Week 1 – Intro to R

Intro to R

- Go to https://cran.r-project.org/ and install version for your operating system
- Go to https://www.rstudio.com/products/rstudio/download/ and install R Studio IDE

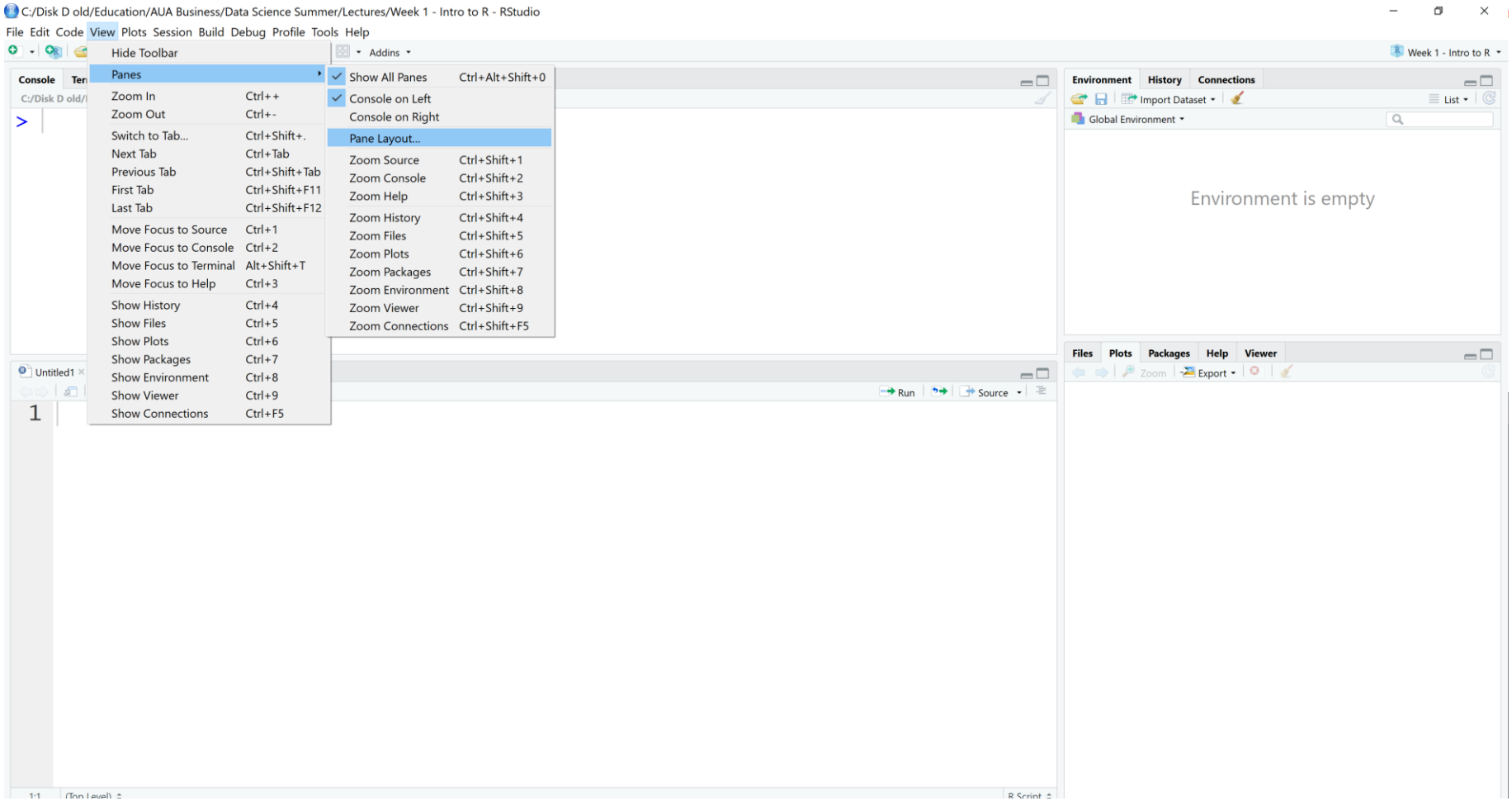
You need to install R before installing R studio

Alternatives

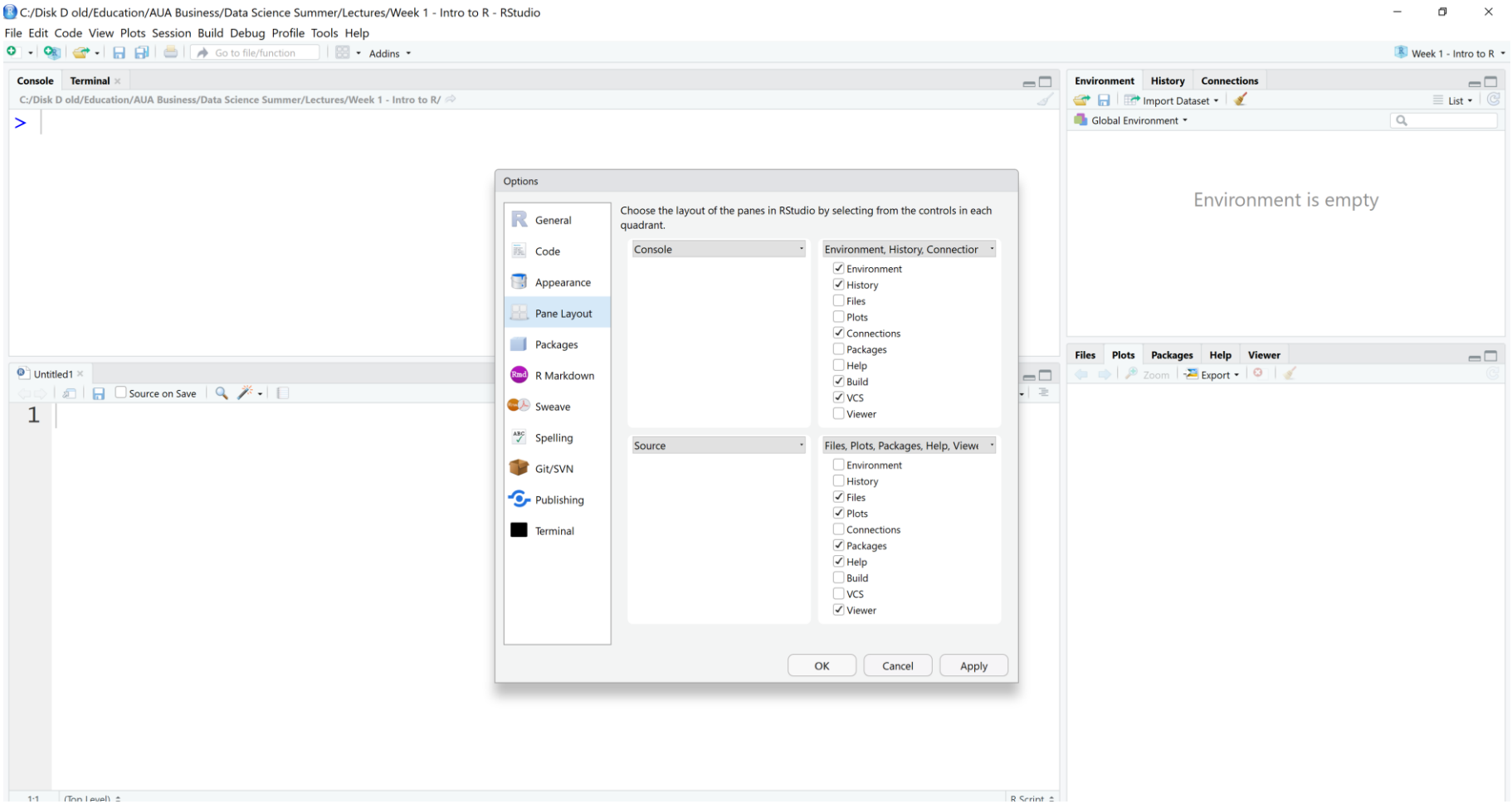
- Install Microsoft R https://mran.microsoft.com/open
- Use Microsoft Visual Studio (free version available)

Intro to R

Configure the pane layout of R studio as you wish

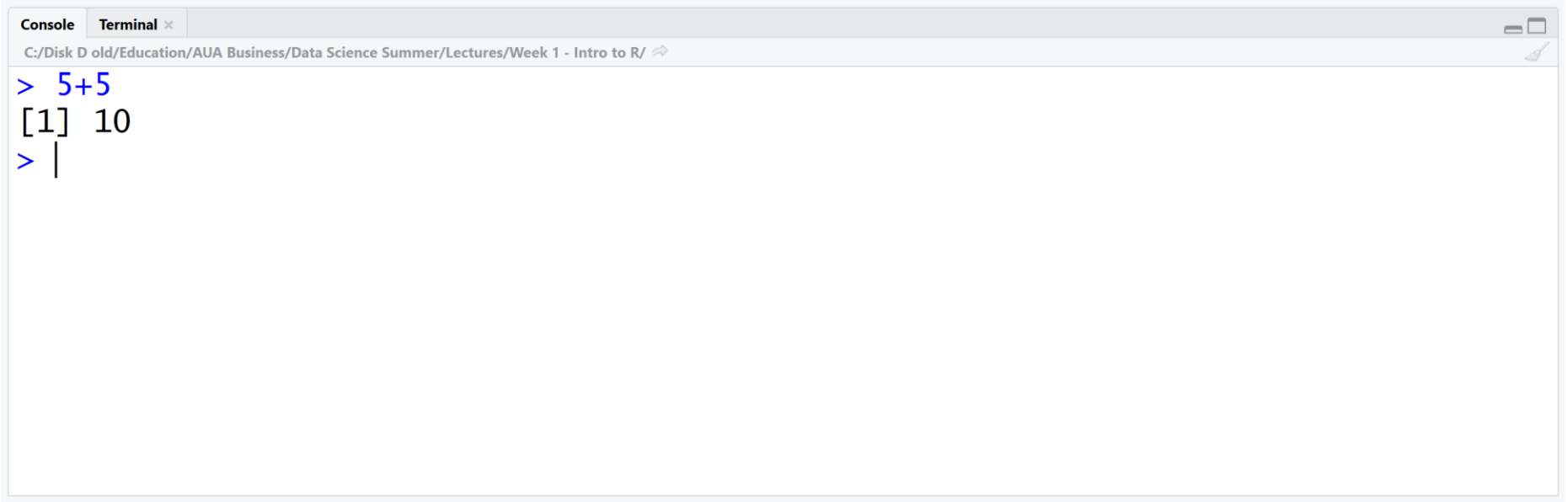


Intro to R


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
Addins
Week 1 - Intro to R
Console
Terminal
C:/Disk D old/Education/AUA Business/Data Science Summer/Lectures/Week 1 - Intro to R/
Environment
History
Connections
Import Dataset
List
Global Environment
Environment is empty
Files
Plots
Packages
Help
Viewer
Zoom
Export
Untitled1
Source on Save
Options
General
Code
Appearance
Pane Layout
Packages
R Markdown
Sweave
Spelling
Git/SVN
Publishing
Terminal
Choose the layout of the panes in RStudio by selecting from the controls in each quadrant.
Console
Environment, History, Connectior
Environment
History
Files
Plots
Connections
Packages
Help
Build
VCS
Viewer
Source
Files, Plots, Packages, Help, Viewe
Environment
History
Files
Plots
Connections
Packages
Help
Build
VCS
Viewer
OK
Cancel
Apply

