DS116 - Data Visualization

Intro to R

Habet Madoyan, Narek Sahakyan

American University of Armenia

Section 1

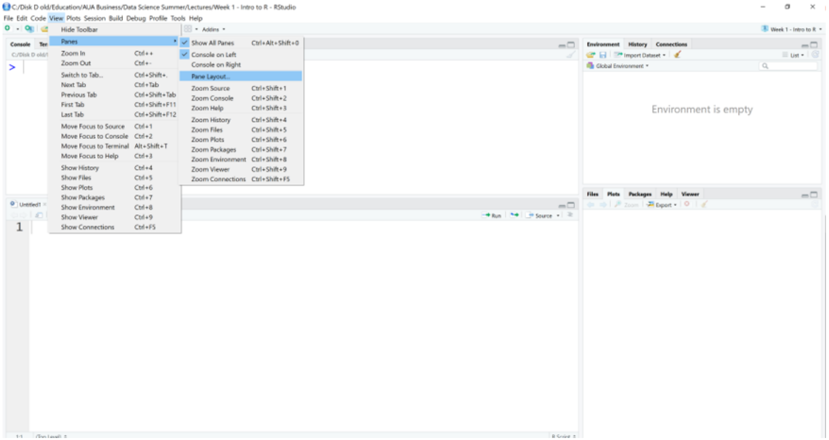
Intro to R

Intro to R

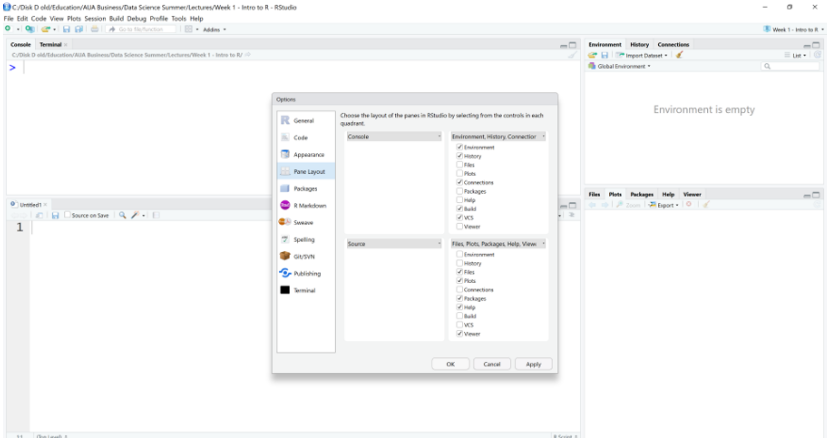
- Go to Comprehensive R Archive Network (CRAN) and install version for your operating system
- Go to RStudio and install RStudio IDE
- You need to install R before installing RStudio

Intro to R

Configure the pane layout of RStudio as you wish


C:/Disk D old/Education/AUA Business/Data Science Summer/Lectures/Week 1 - Intro to R - RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Hide Toolbar
Panes
Zoom In Ctrl++
Zoom Out Ctrl+-
Switch to Tab... Ctrl+Shift+.
Next Tab Ctrl+Tab
Previous Tab Ctrl+Shift+Tab
First Tab Ctrl+Shift+F11
Last Tab Ctrl+Shift+F12
Move Focus to Source Ctrl+1
Move Focus to Console Ctrl+2
Move Focus to Terminal Alt+Shift+T
Move Focus to Help Ctrl+3
Show History Ctrl+4
Show Files Ctrl+5
Show Plots Ctrl+6
Show Packages Ctrl+7
Show Environment Ctrl+8
Show Viewer Ctrl+9
Show Connections Ctrl+F5
Show All Panes Ctrl+Alt+Shift+0
Console on Left
Console on Right
Pane Layout...
Zoom Source Ctrl+Shift+1
Zoom Console Ctrl+Shift+2
Zoom Help Ctrl+Shift+3
Zoom History Ctrl+Shift+4
Zoom Files Ctrl+Shift+5
Zoom Plots Ctrl+Shift+6
Zoom Packages Ctrl+Shift+7
Zoom Environment Ctrl+Shift+8
Zoom Viewer Ctrl+Shift+9
Zoom Connections Ctrl+Shift+F5
Addins
Week 1 - Intro to R
Console
Untitled1
Run
Source
Environment History Connections
Import Dataset
List
Global Environment
Environment is empty
Files Plots Packages Help Viewer
Zoom Export

Intro to R



Choose the layout of the panes in RStudio by selecting from the controls in each quadrant.

Console

Environment, History, Connection

- ✓ Environment
- ✓ History
- ☐ Files
- ☐ Plots
- ✓ Connections
- ☐ Packages
- ☐ Help
- ✓ Build
- ✓ VCS
- ☐ Viewer

Source

Files, Plots, Packages, Help, Viewer

- ☐ Environment
- ☐ History
- ✓ Files
- ✓ Plots
- ☐ Connections
- ✓ Packages
- ✓ Help
- ☐ Build
- ☐ VCS
- ✓ Viewer

Intro to R

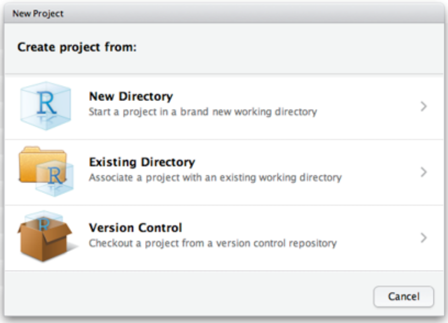
RStudio projects make it straightforward to divide your work into multiple contexts, each with their own working directory, workspace, history, and source documents.

Creating Projects

- RStudio projects are associated with R working directories. You can create an RStudio project:
  - in a brand new directory;
  - in an existing directory where you already have R code and data;
  - by cloning a version control (Git or Subversion) repository.

Intro to R

- Create an empty directory
- Go to File → New Project
- Choose existing directory


New Project

Create project from:

New Directory
Start a project in a brand new working directory

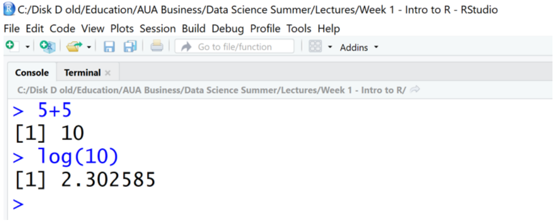
Existing Directory
Associate a project with an existing working directory

Version Control
Checkout a project from a version control repository

Cancel

Intro to R

- Now when you have created your project, all files associated with the project need to be saved in the project directory
- To check the project directory do getwd() in Console


C:/Disk D old/Education/AUA Business/Data Science Summer/Lectures/Week 1 - Intro to R - RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Console
Terminal
C:/Disk D old/Education/AUA Business/Data Science Summer/Lectures/Week 1 - Intro to R/

```
> 5+5
[1] 10
> log(10)
[1] 2.302585
>
```

The name of the project appears at the top of RStudio window

Why do we need RStudio Project?

The benefits of RStudio project:

- Can have different projects with their own environment.
- While opening a project, R restores previous work and project history includes recent commands which had been used in the project.
- Source pane remembers the files which had been opened.

Intro to R

When a new project is created, RStudio:

- Creates a project file (with an .Rproj extension) within the project directory. This file contains various project options (discussed below) and can also be used as a shortcut for opening the project directly from the filesystem.
- Creates a hidden directory (named .Rproj.user) where project-specific temporary files (e.g. auto-saved source documents, window-state, etc.) are stored. This directory is also automatically added to .Rbuildignore, .gitignore, etc. if required.
- Loads the project into RStudio and displays its name in the Projects toolbar (which is located on the far right side of the main toolbar).

Intro to R

Opening Projects

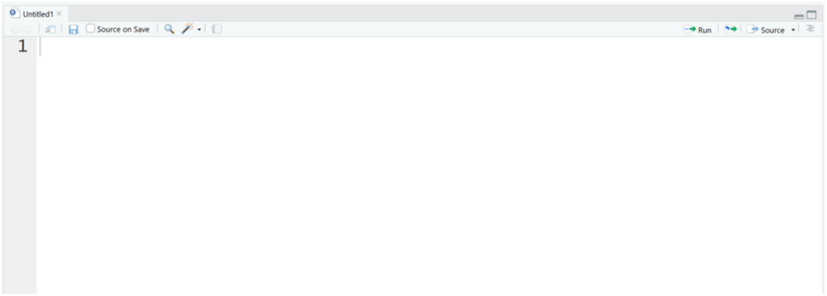
There are several ways to open a project:

- Using the **Open Project** command (available from both the Projects menu and the Projects toolbar) to browse for and select an existing project file (e.g. MyProject.Rproj).
- Selecting a project from the list of the most recently opened projects (also available from both the Projects menu and toolbar).
- Double-clicking on the project file within the system shell (e.g. Windows Explorer, OSX Finder, etc.).

Intro to R

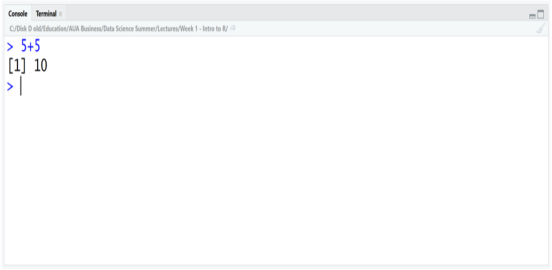
The source editor

- If you plan to reuse your code, write it in source editor



Intro to R

Here you can type any valid R command after the > prompt followed by **Enter** and R will execute that command.


Console
Terminal
C:/Disk D old/Education/AUA Business/Data Science Summer/Lectures/Week 1 - Intro to R/

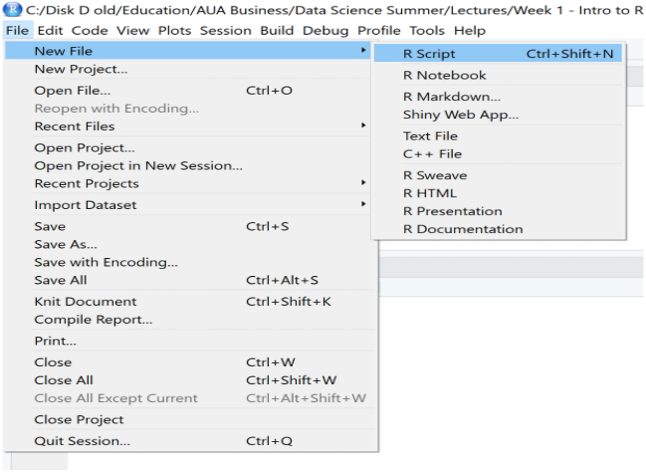
```
> 5+5
[1] 10
> 
```

Use console as a calculator

[1] is the index for the output, just ignore it

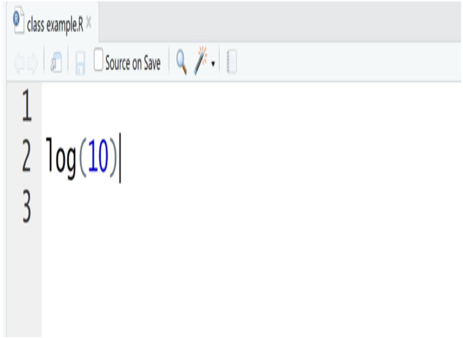
Intro to R

This will create new script file



Intro to R

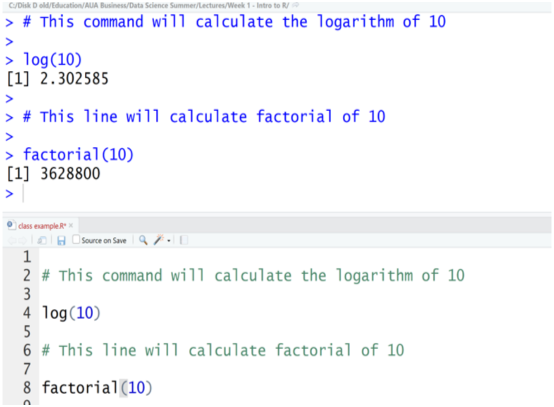
- To run the script line from source editor, put the cursor anywhere on the line and hit **Ctrl+Enter**
- You will see the output in the console



```
log(10)
```

Intro to R

- If you have a piece of text in your editor that is not a code (thus is not executable by R) then you need to comment it, add # before each line
- When you run this line of code, it will be printed in Console as a text
- To comment large chunk of text, use **Ctrl+Shift+C**

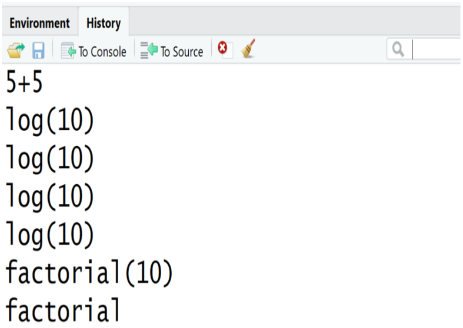

C:/Disk D old/Education/AUA Business/Data Science Summer/Lectures/Week 1 - Intro to R/
class example.R*
Source on Save

```
> # This command will calculate the logarithm of 10
>
> log(10)
[1] 2.302585
>
> # This line will calculate factorial of 10
>
> factorial(10)
[1] 3628800
>
```

```

# This command will calculate the logarithm of 10

log(10)

# This line will calculate factorial of 10

factorial(10)
```

Intro to R

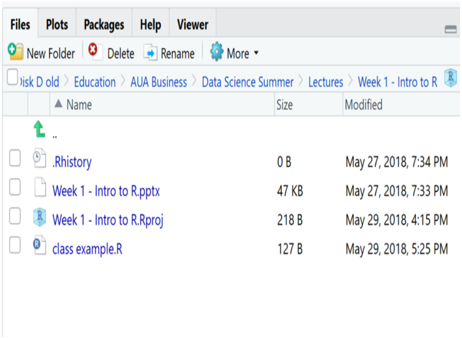
- The environment window contains objects (data, values, functions) R has currently stored in its memory.
- The history window shows all commands that were executed in the Console.



```
5+5
log(10)
log(10)
log(10)
log(10)
factorial(10)
factorial
```

Intro to R

Bottom right: files, plots, packages, help, & viewer pane. Here you can open files, view plots, install and load packages, read main pages, and view markdown and other documents in the viewer tab.

