Statistical Computing with R
Masters in Data Science 503 (S4&5)
Second Batch, SMS, TU, 2023

Shital Bhandary

Associate Professor

Statistics/Bio-statistics, Demography and Medical Education

Patan Academy of Health Sciences, Lalitpur, Nepal

Faculty, Data Analysis and Decision Modeling, MBA, Pokhara University, Nepal

Faculty, FAIMER Fellowship in Health Professions Education, India/USA

Review Preview

- Basics of R
- Chapter from “R for Everyone” book
- **We discussed it in the last class**

- Basics of coding in R
- Chapter from “Hands-on Programming with R” book
- **We will discuss this in today’s class**

Assigning new variable with pipe operator:

- **# Load in the Iris data from internet:**

```
iris <- read.csv(url("http://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"), header = FALSE)
head(iris)
```

 # Add column names for V1, V2, V3, V4 and V5 columns to the Iris data

```
names(iris) <- c("Sepal.Length", "Sepal.Width", "Petal.Length", "Petal.Width", "Species")
```

```
# Compute the square root of `iris$Sepal.Length` and assign it to the new variable
        iris$Sepal.Length.SQRT <- iris$Sepal.Length %>% sqrt()
```

Saving the data frame (e.g. from internet)

- You/we can save the data from internet as CSV file in local computer
- write.csv(DataFrame Name, "Path to export the DataFrame\\File Name.csv", row.names=FALSE)
- write.csv(iris, “iris.csv”) **#Will save CSV file in working directory**
- To know your working directory: getwd()

Using compound assignment with pipes:

- # Compute the square root of `iris$Sepal.Length` and assign it to the **same** variable

```
iris$Sepal.Length %<>% sqrt
```

Return `Sepal.Length` iris$Sepal.Length

 # Be careful while using this as the original data will be lost!

The “tee” pipe operator “%T%”:

set.seed(123) # Why to use this?

```
rnorm(200) %>%
matrix(ncol = 2) %T>%
plot %>%
colSums
```

- Normally, code ends after plot command but the “tee” pipe operator allows it to continue for the next argument

The exposing pipe operator “%$%”:

```
iris %>%
        subset(Sepal.Length > mean(Sepal.Length)) %$%
        cor(Sepal.Length, Sepal.Width)
```

The %$% operation comes handy for functions where “data” argument is not required/used like built-in “cor” function of R!

What will you get with this code:

```
        cor(iris$Sepal.Length, iris$Sepal.Width)
```

When NOT to use pipes?

- In chapter 18 of the web version of the text book “R for Data Science”, the authors have given four suggestions:

  - Your pipes are longer than (say) ten steps
  - You have multiple inputs or outputs
  - You are starting to think about a directed graph with a complex dependency structure
  - You're doing internal package development

More here: https://stackoverflow.com/questions/38880352/should-i-avoid-programming-packages-with-pipe-operators

Functions in R: Built-in functions

- round()
  - round(3.1415)
  - 3

round()
round(3.1415, digits = 2)
3.14

- factorial()
  - factorial(3)
  - 6
  - **3! = 3 x 2 x 1**

factorial()
factorial(2*3)
720
6! = 6 x 5 x 4 x 3 x 2 x 1

- mean()
  - mean(1:6)
  - **=(1+2+3+4+5+6)/6 = 3.5**

mean()
mean(c(1:30)
15.5

Functions in R: random sampling with or without replacement

die <- 1:6

- sample(x = die, size = 1)
- sample (x = die, size = 1)

- sample(x = die, size = 2)
- sample(x = die, size = 2)

- sample(x = die, size = 2, replace = TRUE)
- sample(x = die, size = 2, replace = TRUE)
- sample(x = die, size = 2, replace = TRUE)

This will be very handy when we need to divide our data into Training and Testing sets using Random sampling method!

User-defined function in R:

- my_function <- function() {}
- Where,
- my_function = name of the function e.g. roll (roll the die)

- function() = telling R that it is a user-defined function

- { = We need to start our code after this braces
- } = We need to close our codes before this braces

User-defined function 1: roll()

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

First roll: roll()

Second roll: roll()

Third roll: roll()

User-defined function 2: roll2()

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

First roll: roll2()

Second roll: roll2()

Third roll: roll2()

User-defined function 3: roll3(dice = ?:?)

```
roll3 <- function(dice) {
        dice <- sample(dice, size = 2, replace = TRUE)
        sum(dice)
}
```

First roll: roll3(dice = 1:6)

Second roll: roll3(dice = 1:12)

Third roll: roll3(dice = 1:24) **# Is this possible in two dice?**

Function in R: Continued …

best_practice <- c(“Let”, “the”, “computer”, “do”, “the”, “work”)

```
print_words <- function(sentence) {
        print(sentence[1])
        print(sentence[2])
        print(sentence[3])
        print(sentence[4])
        print(sentence[5])
        print(sentence[6])
}
```

What is wrong with this approach?

- print_words(best_practice) # [1] ”Let” [1] “the” [1] “computer” [1] “do” [1] “the” [1] “work”
- print_words(best_practice[-6]) # [1] ”Let” [1] “the” [1] “computer” [1] “do” [1] “the” [1] “NA”
- best_practice[-6] # [1] ”Let” “the” “computer” “do” “the”

Can we improve it in R?
We can use functions with “for” loop in R!

```
print_words <- function(sentence) {
    for (word in sentence) {
    print(word)
    }
}
print_words(best_practice)
```

[1] “Let” [1] “the” [1] “computer” [1] “do” [1] “the” [1] “work”

```
print_words(best_practice[-6])
```

[1] “Let” [1] “the” [1] “computure” [1] “do” [1] “the”

“for” loop

R:

```
for (variable in collection) {
    do things with variable
}
```

“for” and “while” loops can be very slow in R!

