ATAL Sponsored online five-Days Faculty Development Programme
on
“Data Analytics using Python/R programming”
Organized by
Department of CSE, NIT Puducherry

Introduction to R and RStudio - Nuts and Bolts

Presented by: Siddharth R
Department of Computer Science and Engineering
Amrita School of Engineering, Amrita Vishwa Vidyapeetham,
Chennai Campus

August 26, 2021

Agenda

Session 1 - Introduction

Operators

Objects

Conditional and Looping Statement

Functions

Importing and Loading Your Dataset

Scripts & Packages

Section 1

Session 1 - Introduction

Quote of the day: Tell me and I forget, Teach me and I may remember, Involve me and I learn by Benjamin Franklin

This session treats **R** purely as a programming language. In this session you will get involved in how to

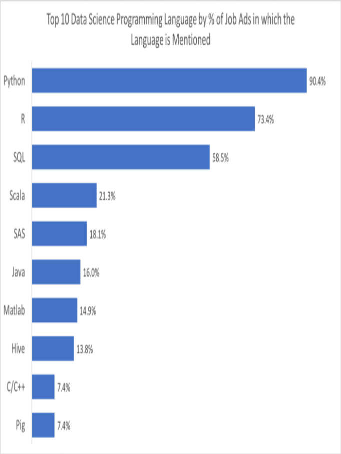
- Use the R and RStudio interfaces
- Run R commands
- Create R objects
- Write your own R functions and scripts
- Load and use R packages
- Generate random samples

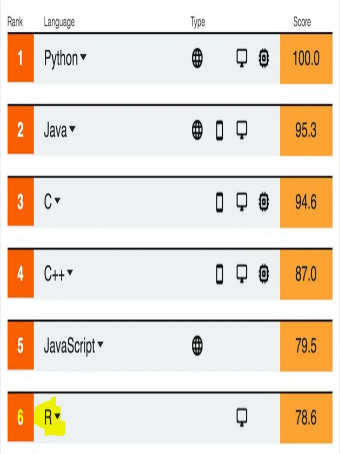
Software Requirements : R and RStudio (Both are free and easy to download)

History of R language

- R is a programming language and software environment for statistical computing
- R was created by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand, and is currently developed by the R Development Core Team
- R can be extended (easily) via packages. There are several packages supplied with the R distribution and many more are available through the CRAN family of Internet sites covering a very wide range of modern statistics.
- R has a large, coherent, integrated collection of intermediate tools for data analysis
- R has a well-developed, simple and effective programming language which includes conditionals, loops, user-defined recursive functions and input and output facilities.

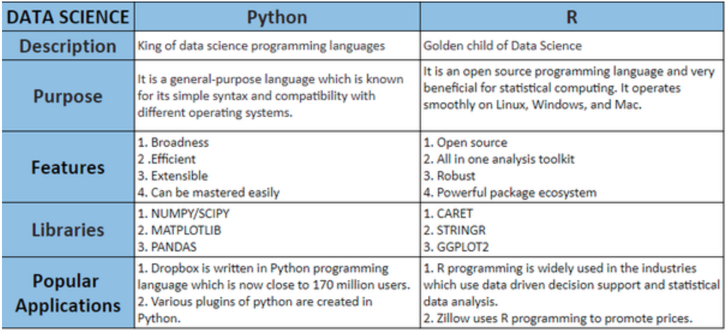
Some Statistics about R





Rank	Language	Type	Score
1	Python ▾	🌐 🖥 ⚙	100.0
2	Java ▾	🌐 📱 🖥	95.3
3	C ▾	📱 🖥 ⚙	94.6
4	C++ ▾	📱 🖥 ⚙	87.0
5	JavaScript ▾	🌐	79.5
6	R ▾	🖥	78.6

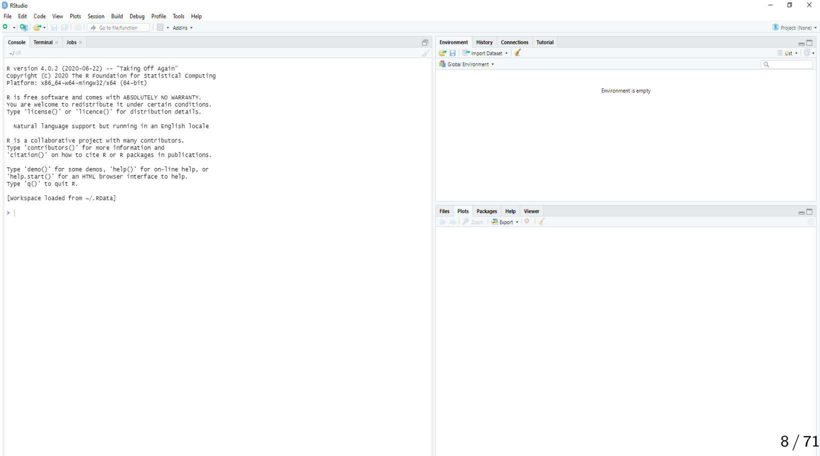
Python vs R



DATA SCIENCE	Python	R
Description	King of data science programming languages	Golden child of Data Science
Purpose	It is a general-purpose language which is known for its simple syntax and compatibility with different operating systems.	It is an open source programming language and very beneficial for statistical computing. It operates smoothly on Linux, Windows, and Mac.
Features	1. Broadness 2 .Efficient 3. Extensible 4. Can be mastered easily	1. Open source 2. All in one analysis toolkit 3. Robust 4. Powerful package ecosystem
Libraries	1. NUMPY/SCIPY 2. MATPLOTLIB 3. PANDAS	1. CARET 2. STRINGR 3. GGPLOT2
Popular Applications	1. Dropbox is written in Python programming language which is now close to 170 million users. 2. Various plugins of python are created in Python.	1. R programming is widely used in the industries which use data driven decision support and statistical data analysis. 2. Zillow uses R programming to promote prices.