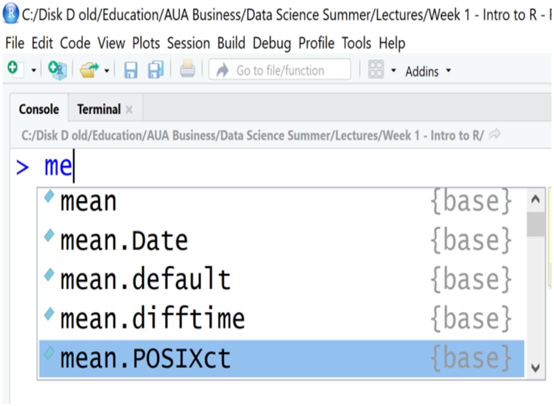


Name	Size	Modified
..		
.Rhistory	0 B	May 27, 2018, 7:34 PM
Week 1 - Intro to R.pptx	47 KB	May 27, 2018, 7:33 PM
Week 1 - Intro to R.Rproj	218 B	May 29, 2018, 4:15 PM
class example.R	127 B	May 29, 2018, 5:25 PM

Intro to R

Some useful shortcuts:

- type first few letters of the function/object then hit Tab to open dropdown menu with possible options



C:/Disk D old/Education/AUA Business/Data Science Summer/Lectures/Week 1 - Intro to R -
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Console
Terminal
C:/Disk D old/Education/AUA Business/Data Science Summer/Lectures/Week 1 - Intro to R/

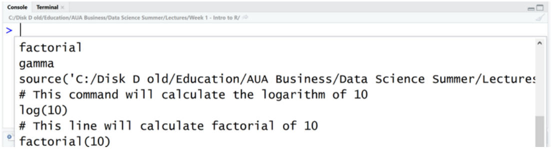
```
> me
mean            {base}
mean.Date       {base}
mean.default    {base}
mean.difftime   {base}
mean.POSIXct    {base}
```

Intro to R

It's often the case that you want to re-execute commands that you previously entered. The RStudio console supports the ability to recall previous commands using the arrow keys:

- **Up** - Recall previous command(s)
- **Down** - Reverse of Up

You can even view a list of your recent commands by pressing **Ctrl+Up** on Windows or **Command+Up** on a Mac.



```
C:/Disk D old/Education/AUA Business/Data Science Summer/Lectures/Week 1 - Intro to R/
> |
factorial
gamma
source('C:/Disk D old/Education/AUA Business/Data Science Summer/Lectures
# This command will calculate the logarithm of 10
log(10)
# This line will calculate factorial of 10
factorial(10)
```

Intro to R

- When you download R from the CRAN, you get that “base” R system.
- The base R system comes with basic functionality; implements the R language.
- One reason R is so useful is the large collection of packages that extend the basic functionality of R.
- R packages are developed and published by the larger R community.

Intro to R

- Packages can be installed with the install.packages() function in R.
- To install a single package, pass the name of the library to the install.packages() function as the first argument.
- The following code installs the devtools package from CRAN.

```
install.packages("devtools")
```

Intro to R

- The package needs to be installed only once
- To load the package into R environment you need to use function library()
- You need to load the library everytime you start a new R

```
library(devtools)
```

Intro to R

- You can also install packages from github
- The following code will install the following package from github

```
library(devtools)
install_github("christophM/iml")
```

If you do not want to load the entire package but want to use some function from it, use the following command

package::function_name

```
devtools::install_github("christophM/iml")
```

Intro to R

To access help/documentation on a function from R base package

```
?mean
```

To access help/documentation on function from a library

```
#??geom_path
```

The same

```
#?ggplot2::geom_path
```

Help on the package

```
help(package='ggplot2')
```

Intro to R

- Each package on CRAN has its own webpage
- This includes documentation and sometimes includes vignettes

- If you have a specific task to do, then look at R Task View
- Here are all the packages and R functionality described for the Time series analysis

Section 2

Intro to R programming language

Intro to R programming language

- Anything in R is an object.
- Objects are assigned values using <- . (An equal sign = can also be used.) For e.g., the following command assigns value 5 to object x.

```
x <- 5
x
```

```
## [1] 5
```

- R is case sensitive, thus Data10 and data10 are two different objects.
- A tidy code requires a space before the assignment operator and a space after.
    - You can see what is inside the object just by simple entering the name of the object in command line
    - When the object is created it should appear in your Environment Window
    - If the object is not in your environment window you cannot work with it
    - The assignment operator works in the opposite direction as well

```
5 -> x
x
```

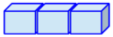
```
## [1] 5
```

Data Structres in R

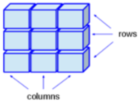
R programming supports five basic types of data structure.

- **Homogeneous**

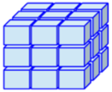
Vector



Matrix

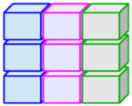


Array

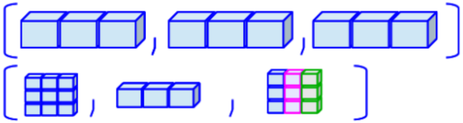


- **Heterogeneous**

Data frame



List



Data Structures: Vector

Vector is a type of data structure that contains similar types of data, i.e., integer, double, logical, complex, etc. In order to create a vector c() function is used.

```
(x <- c(10,5,6))
```

```
## [1] 10  5  6
```

```
(x1 <- c(1:10))
```

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

The same as (only when you have a sequence)

```
(x1 <- 1:10)
```

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

```
class(x)
```

```
## [1] "numeric"
```

A character vector

```
y <- c("CS", "DS", "EC")
class(y)
```

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

Coercion

- All elements of an atomic vector must be the same type, so when you attempt to combine different types they will be coerced to the most flexible type.
- Types from least to most flexible are: logical, numeric, and character.
- For example, combining a character and an integer yields a character:

```
(x3 <- c("A",1))
```

```
## [1] "A" "1"
```

```
class(x3)
```

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

Data Structures: Matrix

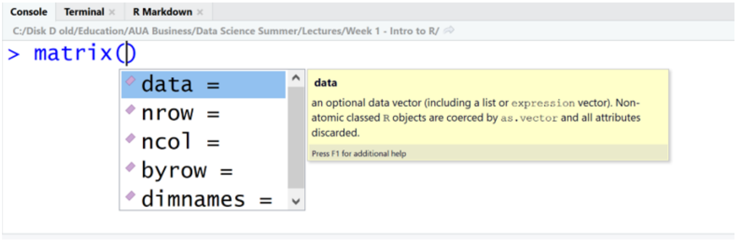
- **Matrix** is a two-dimensional data structure and can be created using matrix() function.
- Matrix is a collection of vectors with the same length and type

```
m <- matrix(data=1:15, nrow=3)
m
```

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

Function

- To look at the function arguments, hit Tab
- You need to name the arguments if they are not in the same order as defined within the function
- If the order is the same you can skip the names


Console
Terminal
R Markdown
C:/Disk D old/Education/AUA Business/Data Science Summer/Lectures/Week 1 - Intro to R/
Press F1 for additional help

```
> matrix()
  data =
  nrow =
  ncol =
  byrow =
  dimnames =
```

data

an optional data vector (including a list or `expression` vector). Non-atomic classed `R` objects are coerced by `as.vector` and all attributes discarded.

Data Structures

Example: function matrix with arguments flipped

The same result

```
(m <- matrix(1:15, 3))
```

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

Different result

```
(m <- matrix(3, 1:15))
```

How can we swap the order of the arguments and still have the same result ?

```
##      [,1]
## [1,]    3
```

Data Sctructures

If arguments are not named, the method matches the arguments based on the positions, so the first argument is considered as data, the second as nrow, and so on.

```
(m <- matrix(nrow=3, data=1:15))
```

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

Data Sctructures

What will be the output of the following expression ?

```r
mixed_arguments_matrix_1 <- matrix(ncol = 3, nrow = 2, FALSE, (1:6))
```

What about this one ?

```r
mixed_arguments_matrix_2 <- matrix(ncol = 3, nrow = 2, (1:6), T)
```

Data Sctructures

The named arguments are handled out of the order, so the remaining arguments are treated as a regular positional argument, with the named ones being removed from the list of positional arguments.

```
(mixed_arguments_matrix_1)
```

```
##       [,1]  [,2]  [,3]
## [1,] FALSE FALSE FALSE
## [2,] FALSE FALSE FALSE
```

```
(mixed_arguments_matrix_2)
```

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

Data Structures

Other ways of building a matrix

- Create two vectors of the same length

```
x <- 0:5
y <- -5:0
```

- Combine together by row

```
(m1 <- rbind(x,y))
```

```
##   [,1] [,2] [,3] [,4] [,5] [,6]
## x    0    1    2    3    4    5
## y   -5   -4   -3   -2   -1    0
```

Data Structures

The matrix has row names but does not have column names

```
colnames(m1)
```

```
## NULL
```

```
rownames(m1)
```

```
## [1] "x" "y"
```

Data Structures

- Combine together by column

```
(m2 <- cbind(x,y))
```

```
##      x  y
## [1,] 0 -5
## [2,] 1 -4
## [3,] 2 -3
## [4,] 3 -2
## [5,] 4 -1
## [6,] 5  0
```

Pay attention, the matrix has column names, but does not have row names

```
colnames(m2)
```

```
## [1] "x" "y"
```

```
rownames(m2)
```

```
## NULL
```

Data Structures

Check if the resulting object is a matrix

```r
is.matrix(m1)
```

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

```r
is.matrix(m2)
```

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

Data Structures

Set the rownames

```r
rownames(m2) <- c('A', 'B', 'C', 'D', 'E', 'F')
m2
```

```
##   x  y
## A 0 -5
## B 1 -4
## C 2 -3
## D 3 -2
## E 4 -1
## F 5  0
```

Data Structures

Subset by either integer position, or by rowname

```
m2[1:3,]
```

```
##   x  y
## A 0 -5
## B 1 -4
## C 2 -3
```

```
m2[c('A', 'B', 'C'),]
```

```
##   x  y
## A 0 -5
## B 1 -4
## C 2 -3
```

Data Structures

What about combining two vectors of different sizes into a matrix ?

```
matrix1 <- rbind(c(1:5), c(1:10), c(1:3))
(is.matrix(matrix1))
```

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

Data Structures

The result:

```
(matrix1)
```

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

R adjusts the length of the shorter vectors by recycling their elements until they match the length of the longest vector.

Data Structures

The same happens with cbind

```
(matrix2 <- cbind(c(1:5), c(1:10), c(1:3)))
```

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

Data Structures

What will happen if you combine vectors of different types ?
Scenario 1

```
a <- (1:5)
b <- c("1","2","3","4","5")


c_matrix <- rbind(a,b)
```

Scenario 2

```
a1 <- c(1,2,3,4,5)
b1 <- c(TRUE, FALSE, TRUE, FALSE, TRUE)
c_matrix1 <- cbind(a1,b1)
```

Scenario 3

```
a2 <- c(1,2,3,4,5)
b2 <- c("1","2","3","A","5")
c_matrix2 <- rbind(a2,b2)
```

Data Structures

Check the structure of the resulting objects

```
(c_matrix)
```

```
##   [,1] [,2] [,3] [,4] [,5]
## a "1"  "2"  "3"  "4"  "5"
## b "1"  "2"  "3"  "4"  "5"
```

```
(c_matrix1)
```

```
##      a1 b1
## [1,]  1  1
## [2,]  2  0
## [3,]  3  1
## [4,]  4  0
## [5,]  5  1
```

```
(c_matrix2)
```

```
##    [,1] [,2] [,3] [,4] [,5]
## a2 "1"  "2"  "3"  "4"  "5"
## b2 "1"  "2"  "3"  "A"  "5"
```

Data Structures

What will happen in this scenario ?

```
c_matrix <- rbind(c("TRUE","FALSE"), c(TRUE, FALSE), c(1,0))
```

Data Structures

The result:

```
(c_matrix)
```

```
##      [,1]   [,2]
## [1,] "TRUE" "FALSE"
## [2,] "TRUE" "FALSE"
## [3,] "1"    "0"
```

This happens of R's built in implicit coercion rules, however you can use explicit coercion to control the choice of the data type

```
(c_matrix <- rbind(
  as.numeric(c("1","2")),
  as.numeric(c(TRUE, FALSE)),
  c(1,0))
)
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    1    0
## [3,]    1    0
```

Data Structures: Data frame

- A **data frame** is the most common way of storing data in R, and if used systematically makes data analysis easier.
- Under the hood, a data frame is a list of equal-length vectors that can also have different types.

```
df <- data.frame(x=1:10, y=11:20)
head(df, n=5)
```

```
##   x  y
## 1 1 11
## 2 2 12
## 3 3 13
## 4 4 14
## 5 5 15
```

Data Structures

- Dataframe always has both column and row names
- If you don't specify row names it is just the row number

```r
colnames(df)
```

```
## [1] "x" "y"
```

```r
rownames(df)
```

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

Data Structures

- You can combine vectors with different type into a dataframe
- First create a matrix with two vectors of different types

```
Club <- c("Juventus", "Napoli", "Roma", "Inter")
Points <- c(95,91,77,72)
```

```
s_matrix <- cbind(Club, Points)
str(s_matrix)
```

```
##  chr [1:4, 1:2] "Juventus" "Napoli" "Roma" "Inter" "95" "91" "77" "72"
##  - attr(*, "dimnames")=List of 2
##   ..$ : NULL
##   ..$ : chr [1:2] "Club" "Points"
```

```
class(s_matrix)
```

```
## [1] "matrix" "array"
```

```
typeof(s_matrix)
```

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

Data Structures

Create a dataframe

```
df_seriea <- data.frame(Club, Points)
str(df_seriea)
```

```
## 'data.frame':    4 obs. of  2 variables:
##  $ Club  : chr  "Juventus" "Napoli" "Roma" "Inter"
##  $ Points: num  95 91 77 72
```

Two columns (vectors) in the data frame have different types

Data Structures: List

List

- In R lists act as containers.
- Unlike vectors, the contents of a list are not restricted to a single type and can encompass any mixture of data types.
- Lists are sometimes called generic vectors, because the elements of a list can be of any type of R object, even lists containing further lists.
- This property makes them fundamentally different from atomic vectors.
- A list is a special type of vector. Each element can be a different type.

