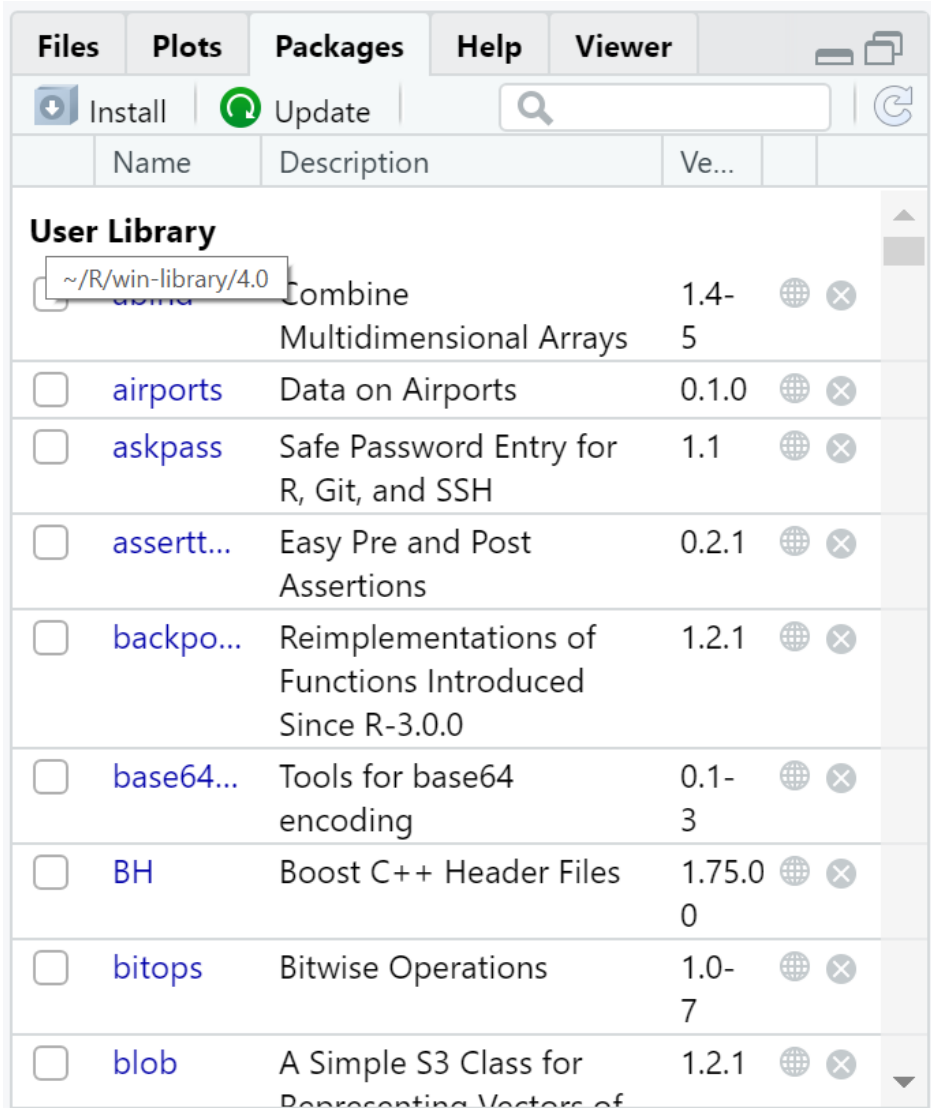
Loads package

**1 x per R Session**

# Vignettes

```
cointoss/  
├── .Rbuildignore  
├── cointoss.Rproj  
├── DESCRIPTION  
├── NAMESPACE  
├── R/  
├── man/  
├── tests/ Vignette files  
└── vignettes/  
    └── introduction.Rmd
```

# Packages



- **Preloaded packages:** R has a decent functionality before installing new packages. However, R's {base} packages are often outdated, or cumbersome to use.
- **Installing packages:** installing a package saves the package in our User Library for future use. A package needs to be installed only once.
- **Loading packages:** If you want to use the functions in a specific package, you must tell R to load it for you. You only need to load a package once per session. You do that with the `library(package.name)` function.
- **Distinguishing between functions:** some packages have functions whose names overlap. You can recognize which package a function belongs to by checking the {package.name} next to it during autocompletion.
- **Removing packages:** you can uninstall a package with the `remove.packages("package.name")` function.