Intro to R

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



Use console as a calculator

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

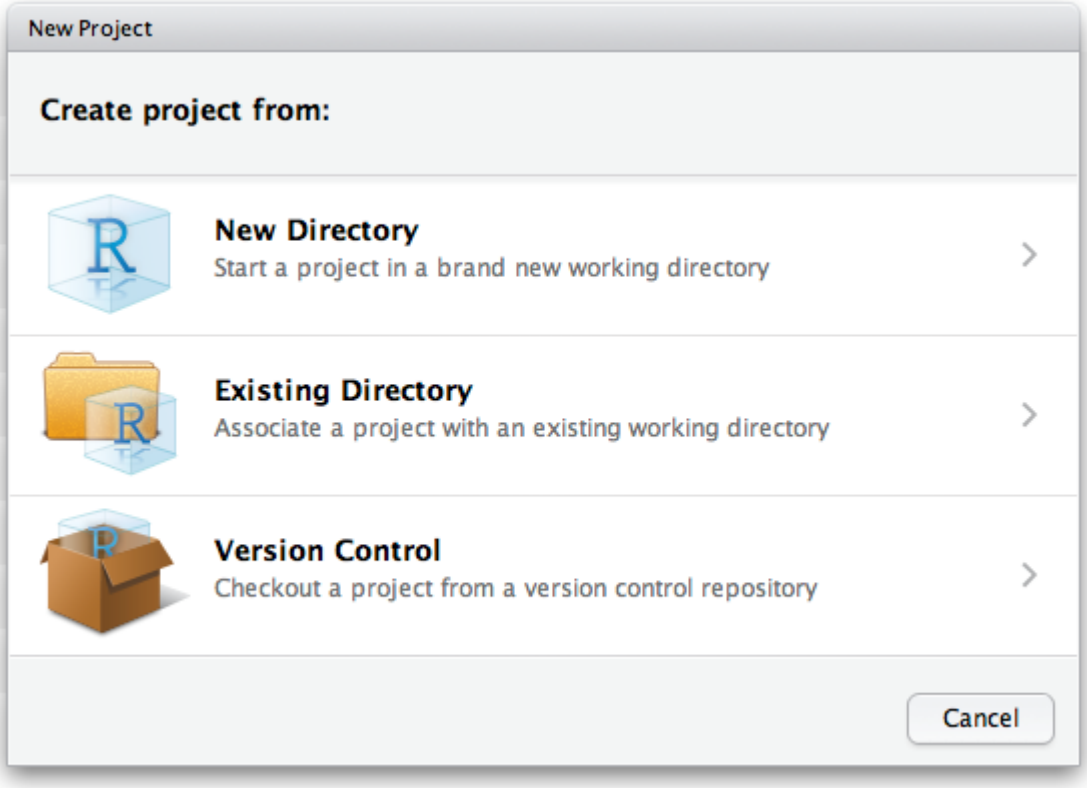
RStudio projects make it straightforward to divide your work into multiple contexts, each with their own working directory, workspace, history, and source documents.

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

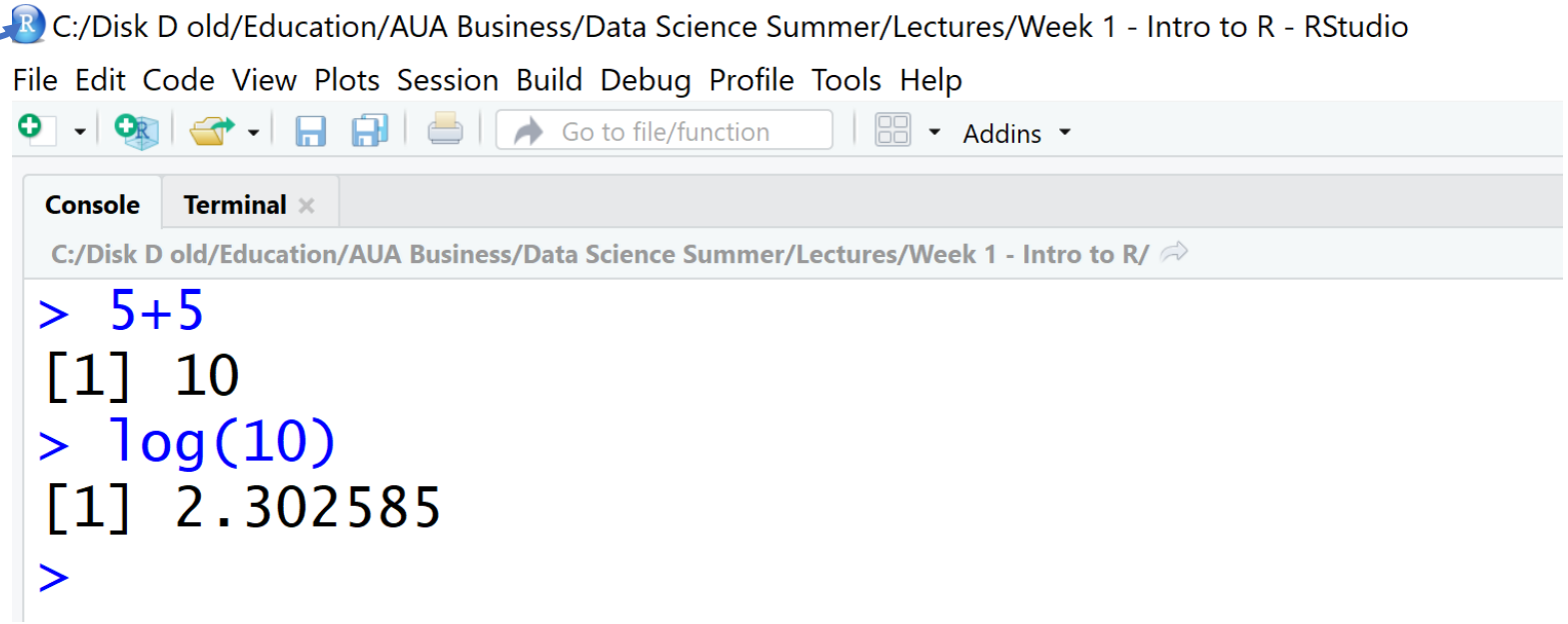
- Create an empty directory
- Go to File > New Project
- Chose existing directory



Intro to R

- Now when you have created your project, all files associated with the project need to be save in the project directory

- To check the project directory do getwd() in console



The name of the project appears at the top of R Studio window

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 display its name in the Projects toolbar (which is located on the far right side of the main toolbar)

Opening Projects

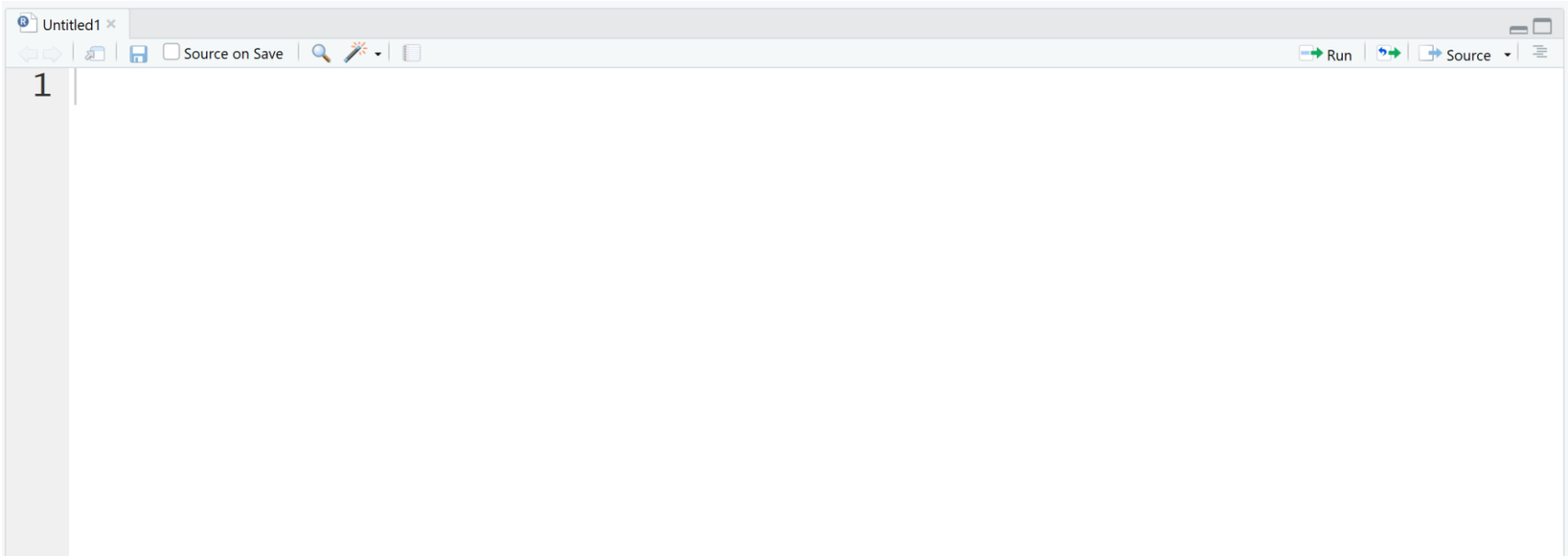
There are several ways to open a project:

- 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 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.).