Data Structures

List containing dataframe and a matrix

```
my_list <- list(df_seriea, m1)
str(my_list)
```

```
## List of 2
##  $ :'data.frame':    4 obs. of  2 variables:
##   ..$ Club  : chr [1:4] "Juventus" "Napoli" "Roma" "Inter"
##   ..$ Points: num [1:4] 95 91 77 72
##  $ : int [1:2, 1:6] 0 -5 1 -4 2 -3 3 -2 4 -1 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:2] "x" "y"
##   .. ..$ : NULL
```

Data Structures

Look what is inside

```
my_list
```

```
## [[1]]
##       Club Points
## 1 Juventus     95
## 2   Napoli     91
## 3     Roma     77
## 4    Inter     72
##
## [[2]]
##   [,1] [,2] [,3] [,4] [,5] [,6]
## x    0    1    2    3    4    5
## y   -5   -4   -3   -2   -1    0
```

Section 3

Data Types

Quantitative vs Qualitative Data

Before diving into R data types, it's important to understand the two main categories of data:

Quantitative (Numerical) Data

- Represents quantities or amounts
- Can be measured and expressed as numbers
- Supports mathematical operations (addition, averaging, etc.)
- Examples: age, height, temperature, number of students

Qualitative (Categorical) Data

- Represents qualities or characteristics
- Describes categories or groups
- Cannot be meaningfully averaged or summed
- Examples: gender, eye color, student grades (A, B, C), country names

Quantitative Data: Types and Examples

Discrete Quantitative Data

- Countable, whole numbers only
- Examples: number of students, number of cars, count of errors

```
students_per_class <- c(25, 30, 28, 32, 27)
sum(students_per_class)
```

```
## [1] 142
```

```
mean(students_per_class)
```

```
## [1] 28.4
```

Continuous Quantitative Data

- Can take any value within a range (including decimals)
- Examples: height, weight, temperature, time

```
heights <- c(165.5, 172.3, 168.0, 175.8, 160.2)
mean(heights)
```

```
## [1] 168.36
```

```
range(heights)
```

```
## [1] 160.2 175.8
```

Qualitative Data: Types and Examples

Nominal Data

- Categories with no natural order
- Examples: colors, countries, gender, blood type

```r
eye_colors <- c("Brown", "Blue", "Green", "Brown", "Blue", "Brown")
table(eye_colors)
```

```
## eye_colors
##  Blue Brown Green
##     2     3     1
```

Ordinal Data

- Categories WITH a natural order
- Examples: grades ($A > B > C$), satisfaction ($\text{High} > \text{Medium} > \text{Low}$)

```r
grades <- c("A", "B", "A", "C", "B", "A", "B", "C", "D", "B")
table(grades)
```

```
## grades
## A B C D
## 3 4 2 1
```

Representing Qualitative Data in R

Character vectors - simple storage, no order

```
colors <- c("Red", "Blue", "Green", "Red")
class(colors)
```

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

Factors - categorical data with defined levels

```
colors_factor <- factor(colors)
levels(colors_factor)
```

```
## [1] "Blue"  "Green" "Red"
```

```
grades <- c("B", "A", "C", "B", "A")
grades_ordered <- factor(grades,
                         levels = c("D", "C", "B", "A"),
                         ordered = TRUE)
grades_ordered
```

```
## [1] B A C B A
## Levels: D < C < B < A
```

Why Ordered Factors Matter

Ordered factors allow meaningful comparisons:

```r
student_grades <- factor(c("B", "A", "C", "B", "D", "A"),
                         levels = c("D", "C", "B", "A"),
                         ordered = TRUE)
```

```r
student_grades[1] > student_grades[3]
```

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

```r
student_grades[student_grades >= "B"]
```

```
## [1] B A B A
## Levels: D < C < B < A
```

Practical Example: Student Dataset

```r
students <- data.frame(
  student_id = 1:6,
  num_courses = c(5, 4, 6, 5, 4, 5),
  gpa = c(3.5, 3.8, 3.2, 3.9, 3.6, 3.4),
  major = c("CS", "DS", "EC", "CS", "DS", "EC"),
  year = factor(c("Junior", "Senior", "Sophomore", "Senior", "Junior", "Fre
                levels = c("Freshman", "Sophomore", "Junior", "Senior"),
                ordered = TRUE)
)
str(students)
```

```
## 'data.frame':    6 obs. of  5 variables:
##  $ student_id : int  1 2 3 4 5 6
##  $ num_courses: num  5 4 6 5 4 5
##  $ gpa        : num  3.5 3.8 3.2 3.9 3.6 3.4
##  $ major      : chr  "CS" "DS" "EC" "CS" ...
##  $ year       : Ord.factor w/ 4 levels "Freshman"<"Sophomore"<..: 3 4 2 4 3 1
```

Working with Different Data Types

```
cat("Average GPA:", mean(students$gpa), "\n")
```

```
## Average GPA: 3.566667
```

```
cat("Total courses:", sum(students$num_courses), "\n")
```

```
## Total courses: 29
```

```
table(students$major)
```

```
##
## CS DS EC
##  2  2  2
```

```
advanced_students <- students[students$year >= "Junior", ]
advanced_students[, c("student_id", "year", "gpa")]
```

```
##   student_id   year gpa
## 1          1 Junior 3.5
## 2          2 Senior 3.8
## 4          4 Senior 3.9
## 5          5 Junior 3.6
```

Choosing the Right Data Type

Data Type	R Representation	Operations	Example
Discrete quantitative	integer/numeric	math operations	`num_students`
Continuous quantitative	numeric	math operations	`height`, `gpa`
Nominal categorical	factor (unordered)	frequencies	`color`, `major`
Ordinal categorical	factor (ordered)	frequencies + comparison	`grade`, `year`

Key principle: Choose the data type that accurately represents your data's nature and enables the analyses you need to perform.

Common Mistakes to Avoid

Mistake 1: Treating categorical data as numeric

```
zip_codes <- c(10001, 90210, 30301)
mean(zip_codes)
```

```
## [1] 43504
```

Mistake 2: Forgetting to order ordinal data

```
satisfaction <- factor(c("High", "Low", "Medium", "High"))
satisfaction_ordered <- factor(c("High", "Low", "Medium", "High"),
                               levels = c("Low", "Medium", "High"),
                               ordered = TRUE)
satisfaction_ordered[1] > satisfaction_ordered[2]
```

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

Data Types

Data types used in R

- Logical
- Numeric
- Factor
- Character

Data Types

Logical data type is one of the frequently used data types usually used for comparing two values. Values that a logical data type takes is TRUE or FALSE.

```
a <- 10
a > 5
```

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

```
log1 <- c(5, 6, TRUE)
typeof(log1)
```

```
## [1] "double"
```

```
log1
```

```
## [1] 5 6 1
```

Coercion of numeric and logical values will result as numeric.

Data Types

String literals or string values are stored as Character objects in R.

```r
b <- c("Armenia", "Georgia", "Azerbaijan")
typeof(b)
```

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

```r
(b1 <- c(b, TRUE))
```

```
## [1] "Armenia"    "Georgia"    "Azerbaijan" "TRUE"
```

Data Types

- A **factor** is a vector that can contain only predefined values, and is used to store categorical data.
- Factors are built on top of integer vectors using two attributes: the class, factor, which makes them behave differently from regular integer vectors, and the levels, which defines the set of allowed values.

```
b <- as.factor(c("Armenia", "Georgia", "Azerbaijan"))
```

Factors in R are stored as a vector of integer values with a corresponding set of character values used when the factor is displayed.

```
typeof(b)
```

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

```
as.numeric(b)
```

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

```
b
```

```
## [1] Armenia    Georgia    Azerbaijan
## Levels: Armenia Azerbaijan Georgia
```

Data Types

You can relevel the factor variable by changing the reference value

```
b<- relevel(b, ref='Azerbaijan')
levels(b)
```

```
## [1] "Azerbaijan" "Armenia"    "Georgia"
```

```
as.numeric(b)
```

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

Section 4

Special Values in R

Special Values in R

Missing Values

The examples and reasons of having missing values:

- The information was not collected (e.g. some people decline to give their age or weight).
- Some attributes are not applicable to all objects (e.g. forms have conditional parts that are filled out only when a person answers a previous question in a certain way).
- The information is not imported in the database (system missing).
- In R missing or undefined values are represented by NA.

Special Values in R

NA

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

```
v1 <- c(1,2,4,NA,5)
```

```
is.na(v1)
```

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

Special Values in R

```r
v2 <- c(10, "A", 20)
```

```r
as.numeric(v2)
```

```
## Warning: NAs introduced by coercion
## [1] 10 NA 20
```

Special Values in R

Inf and -Inf

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

```
120/0
```

```
## [1] Inf
```

```
-120/0
```

```
## [1] -Inf
```

Too big to show

```
45^12500
```

```
## [1] Inf
```

Special Values in R

NaN

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

```
0/0
```

```
## [1] NaN
```

Special Values in R

NULL

- There is a null object in R, represented by the symbol NULL.
- NULL is often used as an argument in functions to mean that no value was assigned to the argument.
- Some functions may return NULL.
- NULL is not the same as NA, Inf, -Inf, or NaN.
- NULL represents the null object in R.
- NULL is used mainly to represent the lists with zero length, and is often returned by expressions and functions whose value is undefined.

Section 5

Importing Data to R

Importing Data to R

Flat files

- A flat file database is a database that stores data in a plain text file.
- Each line of the text file holds one record with fields separated by delimiters, such as commas or tabs.
- While it uses a simple structure, a flat file database cannot contain multiple tables like a relational database can.

Importing Data to R

Delimiter-Separated Values

- Formats that use delimiter-separated values (also DSV) store two-dimensional arrays of data by separating the values in each row with specific delimiter characters.
- Most database and spreadsheet programs are able to read or save data in a delimited format.
- Most widely used delimiters are:
  - comma (CSV – comma separated values)
  - tab (TSV – tab separated values)

Importing Data to R

- File winter.csv contains data on winter Olympic games from 1924 to 2014.
- The file is comma-separated.
- Use readLines() with n=5 to look at the first five lines of the text file.
- You can see that the values are separated by comma.

```
readLines('Data/winter.csv', n=5)
```

```
## [1] "Year,City,Sport,Discipline,Athlete,Country,Gender,Event,Medal"
## [2] "1924,Chamonix,Biathlon,Biathlon,\"BERTHET, G.\",FRA,Men,Military Patrol,Bronze"
## [3] "1924,Chamonix,Biathlon,Biathlon,\"MANDRILLON, C.\",FRA,Men,Military Patrol,Bronze"
## [4] "1924,Chamonix,Biathlon,Biathlon,\"MANDRILLON, Maurice\",FRA,Men,Military Patrol,Bronze"
## [5] "1924,Chamonix,Biathlon,Biathlon,\"VANDELLE, André\",FRA,Men,Military Patrol,Bronze"
```

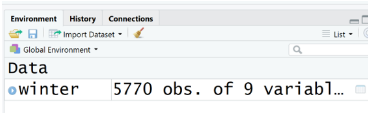
Importing Data to R

Use the function read.csv() to load the file into R environment

```
winter <- read.csv('Data/winter.csv')
str(winter)
```

```
## 'data.frame':    5770 obs. of  9 variables:
##  $ Year      : int  1924 1924 1924 1924 1924 1924 1924 1924 1924 1924 ...
##  $ City      : chr  "Chamonix" "Chamonix" "Chamonix" "Chamonix" ...
##  $ Sport     : chr  "Biathlon" "Biathlon" "Biathlon" "Biathlon" ...
##  $ Discipline: chr  "Biathlon" "Biathlon" "Biathlon" "Biathlon" ...
##  $ Athlete   : chr  "BERTHET, G." "MANDRILLON, C." "MANDRILLON, Maurice" "VANDELLE, André" ...
##  $ Country   : chr  "FRA" "FRA" "FRA" "FRA" ...
##  $ Gender    : chr  "Men" "Men" "Men" "Men" ...
##  $ Event     : chr  "Military Patrol" "Military Patrol" "Military Patrol" "Military Patrol" ...
##  $ Medal     : chr  "Bronze" "Bronze" "Bronze" "Bronze" ...
```

Be sure that you can see the new object in the environment. Otherwise you cannot work with it