# Search Path – list of packages currently loaded into the memory



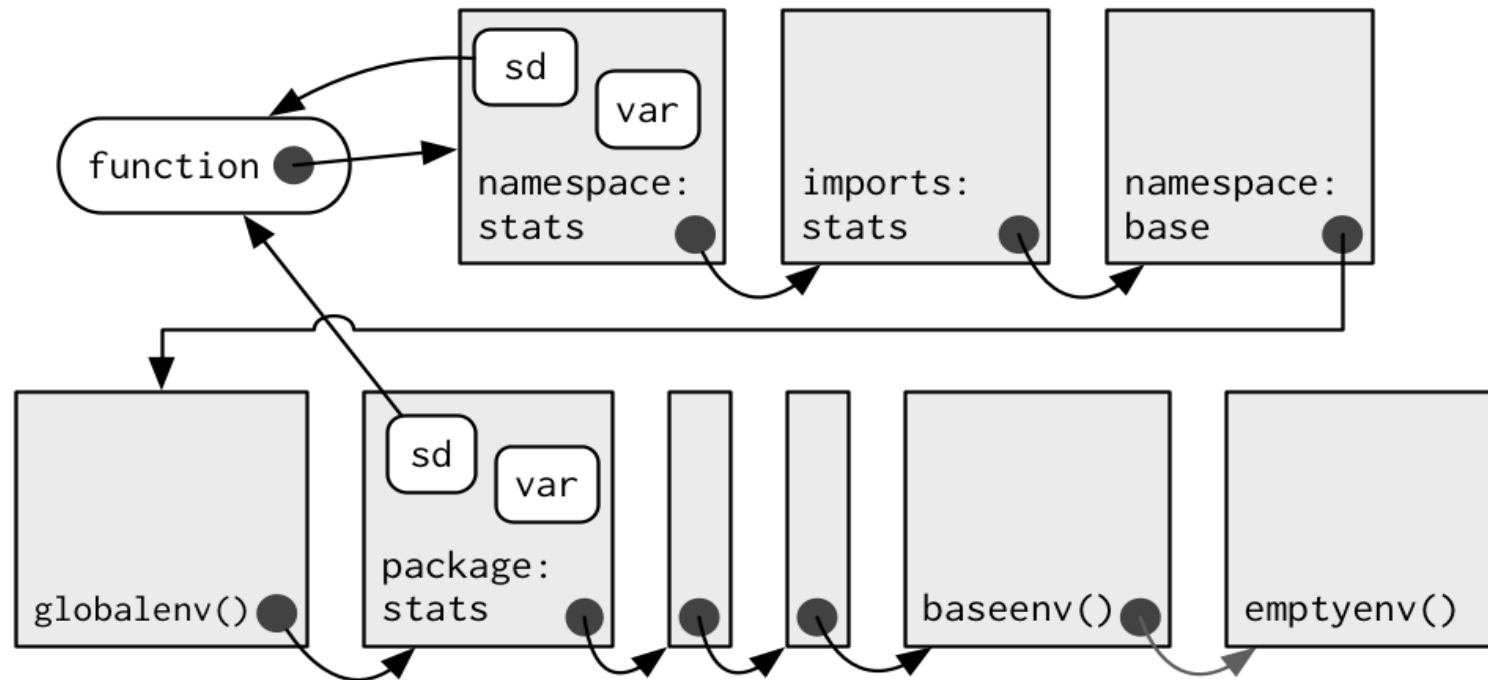
```
> search( )
```

```
[1] ".GlobalEnv"      "tools:rstudio"   "package:stats"
```

```
[4] "package:graphics" "package:grDevices" "package:utils"
```

```
[7] "package:datasets" "package:methods"  "Autoloads"
```

```
[10] "package:base"
```





