CME/STATS 195

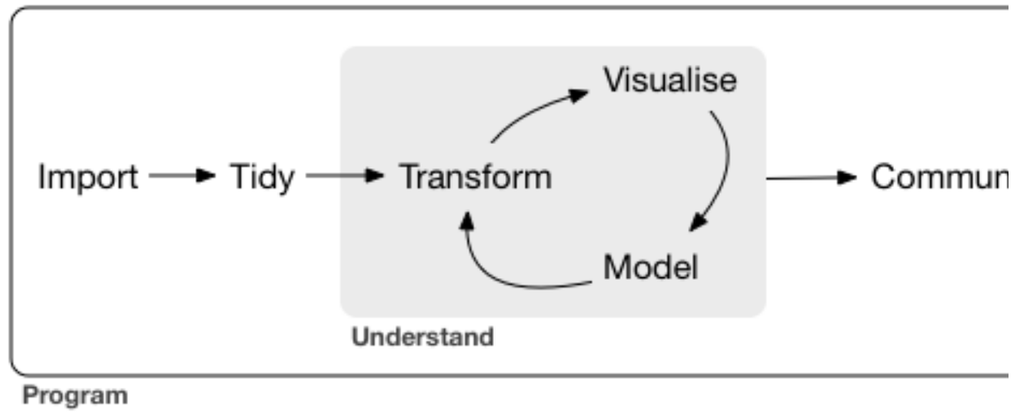
Lecture 2: Communicating using R Markdow & Elements of Programming

Lan Huong Nguyen

October 2, 2018

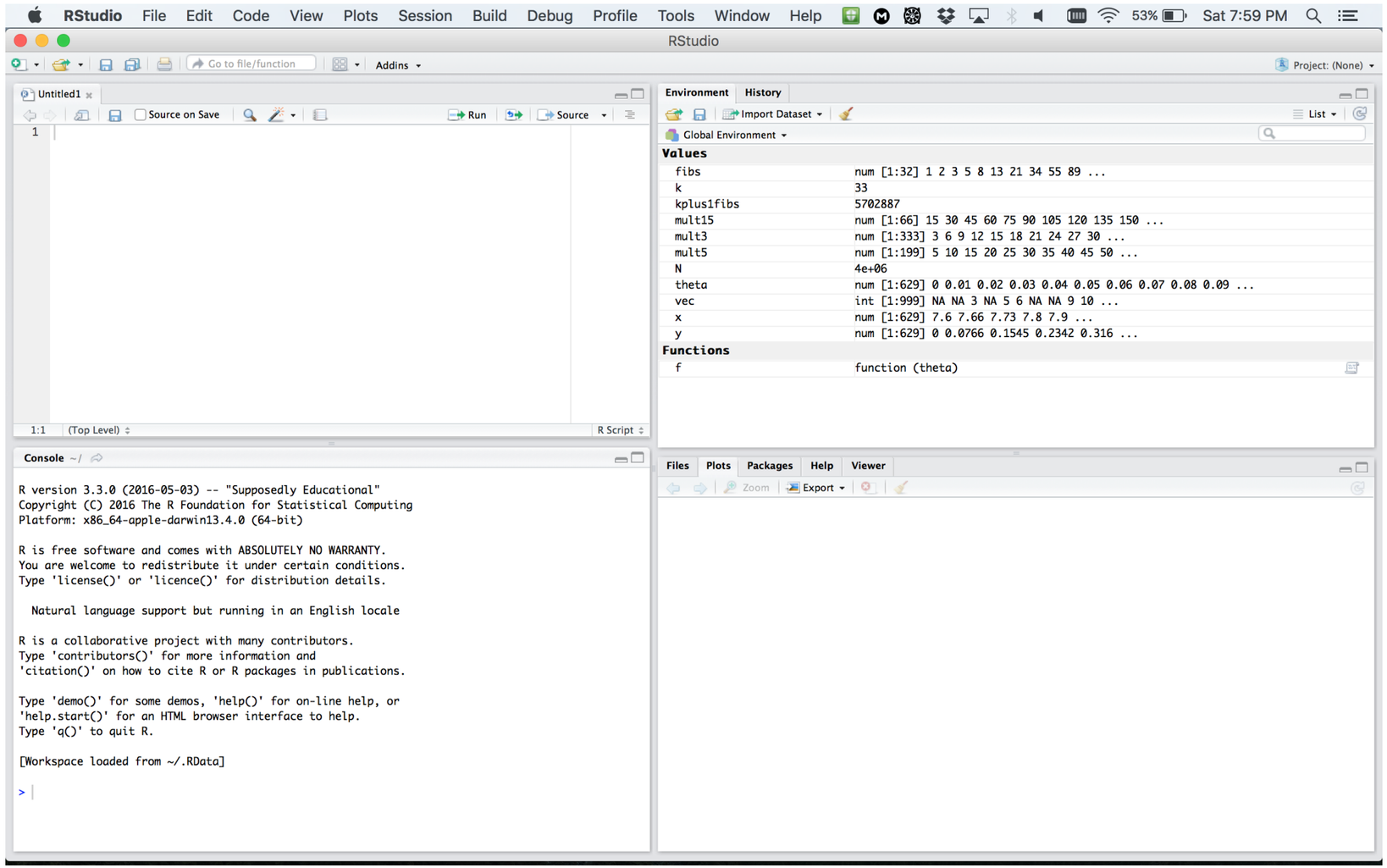
Contents

- Working with R:
  - Using RStudio
  - Communicating with R Markdown
- Programming
  - Syntax
  - Control flow statements
  - Functions