Data

winter 5770 obs. of 9 variabl…

Importing Data to R

- Look the help for read.csv (?read.csv), several options on how the file is imported.
- stringsAsFactors = F – the strings are loaded as a text rather than as a factor.

```r
winter <- read.csv('Data/winter.csv', stringsAsFactors = F)
str(winter)
```

```
## 'data.frame':    5770 obs. of  9 variables:
##  $ Year      : int  1924 1924 1924 1924 1924 1924 1924 1924 1924 1924 ...
##  $ City      : chr  "Chamonix" "Chamonix" "Chamonix" "Chamonix" ...
##  $ Sport     : chr  "Biathlon" "Biathlon" "Biathlon" "Biathlon" ...
##  $ Discipline: chr  "Biathlon" "Biathlon" "Biathlon" "Biathlon" ...
##  $ Athlete   : chr  "BERTHET, G." "MANDRILLON, C." "MANDRILLON, Maurice" "VANDELLE, André" ...
##  $ Country   : chr  "FRA" "FRA" "FRA" "FRA" ...
##  $ Gender    : chr  "Men" "Men" "Men" "Men" ...
##  $ Event     : chr  "Military Patrol" "Military Patrol" "Military Patrol" "Military Patrol" ...
##  $ Medal     : chr  "Bronze" "Bronze" "Bronze" "Bronze" ...
```

Importing Data to R

- The same data is saved in tab delimited file.
- A tab-separated values (TSV) file is a simple text format for storing data in a tabular structure, e.g., database table or spreadsheet data, and a way of exchanging information between databases. Each record in the table is one line of the text file.

```
readLines('Data/winter.txt', n=5)
```

```
## [1] "\"Year\"\t\"City\"\t\"Sport\"\t\"Discipline\"\t\"Athlete\"\t\"Country\"\t\"Gender\"\t\"Event\"\t\"Medal\""
## [2] "\"1\"\t1924\t\"Chamonix\"\t\"Biathlon\"\t\"Biathlon\"\t\"BERTHET, G.\"\t\"FRA\"\t\"Men\"\t\"Military Patrol\"\t\"Bronze\""
## [3] "\"2\"\t1924\t\"Chamonix\"\t\"Biathlon\"\t\"Biathlon\"\t\"MANDRILLON, C.\"\t\"FRA\"\t\"Men\"\t\"Military Patrol\"\t\"Bronze\""
## [4] "\"3\"\t1924\t\"Chamonix\"\t\"Biathlon\"\t\"Biathlon\"\t\"MANDRILLON, Maurice\"\t\"FRA\"\t\"Men\"\t\"Military Patrol\"\t\"Bronz
## [5] "\"4\"\t1924\t\"Chamonix\"\t\"Biathlon\"\t\"Biathlon\"\t\"VANDELLE, André\"\t\"FRA\"\t\"Men\"\t\"Military Patrol\"\t\"Bronze\"
```

Importing Data to R

You can use the same read.csv() file but need to specify the separator (delimiter)

```r
winter <- read.csv('Data/winter.txt', sep="\t", stringsAsFactors = FALSE)
str(winter)
```

```
## 'data.frame':    5770 obs. of  9 variables:
##  $ Year      : int  1924 1924 1924 1924 1924 1924 1924 1924 1924 1924 ...
##  $ City      : chr  "Chamonix" "Chamonix" "Chamonix" "Chamonix" ...
##  $ Sport     : chr  "Biathlon" "Biathlon" "Biathlon" "Biathlon" ...
##  $ Discipline: chr  "Biathlon" "Biathlon" "Biathlon" "Biathlon" ...
##  $ Athlete   : chr  "BERTHET, G." "MANDRILLON, C." "MANDRILLON, Maurice" "VANDELLE, André" ...
##  $ Country   : chr  "FRA" "FRA" "FRA" "FRA" ...
##  $ Gender    : chr  "Men" "Men" "Men" "Men" ...
##  $ Event     : chr  "Military Patrol" "Military Patrol" "Military Patrol" "Military Patrol" ...
##  $ Medal     : chr  "Bronze" "Bronze" "Bronze" "Bronze" ...
```

Importing Data to R

R can work with other data files as well

- STATA files
- SPSS
- SAS
- MS Excel
- Databases
- XML
- HTML
- Etc.

Section 6

Exploring Data: Built-in Functions

Exploring Data: Overview

After importing data, the first step is always to **explore and understand** your data. R provides several built-in functions for this purpose:

Function	Purpose
`str()`	Structure of the object
`summary()`	Statistical summary
`head()` / `tail()`	First/last rows
`dim()`	Dimensions (rows, columns)
`nrow()` / `ncol()`	Number of rows/columns
`names()` / `colnames()`	Column names
`class()` / `typeof()`	Object type
`View()`	Interactive data viewer

str() - Structure of Data

str() displays the **internal structure** of an R object. It's one of the most useful functions for understanding your data.

```r
str(winter)
```

```
## 'data.frame':    5770 obs. of  9 variables:
##  $ Year      : int  1924 1924 1924 1924 1924 1924 1924 1924 1924 1924 ...
##  $ City      : chr  "Chamonix" "Chamonix" "Chamonix" "Chamonix" ...
##  $ Sport     : chr  "Biathlon" "Biathlon" "Biathlon" "Biathlon" ...
##  $ Discipline: chr  "Biathlon" "Biathlon" "Biathlon" "Biathlon" ...
##  $ Athlete   : chr  "BERTHET, G." "MANDRILLON, C." "MANDRILLON, Maurice" "VANDELLE, André" ...
##  $ Country   : chr  "FRA" "FRA" "FRA" "FRA" ...
##  $ Gender    : chr  "Men" "Men" "Men" "Men" ...
##  $ Event     : chr  "Military Patrol" "Military Patrol" "Military Patrol" "Military Patrol" ...
##  $ Medal     : chr  "Bronze" "Bronze" "Bronze" "Bronze" ...
```

What str() shows:

- Object type (data.frame, list, etc.)
- Number of observations and variables
- Column names and their data types
- First few values of each column

str() - Examples with Different Objects

```r
nums <- c(10, 20, 30, 40, 50)
str(nums)
```

```
##  num [1:5] 10 20 30 40 50
```

```r
my_list <- list(name = "John", age = 25, scores = c(85, 90, 88))
str(my_list)
```

```
## List of 3
##  $ name  : chr "John"
##  $ age   : num 25
##  $ scores: num [1:3] 85 90 88
```

```r
mat <- matrix(1:12, nrow = 3, ncol = 4)
str(mat)
```

```
##  int [1:3, 1:4] 1 2 3 4 5 6 7 8 9 10 ...
```

summary() - Statistical Summary

`summary()` provides a **statistical summary** appropriate for each column type.

```r
summary(winter)
```

```
##      Year          City               Sport            Discipline
##  Min.   :1924   Length:5770        Length:5770        Length:5770
##  1st Qu.:1972   Class :character   Class :character   Class :character
##  Median :1994   Mode  :character   Mode  :character   Mode  :character
##  Mean   :1986
##  3rd Qu.:2006
##  Max.   :2014
##    Athlete            Country             Gender             Event
##  Length:5770        Length:5770        Length:5770        Length:5770
##  Class :character   Class :character   Class :character   Class :character
##  Mode  :character   Mode  :character   Mode  :character   Mode  :character
##
##
##
##     Medal
##  Length:5770
##  Class :character
##  Mode  :character
##
##
##
```

summary() - Interpretation

For numeric columns, summary() shows:

- **Min** - minimum value
- **1st Qu** - first quartile (25th percentile)
- **Median** - middle value (50th percentile)
- **Mean** - arithmetic average
- **3rd Qu** - third quartile (75th percentile)
- **Max** - maximum value

For character/factor columns, summary() shows:

- Length (for character)
- Frequency counts (for factor)

```
summary(winter$Year)
```

```
##     Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     1924    1972    1994    1986    2006    2014
```

head() and tail() - First/Last Rows

`head()` shows the **first n rows** (default 6), `tail()` shows the **last n rows**.

```r
head(winter, n = 5)
```

```
##   Year     City    Sport Discipline              Athlete Country Gender
## 1 1924 Chamonix Biathlon   Biathlon         BERTHET, G.     FRA    Men
## 2 1924 Chamonix Biathlon   Biathlon      MANDRILLON, C.     FRA    Men
## 3 1924 Chamonix Biathlon   Biathlon  MANDRILLON, Maurice     FRA    Men
## 4 1924 Chamonix Biathlon   Biathlon     VANDELLE, André     FRA    Men
## 5 1924 Chamonix Biathlon   Biathlon AUFDENBLATTEN, Adolf     SUI    Men
##             Event  Medal
## 1 Military Patrol Bronze
## 2 Military Patrol Bronze
## 3 Military Patrol Bronze
## 4 Military Patrol Bronze
## 5 Military Patrol   Gold
```

```r
tail(winter, n = 3)
```

```
##      Year  City  Sport Discipline            Athlete Country Gender
## 5768 2014 Sochi Skiing   Snowboard MALTAIS, Dominique     CAN  Women
## 5769 2014 Sochi Skiing   Snowboard       SAMKOVA, Eva     CZE  Women
## 5770 2014 Sochi Skiing   Snowboard   TRESPEUCH, Chloe     FRA  Women
##                Event  Medal
## 5768 Snowboard Cross Silver
## 5769 Snowboard Cross   Gold
## 5770 Snowboard Cross Bronze
```

dim(), nrow(), ncol() - Dimensions

These functions tell you the **size** of your data.

```
dim(winter)
```

```
## [1] 5770    9
```

```
nrow(winter)
```

```
## [1] 5770
```

```
ncol(winter)
```

```
## [1] 9
```

```
length(c(1, 2, 3, 4, 5))
```

```
## [1] 5
```

names() and colnames() - Column Names

```
names(winter)
```

```
## [1] "Year"       "City"       "Sport"      "Discipline" "Athlete"
## [6] "Country"    "Gender"     "Event"      "Medal"
```

```
colnames(winter)
```

```
## [1] "Year"       "City"       "Sport"      "Discipline" "Athlete"
## [6] "Country"    "Gender"     "Event"      "Medal"
```

```
head(rownames(winter))
```

```
## [1] "1" "2" "3" "4" "5" "6"
```

class() and typeof() - Object Types

`class()` returns the **high-level class**, `typeof()` returns the **internal storage type**.

```
class(winter)
```

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

```
class(winter$Year)
```

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

```
class(winter$Sport)
```

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

```
typeof(winter)
```

```
## [1] "list"
```

```
typeof(winter$Year)
```

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

```
typeof(TRUE)
```

```
## [1] "logical"
```

Checking Object Types: is.* Functions

R provides many `is.*` functions to check object types:

```r
is.data.frame(winter)
```

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

```r
is.numeric(winter$Year)
```

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

```r
is.character(winter$Sport)
```

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

```r
is.vector(c(1, 2, 3))
```

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

```r
is.matrix(matrix(1:4, 2, 2))
```

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

```r
is.na(c(1, NA, 3))
```

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

unique() and duplicated() - Finding Unique Values

```r
unique(winter$Sport)
```

```
## [1] "Biathlon"   "Bobsleigh"  "Curling"    "Ice Hockey" "Skating"
## [6] "Skiing"     "Luge"
```

```r
length(unique(winter$Sport))
```

```
## [1] 7
```

unique() and duplicated() - continued

```
sample_data <- c("A", "B", "A", "C", "B")
duplicated(sample_data)
```

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

```
sample_data[duplicated(sample_data)]
```

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

```
unique(sample_data)
```

```
## [1] "A" "B" "C"
```

table() - Frequency Tables

`table()` creates **frequency counts** for categorical data.

```r
table(winter$Medal)
```

```
##
## Bronze   Gold Silver
##   1919   1921   1930
```

```r
head(table(winter$Country, winter$Medal), 10)
```

```
##
##       Bronze Gold Silver
##   AUS      7    5      3
##   AUT    103   79     98
##   BEL      7    2      4
##   BLR      5    6      4
##   BUL      3    1      2
##   CAN    107  315    203
##   CHN     36   16     30
##   CRO      1    4      6
##   CZE     35   28     12
##   DEN      0    0      5
```

which() - Finding Positions

which() returns the **indices** where a condition is TRUE.

```
gold_positions <- which(winter$Medal == "Gold")
head(gold_positions, 10)
```

```
##  [1]  5  6  7  8 18 19 20 21 32 33
```

```
scores <- c(75, 92, 88, 95, 81)
which.max(scores)
```

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

```
which.min(scores)
```

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

```
scores[which.max(scores)]
```

```
## [1] 95
```

```
max(scores)
```

```
## [1] 95
```

View() - Interactive Data Viewer

`View()` opens an **interactive spreadsheet-like viewer** in RStudio.

```
View(winter)
```

View() features:

- Sortable columns (click column headers)
- Searchable (filter box)
- Navigate large datasets easily
- Read-only (cannot edit data)

Practical Example: Exploring a Dataset

```
data(mtcars)



cat("Dimensions:", dim(mtcars), "\n")
```

```
## Dimensions: 32 11
```

```
cat("Column names:", paste(names(mtcars), collapse = ", "), "\n")
```

```
## Column names: mpg, cyl, disp, hp, drat, wt, qsec, vs, am, gear, carb
```

```
cat("Missing values:", sum(is.na(mtcars)), "\n")
```

```
## Missing values: 0
```

```
summary(mtcars[, c("mpg", "hp", "wt")])
```

```
##       mpg              hp              wt
##  Min.   :10.40   Min.   : 52.0   Min.   :1.513
##  1st Qu.:15.43   1st Qu.: 96.5   1st Qu.:2.581
##  Median :19.20   Median :123.0   Median :3.325
##  Mean   :20.09   Mean   :146.7   Mean   :3.217
##  3rd Qu.:22.80   3rd Qu.:180.0   3rd Qu.:3.610
##  Max.   :33.90   Max.   :335.0   Max.   :5.424
```