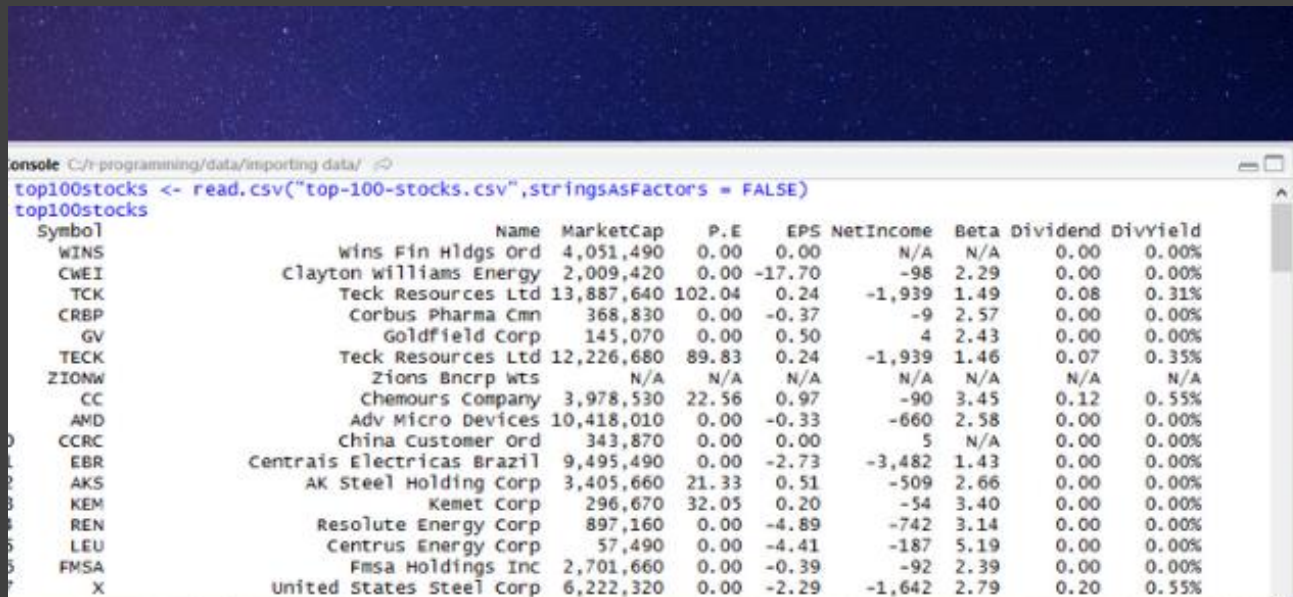


# "package:utils"

This package is loaded by default when you load R.

- `read.table()`: Main function. Reads a file in table format and creates a data frame from it. It offers many arguments to classify the incoming data.
- `read.csv()`: Wrapper function for `read.table()`. Used to read comma-separated (CSV) files.
- `read.delim()`: Wrapper Function used to read tab-separated files. `read.delim()` is used if the numbers in your file use periods(.) as decimals.
- `read.csv2()` : `read.csv()` and `read.csv2()` are identical. The only difference is that they are set up depending on whether you use periods or commas as decimal points in numbers.
- `read.delim2()` : `read.delim2` is used when the numbers in your file use commas(,) as decimals.