What to do?

https://swcarpentry.github.io/r-novice-inflammation/03-loops-R/

Loops in R will not be slow if we:

- Don’t use a loop when a vectorized alternative exists
- Don’t grow objects (via c, cbind, etc) during the loop – R has to create new object and copy across the information just to add new element or row/column
- Allocate an object to hold the result and fill it during the loop
- **Can we do even better in R? Alternative to “loop” in R??**

While working with data.frame in R:

- It is better to use family of “apply” functions from base R:
  - apply
  - lapply
  - sapply
  - vapply

More here: https://www.datacamp.com/tutorial/r-tutorial-apply-family

We will discuss this in detail while doing breakdown analysis session in R later!

- functions instead of “for loop” to run the script much faster in R!

- Same applies to the “while loop” too!

Condition: if and else

```
if (condition) {
        #code executed when condition is TRUE
} else {
        #code executed when condition is FALSE
}
```

Can you think of an example?

What will be the output?

#Checking values of y with x:

```
if (y < 20) {
   x <- "Too low"
} else {
  x <- "Too high"
}
```

#Can you get anything from this?

#Will this work?

```
check.y <- function(y) {
if (y < 20) {
print("Too Low") } else {
print("Two high")
}}
```

```
check.y(10)
```

```
check.y(30)
```

Creating binary variables with “ifelse”

#Will this work?

y <- 1:40

ifelse(y<20, “Too low”, “Too high”)

It’s a logical as:

ifelse(y<20, TRUE, FALSE)

Good to make binary variables with text categories!

#Will this work?

y <- 1:40

ifelse(y<20, 1, 0)

Good to make binary variables with numerical categories!

This one is preferred!

Multiple conditions:

In a function:

```
if (this) {
        # do that
} else if (that) {
        # do something else
} else if (that) {
        # do something else
} else
# remaining
}
```

```
check.x <- function(x){
if (x<20){
print("Less than 20")} else{
if (x<40) {
print("20-39")} else{
if (x<100) {
print("41-99")}
}}}
```

- check.x(20)
- check.x(30)
- check.x(50)

Good to make categorical variables!

Multiple Conditions: combining “ifelse”

Will this work too?

x <- 1:99

x1 <- ifelse(x<20, 1,0) #Binary numbers

x1 <- ifelse(x<20, “<20”, “20+”) #Binary text

x2 ? For x between 20 and less than 40

x3 ? For x between 40 and less than 100

Now combine them in a single column with <20 =1, 20-39 = 2 and 40-99 = 3 for x i.e. create categorical variable of x!

x <- ifelse(x<20,1,ifelse(x<40,2,3)) **works?**

```
#This code shows how Petal. Length categories was created from Petal. Length variable of iris data frame
iris <- within(iris, {
Petal.cat <- NA
Petal.cat[Petal.Length <1.6] <- "Small"
Petal.cat[Petal.Length >=1.6 & Petal.Length<5.1] <- "Medium"
Petal.cat[Petal.Length >=5.1] <- "Large"
})
#The 1.6=Q1 and 5.1=Q3, they were obtained from the “summary” of the Petal.Lenght variable
```

There are multiple ways for doing this but use the one that you feel comfortable with!

Multiple Conditions: If, else if, else if, else if

```
#Make this function work!
if (temp <= 0) {
"freezing"}
else if (temp <= 10) {
"cold"}
else if (temp <= 20) {
"cool"}
else if (temp <= 30) {
"warm"}
else {
"hot"}
```

```
# What is missing?

# How to address it?
```

Naming convention is R?

- It is not properly defined!

- This article "The State of Naming Conventions in R" talks about:

  - alllowercase e.g. adjustcolor
  - period.separated e.g. plot.new
  - **underscore_separated e.g. numeric_version**
  - lowerCamelCase e.g. addTaskCallback
  - UpperCamelCase e.g. SignatureMethod

Also check this out:
https://www.r-bloggers.com/2014/07/consistent-naming-conventions-in-r/
https://bookdown.org/content/d1e53ac9-28ce-472f-bc2c-f499f18264a3/names.html

Reproducible outputs: Markdown

- **Markdown** is described as: "*Text-to-HTML conversion tool/syntax*".

- Markdown is two things: (1) a plain text formatting syntax; and (2) a software tool, written in Perl, that converts the plain text formatting to HTML.

Reproducible outputs: YAML

- On the other hand, **YAML** is detailed as "*A straightforward machine parsable data serialization format designed for human readability and interaction*".

- **YAML** is a human-readable data-serialization language. It is commonly used for configuration files, but could be used in many applications where data is being stored or transmitted.

- YAML = Yet Another Markup Language in 2001 (YAML Ain’t Markup Language from 2002 onwards = NOT FOR DOCUMENT MARKUP)