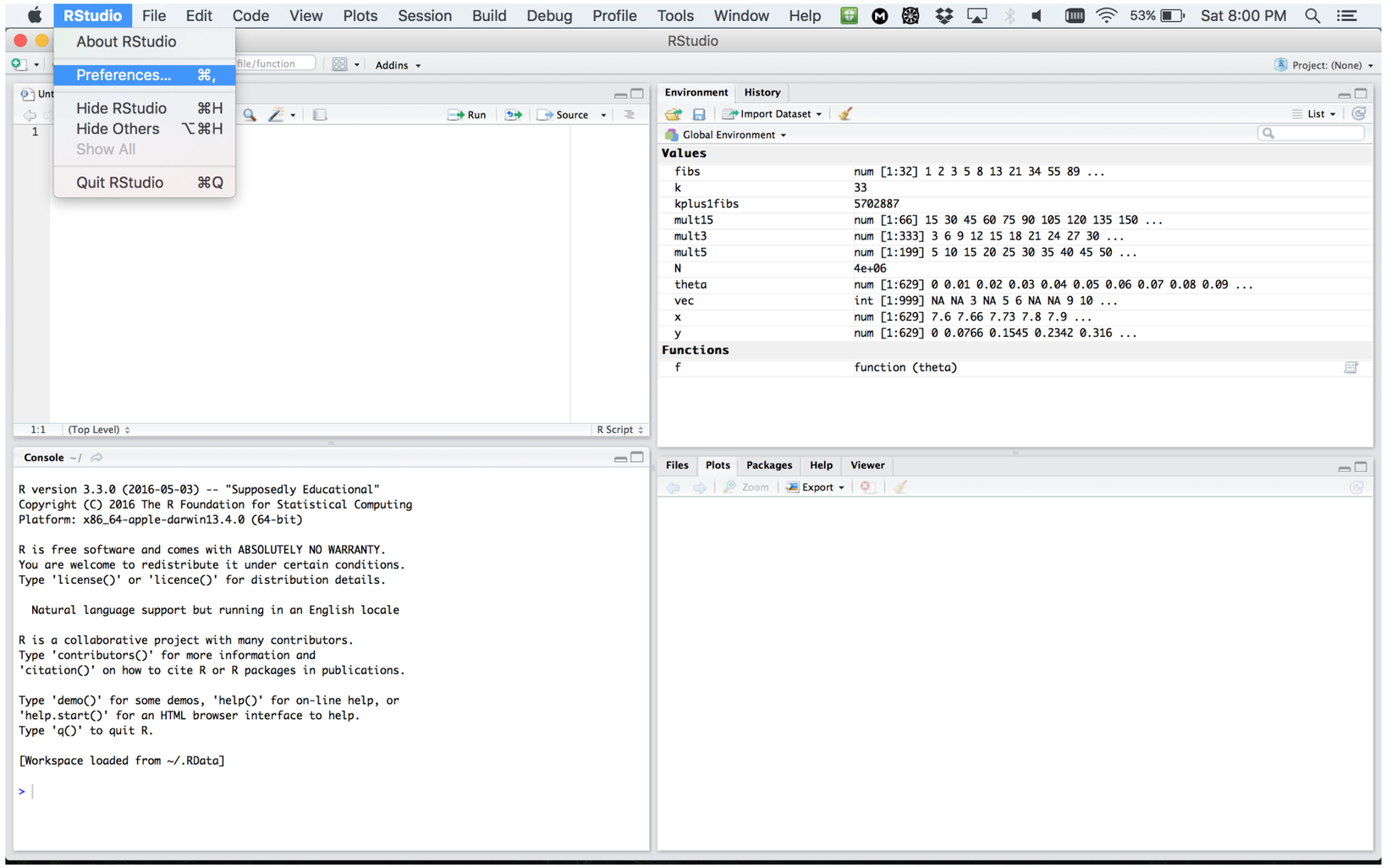


Using RStudio

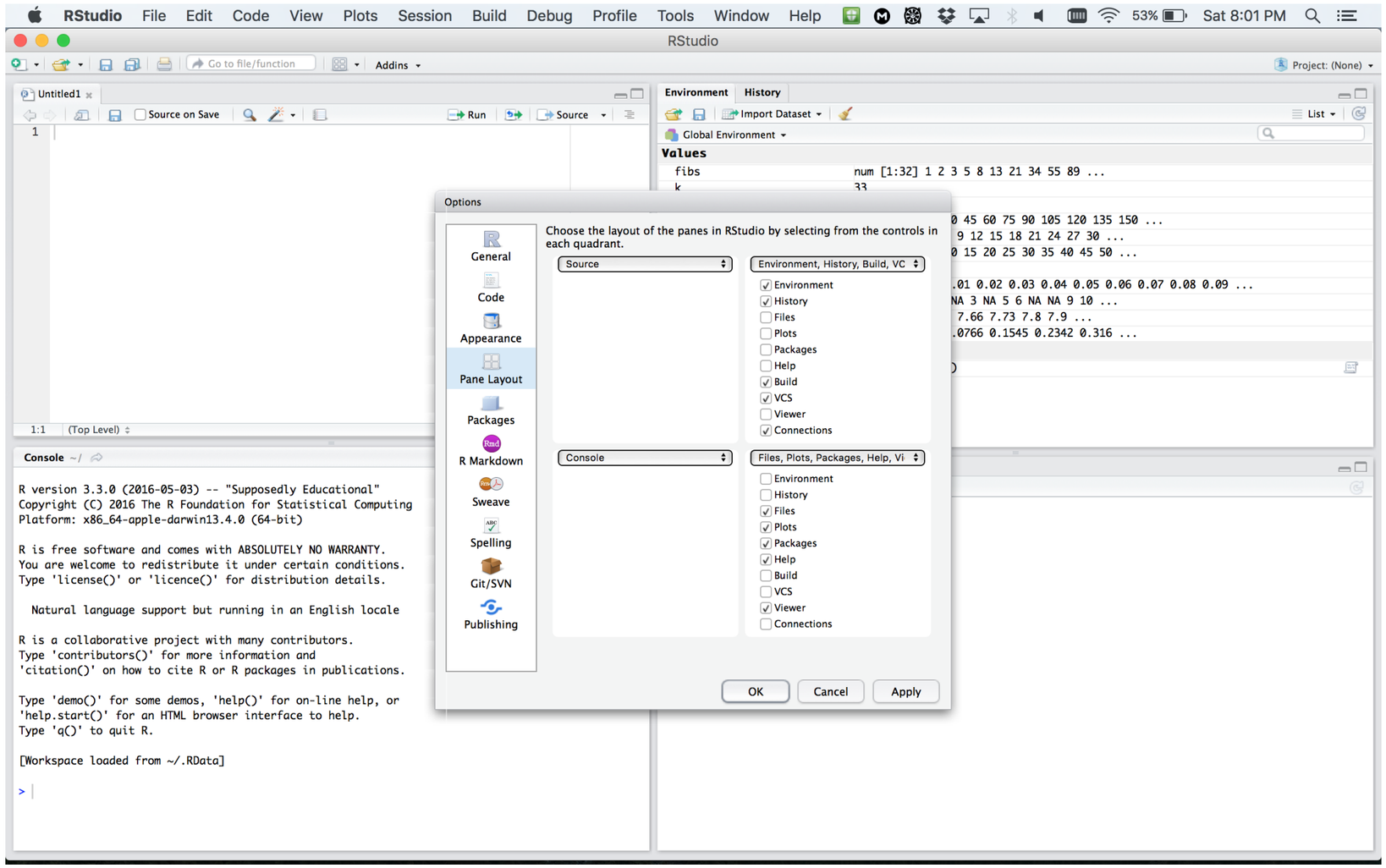
RStudio window



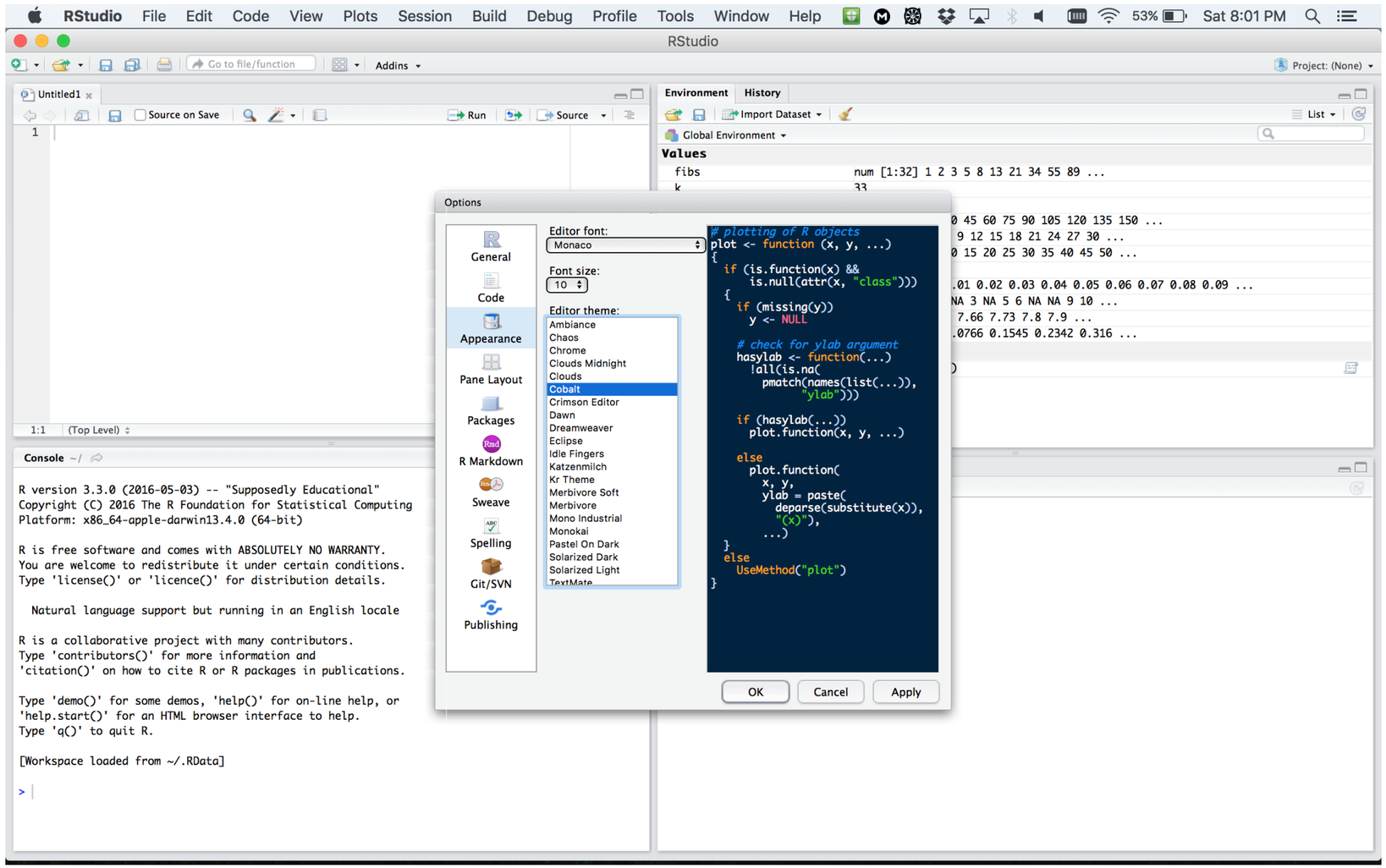
RStudio preferences



RStudio layout

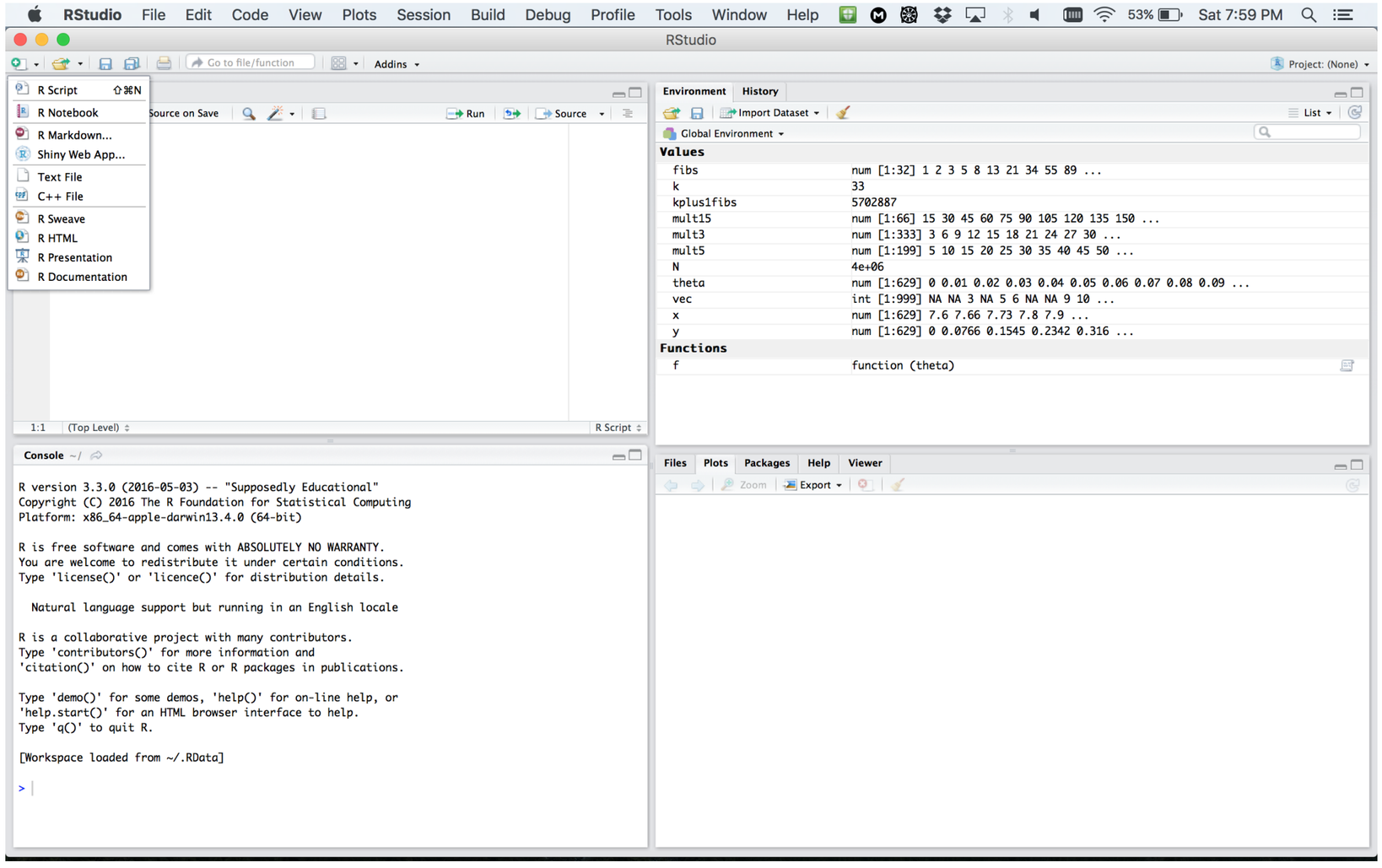


RStudio apprearance



More on RStudio cuztomization can be found here

R document types



R document types

- R Script **a text file** containing R commands stored together.
- R Markdown files can generate high quality reports contatining notes, code and code outputs. **Python and bash code** can also be executed.
- R Notebook is an R Markdown document with **chunks that can be executed independently and interactively**, with output visible immediately beneath the input.
- R presentation let's you author **slides** that make use of R code and LaTeX equations as **straightforward** as possible.
- R Sweave enables the embedding of **R code within LaTeX documents**.
- Other documents