# Importing csv file with read.csv function



```
console C:/r-programming/data/importing data/
top100stocks <- read.csv("top-100-stocks.csv",stringsAsFactors = FALSE)
top100stocks
```

Symbol	Name	MarketCap	P.E	EPS	NetIncome	Beta	Dividend	DivYield
WINS	Wins Fin Hldgs Ord	4,051,490	0.00	0.00	N/A	N/A	0.00	0.00%
CWEI	Clayton Williams Energy	2,009,420	0.00	-17.70	-98	2.29	0.00	0.00%
TCK	Teck Resources Ltd	13,887,640	102.04	0.24	-1,939	1.49	0.08	0.31%
CRBP	Corbus Pharma Cmn	368,830	0.00	-0.37	-9	2.57	0.00	0.00%
GV	Goldfield Corp	145,070	0.00	0.50	4	2.43	0.00	0.00%
TECK	Teck Resources Ltd	12,226,680	89.83	0.24	-1,939	1.46	0.07	0.35%
ZIONW	Zions Bncrp Wts	N/A	N/A	N/A	N/A	N/A	N/A	N/A
CC	Chemours Company	3,978,530	22.56	0.97	-90	3.45	0.12	0.55%
AMD	Adv Micro Devices	10,418,010	0.00	-0.33	-660	2.58	0.00	0.00%
CCRC	China Customer Ord	343,870	0.00	0.00	5	N/A	0.00	0.00%
EBR	Centrais Electricas Brazil	9,495,490	0.00	-2.73	-3,482	1.43	0.00	0.00%
AKS	AK Steel Holding Corp	3,405,660	21.33	0.51	-509	2.66	0.00	0.00%
KEM	Kemet Corp	296,670	32.05	0.20	-54	3.40	0.00	0.00%
REN	Resolute Energy Corp	897,160	0.00	-4.89	-742	3.14	0.00	0.00%
LEU	Centrus Energy Corp	57,490	0.00	-4.41	-187	5.19	0.00	0.00%
FMSA	Fmsa Holdings Inc	2,701,660	0.00	-0.39	-92	2.39	0.00	0.00%
X	United States Steel Corp	6,222,320	0.00	-2.29	-1,642	2.79	0.20	0.55%

read.csv in R

- The function `read.csv()` is used to import data from a csv file.
- This function can take many arguments, but the most important is *file* which is the name of file to be read.
- This function reads the data as a data frame. If the values are separated by a comma use `read.csv()` and if the values are separated by ; (a semi-colon) use `read.csv2()` function. Otherwise, there is no difference between these two functions.

```
> data <- read.csv("testfile.csv")
```

- You don't exactly know the file location or not sure about the name of the file. Use *file.choose* option in read.csv function. This will open a file dialog box to select the file you want to open in R.

```
> data <- read.csv(file.choose())
```

- To read file from that location R code will be

```
> data <- read.csv("http://(web location)/HealthRisks.csv")
```

- use the file variable for storing url and then using it to import file in R

```
> file <- " http://(web location)/HealthRisks.csv "
```

```
> data <- read.csv(file)
```



After importing data in RStudio you can check and see it with some common functions.

- 
1. `View()`: This function will show you the values of csv file in a table format.
  2. `nrow()`: This function returns the total number of rows in your data frame.
  3. `ncol()`: Returns the total number of columns in your data frame.
  4. `colnames()`: This function returns the column headers or column names.
  5. `str()`: Returns the structure of your data frame. Column names with data types and factors.

## Importing your data

R allows for the import of different data formats using specific packages that can make your job easier.

Package	Purpose
readr	.csv files
readxl	excel files (.xlsx or .xls)
haven	SAS, STATA and SPSS data files
RMySQL or RpostgreSQL	Connect to databases, access, and manipulate via DBI
rvest	Web scraping
xml2	XML files
httr	Web APIs

