Day 5 Lecture

Cecilia Y. Sui, and all other TAs

12/29/25

1 Day 5 Outline:

1. Random Sampling
2. Custom Functions
3. Control Statements (if, if-else, for, while) and Random Sampling
4. Bootstrap and Simulation Studies

1.1 Sampling using `sample()` function

Base R comes with `sample()` function, which takes two arguments:

1. A vector named `x`
2. A number named `size`

`sample()` will return the elements sampled from the vector. The return value is **not** a list, which means if you need to do list operations, you need to first convert it to a list.

```
sample(x = 1:4, size = 2)
```

```
## [1] 3 4
```

```
die <- 1:6
sample(x = die, size = 3) # can input object
```

```
## [1] 5 1 3
```

```
# Be careful with the scope of variables here.
# x is a local variable, NOT a global variable.
# This means that x can only be accessed within the function call sample()
```

Many R functions take multiple arguments that help them do their job. You can give a function as many arguments as you like as long as you separate each argument with a comma.

Every argument in every R function has an argument name. For example, x and size are both argument names for the function sample().

You can specify which data should be assigned to which argument by setting a name equal to data, as in the preceding code. This becomes important as you begin to pass multiple arguments to the same function. Argument names help you avoid passing the wrong data to the wrong argument.

However, using names is **optional**. You might notice that R users do not often use the name of the first argument in a function. So you might see the previous code written as:

```
?sample
sample(x = die, size = 3) # best practice
```

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

```r
# or equivalently
sample(die, 3)
```

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

1.2 Setting a random seed

Just like other programming languages, R also has internal RNG (random number generator). Thus, the random processes in R are not true random, and we can fix the state to facilitate reproducibility. This can be done by setting a random seed with `set.seed()` function.

```r
set.seed(63130)
print(rnorm(2)) # draw random numbers from standard normal
```

```
## [1]  1.649446 -1.238885
```

```r
set.seed(63130)
print(rnorm(2)) # draw random numbers from standard normal
```

```
## [1]  1.649446 -1.238885
```

As expected the results from two `rnorm()` calls agree. Remember to set seed when you work with random processes!

1.3 Sampling without Replacement

When we use the **sample()** function to do random sampling, the default is sampling without replacement, where **replace = FALSE**.

```r
?sample()
sample(x = die, size = 3)
# sample(x, size, replace = FALSE, prob = NULL)
```

1.4 Sampling with Replacement

To do sampling with replacement, you can add the argument **replace = TRUE**:

```r
sample(x = die, size = 6, replace = TRUE)
```

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

```r
sample(x = c(0, 1), size = 10, replace = TRUE) # 10 Bernoulli trials
```

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

The argument **replace = TRUE** causes sample() to sample **with replacement**. As a result, sample() may select the same value on the second draw. Each value has a chance of being selected each time. It is as if every draw were the first draw.

"Sampling with replacement is an easy way to create independent random samples." Each value in your sample will be a sample of size one that is independent of the other values. For example, this would be how we simulate a pair of fair dice:

```r
die
```

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

```r
sample(x = die, size = 2, replace = TRUE)
```

```
## [1] 5 3
```

Congratulate yourself! You've just run your first simulation in R!

You now have a method for simulating the result of rolling a pair of dice. If you want to add up the dice, you can feed your result straight into the sum() function:

```
dice <- sample(die, size = 2, replace = T)
dice
```

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

```
# sum(dice)
```

What would happen if you call dice multiple times? Would R generate a new pair of dice values each time? Let's give it a try:

```
dice
```

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

```
dice
```

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

```
dice
```

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

Nope.

Each time you call dice, R will show you the result of that one time you called sample and saved the output to dice. R won't rerun **sample(die, 2, replace = TRUE)** to create a new roll of the dice. This is a relief in a way. Once you save a set of results to an R object, those results do not change. Programming would be quite hard if the values of your objects changed each time you called them.

However, it would be convenient to have an object that can **re-roll** the dice whenever you call it. You can make such an object by writing your own **R function** which we will cover in the following section.