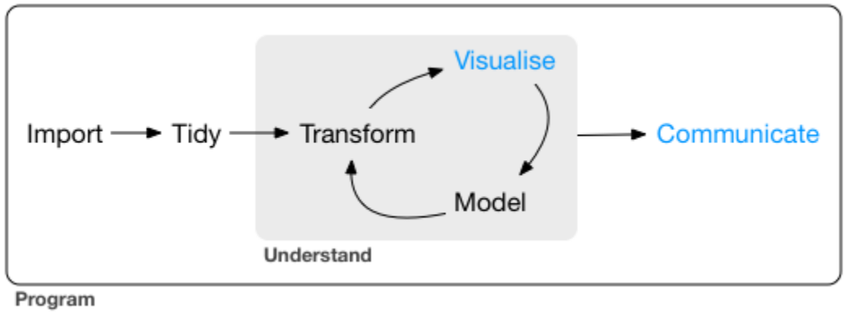
Communicating with R Markdown

R Markdown

R Markdown provides an unified authoring framework for data science, combi
your code, its results, and your prose commentary.

R Markdown was designed to be used:

- for communicating your conclusions with people who do not want to focus on the code behind the analysis.
- for collaborating with other data scientists, interested in both conclusions, and the code.
- as a modern day lab notebook for data science, where you can capture both your work and your thought process.



R Markdown sourse files

R Markdown files are a plain text files with ".Rmd" extension.

````
---
title: "Title of my first document"
date: "2018-09-27"
output: html_document
---

# Section title

```{r chunk-name, include = FALSE}
library(tidyverse)
summary(cars)
```

## Subsection title

```{r pressure, echo=FALSE}
plot(pressure)
```

Note that the `echo = FALSE` parameter was added to the code chunk to
prevent printing of the R code that generated the plot.
````

The documents must contain **YAML header** marked with dashes. You can ass b
code chunks and **plain text**. Sections and subsections are marked with hashtag

Compiling R Markdown files

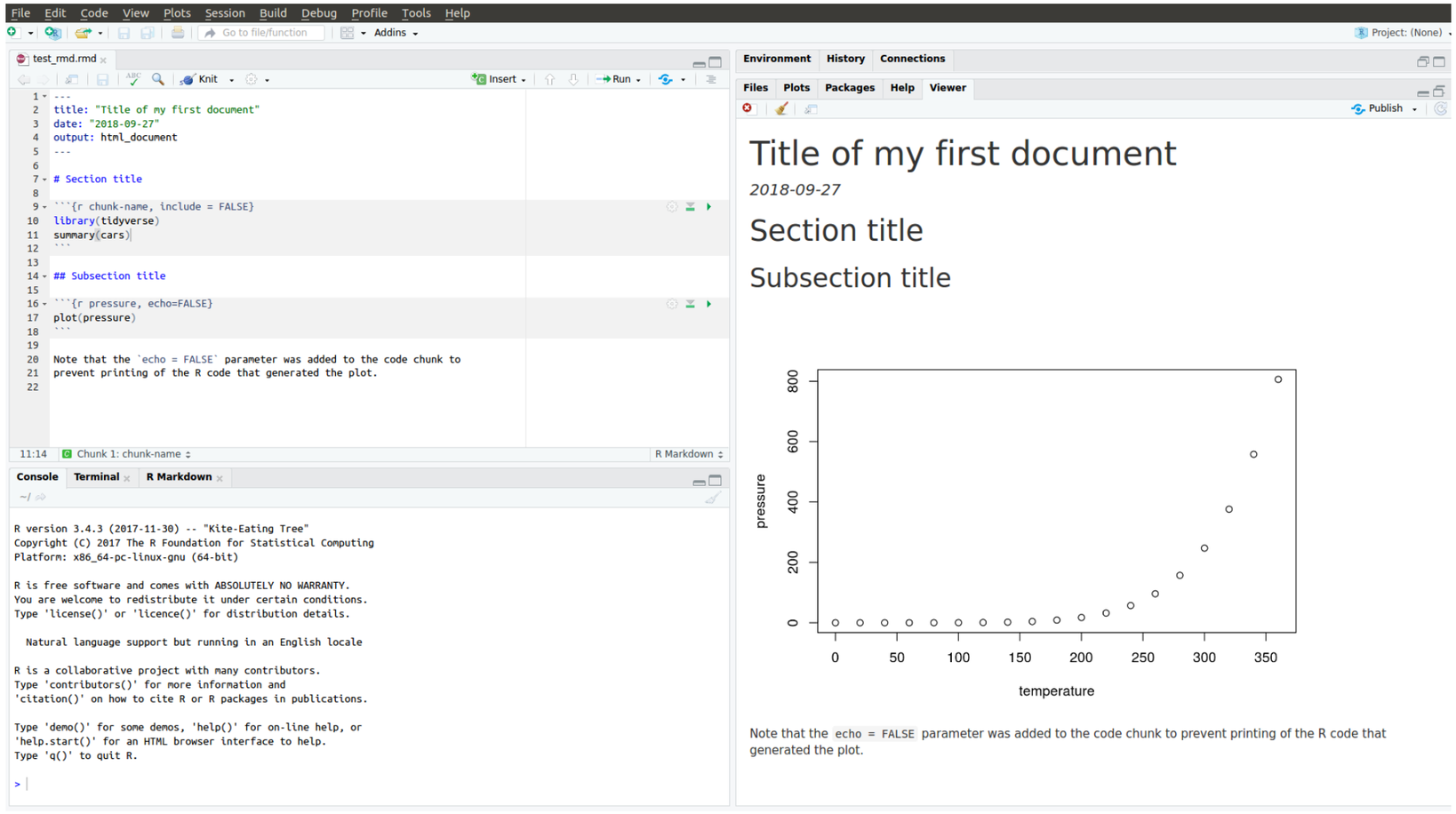
To produce a complete report containing all text, code, and results:

- In RStudio, click on "Knit" or press `Cmd/Ctrl + Shift + K`.
- From the R command line, type rmarkdown::render("filename.Rmd")

This will display the report in the viewer pane, and create a self-contained HTM file that you can share with others.

After compiling the R Markdown document from the previous slide, you get thi html.

Viewing the report in RStudio



YAML header

A YAML header is a set of `key: value` pairs at the start of your file. Begin and the header with a line of three dashes (- - -), e.g.

```
---
title: "Untitled"
author: "Anonymous"
output: html_document
---
```

You can tell R Markdown what type of document you want to render: `html_document` (default), `pdf_document`, `word_document`, `beamer_presentation` etc.

You can print a table of contents (toc) with the following:

```
---
title: "Untitled"
author: "Anonymous"
output:
  html_document:
    toc: true
---
```

Text in R Markdown

In ".Rmd" files, prose is written in Markdown, a lightweight markup language wi
plain text files formating syntax.

Section headers/titles:

```
# 1st Level Header

## 2nd Level Header

### 3rd Level Header
```

Text formatting:

```
*italic* or _italic_
**bold** __bold__

`code`
superscript^2^ and subscript~2~
```

Text in R Markdown

Lists:

```
* unordered list
* item 2
    + sub-item 1
    + sub-item 2

1. ordered list
1. item 2. The numbers are incremented automatically in the output.
```

Links and images:

```
<http://example.com>

[linked phrase](http://example.com)

![optional caption text](path/to/img.png)
```

Text in R Markdown

Tables:

```
Table Header | Second Header
-------------| -------------
Cell 1       | Cell 2
Cell 3       | Cell 4
```

Math formulae

```
$\alpha$ is the first letter of the Greek alphabet.

Using $$ prints a centered equation in the new line.

$$\sqrt{\alpha^2 + \beta^2} = \frac{\gamma}{2}$$
```

Code chunks

In R Markdown R code must go inside code chunks, e.g.:

````
```{r chunk-name}
 x <- runif(10)
 y <- 10 * x + 4
 plot(x, y)
```
````

Keyboard shortcuts:

- Insert a new code chunk: **`Ctrl/Cmd + Alt + I`**
- Run current chunk: **`Ctrl/Cmd + Shift + Enter`**
- Run current line (where the cursor is): **`Ctrl/Cmd + Enter`**

Chunk Options:

Chunk output can be customized with options supplied to chunk header. Some default options are:

- `eval = FALSE` : prevents code from being evaluated
- `include = FALSE` : runs the code, but hides code and its output in the final document
- `echo = FALSE` : hides the code, but not the results, in the final document
- `message = FALSE` : hides messages
- `warning = FALSE` : hides warnings
- `results = 'hide'` : hides printed output
- `fig.show = 'hide'` : hides plots
- `error = TRUE` : does not stop rendering if error occurs

Inline code

You can evealuate R code in a middle of your text:

```
There are 26 in the alphabet, and 12 months in each year.
Today, there are `as.Date("2019-08-23") - Sys.Date()` days left till my next birthday.
```

There are 26 in the alphabet, and 12 months in a year. Today, there are 333 day left till my next birthday.

More on R Markdown

R Markdown is relatively young, and growing rapidly.

Official R Markdown website: (http://rmarkdown.rstudio.com)

Further reading and references:

- http://www.stat.cmu.edu/~cshalizi/rmarkdown
- http://r4ds.had.co.nz
- https://www.rstudio.com/resources/cheatsheets/

Some R Markdown advice

- See your future self as a collaborator.
- Ensure each notebook has a descriptive title and name.
- Use the header date to record start time
- Keep track of failed attempts
- If you discover an error in a data file, write code to fix it.
- Regularly knit the notebook
- Use random seeds before sampling.
- Keep track the versions of the packages you use, e.g. by including `sessionInfo()` command at the end of your document.

All the above will help you increase the reproduciblity of your work.

Programming: Syntax

Style Guide

- There are two main style conventions used in R:
    - Hadley Wickam style
    - Google R style
- You can use either of the two style guides or create your own customized style.
- But you should stay **consistent**, e.g. if you choose to assign variables with `<-`, stick to it and don't use `=`.