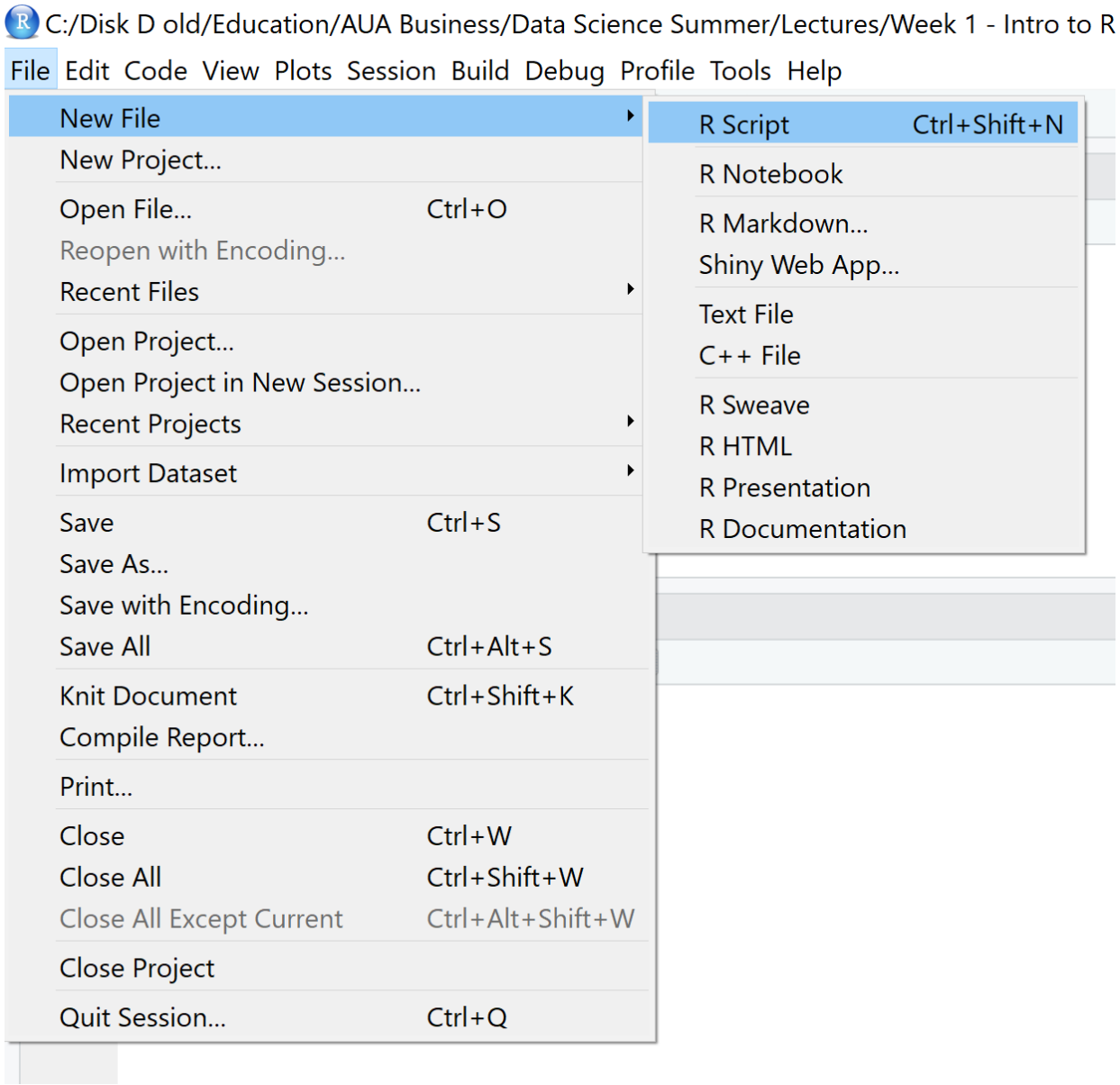
Intro to R

The source editor

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



Intro to R

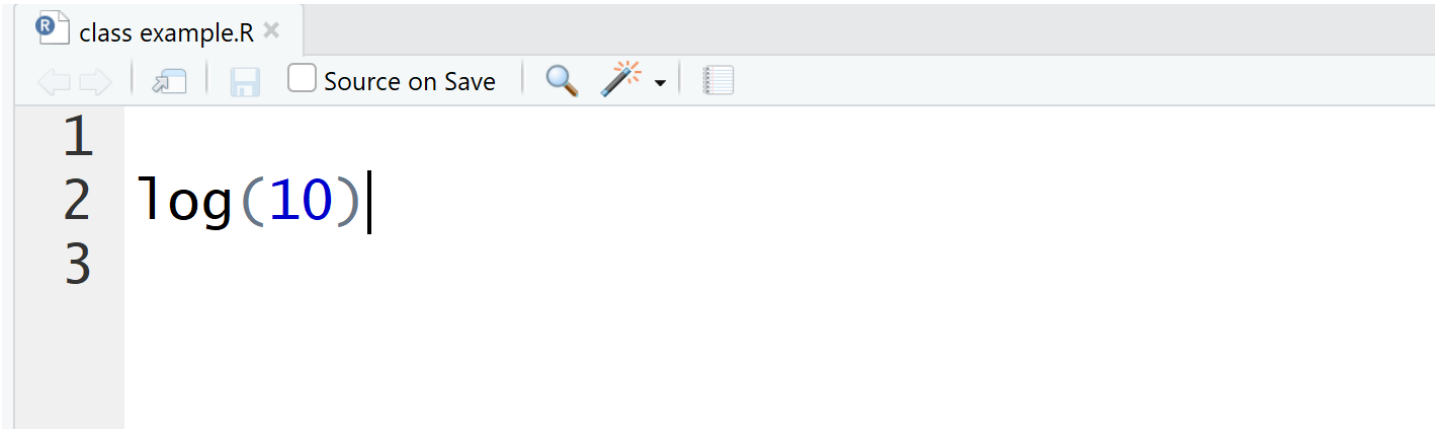


This will create new script file

Intro to R

- To run the script line from source editor, put the coursor anywhere on the line and hit **ctrl+enter**

- You will see the output in the console

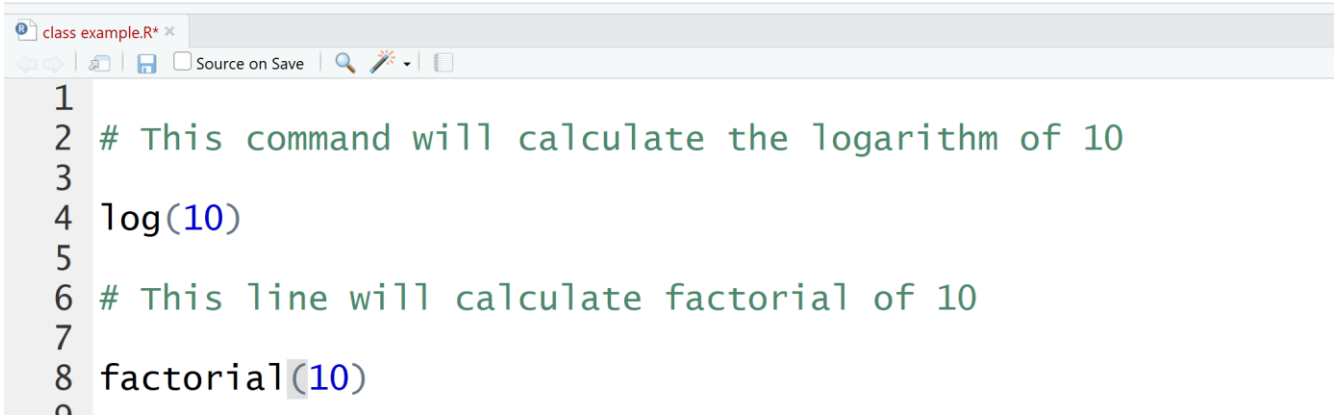


Intro to R

- 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 run this line of code will printed in console as a text

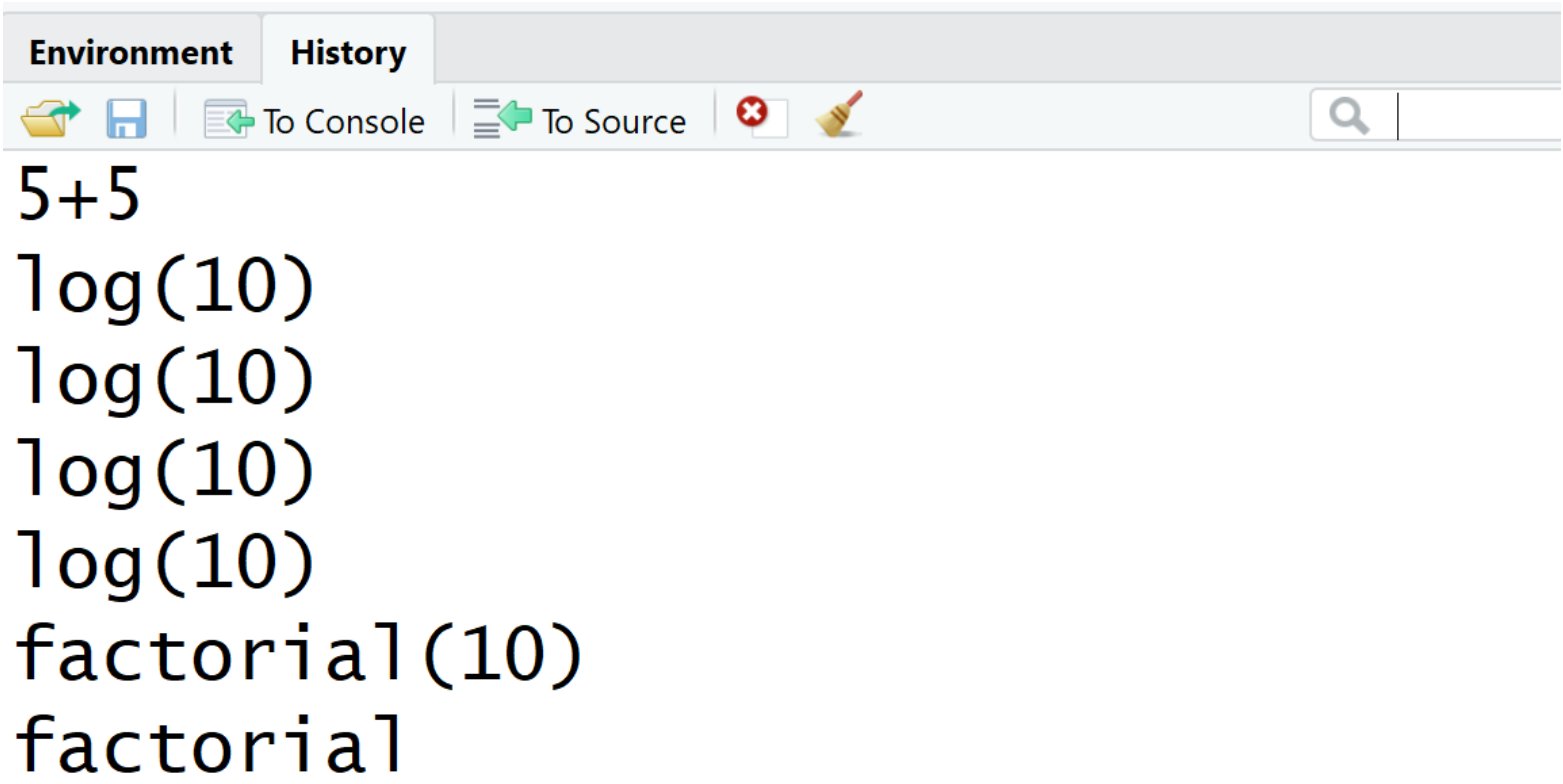
```
C:/Disk D old/Education/AUA Business/Data Science Summer/Lectures/Week 1 - Intro to R/
> # This command will calculate the logarithm of 10
>
> log(10)
[1] 2.302585
>
> # This line will calculate factorial of 10
>
> factorial(10)
[1] 3628800
>
```



To comment large chunk of text, use ctrl+shift+c

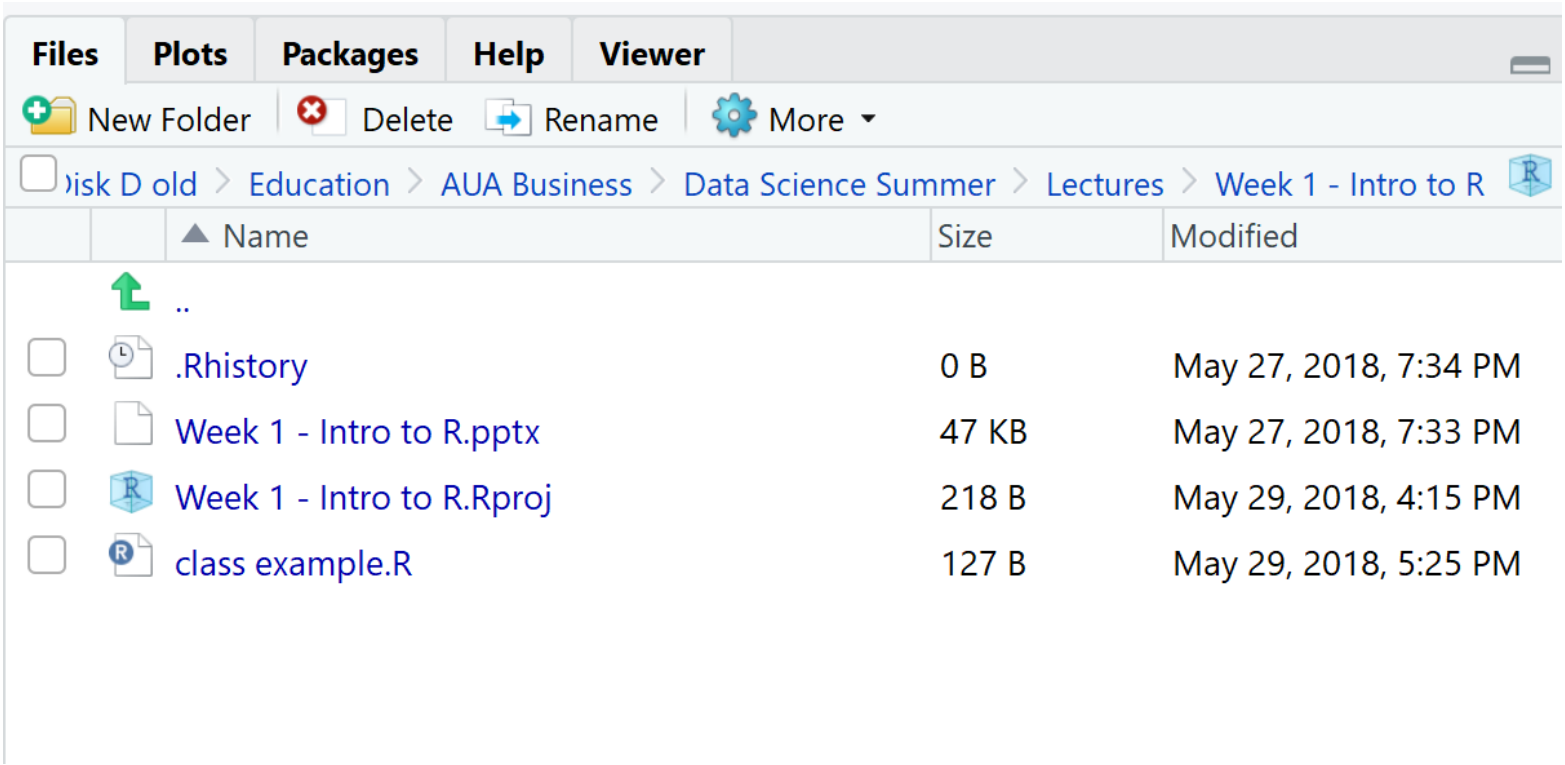
Intro to R

- 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.