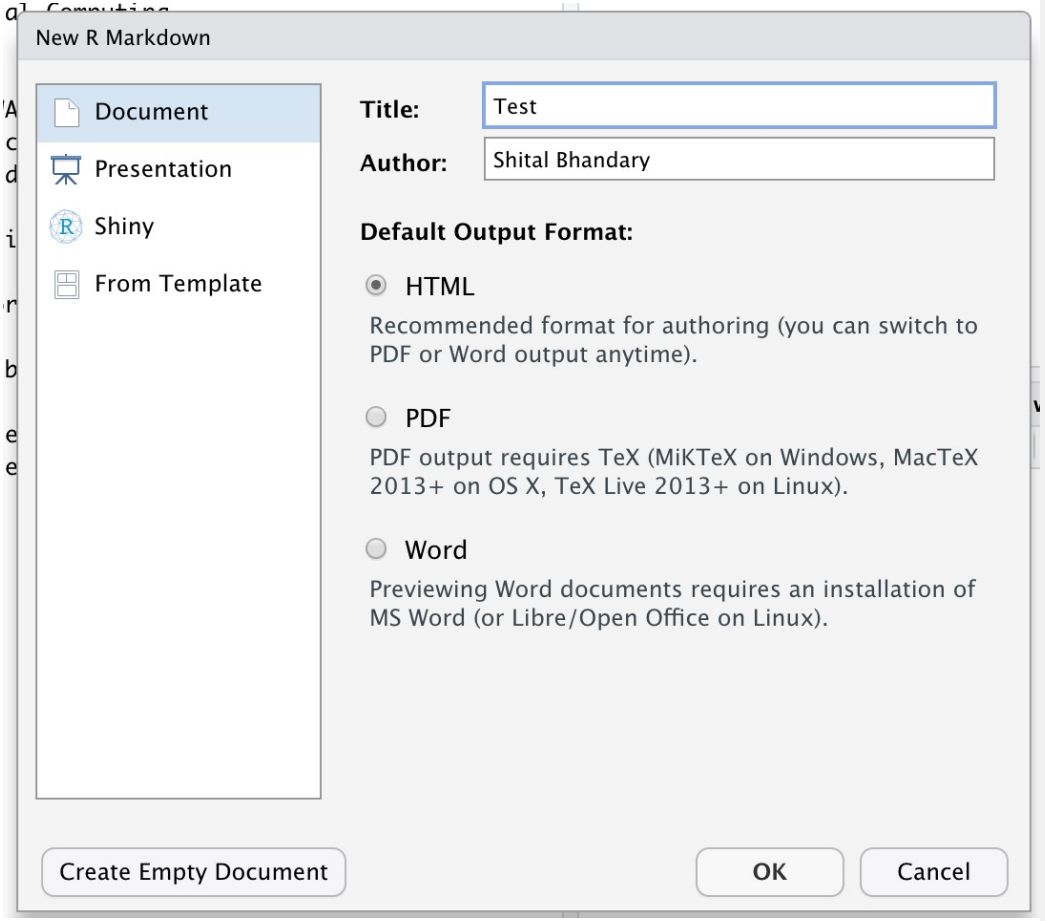
R Markdown and knitr in R Studio: Dynamic Report Generation

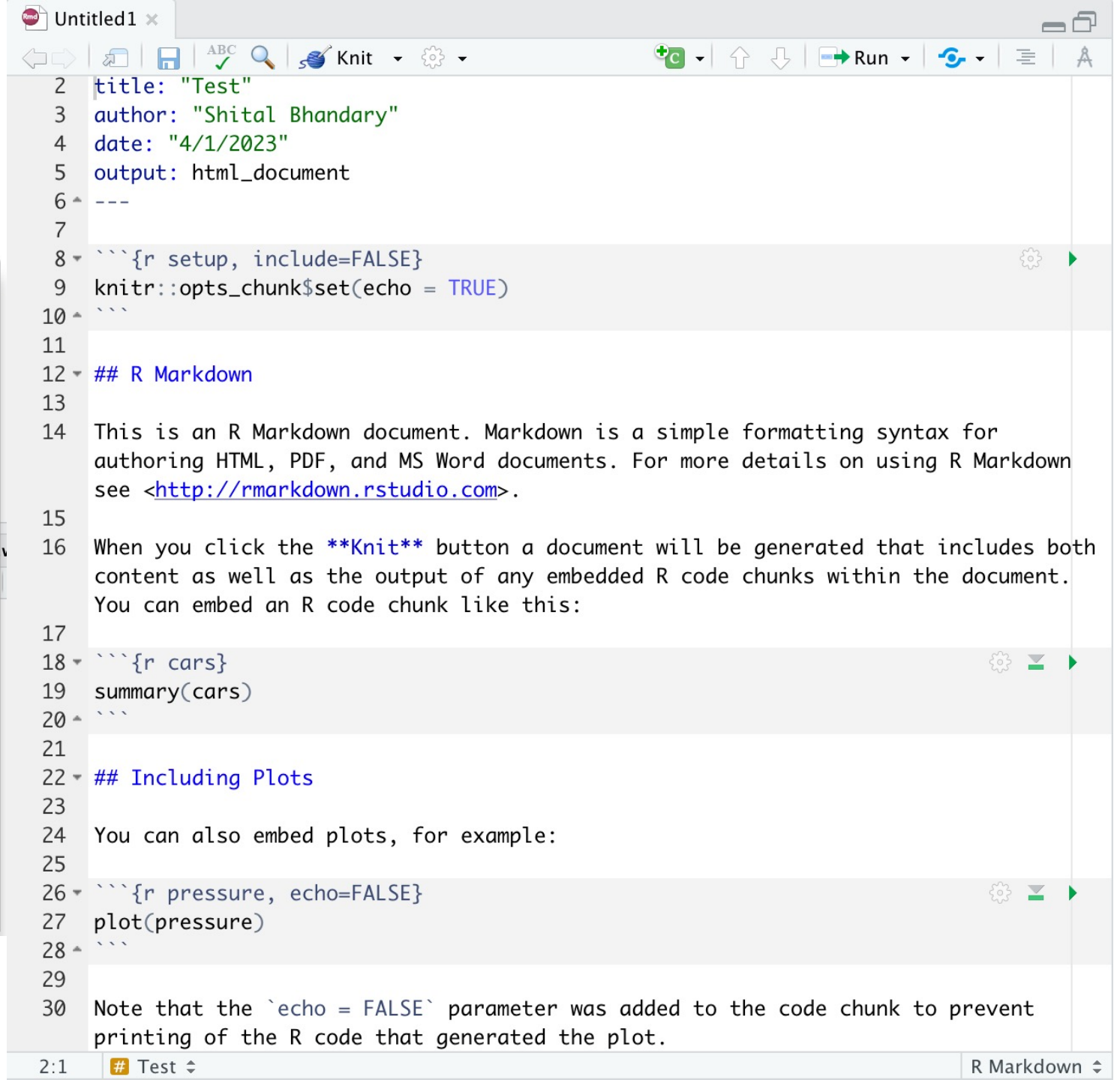
- You cannot execute any R code in a plain Markdown document
- You can embed the R code in plain Markdown using syntax for fenced code block ```r i.e. without curly braces but it will not be executed!
- You can embed R code chunks (```{r}) in an R Markdown document
- More here:
  - https://cran.r-project.org/web/packages/rmarkdown/index.html
  - https://sachsmc.github.io/knit-git-markr-guide/knitr/knit.html
  - https://github.com/rstudio/bookdown