R and RStudio

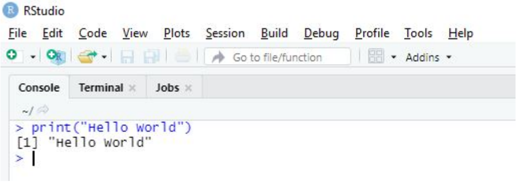
- RStudio gives you a way to talk to your computer. R gives you a language to speak in.



```
R version 4.0.2 (2020-06-22) -- "Taking Off Again"
Copyright (C) 2020 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (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]

> |
```

Write your first code

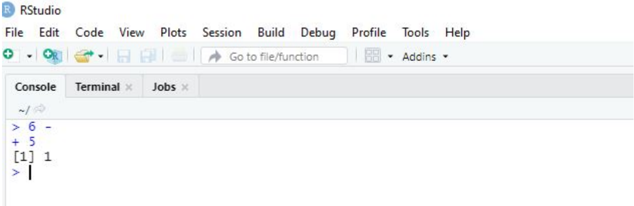
- You type R code into the bottom line of the RStudio console pane and then click Enter to run it
- The code you type is called a command, because it will command your computer to do something for you
- When you type a command at the prompt and hit Enter, your computer executes the command and shows you the results
- Then RStudio displays a fresh prompt for your next command.


RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Console
Terminal
Jobs
~/

```
> print("Hello World")
[1] "Hello World"
> 
```

Compiling R Code

- In languages like C and Java, you have to compile your human-readable code into machine-readable code (often 1s and 0s) before you can run it.
- R is a dynamic programming language, which means R automatically interprets your code as you run it
- If you type an incomplete command and press Enter, R will display a + prompt, which means it is waiting for you to type the rest of your command
- Either finish the command or hit Escape to start over


RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Console
Terminal
Jobs
~/

```
> 6 -
+ 5
[1] 1
> 
```

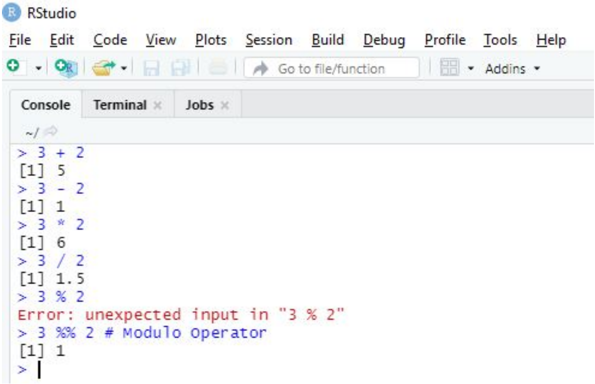
Agenda

- Session 1 - Introduction
- **Operators**
- Objects
- Conditional and Looping Statement
- Functions
- Importing and Loading Your Dataset
- Scripts & Packages

Section 2

Operators

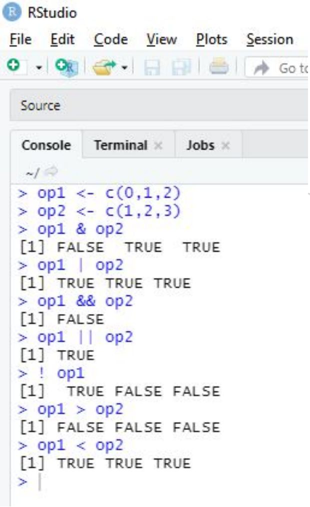
Basic Arithmetic operator in R



RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Console
Terminal
Jobs
~/

```
> 3 + 2
[1] 5
> 3 - 2
[1] 1
> 3 * 2
[1] 6
> 3 / 2
[1] 1.5
> 3 % 2
Error: unexpected input in "3 % 2"
> 3 %% 2 # Modulo Operator
[1] 1
> 
```

Logical Operator



```
> op1 <- c(0,1,2)
> op2 <- c(1,2,3)
> op1 & op2
[1] FALSE  TRUE  TRUE
> op1 | op2
[1] TRUE TRUE TRUE
> op1 && op2
[1] FALSE
> op1 || op2
[1] TRUE
> ! op1
[1]  TRUE FALSE FALSE
> op1 > op2
[1] FALSE FALSE FALSE
> op1 < op2
[1] TRUE TRUE TRUE
>
```

Agenda

Session 1 - Introduction

Operators

Objects

Conditional and Looping Statement

Functions

Importing and Loading Your Dataset

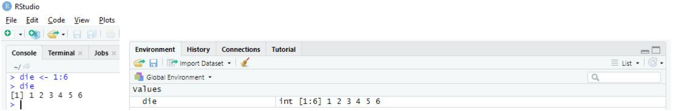
Scripts & Packages

Section 3

Objects

R Objects

- R lets you save data by storing it inside an R object - Just a name that you can use to call up stored data
- You can save data into an object like a or b, wherever R encounters the object, it will replace it with the data
- To create an R object, choose a name and then use the less-than symbol, <, followed by a minus sign, −, to save data into it.
- You can use your object in new R commands, too.

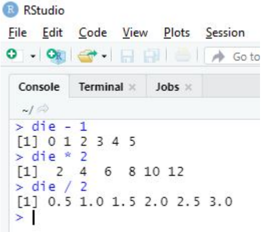


```
> die <- 1:6
> die
[1] 1 2 3 4 5 6
> |
```

```
Values
  die                 int [1:6] 1 2 3 4 5 6
```

R Objects (Contd...)

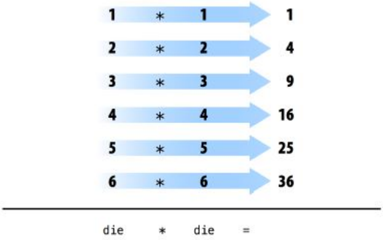
- Object name cannot use some special symbols, like ^, !, $, @, +, -, /, or *
- Object name is **case sensitive**
- *ls()* - object names you have already used / by examining RStudio's environment pane



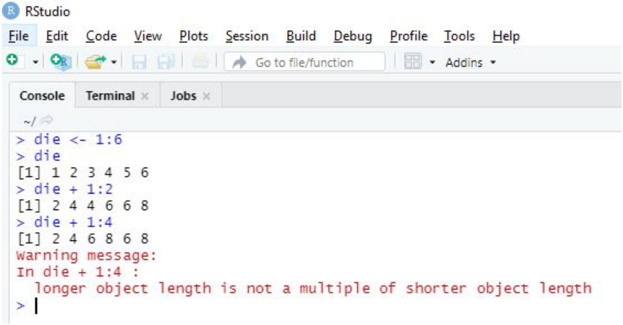
```
> die - 1
[1] 0 1 2 3 4 5
> die * 2
[1]  2  4  6  8 10 12
> die / 2
[1] 0.5 1.0 1.5 2.0 2.5 3.0
>
```