Intro to R

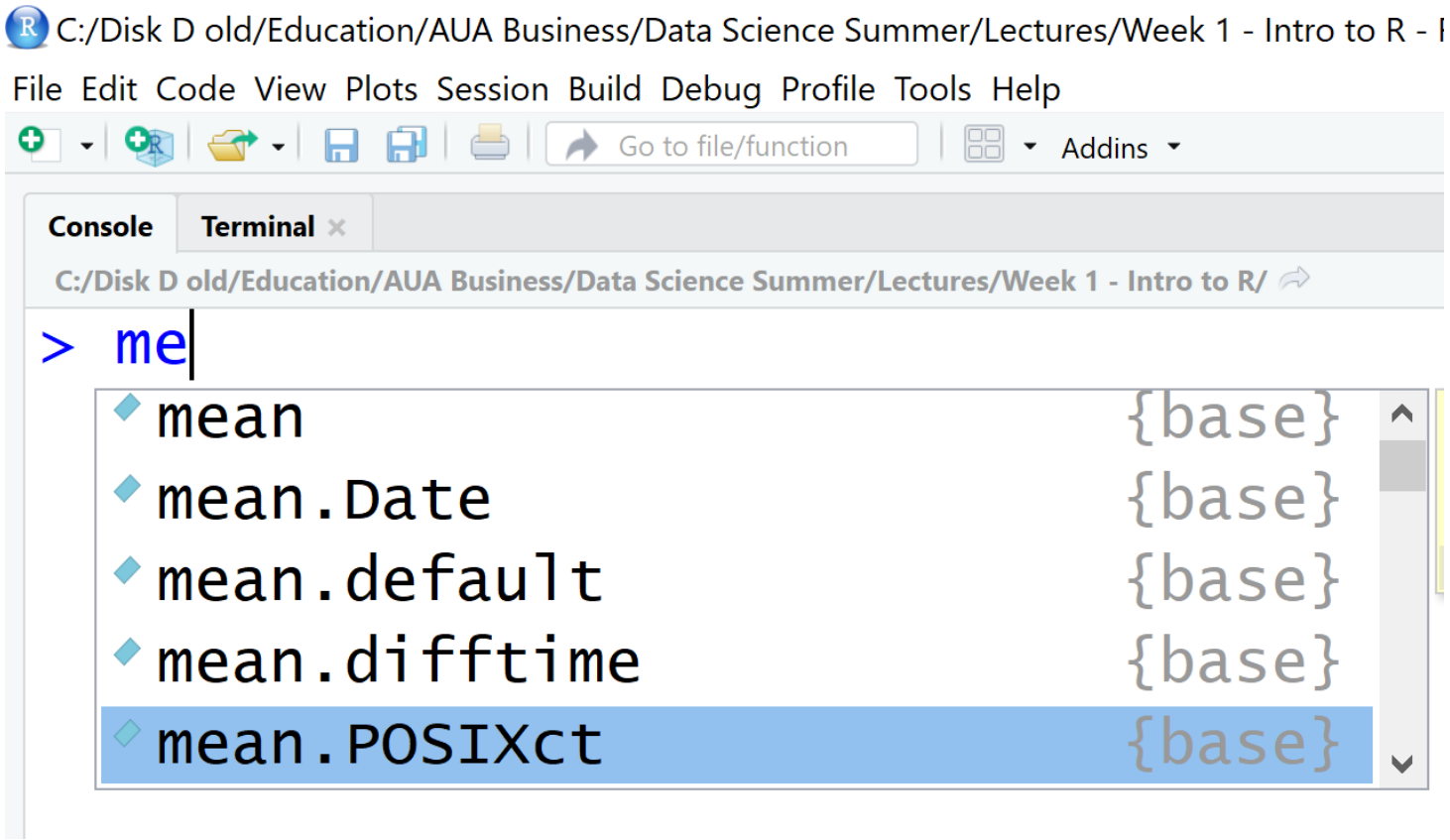
1.**Bottom right**: files, plots, packages, help, & viewer pane. Here you can open files, view plots, install and load packages, read man pages, and view markdown and other documents in the viewer tab.



Intro to R

Some useful shortcuts:

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

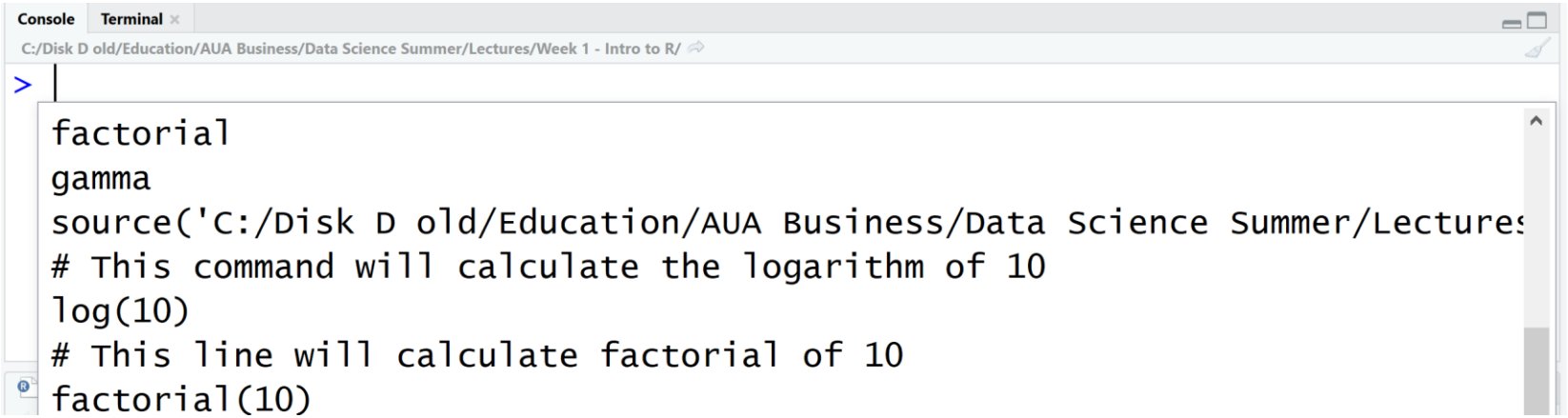


Intro to R

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.



Intro to R

- When you download R from the Comprehensive R Archive Network (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 lecture to the **install.packages()** function as the first argument
- The following the code installs the **devtools** package from CRAN

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

- 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 session

```
library(devtools)
```

```
## Warning: package 'devtools' was built under R version 3.3.3
```

- 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 dont 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")
```

https://github.com/christophM/iml

Working with help

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

```
?mean
```

```
## starting httpd help server ...
```

```
##  done
```

To access help/documentation on function from a library

```
??find.BIB
```

The same

```
?crossdes::find.BIB
```

Help on the package

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

- 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

https://cran.r-project.org/web/views/TimeSeries.html

Intro to R programming language

Intro to R: Data Structures

- Anything in R is an object
- Objects are assigned values using <- , an arrow formed out of < and -. (An equal sign, =, can also be used.) For example, the following command assigns the value 5 to the object x.
- R is case sensitive, thus Data10 and data10 are two different objects

```
x <- 5
x
```

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

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

Vector

Vector – This data structures contain similar types of data, i.e., integer, double, logical, complex, etc. In order to create a vector in R Programming, c() function is used.

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

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

This will create a vector with values from 1 to 10

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

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

Check the class of the vector

```
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, integer, double, and character.

For example, combining a character and an integer yields a character:

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

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

```
class(x3)
```

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

Matrix

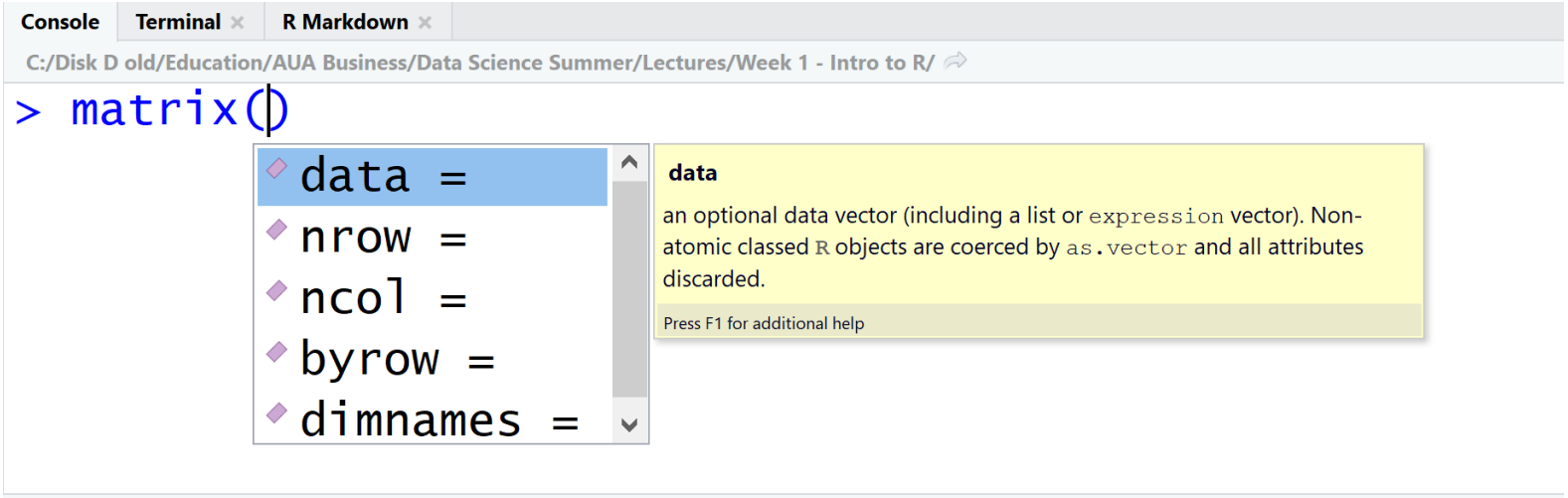
- Matrix – Matrix is a two-dimensional data structure and can be created using matrix () function.
- Matrix is a collection of vectors with the same length and same type

```
m <- matrix(data=c(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
```

Intro to R: 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 in within the function
- If the order is the same you can skip the names



Example: function matrix with arguments flipped

The same result

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

Different result

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

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

Other ways of building a matrix

- create two vectors of the same length

```
x <- c(0:10)
y <- c(-5:5)
```

- Combine together by row

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

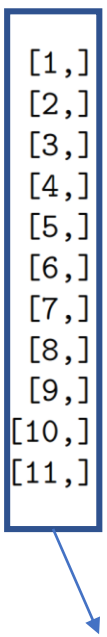
```
##   [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11]
## x    0    1    2    3    4    5    6    7    8     9    10
## y   -5   -4   -3   -2   -1    0    1    2    3     4     5
```

Intro to R: Data Structures

- Combine together by column
- Pay attention: Columns and rows now have names

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

```
##       x  y
##  [1,] 0 -5
##  [2,] 1 -4
##  [3,] 2 -3
##  [4,] 3 -2
##  [5,] 4 -1
##  [6,] 5  0
##  [7,] 6  1
##  [8,] 7  2
##  [9,] 8  3
## [10,] 9  4
## [11,] 10 5
```



Row numbers

- Check if the resulting object is a matrix

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

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

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

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

- 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
```

Row numbers

Dataframe has column and row names

```
colnames(df)
```

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

```
rownames(df)
```

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

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)

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

```
##  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(df_seriea)
```

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

```
typeof(df_seriea)
```

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

Create a dataframe

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

```
## 'data.frame':    4 obs. of  2 variables:
##  $ Club  : Factor w/ 4 levels "Inter","Juventus",..: 2 3 4 1
##  $ Points: num  95 91 77 72
```

Two columns (vectors) in the df have different types

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 by 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.

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  : Factor w/ 4 levels "Inter","Juventus",..: 2 3 4 1
##   ..$ Points: num [1:4] 95 91 77 72
##  $ : int [1:2, 1:11] 0 -5 1 -4 2 -3 3 -2 4 -1 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:2] "x" "y"
##   .. ..$ : NULL
```

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] [,7] [,8] [,9] [,10] [,11]
## x    0    1    2    3    4    5    6    7    8     9    10
## y   -5   -4   -3   -2   -1    0    1    2    3     4     5
```

The unlist() function will flatten the list into a vector

```
z <- unlist(my_list)
z
```

```
##   Club1   Club2   Club3   Club4 Points1 Points2 Points3 Points4
##       2       3       4       1      95      91      77      72       0
##
##      -5       1      -4       2      -3       3      -2       4      -1
##
##       5       0       6       1       7       2       8       3       9
##
##       4      10       5
```

Attributes

- All objects can have arbitrary additional attributes, used to store metadata about the object.
- Attributes can be thought of as a named list (with unique names).
- Attributes can be accessed individually with attr() or all at once (as a list) with attributes().