Curly braces

- An opening curly brace “{” should not go on its own line and be followed by a new line.
- A closing curly “}” brace can go on its own line.
- Indent the code inside curly braces.
- It’s ok to leave very short statements on the same line

```
# Good
if (y < 0 && debug) {
  message("Y is negative")
}
if (y == 0) {
  log(x)
} else {
  y ^ x
}

# Bad
if (y < 0 && debug)
message("Y is negative")

if (y == 0) {
  log(x)
}
else {
  y ^ x
}
```

```
if (y < 0 && debug) message("Y is negative")
```

Code Documentation

- Comment your code! They will be helpful when you read your code a month after you wrote it.
- In R each line of a comment should begin with a comment symbol “#”.

```r
# Function returns the answer to life,
# the universe and everything else
get_answer <- function(){
  return(42)
}
```

- Comments are not subtitles, i.e. don’t repeat the code in the comments.

```r
# Loop through all bananas in the bunch
for(banana in bunch) {
  # make the monkey eat one banana
  MonkeyEat(b)
}
```

- Use commented lines of - and = t break up your file into easily readable chunks.

```r
# Load data ----------------------------
# Plot data ----------------------------
```

Programming: Control flow

Booleans/logicals

Booleans are logical data types (TRUE/FALSE) associated with conditional statements, which allow different actions and change control flow.

```
# equal ""
5 == 5
```

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

```
# not equal: "!=""
5 != 5
```

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

```
# greater than: ">""
5 > 4
```

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

```
# greater than or equal: ">="" (# similarly < an
5 >= 5
```

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

```
# You can combine multiple boolean expressio
TRUE & TRUE
```

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

```
TRUE & FALSE
```

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

```
TRUE | FALSE
```

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

```
!(TRUE)
```

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

Booleans/logicals

In R if you combine 2 vectors of booleans, by each element then use &. Rember **recycling property** for vectors.

```
c(TRUE, TRUE) & c(FALSE, TRUE)
```

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

```
c(5 < 4, 7 == 0, 1< 2) | c(5==5, 6> 2, !FALSE)
```

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

```
c(TRUE, TRUE) & c(TRUE, FALSE, TRUE, FALSE)  # recycling
```

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

Booleans/logicals

If we use double operators && or || is used only the first elements are compare

```
c(TRUE, TRUE) && c(FALSE, TRUE)
```

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

```
c(5 < 4, 7 == 0, 1< 2) || c(5==5, 6> 2, !FALSE)
```

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

```
c(TRUE, TRUE) && c(TRUE, FALSE, TRUE, FALSE)
```

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

Booleans/logicals

- Another possibility to combine booleans is to use `all()` or `any()` functions:

```
all(c(TRUE, FALSE, TRUE))
```

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

```
any(c(TRUE, FALSE, TRUE))
```

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

```
all(c(5 > -1, 3 >= 1, 1 < 1))
```

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

```
any(c(5 > -1, 3 >= 1, 1 < 1))
```

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

Control statements

- **Control flow** is the order in which individual statements, instructions or function calls of a program are evaluated.
- Control statements allow you to do more complicated tasks.
- Their execution results in a choice between which of two or more paths should be followed.
  - `If/else`
  - `For`
  - `While`

If statements

- Decide on whether a block of code should be executed based on the associated boolean expression.
- **Syntax**. The if statements are followed by a boolean expression wrapped in parenthesis. The conditional block of code is inside curly braces {}.

```
if (traffic_light == "green") {
  print("Go.")
}
```

- 'if-else' statements let you introduce more options

```
if (traffic_light == "green") {
   print("Go.")
} else {
   print("Stay.")
}
```

- You can also use `else if()`

```
if (traffic_light == "green") {
   print("Go.")
} else if (traffic_light == "yellow") {
  print("Get ready.")
} else {
   print("Stay.")
}
```

For loops

- A for loop is a statement which **repeats the execution a block of code** a given number of iterations.

```
for (i in 1:5){
  print(i^2)
}
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
## [1] 25
```

While loops

- Similar to for loops, but repeat the execution as long **as the boolean condition supplied is TRUE**.

```
i = 1
while(i <= 5) {
  cat("i =", i, "\n")
  i = i + 1
}
```

```
## i = 1
## i = 2
## i = 3
## i = 4
## i = 5
```

Next

- `next` halts the processing of the current iteration and advances the looping index.

```
for (i in 1:10) {
  if (i <= 5) {
    print("skip")
    next
  }
  cat(i, "is greater than 5.\n")
}
```

```
## [1] "skip"
## [1] "skip"
## [1] "skip"
## [1] "skip"
## [1] "skip"
## 6 is greater than 5.
## 7 is greater than 5.
## 8 is greater than 5.
## 9 is greater than 5.
## 10 is greater than 5.
```

- `next` applies only to the innermo of nested loops.

```
for (i in 1:3) {
  cat("Outer-loop i: ", i, ".\n")
  for (j in 1:4) {
    if(j > i) {
      print("skip")
      next
    }
    cat("Inner-loop j:", j, ".\n")
  }
}
```

```
## Outer-loop i:  1 .
## Inner-loop j: 1 .
## [1] "skip"
## [1] "skip"
## [1] "skip"
## Outer-loop i:  2 .
## Inner-loop j: 1 .
## Inner-loop j: 2 .
## [1] "skip"
## [1] "skip"
## Outer-loop i:  3 .
## Inner-loop j: 1 .
## Inner-loop j: 2 .
## Inner-loop j: 3 .
## [1] "skip"
```

Break

- The `break` statement allows us to break out out of a for, while loop (of the smallest enclosing).
- The control is transferred to the first statement outside the inner-most loop.

```
for (i in 1:10)  {
  if (i == 6)  {
    print(paste("Coming out from for loop Where i = ", i))
    break
  }
  print(paste("i is now: ", i))
}
```

```
## [1] "i is now:  1"
## [1] "i is now:  2"
## [1] "i is now:  3"
## [1] "i is now:  4"
## [1] "i is now:  5"
## [1] "Coming out from for loop Where i =  6"
```

Exercise 1

- Go to “Lec2_Exercises.Rmd” in RStudio.
- Do Exercise 1.