R Element-wise execution

- When you use two or more vectors in an operation, R will line up the vectors and perform a sequence of individual operations
- Element-wise operations are a very useful feature in R because they manipulate groups of values in an orderly way



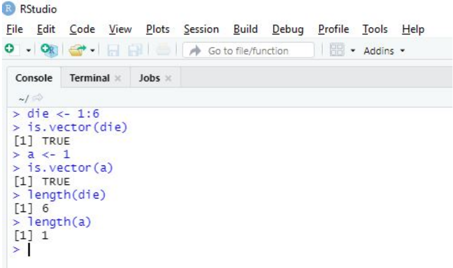
Element wise execution


RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Console Terminal Jobs
~/

```
> die <- 1:6
> die
[1] 1 2 3 4 5 6
> die + 1:2
[1] 2 4 4 6 6 8
> die + 1:4
[1] 2 4 6 8 6 8
Warning message:
In die + 1:4 :
  longer object length is not a multiple of shorter object length
> 
```

Atomic vector

- An atomic vector is just a simple vector of data - one-dimensional, one data type
- You can also make an atomic vector with just one value. R saves single values as an atomic vector of length 1
- length returns the length of an atomic vector
- is.vector tests whether an object is an atomic vector.

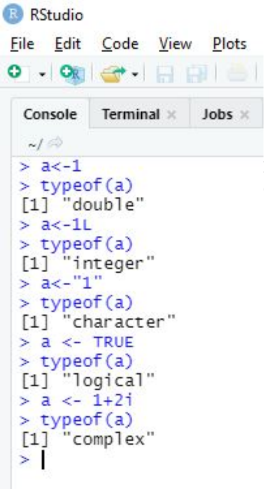

RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Console
Terminal
Jobs
~/

```
> die <- 1:6
> is.vector(die)
[1] TRUE
> a <- 1
> is.vector(a)
[1] TRUE
> length(die)
[1] 6
> length(a)
[1] 1
> 
```

Types of vector based on data type

- **double** - A double vector stores regular numbers. The numbers can be positive or negative, large or small, and have digits to the right of the decimal place or not. In general, R will save any number that you type in R as a double.
- **Integer**- store integers, numbers that can be written without a decimal component. You can specifically create an integer in R by typing a number followed by an uppercase **L**.
- **character**- stores small pieces of text. You can create a character vector in R by typing a character or string of characters surrounded by quotes
- **Logical**- store TRUEs and FALSEs, R's form of Boolean data
- **Complex** - store complex numbers. To create a complex vector, add an imaginary term to a number with i

Examples

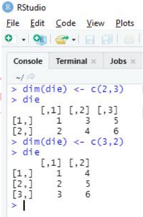


```
> a<-1
> typeof(a)
[1] "double"
> a<-1L
> typeof(a)
[1] "integer"
> a<-"1"
> typeof(a)
[1] "character"
> a <- TRUE
> typeof(a)
[1] "logical"
> a <- 1+2i
> typeof(a)
[1] "complex"
> |
```

of
ring

dim

- **dim** - transform an atomic vector into an n-dimensional array by giving it a dimensions attribute



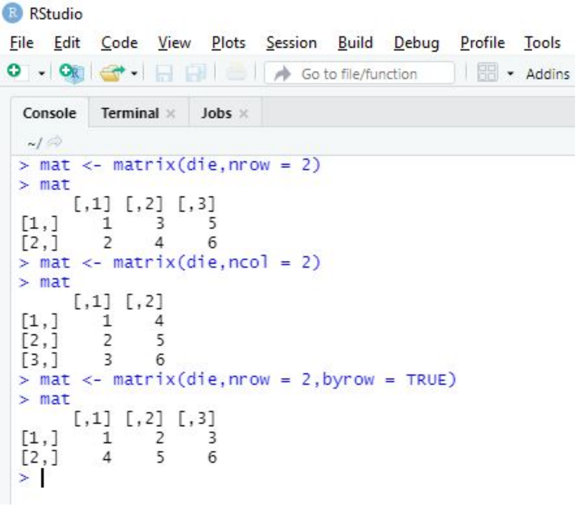
```
> dim(die) <- c(2,3)
> die
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> dim(die) <- c(3,2)
> die
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> |
```

- R will always use the first value in *dim* for the number of rows and the second value for the number of columns.
- You don't have much control - R always fills up each matrix by columns, instead of by rows. that is where we need array or matrix

Matrix

- Matrices store values in a two-dimensional array
- Define how many rows should be in the matrix by setting the *nrow* argument to a number.
- Alternatively, you can set the *ncol* argument
- Matrix will fill up the matrix column by column by default, but you can fill the matrix row by row if you include the argument *byrow* = *TRUE*

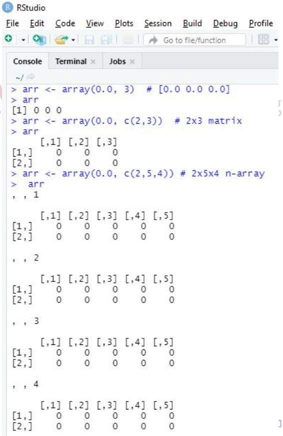
Example-Matrix



```
> mat <- matrix(die,nrow = 2)
> mat
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> mat <- matrix(die,ncol = 2)
> mat
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> mat <- matrix(die,nrow = 2,byrow = TRUE)
> mat
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
> 
```

Array

- An array is a vector with one or more dimensions. So, an array with one dimension is (almost) the same as a vector. An array with two dimensions is (almost) the same as a matrix. An array with three or more dimensions is an n-dimensional array.