```
attr(df_seriea, "topic") <- "sports"
df_seriea
```

```
##       Club Points
## 1 Juventus     95
## 2   Napoli     91
## 3     Roma     77
## 4    Inter     72
```

```
attr(df_seriea, "topic")
```

```
## [1] "sports"
```

Data Types

Data types used in R

- Logical
- numeric
- factor
- character

Intro to R: Data types

Logical data type is one of the frequently used data type usually used for comparing two values. Values 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

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

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

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

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

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

Factors

- One important use of attributes is to define factors.
- 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 to use when the factor is displayed. The factor function is used to create a factor.

```
typeof(b)
```

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

Look at the numeric values associated with factors

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

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

Look at the factor

```
b
```

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

You can relevel the factor variable by changing the reference value

Relevel the variable

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

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

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

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

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

Coercing the character vector to numeric

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

Too big to show

```
120/0
```

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

Too big to show

```
-120/0
```

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

Too big to show

```
45^12500
```

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

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

Additionally, there is a null object in R, represented by the symbol NULL. (The symbol NULL always points to the same object.) NULL is often used as an argument in functions to mean that no value was assigned to the argument. Additionally, some functions may return NULL. Note that 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

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('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, AndrÃ©\",FRA,Men,Military Patrol,Bronze"
```

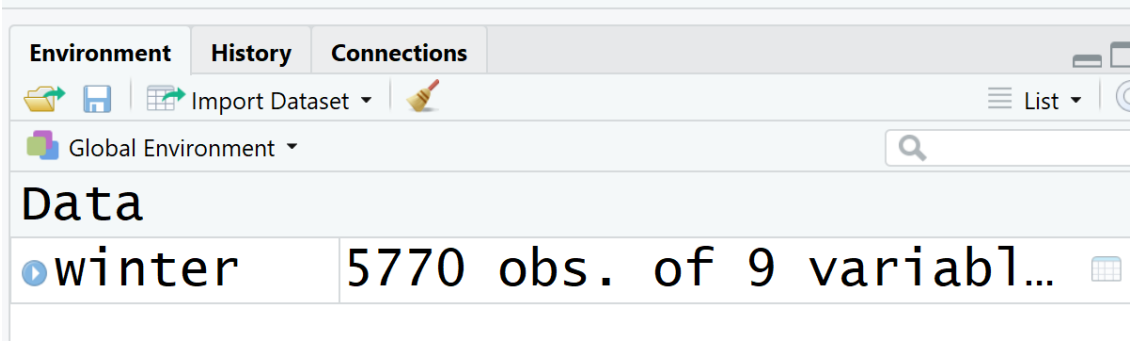
Importing data to R

Use read.csv to load the file into R environment

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

```
## 'data.frame':    5770 obs. of  9 variables:
##  $ Year      : int  1924 1924 1924 1924 1924 1924 1924 1924 1924 1924 ...
##  $ City      : Factor w/ 19 levels "Albertville",..: 3 3 3 3 3 3 3 3 3 3 ...
##  $ Sport     : Factor w/ 7 levels "Biathlon","Bobsleigh",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ Discipline: Factor w/ 15 levels "Alpine Skiing",..: 2 2 2 2 2 2 2 2 2 2 ...
##  $ Athlete   : Factor w/ 3761 levels "Ã-BERG, Carl-GÃ¶ran",..: 301 2095 2096 3472 164 151
##  $ Country   : Factor w/ 45 levels "AUS","AUT","BEL",..: 16 16 16 16 37 37 37 37 15 15 ..
##  $ Gender    : Factor w/ 2 levels "Men","Women": 1 1 1 1 1 1 1 1 1 1 ...
##  $ Event     : Factor w/ 83 levels "10000M","1000M",..: 58 58 58 58 58 58 58 58 58 58 ...
##  $ Medal     : Factor w/ 3 levels "Bronze","Gold",..: 1 1 1 1 2 2 2 2 3 3 ...
```

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



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 then as a factor

```
winter <- read.csv('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, AndrÃ©" ...
##  $ 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('winter.txt', n=5)
```

```
## [1] "Year\tCity\tSport\tDiscipline\tAthlete\tCountry\tGender\tEvent\tMedal"
## [2] "1924\tChamonix\tBiathlon\tBiathlon\t\"BERTHET, G.\"\tFRA\tMen\tMilitary Patrol\tBro
## [3] "1924\tChamonix\tBiathlon\tBiathlon\t\"MANDRILLON, C.\"\tFRA\tMen\tMilitary Patrol\t
## [4] "1924\tChamonix\tBiathlon\tBiathlon\t\"MANDRILLON, Maurice\"\tFRA\tMen\tMilitary Pat
## [5] "1924\tChamonix\tBiathlon\tBiathlon\t\"VANDELLE, AndrÃ©\"\tFRA\tMen\tMilitary Patrol
```

Importing data to R

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

```
winter <- read.csv('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, AndrÃ©
##  $ Country   : chr  "FRA" "FRA" "FRA" "FRA" ...
##  $ Gender    : chr  "Men" "Men" "Men" "Men" ...
##  $ Event     : chr  "Military Patrol" "Military Patrol" "Military Patrol" "Military Patro
##  $ 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

Subseting data

Subseting data: vectors

Create named vector

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

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

In R (unlike in Python) the indexing starts with 1

Subseting 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
```

Subseting 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
```

Subseting by name

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

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

Subseting 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
> 
```

Subseting data: dataframes

Install package from github

- The package contains data and functions for Sports Analytics class.

```
library(devtools)
install_github("HABET/CSE270")
library(SportsAnalytics270)
```

Subseting data: dataframes

```
data("nba2009_2016")
str(nba2009_2016)
```

```
## 'data.frame':    9600 obs. of  9 variables:
##  $ SEASON_ID             : chr  "2009" "2009" "2009" "2009" ...
##  $ GAME_DATE             : Date, format: "2009-10-27" "2009-10-27" ...
##  $ home.TEAM_ABBREVIATION: Factor w/ 30 levels "ATL","BKN","BOS",..: 6 7 25 14 1 22 28 3 16 15 .
##  $ home.TEAM_NAME        : Factor w/ 30 levels "Atlanta Hawks",..: 6 7 25 14 1 22 28 2 16 15 ...
##  $ home.PTS              : num  89 91 96 99 120 120 101 92 115 74 ...
##  $ away.TEAM_ABBREVIATION: Factor w/ 30 levels "ATL","BKN","BOS",..: 3 30 11 13 12 23 6 4 20 9 .
##  $ away.TEAM_NAME        : Factor w/ 30 levels "Atlanta Hawks",..: 2 30 11 13 12 23 6 4 20 9 ...
##  $ away.PTS              : num  95 102 87 92 109 106 91 59 93 96 ...
##  $ home.WL               : chr  "L" "L" "W" "W" ...
```

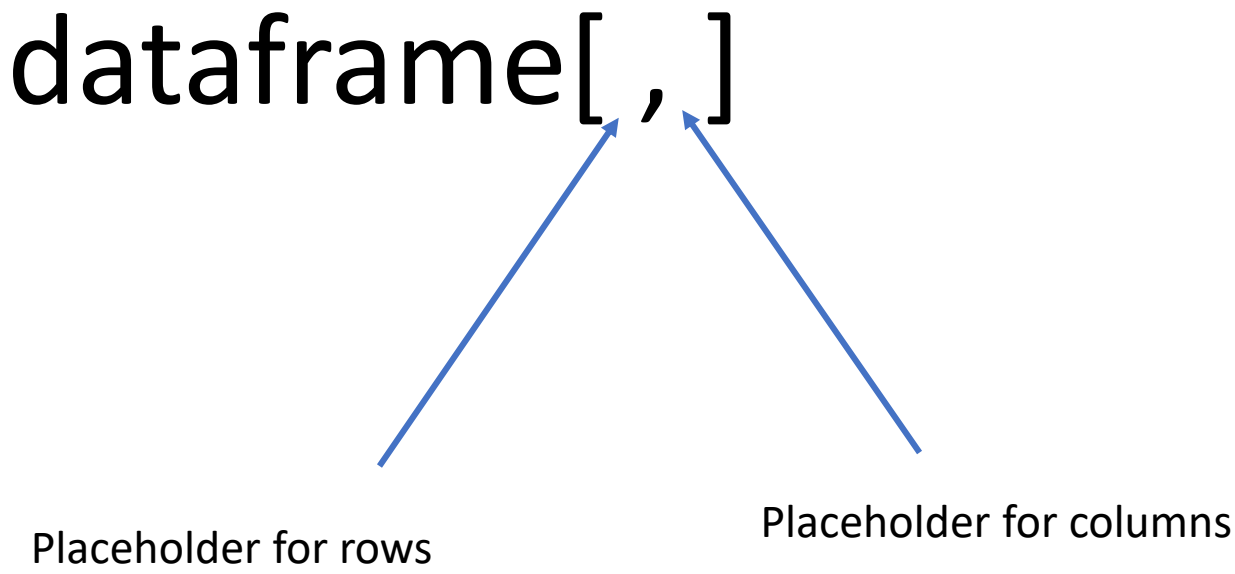
Subseting data: dataframes

```
summary(nba2009_2016)
```

```
##   SEASON_ID           GAME_DATE          home.TEAM_ABBREVIATION
##  Length:9600        Min.   :2009-10-27   ATL    : 320
##  Class :character   1st Qu.:2011-04-06   BKN    : 320
##  Mode  :character   Median :2013-11-13   BOS    : 320
##                     Mean   :2013-08-07   CHA    : 320
##                     3rd Qu.:2015-11-04   CHI    : 320
##                     Max.   :2017-04-12   CLE    : 320
##                                          (Other):7680
##              home.TEAM_NAME    home.PTS    away.TEAM_ABBREVIATION
##  Atlanta Hawks      : 320   Min.   :  0   ATL    : 320
##  Boston Celtics     : 320   1st Qu.: 94   BKN    : 320
##  Brooklyn Nets      : 320   Median :102   BOS    : 320
##  Charlotte Hornets  : 320   Mean   :102   CHA    : 320
##  Chicago Bulls      : 320   3rd Qu.:110   CHI    : 320
##  Cleveland Cavaliers: 320   Max.   :152   CLE    : 320
##  (Other)            :7680                 (Other):7680
##              away.TEAM_NAME    away.PTS        home.WL
##  Atlanta Hawks      : 320   Min.   :  0.00   Length:9600
##  Boston Celtics     : 320   1st Qu.: 91.00   Class :character
##  Brooklyn Nets      : 320   Median : 99.00   Mode  :character
##  Charlotte Hornets  : 320   Mean   : 99.13
##  Chicago Bulls      : 320   3rd Qu.:107.00
##  Cleveland Cavaliers: 320   Max.   :147.00
##  (Other)            :7680
```

Subseting data: dataframes

Dataframe has two dimensions: Rows (first dimension) and Columns (Second dimension)



Will return the value on first row and 4th column

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

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

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

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) ]
```

Subseting data: dataframes

Indexing by column names