R Studio: File → New File → R Markdown

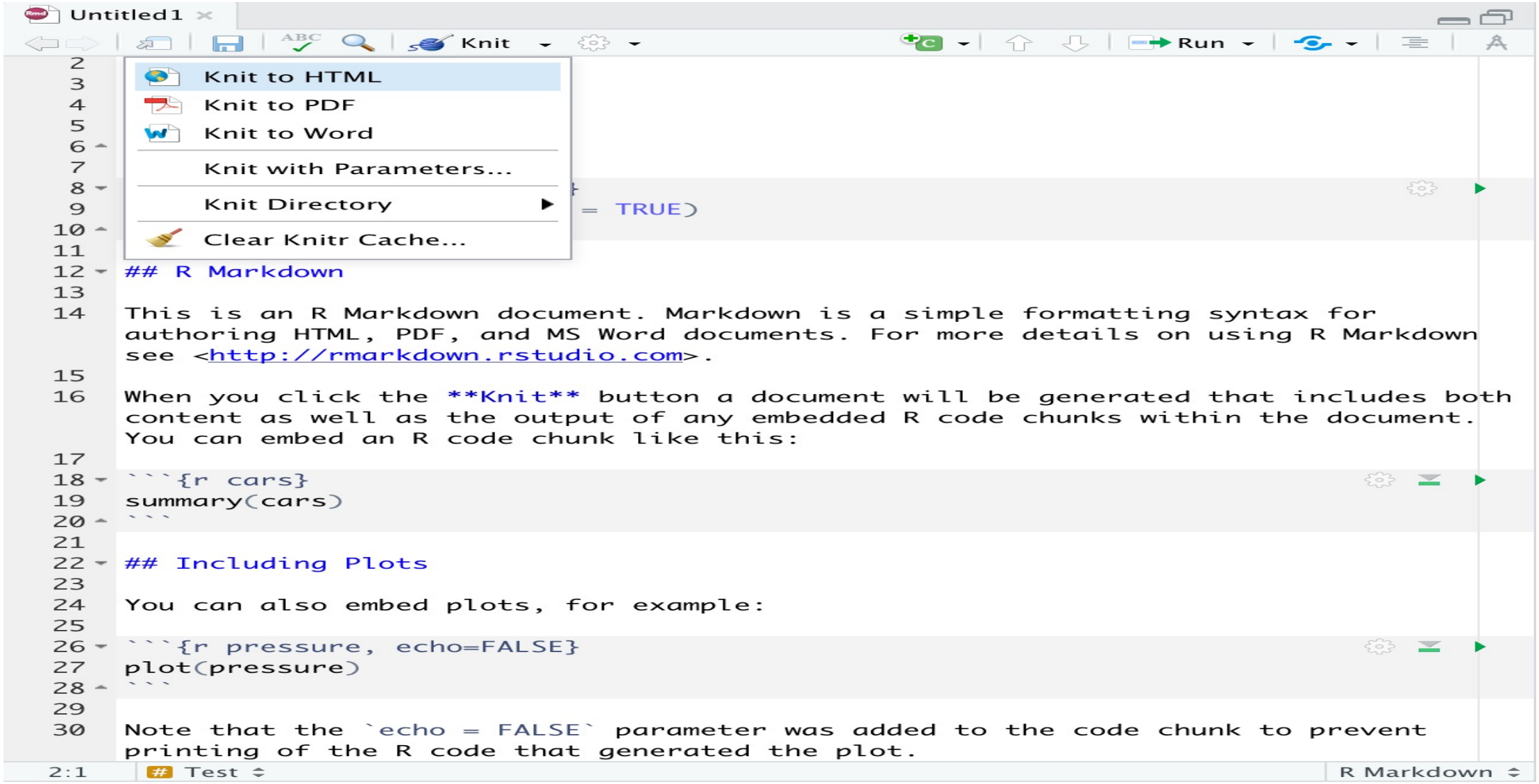
- New R Markdown → Document → Title → Test → OK
- What do you get?
- Click the “knit” button → “Test” → Save
- It will save “Test.html” in your working directory

You will get this:

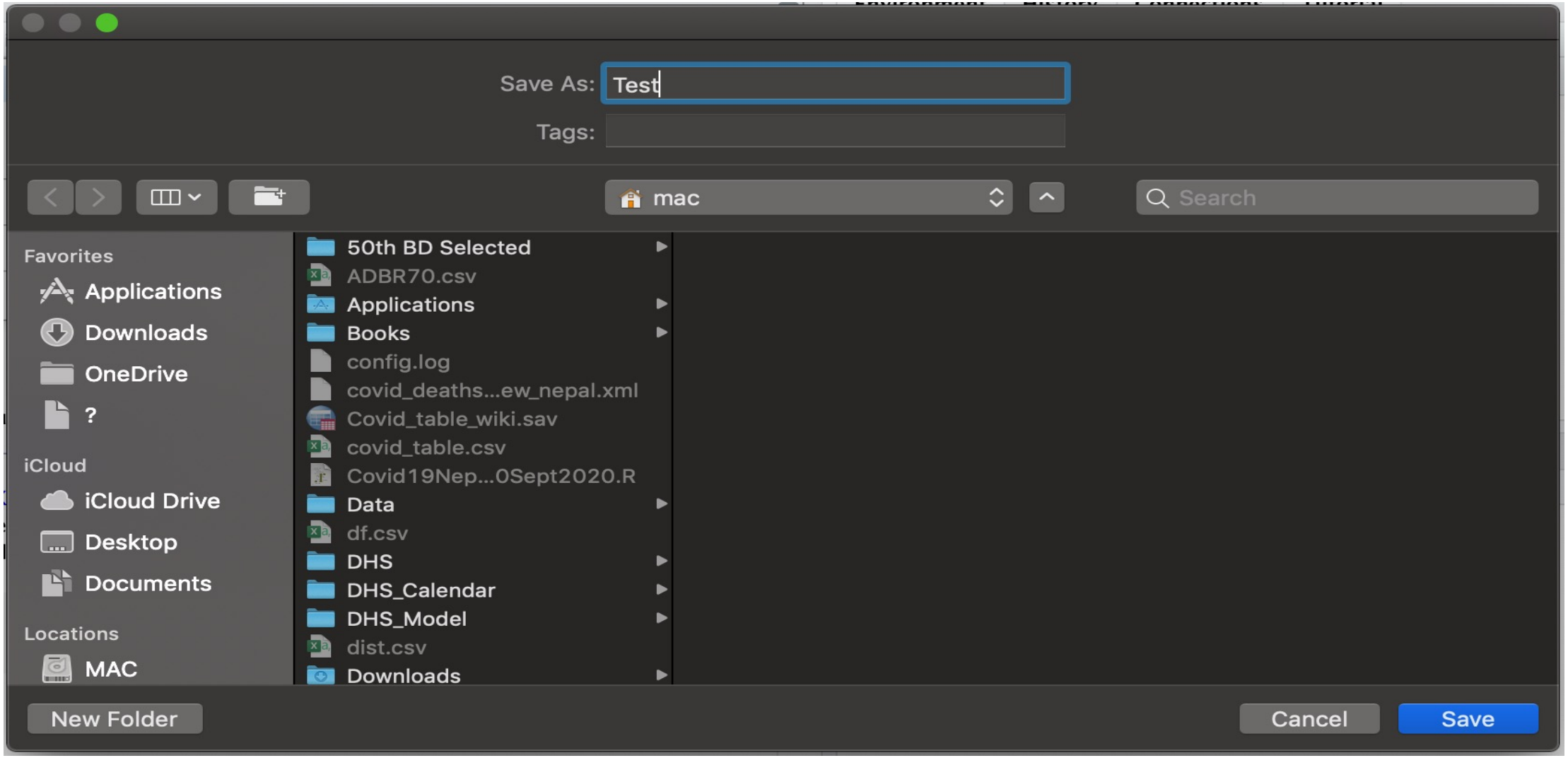




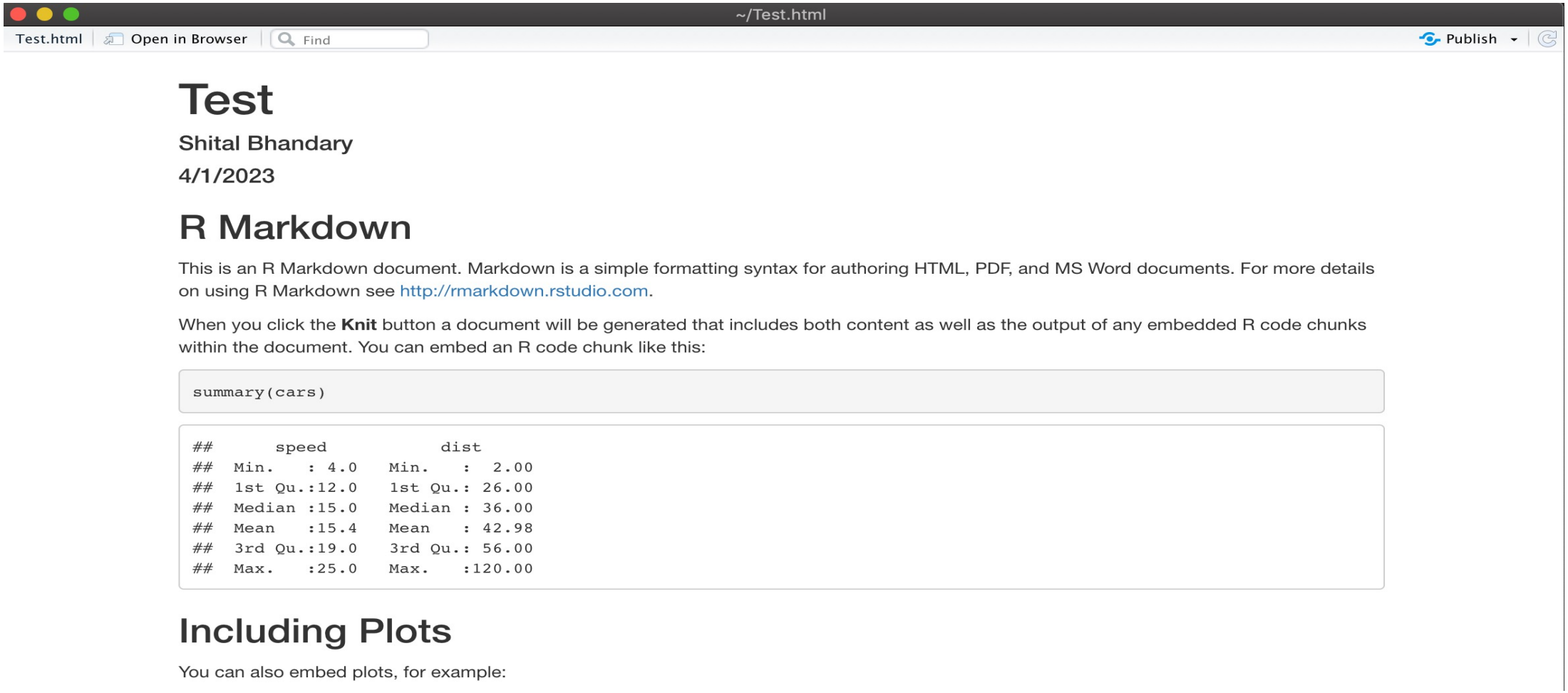
Then “knit” it to get ‘html’ or ‘pdf’ or ‘word’