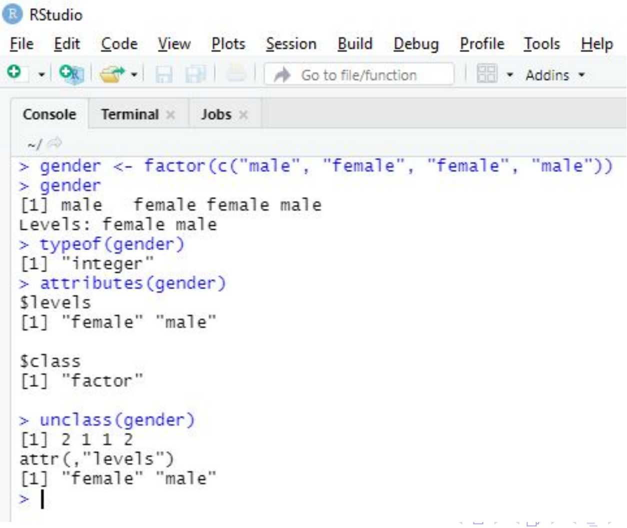


```
> arr <- array(0.0, 3)  # [0.0 0.0 0.0]
> arr
[1] 0 0 0
> arr <- array(0.0, c(2,3))  # 2x3 matrix
> arr
     [,1] [,2] [,3]
[1,]    0    0    0
[2,]    0    0    0
> arr <- array(0.0, c(2,5,4)) # 2x5x4 n-array
>  arr
, , 1

     [,1] [,2] [,3] [,4] [,5]
[1,]    0    0    0    0    0
[2,]    0    0    0    0    0

, , 2

     [,1] [,2] [,3] [,4] [,5]
[1,]    0    0    0    0    0
[2,]    0    0    0    0    0

, , 3

     [,1] [,2] [,3] [,4] [,5]
[1,]    0    0    0    0    0
[2,]    0    0    0    0    0

, , 4

     [,1] [,2] [,3] [,4] [,5]
[1,]    0    0    0    0    0
[2,]    0    0    0    0    0
```

Factors

- Factors are R's way of storing categorical information (Example: Gender may be male or female)
- To make a factor, pass an atomic vector into the factor function. R will recode the data in the vector as integers and store the results in an integer vector
- R will also add a levels attribute to the integer, which contains a set of labels for displaying the factor values and a class attribute, which contains the class factor
- However, factors can be confusing since they look like character strings but behave like integers.
- R will often try to convert character strings to factors when you load and create data

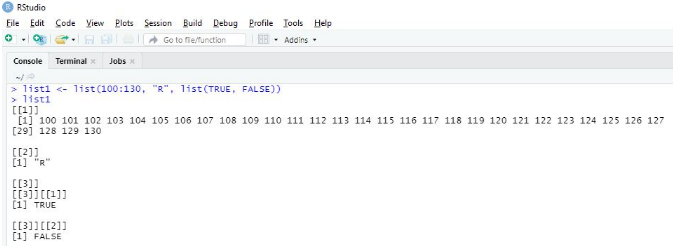
Example-Factors


RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Console Terminal Jobs
~/

```
> gender <- factor(c("male", "female", "female", "male"))
> gender
[1] male   female female male
Levels: female male
> typeof(gender)
[1] "integer"
> attributes(gender)
$levels
[1] "female" "male"

$class
[1] "factor"

> unclass(gender)
[1] 2 1 1 2
attr(,"levels")
[1] "female" "male"
> 
```

Lists

- Lists are like atomic vectors because they group data into a one-dimensional set
- lists group together R objects, such as atomic vectors and other lists.
- The double bracketed indexes tell you which element of the list is being displayed. The single bracket indexes tell you which sub element of an element is being displayed.

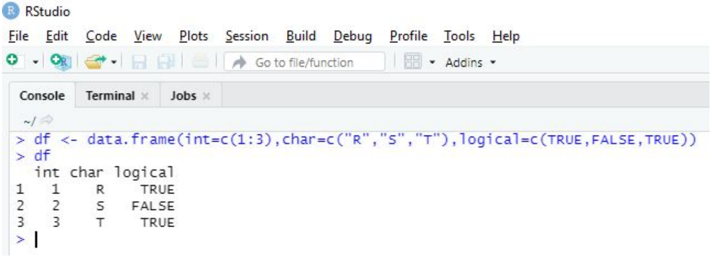

RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Console
Terminal
Jobs
~/

```
> list1 <- list(100:130, "R", list(TRUE, FALSE))
> list1
[[1]]
 [1] 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127
[29] 128 129 130

[[2]]
[1] "R"

[[3]]
[[3]][[1]]
[1] TRUE

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

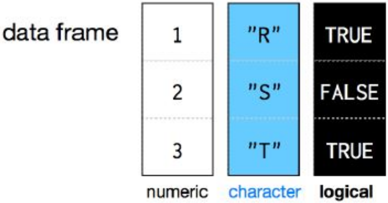
Data frame

- Data frames are the two-dimensional version of a list.
- The most useful storage structure for data analysis, and they provide an ideal way to store an entire dataset
- You can think of a data frame as R's equivalent to the Excel spreadsheet because it stores data in a similar format.
- Each vector should be set equal to a name that describes the vector
- *data.frame* will turn each vector into a column of the new data frame

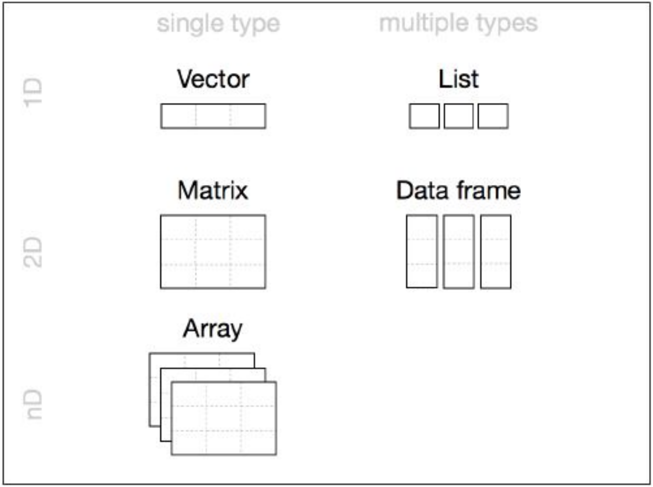
Example-Data frame


RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Console
Terminal
Jobs
~/