```
colnames(nba2009_2016)
```

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

```
nba1 <- nba2009_2016[,c("home.PTS", "away.PTS")]
colnames(nba1)
```

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

```
dim(nba1)
```

```
## [1] 9600    2
```

Subseting data: dataframes

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

```
nba2 <- nba2009_2016[,-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
```

Subseting data: dataframes

Exercises

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

You can access specific column in dataframe by using **$**

```
mean(nba2009_2016$home.PTS)
```

```
## [1] 101.9783
```

```
table(nba2009_2016$home.WL)
```

```
##
##    L    W
## 3930 5669
```

Subseting data: dataframes

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

```
nba4 <- nba2009_2016[nba2009_2016$SEASON_ID=='2009',]
# Check if everything is done right
table(nba4$SEASON_ID)
```

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

Subseting data: dataframes

Take only seasons 2009 and 2010

As the SEASON_ID is character we will do the following

```
nba5 <- nba2009_2016[nba2009_2016$SEASON_ID %in% c("2009", "2010"),]
table(nba5$SEASON_ID)
```

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

Subseting data: dataframes

Workaround: Make SEASON_ID numeric vector

```
nba2009_2016$SEASON_ID <- as.numeric(nba2009_2016$SEASON_ID)
nba5 <- nba2009_2016[nba2009_2016$SEASON_ID < 2011,]
table(nba5$SEASON_ID)
```

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

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

Subseting data: dataframes

Indexing on multiple conditions

Take all the home game records for Detroit Pistons for seasons 2010 and 2011

```
nba6 <- nba2009_2016[nba2009_2016$SEASON_ID %in% c(2010,2011) &
                       nba2009_2016$home.TEAM_NAME =="Detroit Pistons",]

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

```
##
##                          2010 2011
##   Atlanta Hawks             0    0
##   Boston Celtics            0    0
##   Brooklyn Nets             0    0
##   Charlotte Hornets         0    0
##   Chicago Bulls             0    0
##   Cleveland Cavaliers       0    0
##   Dallas Mavericks          0    0
##   Denver Nuggets            0    0
##   Detroit Pistons          41   33
```

Using OR (|) operator in R

```
nba7 <- nba2009_2016[nba2009_2016$away.TEAM_NAME=="Detroit Pistons" |
                     nba2009_2016$home.TEAM_NAME =="Detroit Pistons",]

head(nba7)
```

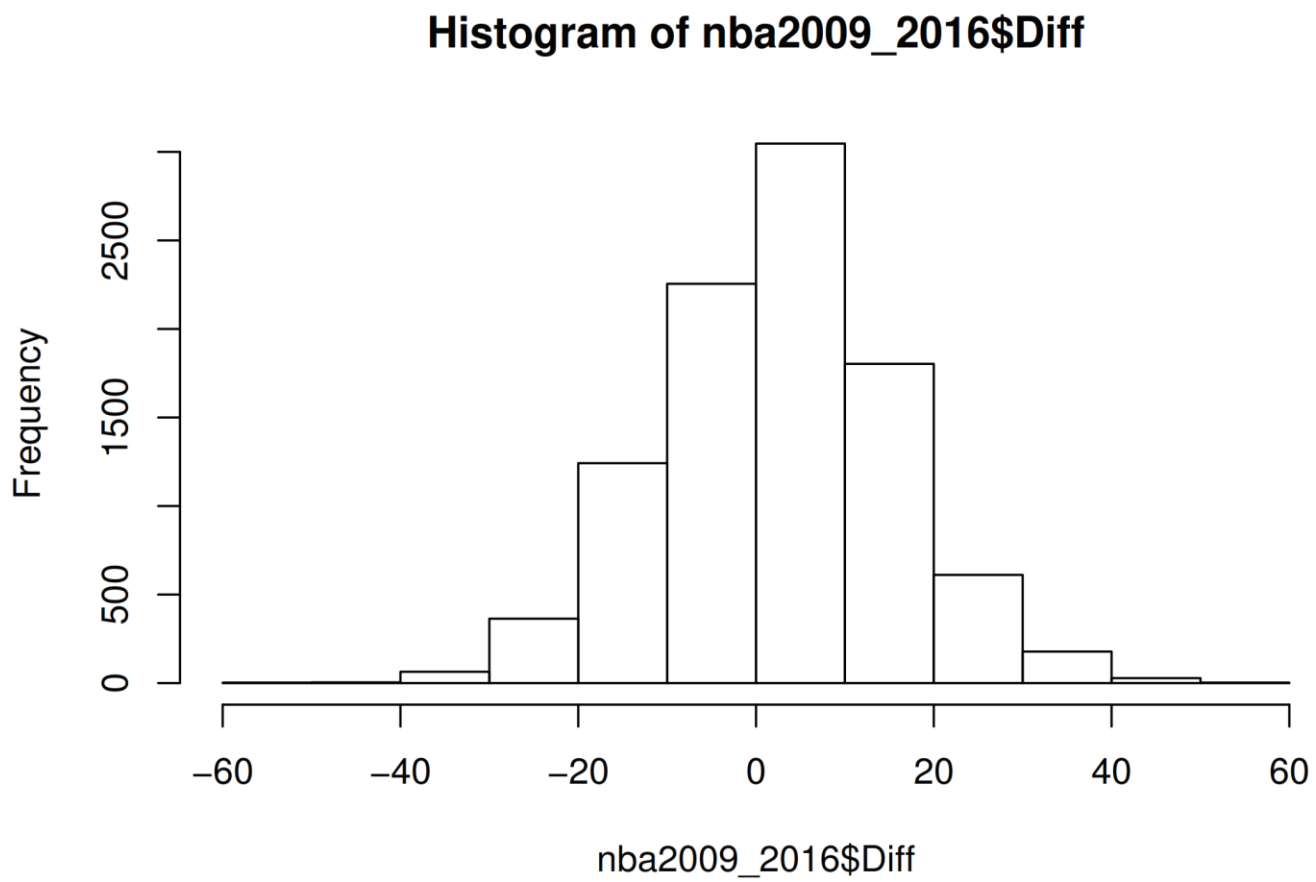
```
##    SEASON_ID  GAME_DATE home.TEAM_ABBREVIATION    home.TEAM_NAME home.PTS
## 10      2009 2009-10-28                     MEM Memphis Grizzlies       74
## 23      2009 2009-10-30                     DET   Detroit Pistons       83
## 36      2009 2009-10-31                     MIL   Milwaukee Bucks       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
```

Subseting data: dataframes

Adding new variable in dataframe

- Point differential

```
nba2009_2016$Diff <- nba2009_2016$home.PTS-nba2009_2016$away.PTS
hist(nba2009_2016$Diff)
```



Indexing list

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

```
##              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)
list1
```

```
## [[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

Indexing list

- [] is used to select multiple elements from the list
- [[]] is used to select multiple elements from the list

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

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

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

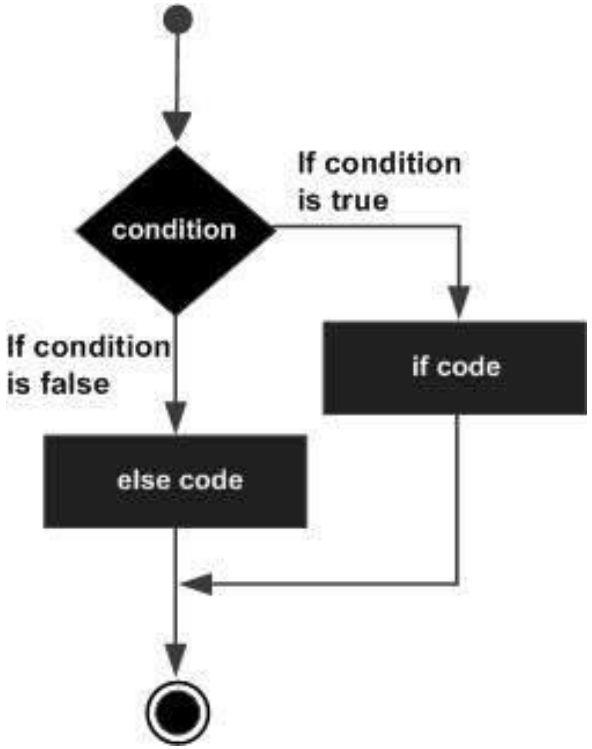
```
list1[[1]][2]
```

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

Loops and statements

If statements

- if statement - executes some code if one condition is true
- if...else statement - executes some code if a condition is true and another code if that condition is false
- if...else if....else statement - executes different codes for more than two conditions



If statements

```
if(boolean_expression) {
        // statement(s) will execute if the boolean expression is true.
} else {
        // statement(s) will execute if the boolean expression is false.
 }
```

If statements

Nested else if

```
if ( test_expression1) {
statement1
 } else if ( test_expression2) {
statement2
} else if ( test_expression3) {
 statement3
 } else {
 statement4 }
```

If statements

```
x <- 5
if (x==5) {
  y <- x*5
}
y
```

```
## [1] 25
```

```
x <- 2
if (x==5) {
  y <- x*5
} else {
  y<-x
}
y
```

If statements

If the value in the vector is negative return square of it If the value in the vector is 0 return 1 If the value in the vector is positive return square root of it

```
x <- 4
if (x<0){
  x <- x^2
} else if (x==0) {
  x <- 1
} else {
  x <- sqrt(x)
}
x
```

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

Try negative number

```
x <- -4
if (x<0){
  x <- x^2
} else if (x==0) {
  x <- 1
} else {
  x <- sqrt(x)
}
x
```

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

for loops

Loops are used in programming to repeat a specific block of code. In this article, you will learn to create a for loop in R programming.

There are three different looping constructs in R.

- repeat
- while
- for

We are going to cover the ***for*** loop

For loop iterates through each item in a vector or list

for (var in array) expression

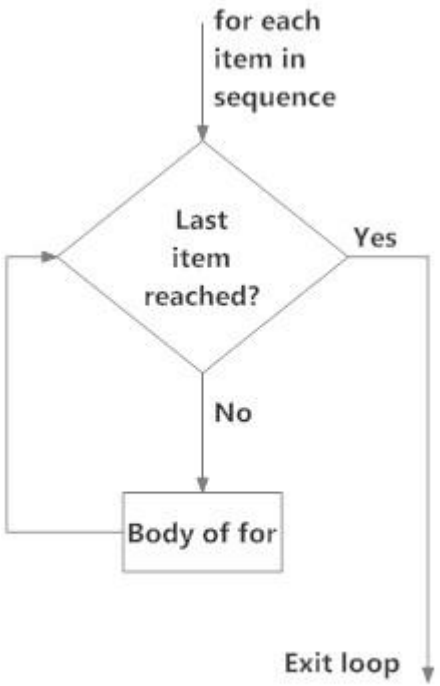


Fig: operation of for loop

for loops

Loop iterates over the vector (1,2,3,4,5) and prints each element

```
for (i in 1:5){
  print(i)
}
```

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

for loops

- Loop iterates over the vector x and add 2 to each value of the vector x.
- Returns vector y

```
x <- c(5,7,8,9)
for (i in x){
  y <- x+2
}

y
```

for loops

- mtcars is a built in dataset in R
- to call it to environment use data(mtcars)
- check dimensions (number of rows and columns) with function dim()
- check number of columns with the function ncol()

```
data(mtcars)
dim(mtcars)
```

```
## [1] 32 11
```

```
ncol(mtcars)
```

```
## [1] 11
```

for loops

We want to make a loop that will calculate mean of each columns

- First we create an empty vector x. This vector is going to be filled with the calculated means
- as we want to iterate over all columns, we set 1:ncol(mtcars)
- x1 is a variable containing mean for a column
- The index i is used to subset mtcars (mtcars[,i])
- Then all the means are appened into a vector x using c()

```
x<-c()
for (i in 1:ncol(mtcars)){
  x1 <- mean(mtcars[,i])
  x <- c(x,x1)
}
 x
```

for loops

Other way to populate the vector x

```
x<-c()
for (i in 1:ncol(mtcars)){
  x1 <- mean(mtcars[,i])
  x[i] <- x1
}
 x
```

```
##   [1]  20.090625   6.187500 230.721875 146.687500   3.596563   3.217250
##   [7]  17.848750   0.437500   0.406250   3.687500   2.812500
```

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.
- **Generically, 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

functions runif generate 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

```
formals(runif)
```

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

```
args(runif)
```

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

Matching arguments:

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] 1.788855 1.073246 1.987114 1.170596 1.047545 1.641882 1.354573
##  [8] 1.841162 1.845753 1.177541
```

Matching by position

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

```
##  [1] 1.105142 1.024699 1.590241 1.502900 1.181286 1.082911 1.168374
##  [8] 1.053954 1.777667 1.686507
```

matched by position, this will give you an error

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

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

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

If the argument has a default value and is not defined then the default value is used.