# Downloading files

## .sas7bdat

- `library("haven")`
- `SASdata <- read.sas("example.sas7bdat")`

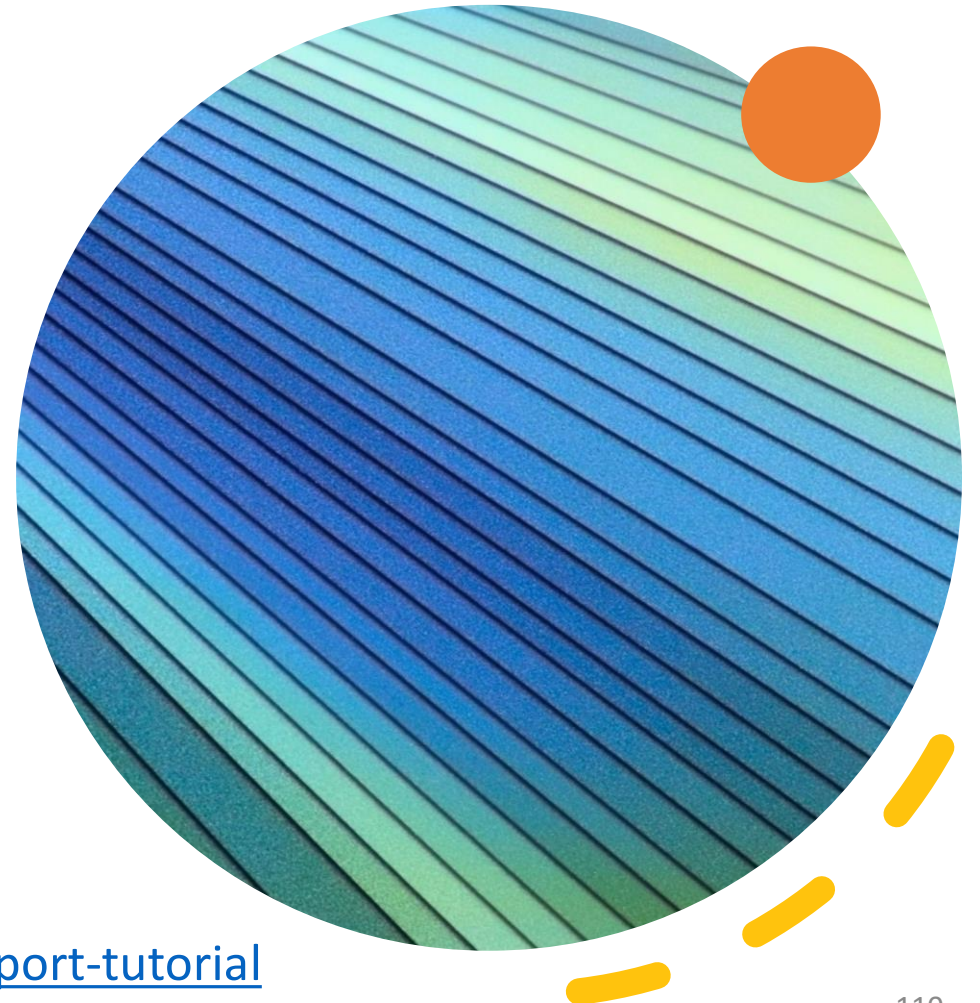
## .xlsx or .xls:

- `library("readxl")`
- `xlsxdata <- read_excel("sheet1.xlsx")`. You can add in a sheet argument.

## From the internet:

- `download.file("http://www..xls",  
"C:/test/students.xls", method="auto",  
quiet=FALSE, mode = "wb", cacheOK =  
TRUE)`

<https://www.datacamp.com/community/tutorials/r-data-import-tutorial>





# Additional resources for R programming basics

- Easy R Programming Basics: <http://www.sthda.com/english/wiki/easy-r-programming-basics>
- An Introduction to R: <https://cran.r-project.org/doc/manuals/R-intro.pdf>
- Packages available in CRAN: <https://cran.r-project.org/web/packages/>
- R for SAS and SPSS Users: <https://docs.google.com/viewer?a=v&pid=sites&srcid=ZGVmYXVsdGRvbWFpbnxyNHN0YXRpc3RpY3N8Z3g6MWNmZDQ4ZjcwODY2Y2I0Yw>