Summary: Essential Exploration Functions

Task	Function	Example
See structure	`str()`	`str(df)`
Statistical summary	`summary()`	`summary(df)`
First rows	`head()`	`head(df, 10)`
Last rows	`tail()`	`tail(df, 5)`
Dimensions	`dim()`	`dim(df)`
Column names	`names()`	`names(df)`
Object type	`class()`	`class(df)`
Unique values	`unique()`	`unique(df$col)`
Frequencies	`table()`	`table(df$col)`
Find positions	`which()`	`which(df$col > 5)`
Interactive view	`View()`	`View(df)`

Section 7

Subsetting Data

Subsetting Data: Vectors

Create named vector

```r
Club <- c("Juventus", "Napoli", "Roma", "Inter")
Points <- c(95,91,77,72)
names(Points) <- Club
Points
```

```
## Juventus   Napoli     Roma    Inter
##       95       91       77       72
```

Subsetting Data: Vectors

- In R (unlike Python) the indexing starts with 1.
- Subsetting by index location.
- The vector has one dimension, so to subset it you need to specify the location of 1 index only.

```
Points[2]
```

```
## Napoli
##     91
```

Subset by name

```
Points["Juventus"]
```

```
## Juventus
##       95
```

Subsetting Data: Vectors

- Subset several elements
- First two elements

```
Points[1:2]
```

```
## Juventus   Napoli
##       95       91
```

- 1 and 3 elements

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

```
## Juventus     Roma
##       95       77
```

- Subsetting by name

```
Points[c("Napoli","Inter")]
```

```
## Napoli  Inter
##     91     72
```

Subsetting Data: Vectors

Why R you getting an eRRoR ?

```
Club[1,2]
```

```
## Error in `Club[1, 2]`:
## ! incorrect number of dimensions
```

```
Club[2,4]
```

```
## Error in `Club[2, 4]`:
## ! incorrect number of dimensions
```

Subsetting Data: Data frames

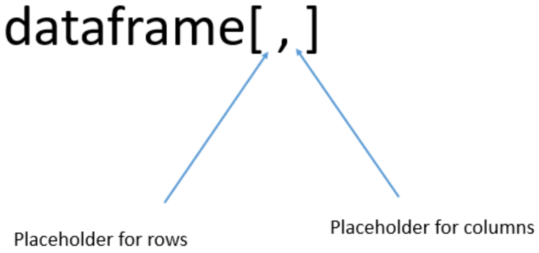
Load nba dataset

```r
load('Data/nba2009_2018.rda')
summary(nba2009_2018)
```

```
##   SEASON_ID           GAME_DATE          home.TEAM_ABBREVIATION
##  Length:12060       Min.   :2009-10-27   Length:12060
##  Class :character   1st Qu.:2012-03-05   Class :character
##  Mode  :character   Median :2014-11-13   Mode  :character
##                     Mean   :2014-08-10
##                     3rd Qu.:2017-01-24
##                     Max.   :2019-04-10
##  home.TEAM_NAME        home.PTS     away.TEAM_ABBREVIATION away.TEAM_NAME
##  Length:12060       Min.   :  0.0   Length:12060           Length:12060
##  Class :character   1st Qu.: 95.0   Class :character       Class :character
##  Mode  :character   Median :103.0   Mode  :character       Mode  :character
##                     Mean   :103.6
##                     3rd Qu.:112.0
##                     Max.   :161.0
##     away.PTS       home.WL
##  Min.   :  0.0   Length:12060
##  1st Qu.: 92.0   Class :character
##  Median :100.0   Mode  :character
##  Mean   :100.9
##  3rd Qu.:109.0
##  Max.   :168.0
```

Subsetting Data: Data frames

Data frame has two dimensions: Rows (first dimension) and Columns (second dimension)



Subsetting Data: Data frames

Will return the value on first row and forth column

```
dataframe[1, 4]
```

Will return the value of the first 10 rows and columns 4,5,6

```
dataframe[1:10, 4:6]
```

Subsetting Data: Data frames

Will return the first 10 rows and all columns

```
dataframe[1:10, ]
```

If you want to select all elements for the given index, just leave the placeholder empty.

What will this command return?

```
dataframe[, c(2,4:6)]
```

Subsetting Data: Data frames

Indexing by column names

```
colnames(nba2009_2018)
```

```
## [1] "SEASON_ID"              "GAME_DATE"              "home.TEAM_ABBREVIATION"
## [4] "home.TEAM_NAME"         "home.PTS"               "away.TEAM_ABBREVIATION"
## [7] "away.TEAM_NAME"         "away.PTS"               "home.WL"
```

```
nba1 <- nba2009_2018[,c("home.PTS", "away.PTS")]
colnames(nba1)
```

```
## [1] "home.PTS" "away.PTS"
```

```
dim(nba1)
```

```
## [1] 12060     2
```

Subsetting Data: Data frames

- Negative indexing is used to exclude certain records from the dataframe.
- This does not work with column names indexing.

```
nba2 <- nba2009_2018[,-c(1,2,4:6)]
head(nba2)
```

```
##   home.TEAM_ABBREVIATION     away.TEAM_NAME away.PTS home.WL
## 1                    CLE     Boston Celtics       95       L
## 2                    DAL Washington Wizards      102       L
## 3                    POR    Houston Rockets       87       W
## 4                    LAL         LA Clippers       92       W
## 5                    ATL     Indiana Pacers      109       W
## 6                    ORL Philadelphia 76ers      106       W
```

Subsetting Data: Data frames

Exercises

- Create new data frame from nba dataset
- Include first 100 rows and columns 2, 3, 5
- Exclude rows 250, 300 to 350 and exclude column 5

Subsetting Data: Data frames

You can access specific column in data frame by using dollar sign

```
mean(nba2009_2018$home.PTS)
```

```
## [1] 103.6102
```

```
table(nba2009_2018$home.WL)
```

```
##
##    L    W
## 4949 7110
```

Subsetting Data: Data frames

Subset one variable

- First approach gives you a dataframe with one columns
- Second gives you a vector

```
home_pts <- nba2009_2018['home.PTS']
str(home_pts)
```

```
## 'data.frame':    12060 obs. of  1 variable:
##  $ home.PTS: num  89 91 96 99 120 120 101 92 115 74 ...
```

```
home_pts <- nba2009_2018[,'home.PTS']
str(home_pts)
```

```
##  num [1:12060] 89 91 96 99 120 120 101 92 115 74 ...
```

Subsetting Data: Data frames

Conditional indexing

- Create new dataframe with games only from season 2009
- We need all the rows where the value for SEASON_ID is 2009
- Note that the type for column SEASON_ID is character

```r
nba4 <- nba2009_2018[nba2009_2018$SEASON_ID=='2009',]
```

Check if everything is done right

```r
table(nba4$SEASON_ID)
```

```
##
## 2009
## 1230
```

Subsetting Data: Data frames

- Take only seasons 2009 and 2010
- As the SEASON_ID is character we will do the following

```
nba5 <- nba2009_2018[nba2009_2018$SEASON_ID %in% c("2009", "2010"),]
table(nba5$SEASON_ID)
```

```
##
## 2009 2010
## 1230 1230
```

Subsetting Data: Data frames

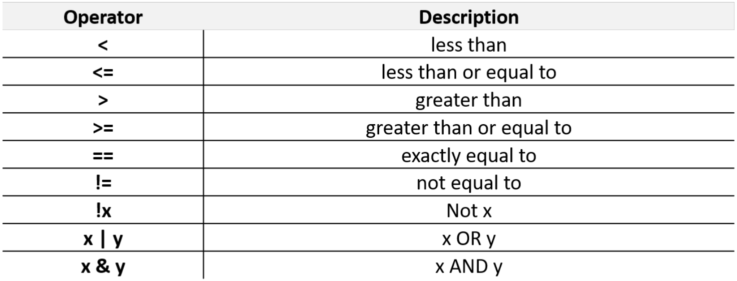
- Workaround: make SEASON_ID numeric vector

```
nba2009_2018$SEASON_ID <- as.numeric(nba2009_2018$SEASON_ID)
nba5 <- nba2009_2018[nba2009_2018$SEASON_ID < 2011,]
table(nba5$SEASON_ID)
```

```
##
## 2009 2010
## 1230 1230
```

Subsetting Data: Data frames

Logical operators in R



Operator	Description
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	exactly equal to
!=	not equal to
!x	Not x
x | y	x OR y
x & y	x AND y

Subsetting Data: Data frames

- Indexing on multiple conditions
- Take all the home game records for Detroit Pistons for seasons 2010 and 2011

```
nba6 <- nba2009_2018[nba2009_2018$SEASON_ID %in% c(2010,2011) &
                          nba2009_2018$home.TEAM_NAME =="Detroit Pistons",]

table(nba6$home.TEAM_NAME, nba6$SEASON_ID)
```

```
##
##                  2010 2011
##   Detroit Pistons   41   33
```

Subsetting Data: Data frames

Using OR (|) operator in R

```
nba7 <- nba2009_2018[nba2009_2018$away.TEAM_NAME=="Detroit Pistons" |
                           nba2009_2018$home.TEAM_NAME =="Detroit Pistons",]

head(nba7)
```

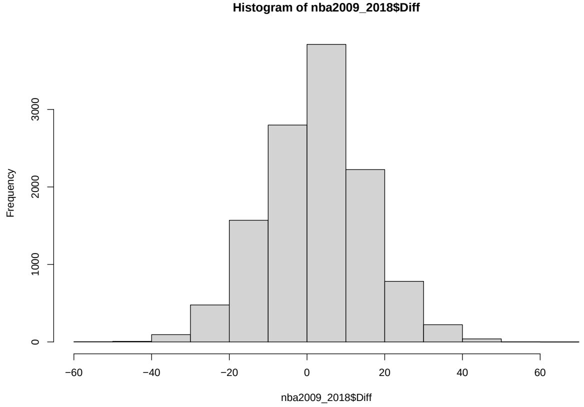
```
##    SEASON_ID  GAME_DATE home.TEAM_ABBREVIATION    home.TEAM_NAME home.PTS
## 10      2009 2009-10-28                    MEM Memphis Grizzlies       74
## 23      2009 2009-10-30                    DET   Detroit Pistons       83
## 36      2009 2009-10-31                    MIL   Milwaukee Bucks       96
## 54      2009 2009-11-03                    DET   Detroit Pistons       85
## 61      2009 2009-11-04                    TOR   Toronto Raptors      110
## 74      2009 2009-11-06                    ORL     Orlando Magic      110
##    away.TEAM_ABBREVIATION        away.TEAM_NAME away.PTS home.WL
## 10                    DET       Detroit Pistons       96       L
## 23                    OKC Oklahoma City Thunder       91       L
## 36                    DET       Detroit Pistons       85       W
## 54                    ORL         Orlando Magic       80       W
## 61                    DET       Detroit Pistons       99       W
## 74                    DET       Detroit Pistons      103       W
```

Subsetting Data: Data frames

Adding new variable in dataframe

- Point differential

```
nba2009_2018$Diff <- nba2009_2018$home.PTS-nba2009_2018$away.PTS
hist(nba2009_2018$Diff)
```



Indexing List

```
(seriea <- data.frame(Club, Points))
```

```
##              Club Points
## Juventus Juventus     95
## Napoli     Napoli     91
## Roma         Roma     77
## Inter       Inter     72
```

- We have created a dataframe. Note that the dataframe has rownames
- Create a list

```
(list1 <- list(Club, "Italy", 2017, seriea))
```

```
## [[1]]
## [1] "Juventus" "Napoli"   "Roma"     "Inter"
##
## [[2]]
## [1] "Italy"
##
## [[3]]
## [1] 2017
##
## [[4]]
##              Club Points
## Juventus Juventus     95
## Napoli     Napoli     91
## Roma         Roma     77
## Inter       Inter     72
```

Indexing List

- If you index the list with [], the result is a list.
- If you index the list with [[]] the result has the same structure as the list element

```
list1[4]
```

```
## [[1]]
##              Club Points
## Juventus Juventus     95
## Napoli     Napoli     91
## Roma         Roma     77
## Inter       Inter     72
```

```
list1[[4]]
```

```
##              Club Points
## Juventus Juventus     95
## Napoli     Napoli     91
## Roma         Roma     77
## Inter       Inter     72
```

```
is.data.frame(list1[[4]])
```

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

```
is.data.frame(list1[4])
```

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

Indexing List

Two elements

```
list1[1:2]
```

```
## [[1]]
## [1] "Juventus" "Napoli"   "Roma"     "Inter"
##
## [[2]]
## [1] "Italy"
```

Slicing element from the element

This will bring the second element from the first element in the list

```
list1[[1]][2]
```

```
## [1] "Napoli"
```

Indexing List

Named elements in the list

```
list1 <- list(Teams = Club, country = "Italy",
              year = 2017, standings = seriea)
list1
```

```
## $Teams
## [1] "Juventus" "Napoli"   "Roma"     "Inter"
##
## $country
## [1] "Italy"
##
## $year
## [1] 2017
##
## $standings
##              Club Points
## Juventus Juventus     95
## Napoli     Napoli     91
## Roma         Roma     77
## Inter       Inter     72
```

Indexing List

Indexing by name

```
list1$standings
```

```
##              Club Points
## Juventus Juventus     95
## Napoli     Napoli     91
## Roma         Roma     77
## Inter       Inter     72
```

```
list1$year
```

```
## [1] 2017
```

Section 8

Functions

Functions

- In programming, you use functions to incorporate sets of instructions that you want to use repeatedly or that, because of their complexity, are better self-contained in a sub program and called when needed.
- A function is a piece of code written to carry out a specified task; it can or can not accept arguments or parameters and it can or can not return one or more values.
- In fact, there are several possible formal definitions of ‘function’ spanning from mathematics to computer science.
- **Genericaly, its arguments constitute the input and return values as output.**

Functions

- Functions have named arguments which potentially have default values.
- The formal arguments are the arguments included in the function definition.
- The formals function returns a list of all the formal arguments of a function.
- Not every function call in R makes use of all the formal arguments.
- Function arguments can be missing or might have default values.