```
> df <- data.frame(int=c(1:3),char=c("R","S","T"),logical=c(TRUE,FALSE,TRUE))
> df
  int char logical
1   1    R    TRUE
2   2    S   FALSE
3   3    T    TRUE
> 
```



Overview


single type
multiple types
1D
Vector
List
2D
Matrix
Data frame
nD
Array

Agenda

Session 1 - Introduction

Operators

Objects

Conditional and Looping Statement

Functions

Importing and Loading Your Dataset

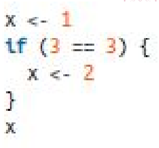
Scripts & Packages

Section 4

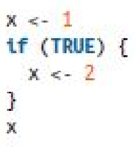
Conditional and Looping Statement

if statements

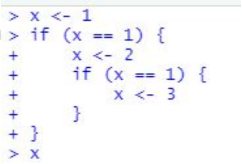
Syntax: if (condition) { statement(s) }



```
x <- 1
if (3 == 3) {
  x <- 2
}
x
```

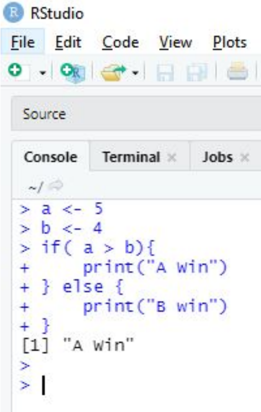


```
x <- 1
if (TRUE) {
  x <- 2
}
x
```

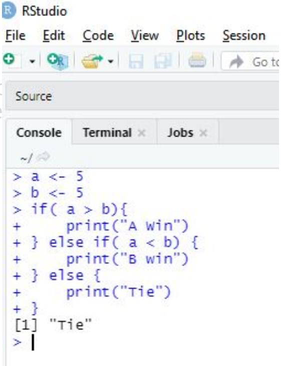


```
> x <- 1
> if (x == 1) {
+     x <- 2
+     if (x == 1) {
+         x <- 3
+     }
+ }
> x
```

if-else and nested if-else

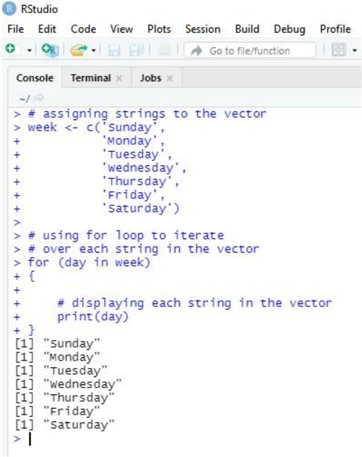


```
> a <- 5
> b <- 4
> if( a > b){
+       print("A win")
+ } else {
+       print("B win")
+ }
[1] "A win"
>
> |
```



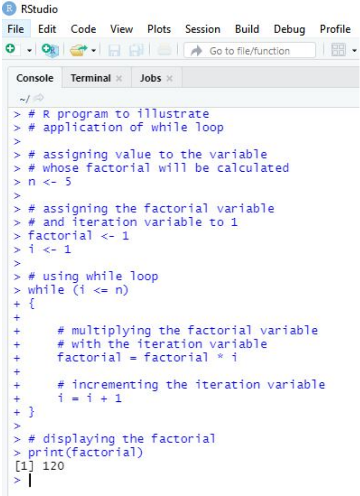
```
> a <- 5
> b <- 5
> if( a > b){
+     print("A win")
+ } else if( a < b) {
+     print("B win")
+ } else {
+     print("Tie")
+ }
[1] "Tie"
> |
```

for - loop



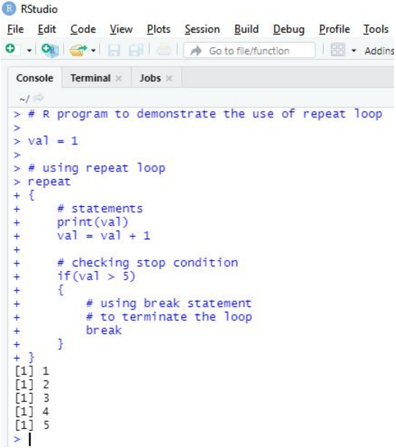
```
> # assigning strings to the vector
> week <- c('Sunday',
+           'Monday',
+           'Tuesday',
+           'Wednesday',
+           'Thursday',
+           'Friday',
+           'Saturday')
>
> # using for loop to iterate
> # over each string in the vector
> for (day in week)
+ {
+
+     # displaying each string in the vector
+     print(day)
+ }
[1] "Sunday"
[1] "Monday"
[1] "Tuesday"
[1] "Wednesday"
[1] "Thursday"
[1] "Friday"
[1] "Saturday"
> |
```

while - loop



```
> # R program to illustrate
> # application of while loop
>
> # assigning value to the variable
> # whose factorial will be calculated
> n <- 5
>
> # assigning the factorial variable
> # and iteration variable to 1
> factorial <- 1
> i <- 1
>
> # using while loop
> while (i <= n)
+ {
+
+     # multiplying the factorial variable
+     # with the iteration variable
+     factorial = factorial * i
+
+     # incrementing the iteration variable
+     i = i + 1
+ }
>
> # displaying the factorial
> print(factorial)
[1] 120
> |
```

Repeat - loop