Programming: Functions

Functions

- A **function** is a procedure/routine that performs a specific task.
- Functions are used to **abstract** components of larger program.
- They are like a mathematical functions. They **take some input and then do something to find the result**.
- Functions allow you to **automate common tasks** in a more powerful and general way than copy-and-pasting.
- A general rule is that you should **use a function, whenever you've copied and pasted a block of code more than twice**.

Function Definition

- To define a function you assigne a variable name to a `function` object.
- Functions take **arguments, mandatory and optional**.
- Provide the brief **description of your function in comments** before the function definition.

```r
# Computes mean and standard deviation
# and optionally prints the results.
mysummary <- function(x, print=TRUE) {
  center <- mean(x)
  spread <- sd(x)
  if (print) {
    cat("Mean =", center, "\n",
        "SD = ", spread, "\n")
  }
  result <- list(mean=center,
                 sd=spread)
  return(result)
}
```

Calling functions

```
x <- rnorm(n = 500, mean = 4, sd = 1)
y <- mysummary(x)
```

```
## Mean = 4.032558
##  SD =  1.010826
```

```
# without printing
y <- mysummary(x, print = FALSE)
```

```
# Results are stored in list "y"
y$mean
```

```
## [1] 4.032558
```

```
y$sd
```

```
## [1] 1.010826
```

```
# The order of arguments does not matter if the names are specified
y <- mysummary(print=FALSE, x = x)
```

apply, lapply, sapply functions

- The `apply` family functions, are **functions which manipulate slices of data** stored as matrices, arrays, lists and data-frames **in a repetitive way**.
- These functions **avoid the explicit use of loops**, and might be **more computationally efficient**, depending on how big a dataset is. For more details on runtimes see this link.
- `apply` allow you to perform operations with **very few lines of code.**
- The family comprises: **apply, lapply , sapply, vapply, mapply, rapply, and tapply**. The difference lies in the structure of input data and the desired format of the output).

apply function

`apply` operates on arrays/matrices.

In the example below we obtain column sums of matrix X.

```
(X <- matrix(sample(30), nrow = 5, ncol = 6))
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]   25   18   28   17   15   16
## [2,]    1    5   27   10    4   13
## [3,]   30    6   29   23    2   20
## [4,]   24   11    8   12   21    9
## [5,]   26    7    3   19   14   22
```

```
apply(X, MARGIN = 2 , FUN = sum)
```

```
## [1] 106  47  95  81  56  80
```

Note: that in a matrix `MARGIN = 1` indicates rows and `MARGIN = 2` indicates columns.

apply function

- `apply` can be used with **user-defined functions**:

```
print(X)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]   25   18   28   17   15   16
## [2,]    1    5   27   10    4   13
## [3,]   30    6   29   23    2   20
## [4,]   24   11    8   12   21    9
## [5,]   26    7    3   19   14   22
```

```
# number entries < 15
apply(X, 2, function(x) 10*x + 2)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]  252  182  282  172  152  162
## [2,]   12   52  272  102   42  132
## [3,]  302   62  292  232   22  202
## [4,]  242  112   82  122  212   92
## [5,]  262   72   32  192  142  222
```

- The function can be defined outside `apply()`,

```
logColMeans <- function(x, eps = NULL) {
  if (!is.null(eps)) x <- x + eps
  return(mean(x))
}
apply(X, 2, logColMeans)
```

```
## [1] 21.2  9.4 19.0 16.2 11.2 16.0
```

```
apply(X, 2, logColMeans, eps = 0.1)
```

```
## [1] 21.3  9.5 19.1 16.3 11.3 16.1
```

lapply/sapply functions

- `lapply()` is used to **repeatedly apply a function to elements of a sequential object** such as a vector, list, or data-frame (applies to columns).
- The **output is a list** with the same number of elements as the input object.
- `sapply` is the same as `lapply` but **returns a "simplified" output**.
- user-defined functions can be used with `sapply/lapply`

```r
# lapply returns a list
lapply(1:3, function(x) x^2)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 4
##
## [[3]]
## [1] 9
```

```r
# which you can 'simplify' with unlist()
unlist(lapply(1:3, function(x) x^2))
```

```
## [1] 1 4 9
```

```r
# Or you could use sapply() instead
sapply(1:3, function(x) x^2)
```

```
## [1] 1 4 9
```

mapply functions

- `mapply` stands for 'multivariate' apply. It **applies a function to a multiple list or multiple vectors as arguments**.
- The goal is to vectorize arguments to a function which usually does not accept vectors as arguments.

```
# function word() returns a string of character C repeated k times.
word <- function(C,k) paste(rep.int(C,k), collapse='')
mapply(word, LETTERS[1:6], 6:1, SIMPLIFY = FALSE)
```

```
## $A
## [1] "AAAAAA"
##
## $B
## [1] "BBBBB"
##
## $C
## [1] "CCCC"
##
## $D
## [1] "DDD"
##
## $E
## [1] "EE"
##
## $F
## [1] "F"
```

Exercise 2 and 3

- Go back to “Lec2_Exercises.Rmd” in RStudio.
- Do Exercise 2 and 3.