You will be asked to save it:



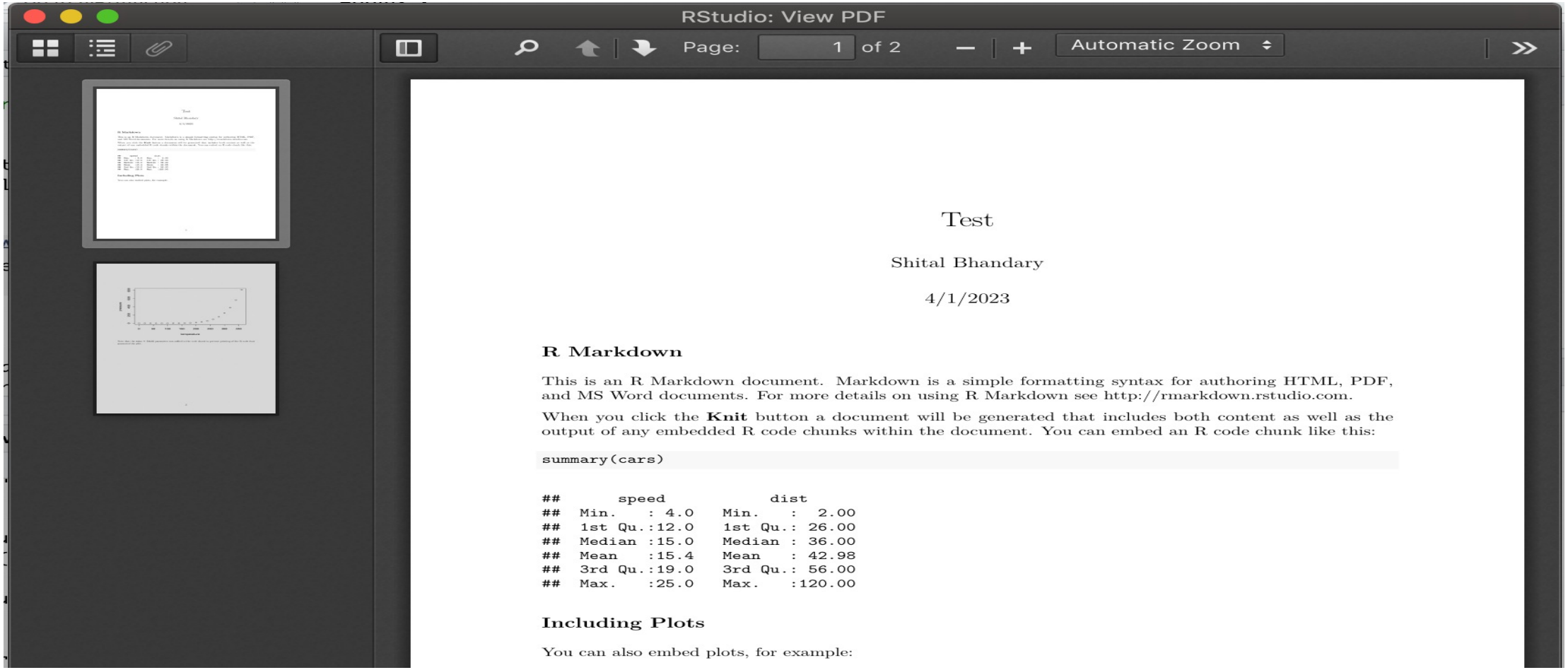
To get the HTML file with R Markdown:



R Studio: File → New File → R Markdown

- New R Markdown → Document → Title → Test → OK
- What do you get?
- Click the “knit” button → ”Knit to PDF” → “Test” → Save
- It will save “Test.pdf” in your working directory if you have the required LaTex to PDF package like TinyTex (you can install it with this command in R: tinytex::install_tinytex() if required!)