```
> # R program to demonstrate the use of repeat loop
>
> val = 1
>
> # using repeat loop
> repeat
+ {
+     # statements
+     print(val)
+     val = val + 1
+
+     # checking stop condition
+     if(val > 5)
+     {
+         # using break statement
+         # to terminate the loop
+         break
+     }
+ }
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
> 
```

Agenda

- Session 1 - Introduction
- Operators
- Objects
- Conditional and Looping Statement
- Functions
- Importing and Loading Your Dataset
- Scripts & Packages

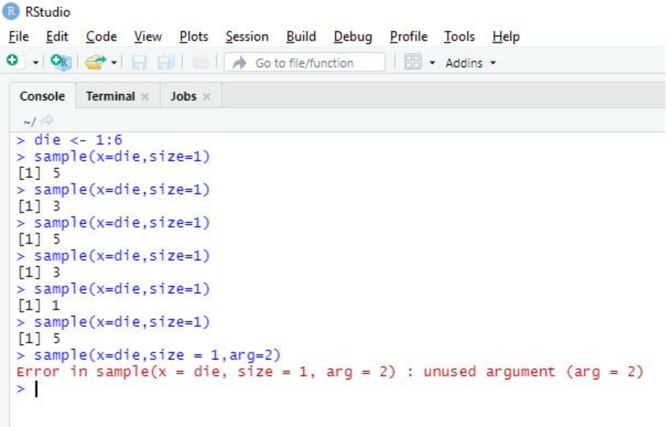
Section 5

Functions

Simulating a die using sample()

- To roll your die and get a number back, set x to die and sample one element from it using **sample()** - You'll get a new number each time you roll it
- Every argument in R function has a *name*. You can specify which data should be assigned to which argument by setting a name equal to
- Names help you avoid passing the wrong data to the wrong argument
- If you try to use a name that a function does not expect, you will likely get an error

Example

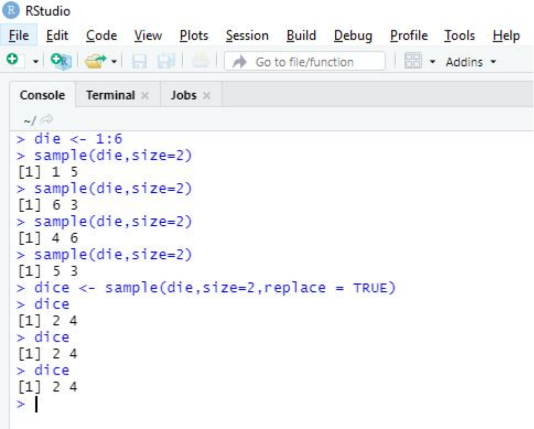

RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Console
Terminal
Jobs
~/

```
> die <- 1:6
> sample(x=die,size=1)
[1] 5
> sample(x=die,size=1)
[1] 3
> sample(x=die,size=1)
[1] 5
> sample(x=die,size=1)
[1] 3
> sample(x=die,size=1)
[1] 1
> sample(x=die,size=1)
[1] 5
> sample(x=die,size = 1,arg=2)
Error in sample(x = die, size = 1, arg = 2) : unused argument (arg = 2)
> 
```

Simulating a pair of dice

- If you set size = 2 in *sample()*, you can almost simulate a pair of dice.
- By default, sample() builds a sample without replacement i.e second die never has the same value as the first die
- The argument replace = TRUE causes sample() to sample with replacement
- Sampling with replacement is an easy way to create independent random samples.

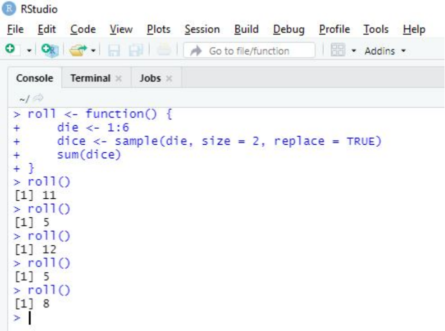
Example


RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Console
Terminal
Jobs
~/

```
> die <- 1:6
> sample(die,size=2)
[1] 1 5
> sample(die,size=2)
[1] 6 3
> sample(die,size=2)
[1] 4 6
> sample(die,size=2)
[1] 5 3
> dice <- sample(die,size=2,replace = TRUE)
> dice
[1] 2 4
> dice
[1] 2 4
> dice
[1] 2 4
> 
```

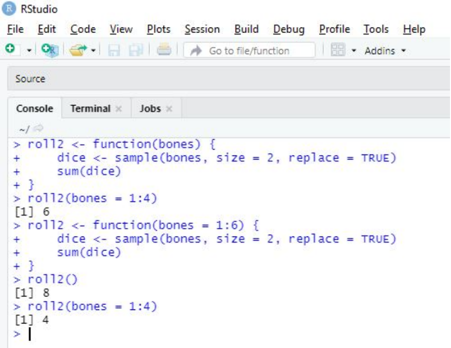
Writing your own R Functions

- Every function in R has three basic parts: a name, a body of code, and a set of arguments
- To make your own function, you need to replicate these parts and store them in an R object
- **Syntax:** $my\_function() <- function() \{\}$


RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Console
Terminal
Jobs
~/

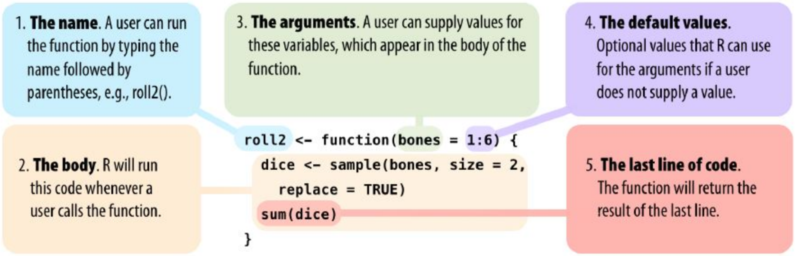
```
> roll <- function() {
+     die <- 1:6
+     dice <- sample(die, size = 2, replace = TRUE)
+     sum(dice)
+ }
> roll()
[1] 11
> roll()
[1] 5
> roll()
[1] 12
> roll()
[1] 5
> roll()
[1] 8
> 
```

Functions with Argument & default values


RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Source
Console
Terminal
Jobs
~/

```
> roll2 <- function(bones) {
+     dice <- sample(bones, size = 2, replace = TRUE)
+     sum(dice)
+ }
> roll2(bones = 1:4)
[1] 6
> roll2 <- function(bones = 1:6) {
+     dice <- sample(bones, size = 2, replace = TRUE)
+     sum(dice)
+ }
> roll2()
[1] 8
> roll2(bones = 1:4)
[1] 4
> 
```

Functions - Overview



1. **The name.** A user can run the function by typing the name followed by parentheses, e.g., roll2().

3. **The arguments.** A user can supply values for these variables, which appear in the body of the function.

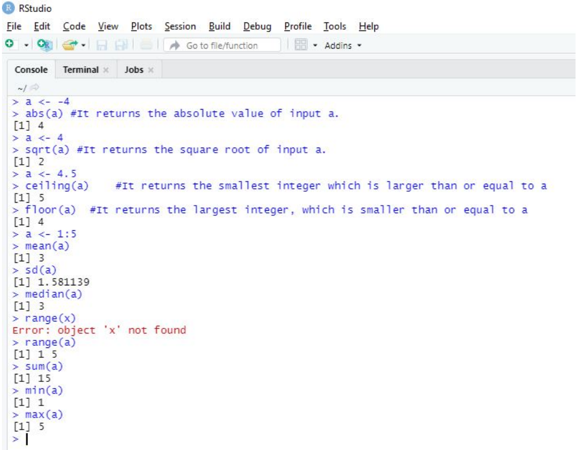
4. **The default values.** Optional values that R can use for the arguments if a user does not supply a value.