Functions

The function runif() generates random numbers from continuous uniform distribution

Arguments are:

- n, number of observations, with no default - so needs to be specified
- min, the minimum
- max, the maximum

```r
formals(runif)
```

```
## $n
##
##
## $min
## [1] 0
##
## $max
## [1] 1
```

```r
args(runif)
```

```
## function (n, min = 0, max = 1)
## NULL
```

Functions

R functions' arguments can be matched **positionally** or **by name**. So the following calls to runif() are all equivalent (you can mix them as well).

- Matching by name

```
runif(min=1, n=10, max=2)
```

```
##  [1] 1.613527 1.771101 1.129534 1.750953 1.828312 1.264977 1.063733 1.083328
##  [9] 1.579575 1.395055
```

- Matching by position

```
runif(10,1,2)
```

```
##  [1] 1.780398 1.418638 1.828284 1.572352 1.022267 1.773040 1.634749 1.159017
##  [9] 1.268472 1.546173
```

Note, as the process is random you might get different results.

Functions

Matched by position, this will give an error

```
runif(1,10,2)
```

```
## Warning in runif(1, 10, 2): NAs produced
## [1] NaN
```

Functions

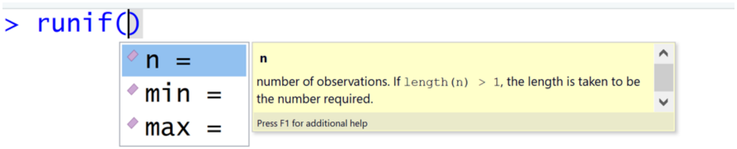
- If the argument has a default value and is not defined then the default value is used.
- Generated 10 numbers with min=0, max=1

```r
runif(10)
```

```
##  [1] 0.67661014 0.65872349 0.10309254 0.31487621 0.08606530 0.98953927
##  [7] 0.78064446 0.06508086 0.99577982 0.31792134
```

Functions

You can look at the arguments of the function and their description by hitting **Tab** inside the brackets



```
> runif()
```

n

number of observations. If `length(n) > 1`, the length is taken to be the number required.

Press F1 for additional help

Functions

General rule: if you have to copy a code more than twice, write a function.

```
f <- function(a, b = 1, c = 2, d = NULL) {
what needs to be done
}
```

In addition to not specifying a default value, you can also set an argument value to NULL.

Functions

- User-defined functions are stored in the global environment and can be accessed easily.
- The following function will calculate the x power of y (default value of y is 2).

```r
foo <- function (x, y=2){
  x^y
}

foo(4)
```

```
## [1] 16
```

Functions

Arguments to functions are evaluated lazily, so they are evaluated only as needed.

```
foo <- function (x,y){
  x^2
}

foo(3)
```

```
## [1] 9
```

This function never actually uses the argument y, so calling f(2) will not produce an error because the 2 gets positionally matched to x.

Functions

- The return value of a function is the last expression in the function body to be evaluated.
- It can be also specified with the function return()

Functions

The following function will calculate the z-score of a vector

```
norm <- function(x) {
  return((x-mean(x))/sd(x))
}

norm(mtcars$mpg)
```

```
##  [1]  0.15088482  0.15088482  0.44954345  0.21725341 -0.23073453 -0.33028740
##  [7] -0.96078893  0.71501778  0.44954345 -0.14777380 -0.38006384 -0.61235388
## [13] -0.46302456 -0.81145962 -1.60788262 -1.60788262 -0.89442035  2.04238943
## [19]  1.71054652  2.29127162  0.23384555 -0.76168319 -0.81145962 -1.12671039
## [25] -0.14777380  1.19619000  0.98049211  1.71054652 -0.71190675 -0.06481307
## [31] -0.84464392  0.21725341
```

Functions

Different ways of assigning values to variables inside functions, when using <- the value of the argument will be stored inside the variable in the environment.

```
x <- c(1,2,3)
(median(x = (1:10)))
```

```
## [1] 5.5
```

```
(x)
```

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

```
(median(x <- (1:10)))
```

```
## [1] 5.5
```

```
(x)
```

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

Section 9

Dates and Times in R

Dates in R

- R provides several options for dealing with date and date/time data.
- The built in function as.Date() handles dates (without times)
- Library chron handles dates and times, but does not control for time zones
- The POSIXct and POSIXlt classes allow for dates and times with control for time zones.
- The general rule for date/time data in R is to use the simplest technique possible.
- Thus, for date only data, as.Date() will usually be the best choice. If you need to handle dates and times without time-zone information, the chron library is a good choice; the POSIX classes are especially useful when time-zone manipulation is important.

Dates in R

- Except for the **POSIXlt** class, dates are stored internally as the number of days or seconds from some reference date.
- The function as.Date() stores the number of days passed from 1970-01-01

```
x <- as.Date("2018-06-18")
class(x)
```

```
## [1] "Date"
```

```
as.numeric(x)
```

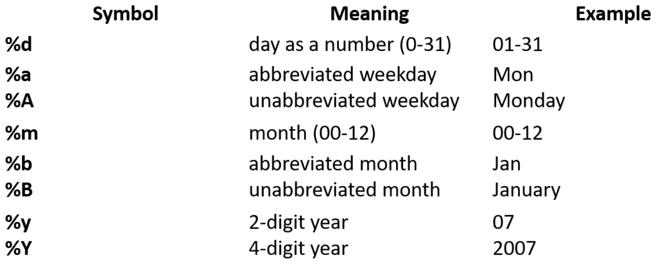
```
## [1] 17700
```

```
as.numeric(as.Date("1970-01-01"))
```

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

Dates in R

- The date can come in different formats.
- Usually, you need to tell R in which format the date is.



Symbol	Meaning	Example
%d	day as a number (0-31)	01-31
%a	abbreviated weekday	Mon
%A	unabbreviated weekday	Monday
%m	month (00-12)	00-12
%b	abbreviated month	Jan
%B	unabbreviated month	January
%y	2-digit year	07
%Y	4-digit year	2007

```r
as.Date("2018/01/15", format = "%Y/%m/%d")
```

```
## [1] "2018-01-15"
```

```r
as.Date("01-15-2018", format = "%m-%d-%Y")
```

```
## [1] "2018-01-15"
```

Dates in R

```r
oil <- read.csv("Data/oil.csv", stringsAsFactors = FALSE)
head(oil)
```

```
##       DATE   OPEN   HIGH    LOW  CLOSE   VOL
## 1 1-Oct-12 112.14 113.27 110.76 111.40 80055
## 2 2-Oct-12 111.40 111.70 110.55 110.55 29332
## 3 3-Oct-12 110.55 110.59 106.95 107.34 56307
## 4 4-Oct-12 107.44 111.79 107.24 111.36 61664
## 5 5-Oct-12 111.27 112.09 109.64 111.13 51704
## 6 7-Oct-12 111.11 111.13 111.02 111.02    57
```

```r
str(oil)
```

```
## 'data.frame':    1643 obs. of  6 variables:
##  $ DATE : chr  "1-Oct-12" "2-Oct-12" "3-Oct-12" "4-Oct-12" ...
##  $ OPEN : num  112 111 111 107 111 ...
##  $ HIGH : num  113 112 111 112 112 ...
##  $ LOW  : num  111 111 107 107 110 ...
##  $ CLOSE: num  111 111 107 111 111 ...
##  $ VOL  : int  80055 29332 56307 61664 51704 57 40638 61077 68836 68708 ...
```

Dates in R

Format the DATE column with as.Date

```
oil$DATE<-as.Date(oil$DATE, format="%d-%b-%y")
str(oil)
```

```
## 'data.frame':    1643 obs. of  6 variables:
##  $ DATE : Date, format: "2012-10-01" "2012-10-02" ...
##  $ OPEN : num  112 111 111 107 111 ...
##  $ HIGH : num  113 112 111 112 112 ...
##  $ LOW  : num  111 111 107 107 110 ...
##  $ CLOSE: num  111 111 107 111 111 ...
##  $ VOL  : int  80055 29332 56307 61664 51704 57 40638 61077 68836 68708 ...
```

Dates in R

- Now we can subset the data frame with DATE column.
- The following code is going to create a data frame with cases only from 2014

```
oil_2014 <- oil[oil$DATE >= "2014-01-01" & oil$DATE < "2015-01-01",]
head(oil_2014)
```

```
##           DATE   OPEN   HIGH    LOW  CLOSE   VOL
## 389 2014-01-02 110.74 111.07 107.49 107.65 32977
## 390 2014-01-03 107.66 108.42 106.44 106.65 35605
## 391 2014-01-06 106.68 107.60 106.28 106.67 35795
## 392 2014-01-07 106.73 107.33 106.56 107.01 30686
## 393 2014-01-08 107.08 107.54 106.60 106.83 36728
## 394 2014-01-09 106.83 107.71 105.70 106.00 55777
```

Dates in R

You can do arithmetic operations with class Date

```
as.Date("2018-06-18")+1
```

```
## [1] "2018-06-19"
```

```
as.Date("2018-06-18")-as.Date("2018-05-18")
```

```
## Time difference of 31 days
```

Dates in R

Extracting weekday

```
oil$WEEKDAY <- weekdays(oil$DATE)
head(oil)
```

```
##         DATE   OPEN   HIGH    LOW  CLOSE   VOL   WEEKDAY
## 1 2012-10-01 112.14 113.27 110.76 111.40 80055    Monday
## 2 2012-10-02 111.40 111.70 110.55 110.55 29332   Tuesday
## 3 2012-10-03 110.55 110.59 106.95 107.34 56307 Wednesday
## 4 2012-10-04 107.44 111.79 107.24 111.36 61664  Thursday
## 5 2012-10-05 111.27 112.09 109.64 111.13 51704    Friday
## 6 2012-10-07 111.11 111.13 111.02 111.02    57    Sunday
```

Dates in R

Extract month

```
oil$MONTH <- months(oil$DATE)
head(oil)
```

```
##         DATE   OPEN   HIGH    LOW  CLOSE   VOL   WEEKDAY   MONTH
## 1 2012-10-01 112.14 113.27 110.76 111.40 80055    Monday October
## 2 2012-10-02 111.40 111.70 110.55 110.55 29332   Tuesday October
## 3 2012-10-03 110.55 110.59 106.95 107.34 56307 Wednesday October
## 4 2012-10-04 107.44 111.79 107.24 111.36 61664  Thursday October
## 5 2012-10-05 111.27 112.09 109.64 111.13 51704    Friday October
## 6 2012-10-07 111.11 111.13 111.02 111.02    57    Sunday October
```

Dates in R

- To extract the day of the month, use format()
- The returned column is character, but you can make it numeric

```r
oil$DAY <- format(oil$DATE, "%d")
str(oil)
```

```
## 'data.frame':    1643 obs. of  9 variables:
##  $ DATE   : Date, format: "2012-10-01" "2012-10-02" ...
##  $ OPEN   : num  112 111 111 107 111 ...
##  $ HIGH   : num  113 112 111 112 112 ...
##  $ LOW    : num  111 111 107 107 110 ...
##  $ CLOSE  : num  111 111 107 111 111 ...
##  $ VOL    : int  80055 29332 56307 61664 51704 57 40638 61077 68836 68708 ...
##  $ WEEKDAY: chr  "Monday" "Tuesday" "Wednesday" "Thursday" ...
##  $ MONTH  : chr  "October" "October" "October" "October" ...
##  $ DAY    : chr  "01" "02" "03" "04" ...
```

Dates in R

Make the variable numeric

```
oil$DAY <- as.numeric(oil$DAY)
head(oil)
```

```
##         DATE   OPEN   HIGH    LOW  CLOSE   VOL   WEEKDAY   MONTH DAY
## 1 2012-10-01 112.14 113.27 110.76 111.40 80055    Monday October   1
## 2 2012-10-02 111.40 111.70 110.55 110.55 29332   Tuesday October   2
## 3 2012-10-03 110.55 110.59 106.95 107.34 56307 Wednesday October   3
## 4 2012-10-04 107.44 111.79 107.24 111.36 61664  Thursday October   4
## 5 2012-10-05 111.27 112.09 109.64 111.13 51704    Friday October   5
## 6 2012-10-07 111.11 111.13 111.02 111.02    57    Sunday October   7
```

Section 10

Data Transformation: dplyr

Grammar of data manipulation

dplyr is a grammar of data manipulation, providing a consistent set of verbs that help you solve the most common data manipulation challenges:

- **mutate()** adds new variables that are functions of existing variables
- **select()** picks variables based on their names.
- **filter()** picks cases based on their values.
- **summarise()** reduces multiple values down to a single summary.
- **arrange()** changes the ordering of the rows.

These all combine naturally with group_by() which allows you to perform any operation “by group”.

Grammar of data manipulation

- Not necessary, but dplyr works the best with pipe like operator from **magrittr** package
- %>% operator takes the object from its left-hand side and use it as an argument in the function on the right-hand side

Grammar of data manipulation

- The "pipe" operation is a handy tool to make your code more legible: %>%.

Key points:

- It takes the output of your previous operation and uses it as an input to your next operation.
- You can determine where the previous argument goes with the period symbol, . , which acts as a placeholder.
- Understand how to use it by replacing the pipe operation with "then" (in your mind, not in the code). For example, **filter(data, ...) %>% select(...)** filters first then selects columns from the output of filter.