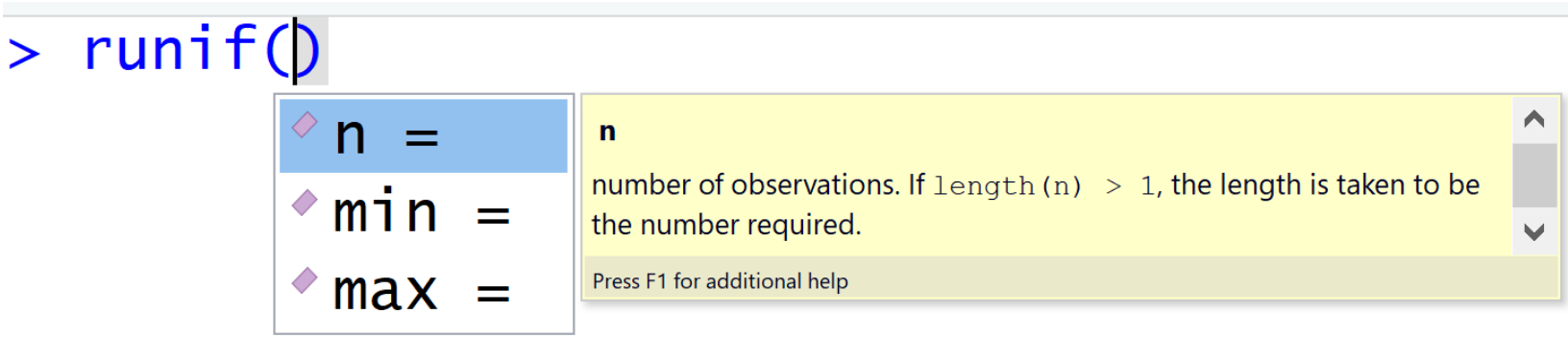
Generated 10 numbers with min=0, max=1

```
runif(10)
```

```
##  [1] 0.47445298 0.08305635 0.45821497 0.66941373 0.31394192 0.96459659
##  [7] 0.60195719 0.43008084 0.43477999 0.44796031
```

Functions

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



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 need 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)

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

foo(4)
```

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

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

foo(3)
```

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

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
##  [6] -0.33028740 -0.96078893  0.71501778  0.44954345 -0.14777380
## [11] -0.38006384 -0.61235388 -0.46302456 -0.81145962 -1.60788262
## [16] -1.60788262 -0.89442035  2.04238943  1.71054652  2.29127162
## [21]  0.23384555 -0.76168319 -0.81145962 -1.12671039 -0.14777380
## [26]  1.19619000  0.98049211  1.71054652 -0.71190675 -0.06481307
## [31] -0.84464392  0.21725341
```

- Create a function that will do Max-Min normalization
- use functions max() and min() to get the minimum and maximum values in the vector

$$z_i = \frac{x_i - \min(x)}{\max(x) - \min(x)}$$

Test it on one of the columns in mtcars

Functions

- Create a function that will do the following
  - apply max-min normalization on each variable in the dataframe
  - return new dataframe with normalized variables
    - You need to define your max-min function for a vector inside the function
    - You can use for loop inside the function
    - to create a dataframe use cbind()

- What you think what is the most likely situation when the function will give an error ?

- Apply function on mtcars

- How can you make the function any better ?

Dates and Times in R

Dates in R

- R provides several options for dealing with date and date/time data.

- The builtin ***as.Date*** function handles dates (without times);
- She contributed 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 timezone information, the chron library is a good choice; the POSIX classes are especially useful when timezone 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.

- as.Date function stores the number of days passed from 1971-01-01

```
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

Dates in R

```
oil <- read.csv("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
```

```
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 dataframe with DATE column.
- The following code is going to create ne dataframe 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
```

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
```

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 columns is character, but you can make it numeric

```
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" ...
```

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
```

Tidy data

Definitions

- **Variable**: A quantity, quality, or property that you can measure.
- **Observation**: A set of values that display the relationship between variables. To be an observation, values need to be measured under similar conditions, usually measured on the same observational unit at the same time.
- **Value**: The state of a variable that you observe when you measure it.

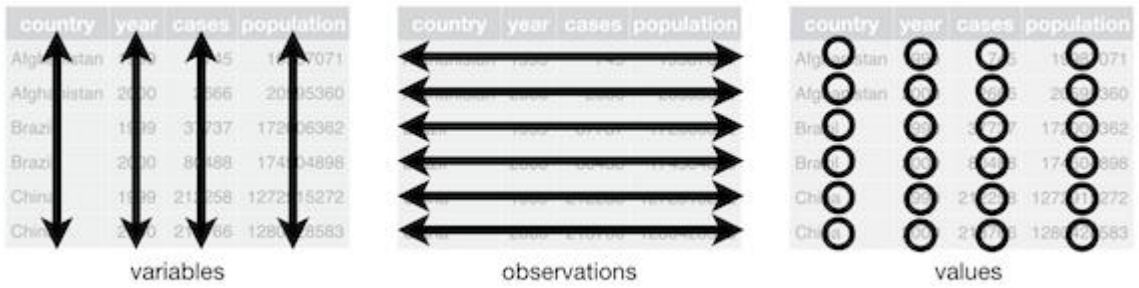
Source: vignette("tidy-data")

Principles of Tidy data

1.Each variable is in its own column

2.Each observation is in its own row

3.Each value is in its own cell



Tidy Data: Column names are values not variables

Example of messy data

Tidy Data

religion	<$10k	$10–20k	$20–30k	$30–40k	$40–50k	$50–75k
Agnostic	27	34	60	81	76	137
Atheist	12	27	37	52	35	70
Buddhist	27	21	30	34	33	58
Catholic	418	617	732	670	638	1116
Don't know/refused	15	14	15	11	10	35
Evangelical Prot	575	869	1064	982	881	1486
Hindu	1	9	7	9	11	34
Historically Black Prot	228	244	236	238	197	223
Jehovah's Witness	20	27	24	24	21	30
Jewish	19	19	25	25	30	95

Tidy Data: Column names are values not variables

Tidy data for the previous example

religion	income	freq
Agnostic	<$10k	27
Agnostic	$10–20k	34
Agnostic	$20–30k	60
Agnostic	$30–40k	81
Agnostic	$40–50k	76
Agnostic	$50–75k	137
Agnostic	$75–100k	122
Agnostic	$100–150k	109
Agnostic	>150k	84
Agnostic	Don't know/refused	96

Tidy Data: Multiple variables stored in one column

country	year	column	cases
AD	2000	m014	0
AD	2000	m1524	0
AD	2000	m2534	1
AD	2000	m3544	0
AD	2000	m4554	0
AD	2000	m5564	0
AD	2000	m65	0
AE	2000	m014	2
AE	2000	m1524	4
AE	2000	m2534	4
AE	2000	m3544	6
AE	2000	m4554	5
AE	2000	m5564	12
AE	2000	m65	10
AE	2000	f014	3

Not tidy yet

country	year	sex	age	cases
AD	2000	m	0–14	0
AD	2000	m	15–24	0
AD	2000	m	25–34	1
AD	2000	m	35–44	0
AD	2000	m	45–54	0
AD	2000	m	55–64	0
AD	2000	m	65+	0
AE	2000	m	0–14	2
AE	2000	m	15–24	4
AE	2000	m	25–34	4
AE	2000	m	35–44	6
AE	2000	m	45–54	5
AE	2000	m	55–64	12
AE	2000	m	65+	10
AE	2000	f	0-14	3

Tidy

Tidy Data: Wide and Long formats

Wide data has an individual's multiple observations distributed across columns, and long data has an individual's repeated observations distributed across rows.

Wide format: every row is an observation

```
data("nba_east")
head(nba_east)
```

```
##   Rank              Team  W  L   Pct   GB Season
## 1    1      Chicago Bulls 61 21 0.744    -   1991
## 2    2     Boston Celtics 56 26 0.683  5.0   1991
## 3    3    Detroit Pistons 50 32 0.610 11.0   1991
## 4    4    Milwaukee Bucks 48 34 0.585 13.0   1991
## 5    5 Philadelphia 76ers 44 38 0.537 17.0   1991
## 6    6      Atlanta Hawks 43 39 0.524 18.0   1991
```

Long format: every row is key-value combination

```
##                Team variable value
## 1      Chicago Bulls     Rank     1
## 2     Boston Celtics     Rank     2
## 3    Detroit Pistons     Rank     3
## 4    Milwaukee Bucks     Rank     4
## 5 Philadelphia 76ers     Rank     5
## 6      Atlanta Hawks     Rank     6
```

1 2 3

1. ID
2. Key
3. Value

Tidy Data

There are many reasons to prefer datasets structured in long form. Repeating some of the points made in Hadley Wickham's paper on the topic, here are three reasons why you should attempt to structure your data in long form:

- If you have many value variables, it is difficult to summarize wide-form datasets at a glance (which in turn makes it hard to identify mistakes in the data). For example, imagine we have a dataset with 50 years and 10 value variables of interest — this would result in 500 columns in wide form. Summarizing each column to look for strange observations, or simply understanding which variables are included in the dataset, becomes difficult in this case.

- Structuring data as key-value pairs — as is done in long-form datasets — facilitates conceptual clarity.

- Long-form datasets are often required for advanced statistical analysis and graphing. Many graphing packages, including ggplot, rely on your data being in long form.

Tidy Data

- There are several R packages that can help with making data tidy or converting between data types such as tidyr, reshape, dplyr etc
- R basic functionality allows to do this data manipulation as well.

```
library(reshape2)
nba_long <- melt(nba_east, id.vars = c("Team", "Season"))
head(nba_long)
```

```
##                 Team Season variable value
## 1      Chicago Bulls   1991     Rank     1
## 2     Boston Celtics   1991     Rank     2
## 3    Detroit Pistons   1991     Rank     3
## 4    Milwaukee Bucks   1991     Rank     4
## 5 Philadelphia 76ers   1991     Rank     5
## 6      Atlanta Hawks   1991     Rank     6
```

Tidy Data: From Long to wide

- The data combines information on NBA 2015 regular season.
- Each game (given by GAME_ID) is recorded by two rows, one for Home team and another for Away team (column H)

```
nba15 <- read.csv("nba15.csv")
head(nba15)
```

```
##         GAME_DATE_EST  GAME_ID TEAM_ABBREVIATION PTS H
## 1 2010-10-26T00:00:00 21000001               BOS  88 H
## 2 2010-10-26T00:00:00 21000001               MIA  80 A
## 3 2010-10-26T00:00:00 21000002               POR 106 H
## 4 2010-10-26T00:00:00 21000002               PHX  92 A
## 5 2010-10-26T00:00:00 21000003               LAL 112 H
## 6 2010-10-26T00:00:00 21000003               HOU 110 A
```

The data is given in a long format, we want to get it into wide format such that each row will represent one game

Tidy Data: From Long to wide

Arguments

- timevar –the variable in long format that differentiates multiple records from the same group or individual.
- idvar – variable(s) in long format that identify multiple records from the same group/individual.
- direction – whether you want to get the output in ***long*** or ***wide*** format