```
roll2 <- function(bones = 1:6) {
  dice <- sample(bones, size = 2,
    replace = TRUE)
  sum(dice)
}
```

2. **The body.** R will run this code whenever a user calls the function.

5. **The last line of code.** The function will return the result of the last line.

Some of popular Built-in functions


RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Console Terminal Jobs
~/

```
> a <- -4
> abs(a) #It returns the absolute value of input a.
[1] 4
> a <- 4
> sqrt(a) #It returns the square root of input a.
[1] 2
> a <- 4.5
> ceiling(a)    #It returns the smallest integer which is larger than or equal to a
[1] 5
> floor(a)  #It returns the largest integer, which is smaller than or equal to a
[1] 4
> a <- 1:5
> mean(a)
[1] 3
> sd(a)
[1] 1.581139
> median(a)
[1] 3
> range(x)
Error: object 'x' not found
> range(a)
[1] 1 5
> sum(a)
[1] 15
> min(a)
[1] 1
> max(a)
[1] 5
> 
```

Agenda

Session 1 - Introduction

Operators

Objects

Conditional and Looping Statement

Functions

Importing and Loading Your Dataset

Scripts & Packages

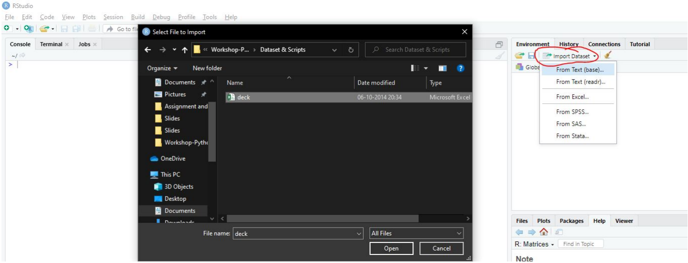
Section 6

Importing and Loading Your Dataset

Loading your data

- You should avoid typing large data sets in by hand - Typing invites typos and errors
- It is always better to acquire large data sets as a computer file. You can then ask R to read the file and store the contents as an object.
- *deck.csv* is a comma-separated values file, or CSV for short. CSVs are plain-text files, which means you can open them in a text editor
- To load a plain-text file into R, click the Import Dataset icon in RStudio
- RStudio will ask you to select the file you want to import, then it will open a wizard to help you import the data

Loading your data in RStudio



Name	Date modified	Type
deck	06-10-2014 20:34	Microsoft Excel

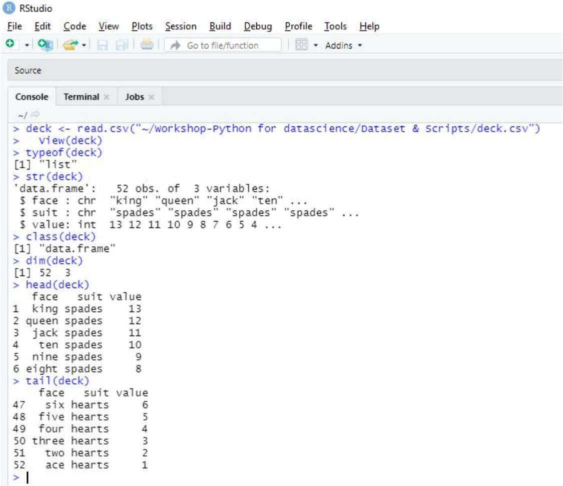
File name: deck

All Files

Additional Information

- You can load a plain-text file straight from the Internet by clicking the "From Web URL..." option under Import Dataset. The file will need to have its own URL, and you will need to be connected.
- When you import a data set, RStudio will save the data to a data frame and then display the data frame in a View tab. You can open any data frame in a View tab at any time with the View function
- head and tail are two functions that provide an easy way to peek at large data sets. head will return just the first six rows of the data set, and tail will return just the last six rows.

Import dataset - options


RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Source
Console
Terminal
Jobs
~/

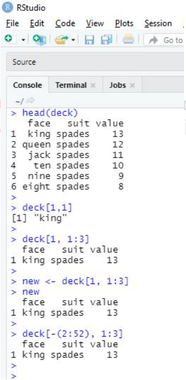
```
> deck <- read.csv("~/Workshop-Python for datascience/Dataset & Scripts/deck.csv")
>   View(deck)
> typeof(deck)
[1] "list"
> str(deck)
'data.frame':   52 obs. of  3 variables:
 $ face : chr  "king" "queen" "jack" "ten" ...
 $ suit : chr  "spades" "spades" "spades" "spades" ...
 $ value: int  13 12 11 10 9 8 7 6 5 4 ...
> class(deck)
[1] "data.frame"
> dim(deck)
[1] 52  3
> head(deck)
   face   suit value
1  king spades    13
2 queen spades    12
3  jack spades    11
4   ten spades    10
5  nine spades     9
6 eight spades     8
> tail(deck)
    face   suit value
47   six hearts     6
48  five hearts     5
49  four hearts     4
50 three hearts     3
51   two hearts     2
52   ace hearts     1
> 
```

Selecting values

- Commonly used index in data analysis:
  1. Positive integer
  2. Negative integer
  3. Blank space
  4. Names
- Indexing begins at **1**

Positive and Negative index

Negative index - R will return every element except the elements in a negative index.

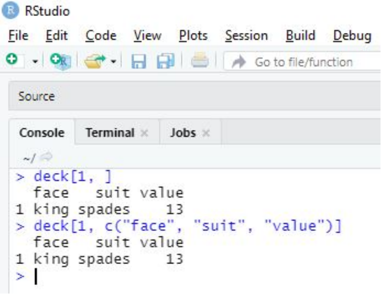


```
> head(deck)
   face   suit value
1  king spades    13
2 queen spades    12
3  jack spades    11
4   ten spades    10
5  nine spades     9
6 eight spades     8
>
> deck[1,1]
[1] "king"
>
> deck[1, 1:3]
  face   suit value
1 king spades    13
>
> new <- deck[1, 1:3]
> new
  face   suit value
1 king spades    13
>
> deck[-(2:52), 1:3]
  face   suit value
1 king spades    13
>
>
```

Blank space and Names

Blank space - tell R to extract every value in a dimension

Names - you can ask for the elements you want by name, if your object has names