Grammar of data manipulation

Filtering USA games only

```
summer <- read.csv("Data/summer.csv", stringsAsFactors = F)
```

```
summer_usa <- summer %>% filter(Country=="USA")
```

```
table(summer_usa$Country, summer_usa$Medal)
```

```
##
##       Bronze Gold Silver
##   USA   1098 2235   1252
```

Grammar of data manipulation

Filter and group_by

```
summer %>%
  filter(Country %in% c("USA", "FRA", "GBR")) %>%
  group_by(Country) %>%
  summarise(Count=n())
```

```
## # A tibble: 3 x 2
##   Country Count
##   <chr>   <int>
## 1 FRA      1396
## 2 GBR      1720
## 3 USA      4585
```

- The result is a tibble
- Count=n() creates a new variable named Count with frequencies, n() calculates frequencies

Grammar of data manipulation

Number of medals by country

```
summer %>%
  filter(Country %in% c("USA", "FRA", "GBR")) %>%
  group_by(Country, Medal) %>%
  summarise(Count=n())
```

```
## # A tibble: 9 x 3
## # Groups:   Country [3]
##   Country Medal  Count
##   <chr>   <chr>  <int>
## 1 FRA     Bronze   497
## 2 FRA     Gold     408
## 3 FRA     Silver   491
## 4 GBR     Bronze   553
## 5 GBR     Gold     546
## 6 GBR     Silver   621
## 7 USA     Bronze  1098
## 8 USA     Gold    2235
## 9 USA     Silver  1252
```

Section 11

Summary Functions

Summary Functions: Overview

Summary functions take a vector of values and return a single value. They are essential for understanding your data.

Common summary functions:

- `mean()` - arithmetic average
- `median()` - middle value
- `min()` / `max()` - minimum and maximum
- `sum()` - total of all values
- `sd()` / `var()` - standard deviation and variance
- `n()` - count (in dplyr context)
- `first()` / `last()` - first and last values
- `quantile()` - percentiles

Summary Functions: Basic Examples

```r
scores <- c(85, 92, 78, 95, 88, 72, 90, 85, 91, 76)
```

```r
(mean(scores))
```

```
## [1] 85.2
```

```r
(median(scores))
```

```
## [1] 86.5
```

```r
(sd(scores))
```

```
## [1] 7.583608
```

```r
(var(scores))
```

```
## [1] 57.51111
```

Mean vs Median: Key Differences

Mean (Arithmetic Average)

- Calculated by summing all values and dividing by the count
- Formula: $\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}$
- **Sensitive to outliers** - extreme values pull the mean toward them
- Best for symmetric, normally distributed data

Median (Middle Value)

- The value that separates the higher half from the lower half
- **Robust to outliers** - extreme values don't affect it much
- Best for skewed data or when outliers are present

Mean vs Median: Outlier Sensitivity

```r
salaries <- c(45, 52, 48, 55, 50, 47, 53, 49, 51, 46)

mean(salaries)
```

```
## [1] 49.6
```

```r
median(salaries)
```

```
## [1] 49.5
```

```r
salaries_with_ceo <- c(salaries, 500)

mean(salaries_with_ceo)
```

```
## [1] 90.54545
```

```r
median(salaries_with_ceo)
```

```
## [1] 50
```

When to Use Mean vs Median

Situation	Use Mean	Use Median
Symmetric data	Yes	Yes
Skewed data	No	**Yes**
Data with outliers	No	**Yes**
Income/salary data	No	**Yes**
Test scores (no outliers)	**Yes**	Yes
Scientific measurements	**Yes**	Sometimes

Rule of thumb: If mean and median are very different, your data is likely skewed or has outliers. In such cases, prefer the median.

```r
mean(salaries_with_ceo) - median(salaries_with_ceo)
```

```
## [1] 40.54545
```

Other Measures of Central Tendency

Mode - The most frequently occurring value

- R doesn't have a built-in mode function for general use
- Useful for categorical data

```
get_mode <- function(x) {
  uniq_x <- unique(x)
  uniq_x[which.max(tabulate(match(x, uniq_x)))]
}

shoe_sizes <- c(9, 10, 10, 11, 10, 9, 10, 12, 10, 11)
get_mode(shoe_sizes)
```

```
## [1] 10
```

Trimmed Mean - Mean after removing extreme values

```
mean(salaries_with_ceo, trim = 0.1)
```

```
## [1] 50.11111
```

Summary Functions: Range and Extremes

```
min(scores)
```

```
## [1] 72
```

```
max(scores)
```

```
## [1] 95
```

```
range(scores)
```

```
## [1] 72 95
```

```
sum(scores)
```

```
## [1] 852
```

```
length(scores)
```

```
## [1] 10
```

Understanding Quantiles

What are quantiles?
Quantiles divide your data into equal-sized groups when sorted from smallest to largest.

Common quantile types:

- **Quartiles** (4 groups): Q1 (25%), Q2 (50% = median), Q3 (75%)
- **Percentiles** (100 groups): e.g., 10th percentile, 90th percentile
- **Deciles** (10 groups): 10%, 20%, 30%, etc.

Interpretation:

- Q1 (25th percentile): 25% of data falls below this value
- Q2 (50th percentile): 50% of data falls below (this is the median)
- Q3 (75th percentile): 75% of data falls below this value

Quantiles in R: Basic Usage

```r
exam_scores <- c(55, 62, 67, 70, 72, 75, 78, 80,
                 82, 85, 88, 90, 92, 95, 98)
```

```r
quantile(exam_scores)
```

```
##   0%  25%  50%  75% 100%
##   55   71   80   89   98
```

```r
quantile(exam_scores, probs = c(0.10, 0.90))
```

```
##  10%  90%
## 64.0 93.8
```

```r
quantile(exam_scores, probs = seq(0, 1, by = 0.1))
```

```
##   0%  10%  20%  30%  40%  50%  60%  70%  80%  90% 100%
## 55.0 64.0 69.4 72.6 76.8 80.0 83.2 87.4 90.4 93.8 98.0
```

Quantiles: Practical Applications

Use cases for quantiles:

- **Grading on a curve:** Top 10% get an A
- **Income analysis:** What salary puts you in the top 25%?
- **Performance benchmarks:** Is this value above the 90th percentile?

```
top_25_cutoff <- quantile(exam_scores, 0.75)
top_25_cutoff
```

```
## 75%
##  89
```

```
exam_scores[exam_scores >= top_25_cutoff]
```

```
## [1] 90 92 95 98
```

```
mean(exam_scores <= 80)
```

```
## [1] 0.5333333
```

Interquartile Range (IQR)

The **Interquartile Range (IQR)** measures the spread of the middle 50% of your data.

$$IQR = Q3 - Q1$$

```r
Q1 <- quantile(exam_scores, 0.25)
Q3 <- quantile(exam_scores, 0.75)

Q3 - Q1
```

```
## 75%
##  18
```

```r
IQR(exam_scores)
```

```
## [1] 18
```

Detecting Outliers with IQR

The **IQR method** is a standard approach for identifying outliers:
Outlier boundaries:

- **Lower bound:** $Q1 - 1.5 \times IQR$
- **Upper bound:** $Q3 + 1.5 \times IQR$

Any value below the lower bound or above the upper bound is considered an outlier.

```r
Q1 <- quantile(exam_scores, 0.25)
Q3 <- quantile(exam_scores, 0.75)
iqr_value <- IQR(exam_scores)
lower_bound <- Q1 - 1.5 * iqr_value
upper_bound <- Q3 + 1.5 * iqr_value
cat("Lower bound:", lower_bound, "\n")
```

```
## Lower bound: 44
```

```r
cat("Upper bound:", upper_bound, "\n")
```

```
## Upper bound: 116
```

Detecting Outliers: Practical Example

```r
house_prices <- c(150, 180, 195, 210, 225, 240,
                  255, 270, 285, 300,
                  315, 330, 350, 380, 950)
```

```r
Q1 <- quantile(house_prices, 0.25)
Q3 <- quantile(house_prices, 0.75)
iqr_val <- IQR(house_prices)

lower <- Q1 - 1.5 * iqr_val
upper <- Q3 + 1.5 * iqr_val

cat("Q1:", Q1, "| Q3:", Q3, "| IQR:", iqr_val, "\n")
```

```
## Q1: 217.5 | Q3: 322.5 | IQR: 105
```

```r
cat("Lower bound:", lower, "| Upper bound:", upper, "\n")
```

```
## Lower bound: 60 | Upper bound: 480
```

Creating an Outlier Detection Function

```r
find_outliers <- function(x, multiplier = 1.5) {
  Q1 <- quantile(x, 0.25, na.rm = TRUE)
  Q3 <- quantile(x, 0.75, na.rm = TRUE)
  iqr_val <- IQR(x, na.rm = TRUE)

  lower <- Q1 - multiplier * iqr_val
  upper <- Q3 + multiplier * iqr_val

  outliers <- x[x < lower | x > upper]

  list(
    lower_bound = lower,
    upper_bound = upper,
    outliers = outliers,
    outlier_count = length(outliers)
  )
}
```

Summary Functions: Handling Missing Values

Most summary functions return NA if the data contains missing values. Use the `na.rm = TRUE` argument to exclude NAs.

```r
scores_with_na <- c(85, 92, NA, 95, 88, NA, 90)
```

```r
mean(scores_with_na)
```

```
## [1] NA
```

```r
mean(scores_with_na, na.rm = TRUE)
```

```
## [1] 90
```

```r
median(scores_with_na, na.rm = TRUE)
```

```
## [1] 90
```

```r
sum(scores_with_na, na.rm = TRUE)
```

```
## [1] 450
```

Summary Functions: The summary() Function

The `summary()` function provides a quick overview of data, adapting to the data type.

```r
summary(scores)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
##   72.00   79.75   86.50   85.20   90.75   95.00 
```

```r
summary(mtcars[, 1:4])
```

```
##       mpg             cyl             disp             hp       
##  Min.   :10.40   Min.   :4.000   Min.   : 71.1   Min.   : 52.0  
##  1st Qu.:15.43   1st Qu.:4.000   1st Qu.:120.8   1st Qu.: 96.5  
##  Median :19.20   Median :6.000   Median :196.3   Median :123.0  
##  Mean   :20.09   Mean   :6.188   Mean   :230.7   Mean   :146.7  
##  3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0   3rd Qu.:180.0  
##  Max.   :33.90   Max.   :8.000   Max.   :472.0   Max.   :335.0  
```

Summary Functions with dplyr

Using `summarise()` with summary functions allows powerful grouped calculations.

```
nba2009_2018 %>%
  group_by(home.WL) %>%
  summarise(
    avg_points = mean(home.PTS, na.rm = TRUE),
    median_points = median(home.PTS, na.rm = TRUE),
    min_points = min(home.PTS, na.rm = TRUE),
    max_points = max(home.PTS, na.rm = TRUE),
    games = n()
  )
```

```
## # A tibble: 3 x 6
##   home.WL avg_points median_points min_points max_points games
##   <chr>        <dbl>         <dbl>      <dbl>      <dbl> <int>
## 1 L             97.1            96         59        161  4949
## 2 W            108.            108         71        154  7110
## 3 <NA>           0               0          0          0     1
```

Summary Functions with dplyr: Multiple Groups

```r
nba2009_2018 %>%
  filter(SEASON_ID %in% c(2015, 2016, 2017)) %>%
  group_by(SEASON_ID, home.WL) %>%
  summarise(
    avg_home_pts = mean(home.PTS, na.rm = TRUE),
    avg_away_pts = mean(away.PTS, na.rm = TRUE),
    total_games = n(),
    .groups = 'drop'
  )
```

```
## # A tibble: 6 x 5
##   SEASON_ID home.WL avg_home_pts avg_away_pts total_games
##       <dbl> <chr>          <dbl>        <dbl>       <int>
## 1      2015 L               97.4        108.          506
## 2      2015 W              109.          96.9         724
## 3      2016 L              101.         111.          512
## 4      2016 W              111.          99.1         718
## 5      2017 L              101.         112.          518
## 6      2017 W              112.         100.          712
```

Summary Functions: across() for Multiple Columns

Use `across()` to apply the same summary function to multiple columns at once.

```
mtcars %>%
  summarise(across(c(mpg, hp, wt),
                   list(mean = mean, sd = sd, median = median)))
```

```
##   mpg_mean   mpg_sd mpg_median  hp_mean    hp_sd hp_median wt_mean     wt_sd
## 1 20.09062 6.026948       19.2 146.6875 68.56287       123 3.21725 0.9784574
##   wt_median
## 1     3.325
```

Section 12

Joining Data

Why Join Data?

In real-world analysis, data often comes from multiple sources that need to be combined.