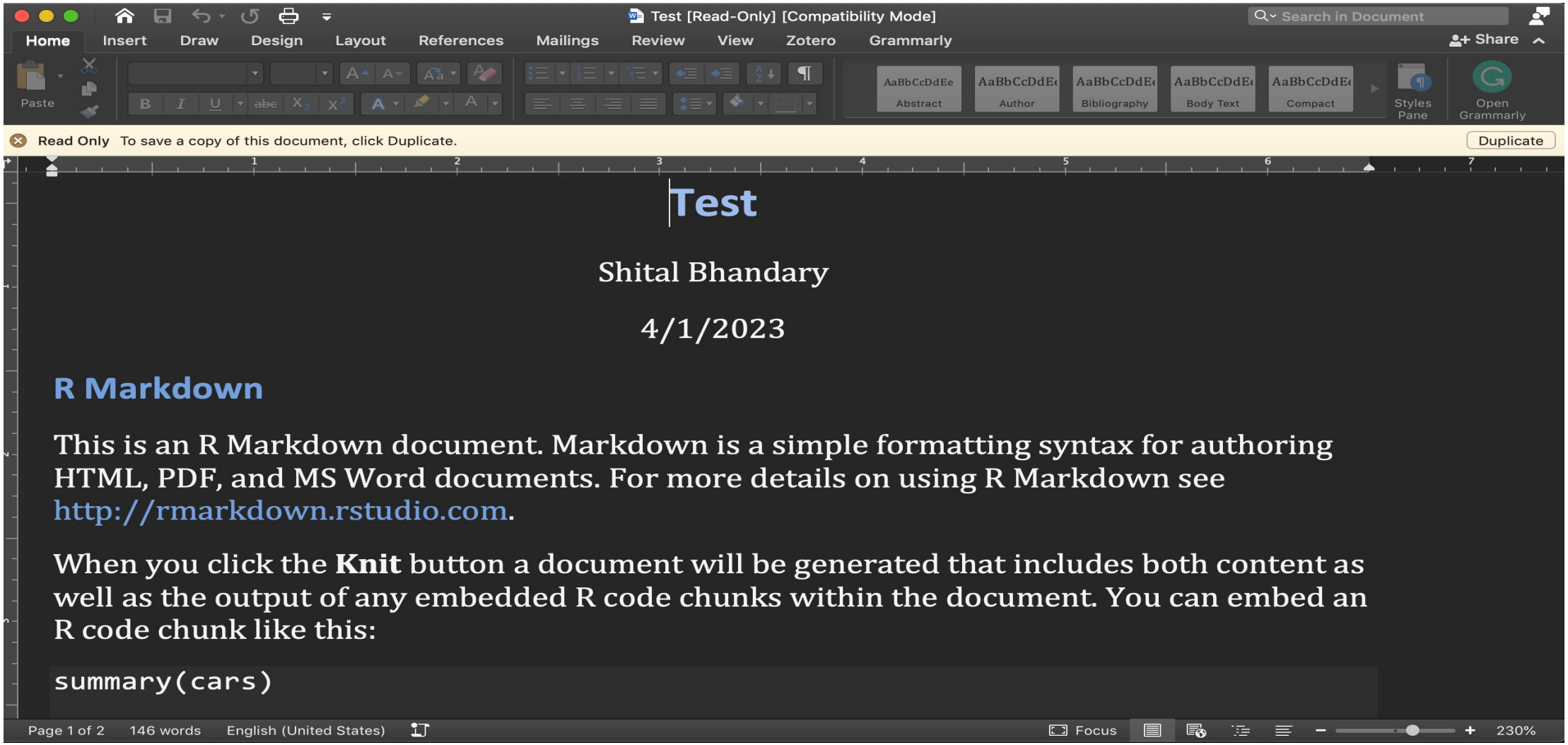
You will get this then:



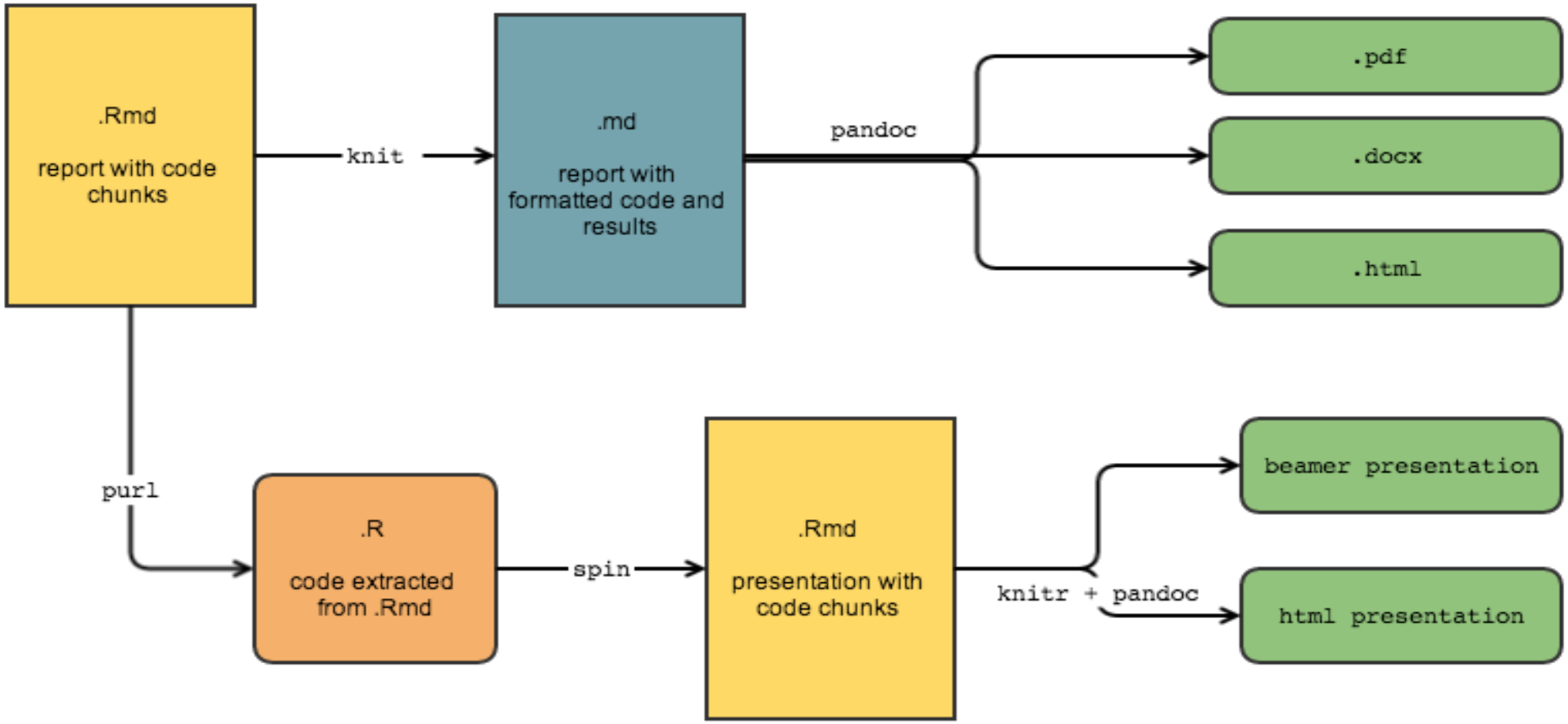
R Studio: File → New File → R Markdown

- New R Markdown → Document → Title → Test → OK
- What do you get?
- Click the “knit” button → ”Knit to Word” → “Test” → Save
- It will save “Test.docx” in your working directory if you have the MS Word software in your computer (you need to provide access to write using MS Word when asked!)

You will get this if all goes well:



R Markdown Workflow in R Studio:



https://sachsmc.github.io/knit-git-markr-guide/knitr/knit.html

Profiling and Optimizing Codes in R

https://bookdown.org/rdpeng/rprogdatascience/profiling-r-code.html

- R comes with a profiler to help you optimize your code and improve its performance.
- In general, it's usually a bad idea to focus on optimizing your code at the very beginning of development. Rather, in the beginning it's better to focus on translating your ideas into code and writing code that's coherent and readable.
- The problem is that heavily optimized code tends to be obscure and difficult to read, making it harder to debug and revise. Better to get all the bugs out first, then focus on optimizing.

Profiling

- Profiling is a systematic way to examine how much time is spent in different parts of a program.

- The reality is that *profiling is better than guessing*.

- The system.time() function computes the time (in seconds) needed to execute an expression and if there's an error, gives the time until the error occurred.

R profiler

- Rprof() #Turn on the R profiler
  - In conjunction with Rprof(), we will use the summaryRprof() function which summarizes the output from Rprof() (otherwise it's not really readable)
  - You should NOT use system.time() and Rprof() together!
  - Once you call the Rprof() function, everything that you do from then on will be measured by the profiler.

- Rprof(NULL) #Turn off the profiler

- **Read: Chapter 19- Profiling R code (R Programming for Data Science)**

Profiling R code with R Studio IDE

https://support.posit.co/hc/en-us/articles/218221837-Profiling-R-code-with-the-RStudio-IDE

- As R users, many, perhaps most, of us have had times where we've wanted our code to run faster. However, it's not always clear how to accomplish this. A common approach is to rely on our intuitions, and on wisdom from the broader R community about speeding up R code.
- **e.g., that apply functions are inherently faster than for loops**

- One drawback to this is it can lead to a focus on optimizing things that actually take a small proportion of the overall running time.

Example: With “loop” in R for row mean

- N <- 10000
- x1 <- runif(N)
- x2 <- runif(N)
- d <- as.data.frame(cbind(x1, x2))

- system.time(for (loop in c(1:length(d[, 1]))) { d$mean2[loop] <- mean(c(d[loop, 1], d[loop, 2])) })

- # user system elapsed
- # 13.912 0.204 14.150

Example: With built-in “apply” function

- N <- 10000
- x1 <- runif(N)
- x2 <- runif(N)
- d <- as.data.frame(cbind(x1, x2))

- system.time(d$mean1 <- apply(d, 1, mean))

```
# user    system   elapsed
# 0.180   0.000      0.179
```

```
#apply (x, 1 or 2, function)
# 1=Row; 2=Column
```

E.G.: With vectorized ‘rowMeans’ function

- N <- 10000
- x1 <- runif(N)
- x2 <- runif(N)
- d <- as.data.frame(cbind(x1, x2))

- system.time(d$mean3 <- rowMeans(d[, c(1, 2)]))

```
# user    system   elapsed
# 0.004     0.000     0.002
```

Comparison:

- Bad way

```
x <- c() for (i in 1:1e+05)
  { x <- c(x, i) }
```

- #15 seconds

- Good way (0.001 seconds)

```
y <- seq(1, 1e+05)
```

- Better way (<0 seconds)

```
z <- 1:1e+05
```

Comparison: Which one is better?

- x <- runif(1e+06)

```
for (i in 1:length(x)) {
if (x[i] < 0.05) {
x[i] <- NA
}
}
```

What is this doing?

- x <- runif(1e+06)
  x[which(x < 0.05)] <- NA

- What is this doing?

Profiling R code with R Studio IDE

https://support.posit.co/hc/en-us/articles/218221837-Profiling-R-code-with-the-RStudio-IDE

- The profiler is a tool for helping you to understand how R spends its time. It provides a interactive graphical interface for visualizing data from Rprof, R's built-in tool for collecting profiling data and, profvis, a tool for visualizing profiles from R.

- Example: library(profvis)

```
profvis({ data(diamonds, package = "ggplot2")
plot(price ~ carat, data = diamonds)
m <- lm(price ~ carat, data = diamonds) abline(m, col = "red") })
```

More here: https://support.posit.co/hc/en-us/articles/218221837-Profiling-R-code-with-the-RStudio-IDE

Questions/queries?

Thank you!

@shitalbhandary