```
> deck[1, ]
  face   suit value
1 king spades    13
> deck[1, c("face", "suit", "value")]
  face   suit value
1 king spades    13
> |
```

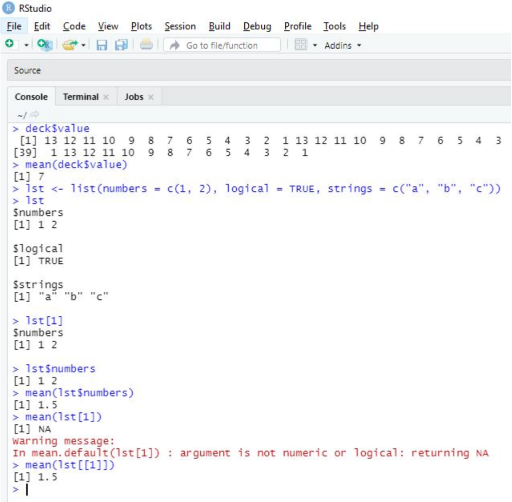
Dollar Sign

- You can extract values from data frames and lists with the $ syntax
- To select a column from a data frame, write the data frame's name and the column name separated by a $.
- R will return all of the values in the column as a vector
- Often we will store the variables of our data sets as columns and we want to run a function like mean or median on the values
- You can use the same $ notation with the elements of a list

Double bracket

- Subset using [] and $ look similar but you many R functions do not work with lists subset using [].
- When you use the $ notation, R will return the selected values as they are, with no list structure around them
- When you use the [] notation, The result is a *smaller* list
- **Double bracket** will do the same thing as the $ notation:
- If you subset a list with single-bracket notation, R will return a smaller list. If you subset a list with double-bracket notation, R will return just the values

Example screenshot


RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Source
Console
Terminal
Jobs
~/

```
> deck$value
 [1] 13 12 11 10  9  8  7  6  5  4  3  2  1 13 12 11 10  9  8  7  6  5  4  3
[39]  1 13 12 11 10  9  8  7  6  5  4  3  2  1
> mean(deck$value)
[1] 7
> lst <- list(numbers = c(1, 2), logical = TRUE, strings = c("a", "b", "c"))
> lst
$numbers
[1] 1 2

$logical
[1] TRUE

$strings
[1] "a" "b" "c"

> lst[1]
$numbers
[1] 1 2

> lst$numbers
[1] 1 2
> mean(lst$numbers)
[1] 1.5
> mean(lst[1])
[1] NA
Warning message:
In mean.default(lst[1]) : argument is not numeric or logical: returning NA
> mean(lst[[1]])
[1] 1.5
> 
```

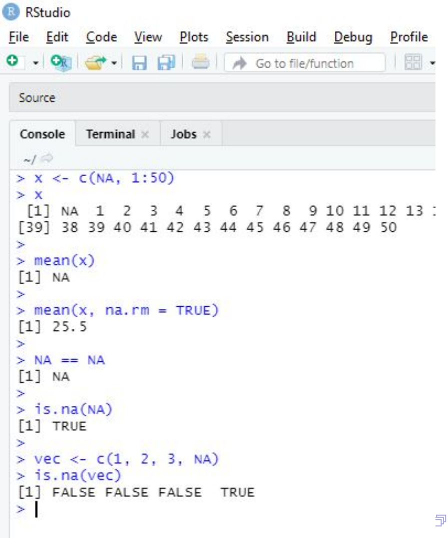
Missing information

- Missing information problems happen frequently in data science
- R has a way to help you manage these missing values
- The **NA** character is a special symbol in R. It stands for **“not available”** and can be used as a placeholder for missing information.
- Save you from making errors based on missing data.
- If even one of the values is NA, your result will be NA.

Handling NA

- Most R functions come with the optional argument, **na.rm**, which stands for NA remove
- R will ignore NAs when it evaluates a function if you add the argument na.rm = TRUE
- On occasion, you may want to identify the NAs in your data set with a logical test
- R supplies a special function that can test whether a value is an NA using is.na()

Example



```
> x <- c(NA, 1:50)
> x
 [1] NA  1  2  3  4  5  6  7  8  9 10 11 12 13 1
[39] 38 39 40 41 42 43 44 45 46 47 48 49 50
>
> mean(x)
[1] NA
>
> mean(x, na.rm = TRUE)
[1] 25.5
>
> NA == NA
[1] NA
>
> is.na(NA)
[1] TRUE
>
> vec <- c(1, 2, 3, NA)
> is.na(vec)
[1] FALSE FALSE FALSE  TRUE
> |
```

Agenda

Session 1 - Introduction

Operators

Objects

Conditional and Looping Statement

Functions

Importing and Loading Your Dataset

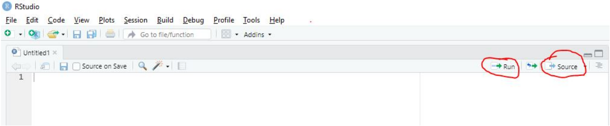
Scripts & Packages

Section 7

Scripts & Packages

Scripts

- Create a draft of your code as you go by using an *R script*
- You can open an R script in RStudio by going to File — New File — R script
- Creates a reproducible record of your work
- R will run whichever line of code your cursor is on. If you have a whole section highlighted, R will run the highlighted code. Alternatively, you can run the entire script by clicking the Source button.



Packages

- Many professors, programmers, and statisticians use R to design tools that can help people analyze data.
- They then make these tools free for anyone to use.
- To use these tools, you just have to download them. They come as preassembled collections of functions and objects called *packages*.
- Each R package is hosted at http://cran.r-project.org, the same website that hosts R.
- Open RStudio — Make sure you are connected to the Internet —- Run **install.packages(”package name”)** at the command line.

References

1. Garrett Grolemund, ”Hands-On Programming with R WRITE YOUR OWN FUNCTIONS AND SIMULATIONS”, O’Reilly Media, 2014
2. https://www.datamentor.io/r-programming/operator/

Thank You..