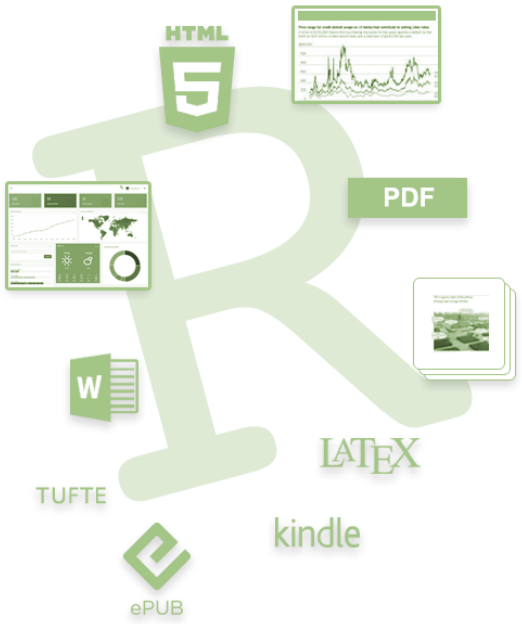
```
nba15_long <- reshape(nba15, timevar = "H", idvar="GAME_ID", direction="wide")
head(nba15_long)
```

```
##      GAME_ID     GAME_DATE_EST.H TEAM_ABBREVIATION.H PTS.H
## 1   21000001 2010-10-26T00:00:00                 BOS    88
## 3   21000002 2010-10-26T00:00:00                 POR   106
## 5   21000003 2010-10-26T00:00:00                 LAL   112
## 7   21000004 2010-10-27T00:00:00                 CLE    95
## 9   21000005 2010-10-27T00:00:00                 NJN   101
## 11  21000006 2010-10-27T00:00:00                 PHI    87
##         GAME_DATE_EST.A TEAM_ABBREVIATION.A PTS.A
## 1   2010-10-26T00:00:00                 MIA    80
## 3   2010-10-26T00:00:00                 PHX    92
## 5   2010-10-26T00:00:00                 HOU   110
## 7   2010-10-27T00:00:00                 BOS    87
## 9   2010-10-27T00:00:00                 DET    98
## 11  2010-10-27T00:00:00                 MIA    97
```

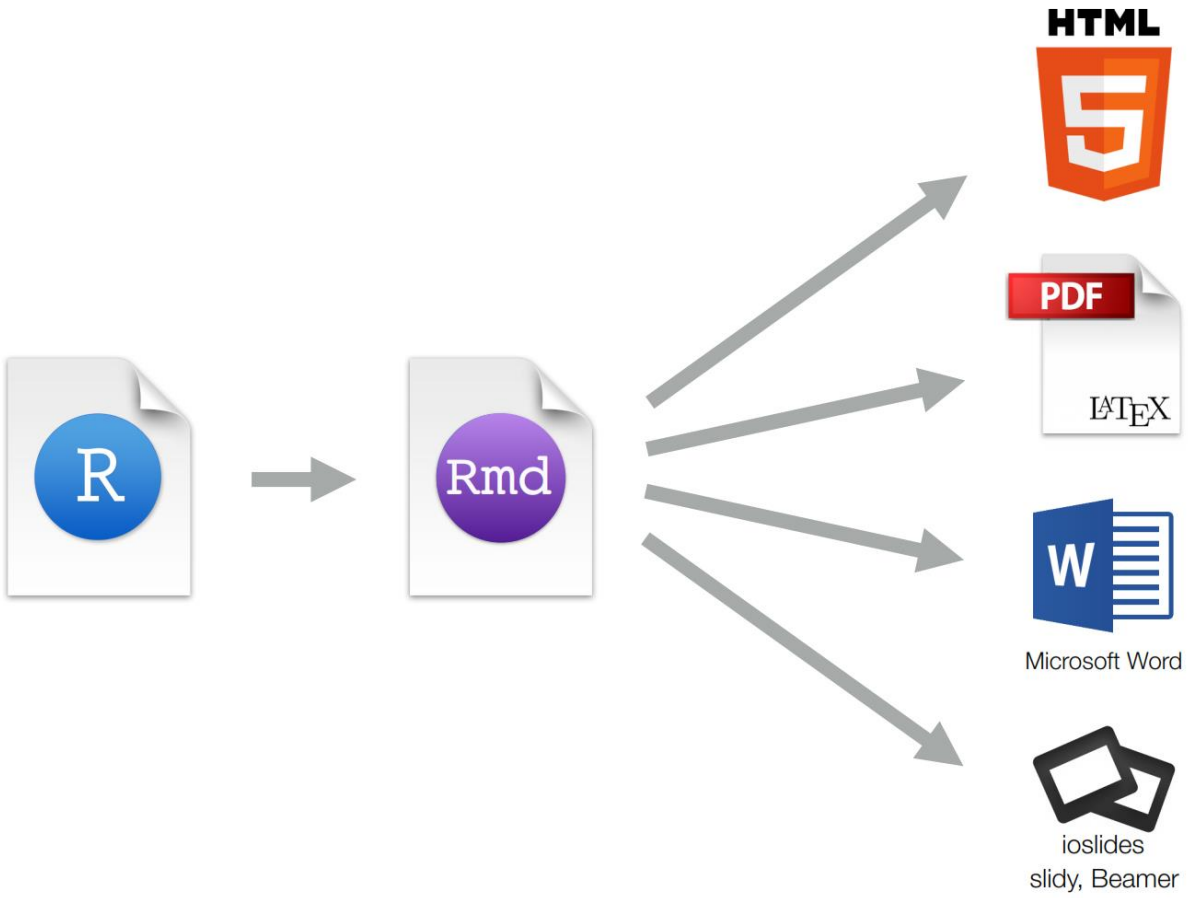
R markdown

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

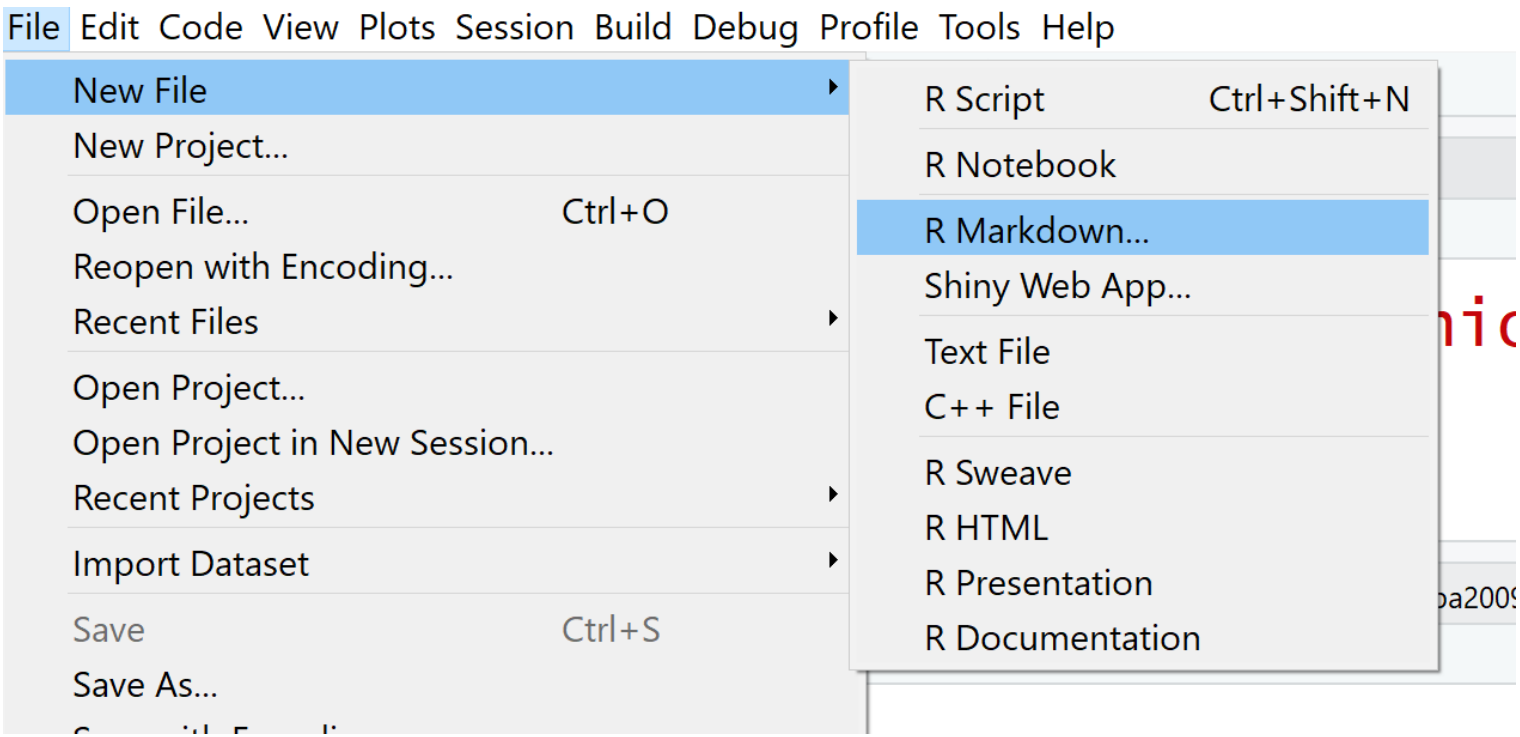


The workflow



R markdown

To open new markdown file



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 ```**{r}** and ```; 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)
```
```

1

2

3

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. This is what appears in the output document:

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 automatable.

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.

LaTex formula example

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

In Tex

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

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

Lists in R Markdown

To make a bulleted list in Markdown, place each item on a new line after an asterisk 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 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 ```` ```{r} ```` and one that contains ```` ``` ````. 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.

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).

Other important stuff

Emphasis

```
*italic*   **bold**

_italic_   __bold__
```

Headers

```
# Header 1

## Header 2

### Header 3
```

If you want to start from a new page

Put in text part of the markdown

\newpage

\pagebreak

Running python script in markdown

- you need to install package ***reticulate***
- and of course all python distributions

Lib reticulate is developed by R Studio to add functionality of using Python and other languages in R Studio environment.

- You can access objects created with python in R using py$object and objects created with R in python using r$object

R markdown

```{python}
import pandas as pd
winter_pd = pd.read_csv("winter.csv")
print(winter_pd.head())
```

Access object created with python in R
first call the library(reticulate)

```{r}
library(reticulate)
```

```{r}
head(py$winter_pd)
```

Knitting with parameter

- Allows you to knit a report based on some parameter
- Lets say you want to generate report on number of medals won during Olympics games for each country

If you change the value for parameter c to the name of another country you will get report for that country, try for “FRA”

```
---
title: "winter games"
author: "Habet Madoyan"
date: "June 16, 2018"
output: pdf_document

params:
     c: "RUS"
---

```{r}
## Read the parameter
country <- params$c
## Run the code
winter <- read.csv("winter.csv")
winter_count <- winter[winter$Country==country,]
head(winter_count)
```

Count number of medals for `r country`

```{r}
table(winter_count$Medal)
```
```

R markdown

- Can we generate reports for all countries with a single code

```
# Dont run this if you already have the dataframe in the environment
winter <- read.csv("winter.csv")
```

```
countries <- unique(winter$Country)
countries
```

```
##  [1] FRA SUI FIN BEL GBR SWE CAN USA AUT NOR GER TCH HUN ITA FRG NED URS
## [18] EUA JPN POL PRK ROU GDR ESP LIE BUL YUG EUN KOR CHN LUX NZL RUS UKR
## [35] BLR AUS SLO KAZ UZB DEN CZE CRO EST LAT SVK
## 45 Levels: AUS AUT BEL BLR BUL CAN CHN CRO CZE DEN ESP EST EUA EUN ... YUG
```

Subset the first 5 countries

```
countries <- countries[1:5]
```

The loop in R script

```
for (x in countries){
  rmarkdown::render(input = 'by country games all.Rmd',
                    # Generate the output pdf file
                    output_file =  paste("games_", x,  ".pdf", sep=''),
                    # Save in directory
                    output_dir = 'reports')
}
```

Markdown

````
```{r}
## Read the x
country <- x
## Run the code
winter <- read.csv("winter.csv")
winter_count <- winter[winter$Country==country,]
head(winter_count)
```

## Count number of medals for `r country`

```{r}
table(winter_count$Medal)
```
````