Common scenarios:

- Customer information in one table, transactions in another
- Player statistics in one dataset, team information in another
- Survey responses linked by respondent ID

dplyr provides several join functions:

- `left_join()` - keep all rows from the left table
- `right_join()` - keep all rows from the right table
- `inner_join()` - keep only matching rows
- `full_join()` - keep all rows from both tables

Joining Data: Example Setup

Let's create two sample datasets to demonstrate joins.

```r
teams <- data.frame(
  team_id = c(1, 2, 3, 4),
  team_name = c("Lakers", "Celtics", "Bulls", "Heat"),
  city = c("Los Angeles", "Boston", "Chicago", "Miami")
)
teams
```

```
##   team_id team_name        city
## 1       1    Lakers Los Angeles
## 2       2   Celtics      Boston
## 3       3     Bulls     Chicago
## 4       4      Heat       Miami
```

```r
players <- data.frame(
  player_name = c("LeBron", "Tatum", "LaVine", "Butler", "Curry"),
  team_id = c(1, 2, 3, 4, 5),
  points_avg = c(27.2, 26.9, 24.8, 22.9, 29.4)
)
players
```

```
##   player_name team_id points_avg
## 1      LeBron       1       27.2
## 2       Tatum       2       26.9
## 3      LaVine       3       24.8
```

Joining Data: left_join()

`left_join()` keeps ALL rows from the left table, matching where possible.

```
players %>%
  left_join(teams, by = "team_id")
```

```
##   player_name team_id points_avg team_name        city
## 1      LeBron       1       27.2    Lakers Los Angeles
## 2       Tatum       2       26.9   Celtics      Boston
## 3      LaVine       3       24.8     Bulls     Chicago
## 4      Butler       4       22.9      Heat       Miami
## 5       Curry       5       29.4      <NA>        <NA>
```

- Curry has `NA` for team_name and city because team_id 5 doesn't exist in the teams table
- Use left_join when you want to keep all records from your primary dataset

Joining Data: right_join()

`right_join()` keeps ALL rows from the right table, matching where possible.

```
players %>%
  right_join(teams, by = "team_id")
```

```
##   player_name team_id points_avg team_name        city
## 1      LeBron       1       27.2    Lakers Los Angeles
## 2       Tatum       2       26.9   Celtics      Boston
## 3      LaVine       3       24.8     Bulls     Chicago
## 4      Butler       4       22.9      Heat       Miami
```

- All teams are kept, even if no players match
- This is equivalent to `teams %>% left_join(players, by = "team_id")`

Joining Data: inner_join()

inner_join() keeps ONLY rows that match in BOTH tables.

```
players %>%
  inner_join(teams, by = "team_id")
```

```
##   player_name team_id points_avg team_name        city
## 1      LeBron       1       27.2    Lakers Los Angeles
## 2       Tatum       2       26.9   Celtics      Boston
## 3      LaVine       3       24.8     Bulls     Chicago
## 4      Butler       4       22.9      Heat       Miami
```

- Curry is excluded because team_id 5 doesn't exist in teams
- Use inner_join when you only want complete matches

Joining Data: full_join()

full_join() keeps ALL rows from BOTH tables, matching where possible.

```
teams2 <- rbind(teams, data.frame(team_id = 6, team_name = "Knicks", city =

players %>%
  full_join(teams2, by = "team_id")
```

```
##   player_name team_id points_avg team_name        city
## 1      LeBron       1       27.2    Lakers Los Angeles
## 2       Tatum       2       26.9   Celtics      Boston
## 3      LaVine       3       24.8     Bulls     Chicago
## 4      Butler       4       22.9      Heat       Miami
## 5       Curry       5       29.4      <NA>        <NA>
## 6        <NA>       6         NA    Knicks    New York
```

- Keeps all players AND all teams
- NAs appear where no match exists

Joining Data: Multiple Key Columns

You can join on multiple columns when needed.

```
sales <- data.frame(
  region = c("East", "East", "West", "West"),
  year = c(2023, 2024, 2023, 2024),
  revenue = c(100, 120, 150, 180)
)

targets <- data.frame(
  region = c("East", "East", "West"),
  year = c(2023, 2024, 2023),
  target = c(95, 115, 140)
)

sales %>%
  left_join(targets, by = c("region", "year"))
```

```
##   region year revenue target
## 1   East 2023     100     95
## 2   East 2024     120    115
## 3   West 2023     150    140
## 4   West 2024     180     NA
```

Joining Data: Different Column Names

Use `by = c("left_name" = "right_name")` when column names differ.

```
team_stats <- data.frame(
  t_id = c(1, 2, 3),
  wins = c(52, 57, 40)
)

teams %>%
  left_join(team_stats, by = c("team_id" = "t_id"))
```

```
##   team_id team_name        city wins
## 1       1    Lakers Los Angeles   52
## 2       2   Celtics      Boston   57
## 3       3     Bulls     Chicago   40
## 4       4      Heat       Miami   NA
```

Joining Data: Practical Example

```r
player_totals <- players %>%
  group_by(team_id) %>%
  summarise(
    num_players = n(),
    team_avg_points = mean(points_avg)
  )

teams %>%
  left_join(player_totals, by = "team_id")
```

```
##   team_id team_name        city num_players team_avg_points
## 1       1    Lakers Los Angeles           1            27.2
## 2       2   Celtics      Boston           1            26.9
## 3       3     Bulls     Chicago           1            24.8
## 4       4      Heat       Miami           1            22.9
```

Section 13

In Class Assignment

In Class Assignment: Olympic Medal Analysis

Using the `summer` Olympic dataset and concepts learned in this lecture, complete the following tasks. Time estimate: 15-20 minutes.

Dataset: summer.csv (already loaded as `summer`)

Tasks:

1. Create a summary showing the **total number of medals** and **number of unique sports** for each country. Filter to show only the top 5 countries by total medals.
2. Calculate the **mean** and **median** number of medals per year for the USA. Are they similar? What does this tell you about the distribution?
3. Create two data frames: one with country populations and one with medal counts. Then **join** them to calculate medals per million population.

In Class Assignment: Task Details

Task 1: Summary by country

- Group by Country
- Use `summarise()` with `n()` for total medals
- Use `n_distinct(Sport)` for unique sports count
- Arrange in descending order and show top 5

Task 2: Mean vs Median analysis

- Filter for USA only
- Group by Year and count medals per year
- Calculate mean and median of these yearly counts
- Compare and interpret the difference

Task 3: Joining data

- Create a simple population data frame for 3-4 countries
- Join with medal counts
- Calculate medals per capita

In Class Assignment: Task Details (continued)

Hints:

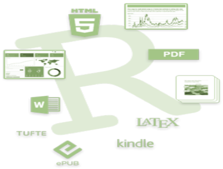
- Remember to use %>% to chain operations
- For Task 2, first create a yearly summary, then summarize that
- For Task 4, you'll need to summarize before pivoting

Section 14

R Markdown

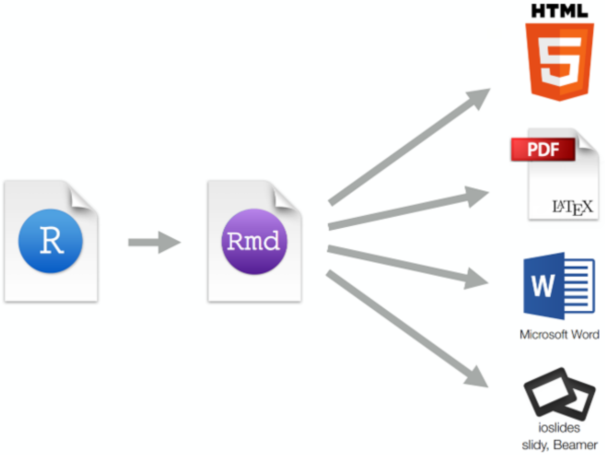
R Markdown

- R Markdown documents are fully reproducible.
- It allows to combine text and code together in one report.
- You can use multiple languages within markdown file (R, python, SQL).
- R Markdown supports dozens of static and dynamic output formats including HTML, PDF, MS Word, Beamer, HTML5 slides, etc.
- R markdown is run in its own environment.



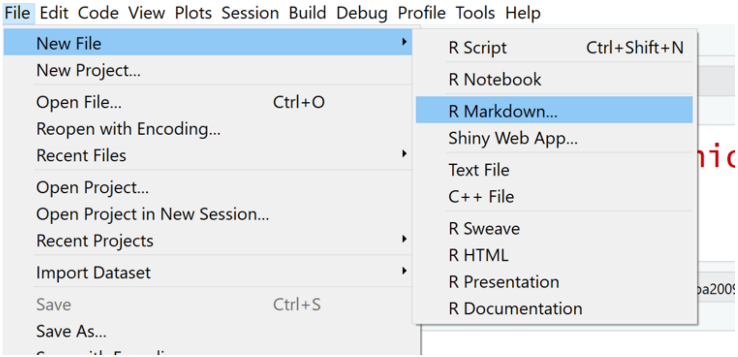
R Markdown

The Workflow



R Markdown

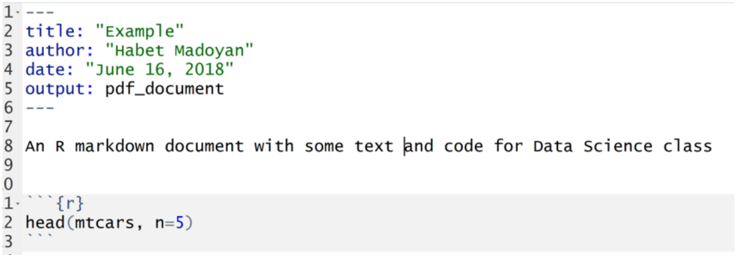
To open new markdown file



R Markdown

The document below is a template R Markdown document. It includes the most familiar parts of an R Markdown document:

- A YAML header that contains some metadata (1)
- Narrative text written in Markdown (2)
- R code chunks surrounded by triple backticks {r} and followed by triple backticks; a syntax that comes from the knitr package (3)

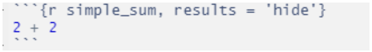


~~~
---
title: "Example"
author: "Habet Madoyan"
date: "June 16, 2018"
output: pdf_document
---

An R markdown document with some text and code for Data Science class


```{r}
head(mtcars, n=5)
```
~~~

R Markdown

Labeling and reusing code chunks

- Apart from the popular code chunk options you have learned by now, you can define even more things in the curly braces that follow the triple backticks.
- An interesting feature available in knitr is the labeling of code snippets. The code chunk below would be assigned the label simple_sum:

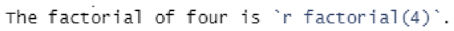


~~~
```{r simple_sum, results = 'hide'}
2 + 2
```
~~~

- However, because the results option is equal to hide, no output is shown.

R Markdown

- You can embed R code into the text of your document with the `r` syntax. Be sure to include the lower case r in order for this to work properly.
- R Markdown will run the code and replace it with its result, which should be a piece of text, such as a character string or a number.
- For example, the line below uses embedded R code to create a complete sentence:



```
The factorial of four is `r factorial(4)`.
```

- When you render the document the result will appear as:

The factorial of four is 24.

- Inline code provides a useful way to make your reports completely automated.

R Markdown

LaTeX equations

- You can also use the Markdown syntax to embed latex math equations into your reports. To embed an equation in its own centered equation block, surround the equation with two pairs of dollar signs like this,

```
$$1 + 1 = 2$$
```

- To embed an equation inline, surround it with a single pair of dollar signs, like this:

```
$1 + 1 = 2$
```

- You can use all of the standard latex math symbols to create attractive equations.

R Markdown

LaTex formula example

$$\frac{n!}{k!(n-k)!} = \binom{n}{k}$$

In Tex

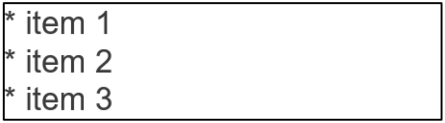
```
\frac{n!}{k!(n-k)!} = \binom{n}{k}
```

Tex script needs to be written in the text portion of the document

R Markdown

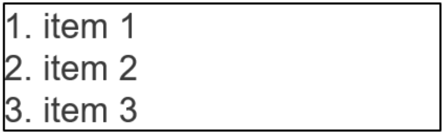
Lists in R Markdown

To make a bullet list in Markdown, place each item on a new line after an asterisk or hyphen and a space, like this:



```
* item 1
* item 2
* item 3
```

You can make an ordered list by placing each item on a new line after a number followed by a period followed by a space, like this



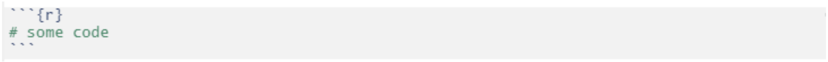
```
1. item 1
2. item 2
3. item 3
```

In each case, you need to place a blank line between the list and any paragraphs that come before it.

R Markdown

R code chunks

You can embed R code into your R Markdown report with the knitr syntax. To do this, surround your code with two lines: one that contains triple backticks {r} and the below one contains triple backticks. The result is a code chunk that looks like this:



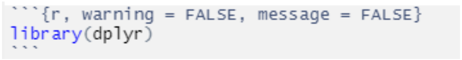
~~~
```{r}
# some code
```
~~~

When you render the report, R will execute the code. If the code returns any results, R will add them to your report.

R Markdown

Customize R code chunks

- You can customize each R code chunk in your report by providing optional arguments after the r in “‘{r}, which appears at the start of the code chunk. Let’s look at one set of options.
- R functions sometimes return messages, warnings, and even error messages. By default, R Markdown will include these messages in your report. You can use the **message, warning and error** options to prevent R Markdown from displaying these. If any of the options are set to FALSE, R Markdown will not include the corresponding type of message in the output. Packages often generate messages when you first load them with library().
- For example, R Markdown would ignore any messages or warnings generated by the chunk below.



~~~
```{r, warning = FALSE, message = FALSE}
library(dplyr)
```
~~~

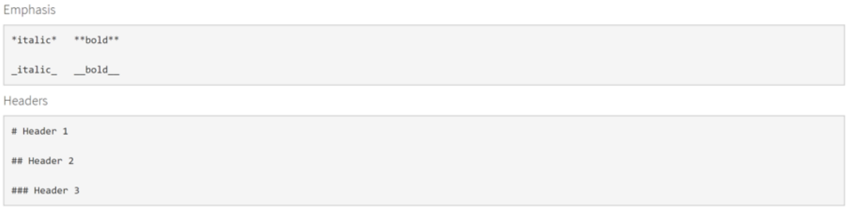
R Markdown

Three of the most popular chunk options are echo, eval and results.

- If ***echo = FALSE***, R Markdown will not display the code in the final document (but it will still run the code and display its results unless told otherwise).
- If ***eval = FALSE***, R Markdown will not run the code or include its results, (but it will still display the code unless told otherwise).
- If ***results = 'hide'***, R Markdown will not display the results of the code (but it will still run the code and display the code itself unless told otherwise).

R Markdown

Other important stuff



Emphasis

```
*italic*   **bold**

_italic_   __bold__
```

Headers

```
# Header 1

## Header 2

### Header 3
```

If you want to start from a new page in pdf file, put in text part of the markdown

```
\newpage

\pagebreak
```