Any  
Questions



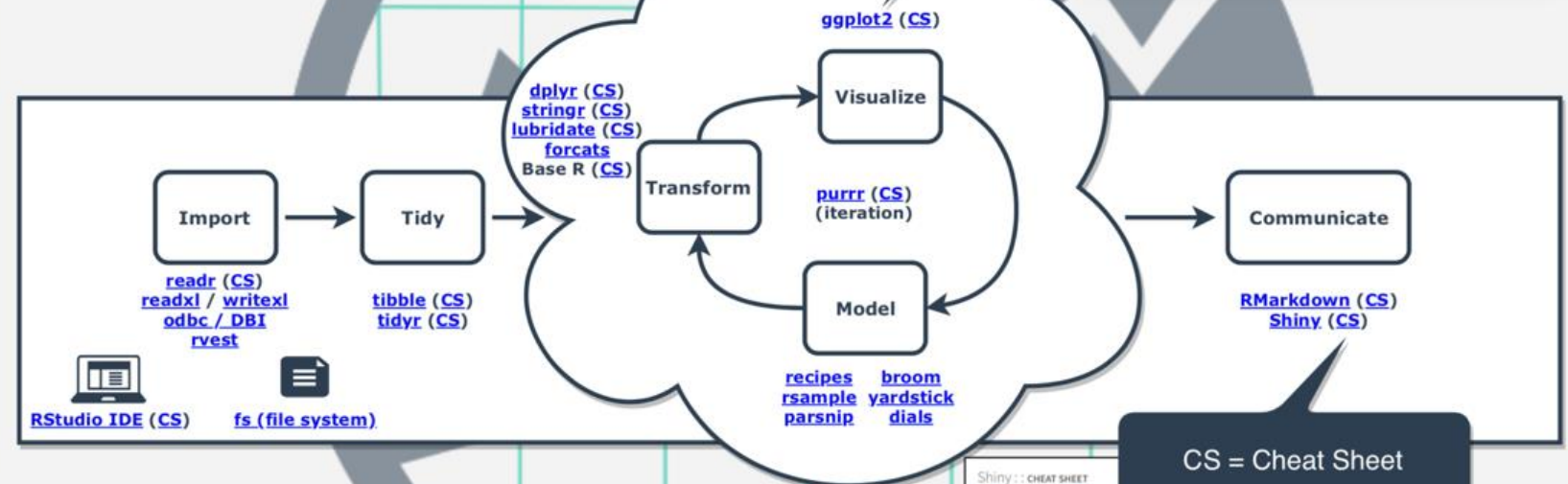
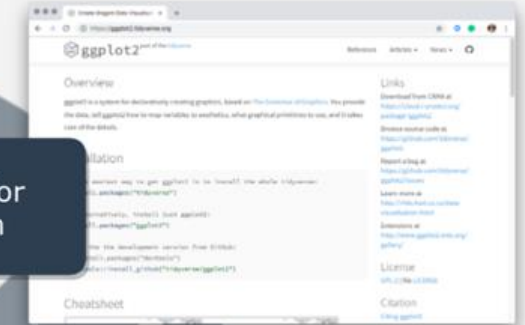
# NEXT CLASS

## Data Science with R Workflow

The Data Science With R Workflow is available in the book: [R For Data Science](#). If you want to learn R and this workflow *for business analysis*, take the [R For Business Analysis \(DS4B 101-R\) course](#) through Business Science University.



Click the links for Documentation



## Important Resources

- R For Data Science Book: <http://r4ds.had.co.nz/>
- Rmarkdown Book: <https://bookdown.org/yihui/rmarkdown/>
- Data Visualization Book: <https://rkabacoff.github.io/datavis/>
- More Cheatsheets: <https://www.rstudio.com/resources/cheatsheets/>
- tidyverse packages: <https://www.tidyverse.org/>
- Connecting to databases: <https://db.rstudio.com/>
- RMarkdown website: <https://rmarkdown.rstudio.com/>
- Shiny web applications website: <http://shiny.rstudio.com/>
- Jenny Bryan's purrr tutorial: <https://jennybryan.org/>



"Data Science Education for the Enterprise"



Business Science University  
[university.business-science.io](http://university.business-science.io)



Yvonne Phillips  
[yphillips@beverasolutions.com](mailto:yphillips@beverasolutions.com)