1.5 In-class exercises 3.1:

1. Sample with replacement from 1 to 1000 inclusive of size 20.

```
set.seed(12345)
sample(1:100, 5)
```

```
## [1] 14 51 80 90 92
```

2. Sample without replacement from 1 to 1000 inclusive of size 20.
3. What does the following code do?

```
x <- 1:10
sample(x[x > 10])
```

```
## integer(0)
```

2 Writing Your Own Functions

To recap, you already have working R code that simulates rolling a pair of dice.

```
die <- 1:6
dice <- sample(die, size = 2, replace = TRUE)
dice
```

```
## [1] 2 5
```

```
sum(dice)
```

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

You can retype this code into the console anytime you want to re-roll your dice. However, this is an awkward way to work with the code. It would be easier to use your code if you wrapped it into its own **function**, which is exactly what we will do now. You are going to write a function named roll that you can use to roll two virtual dice. When you are finished, the function will work like this: each time you call **roll()**, R will return the sum of rolling two dice:

```
# This code block is just for illustration.
# It is not executable yet, since we have not defined roll().
roll()
# 10
roll()
# 8
roll()
# 7
```

Functions might sound fancy when first hearing about it. They are just another type of R object. Instead of containing data, like vectors, matrices, or dataframes, functions contain code. This code is stored in a special format that makes it easy to reuse the code in new situations throughout your program. You can write your own functions by recreating the following format.

2.1 The Function Constructor

Every function in R has three basic parts:

1. a function name
2. a body of code
3. a set of arguments

To make your own function, you need to replicate these parts and store them in an R object, which you can do with the function **function()**. To do this, call function() and follow it with a pair of braces, {}:

```
my_function <- function( ) {
  ### the body of the function
  return( ) # the return value(s)
}
```

function() will build a function out of whatever R code you place between the braces.

For example, you can turn your dice code into a function by running:

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

roll()
```

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

Notice that I indented each line of code between the braces here. The indentation is just for readability, which makes the code chunk easier for you to read. It has **NO** impact on how the code executes. R ignores spaces and line breaks inside these braces, and executes one complete expression at a time.

When writing the code between the braces, just hit the Enter key between each line after the first brace, {. R will wait for you to type the last brace, }, before it responds. If you are using RStudio, when starting a new line between braces, it automatically aligns with your code above.

Don't forget to save the output of function to an R object. This object will become your new function that you can call. To use it, write the object's name followed by an open and closed parenthesis. For example, if we call **roll()** three times, it should output three different outputs.

```
roll()
```

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

```
roll()
```

```
## [1] 12
```

```
roll()
```

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

You can think of the parentheses as the "trigger" that causes R to run the function. If you type in a function's name without the parentheses, R will show you the code that is stored inside the function. If you type in the name with the parentheses, R will run that code:

```
roll # view the code
```

```
## function ()
## {
##     die <- 1:6
##     dice <- sample(die, size = 2, replace = TRUE)
##     s <- sum(dice)
##     return(s)
## }
## <bytecode: 0x116e89b90>
```

```
roll() # run the function
```

```
## [1] 12
```

The code that you place inside your function is known as the **body of the function**. When you run a function in R, R will execute all of the code in the body and then return the value you put in return().

return() is not required to write R functions. If you do not have a return value specified, R will return the result of the last line of code inside the braces.

If the last line of code does not return a value, neither will your function. So if you want your function to have return values, you need to make sure that either you final line of code returns a value or you specify the return value inside your return().

One way to check this is to think about what would happen if you ran the body of code line by line in the command line. Would R display a result after the last line, or would it not?

Here's some code that would display a result:

```
dice
1 + 1
sqrt(2)
```

And here's some code that would not:

```
dice <- sample(die, size = 2, replace = TRUE)
two <- 1 + 1
```

```r
a <- sqrt(2)
# df[1, ] <- 0
```

Do you notice the pattern? These lines of code do not return a value to the command line; they save a value to an object. Generally, if the last line of code in your function definition is an assignment statement, it will not return anything.

2.2 Arguments

What if we removed one line of code from our function and changed the name die to custom_die, like this?

```r
roll2 <- function() {
  dice <- sample(custom_die, size = 2, replace = TRUE)
  sum(dice)
}
```

Now we will get an error when we run the function. The function needs the object custom_die to do its job, but there is no object named custom_die to be found in the environment:

```r
roll2()
# Error in sample(custom_die, size = 2, replace = TRUE) :
# object 'custom_die' not found
```

You can provide custom_die when you call roll2 if you make custom_die an argument of the function. To do this, put the name custom_die in the parentheses that follow function when you define roll2:

Say you want to make a different sided die

```r
roll2 <- function(custom_die) {
  dice <- sample(custom_die, size = 2, replace = TRUE)
  return(sum(dice))
}
```

Now roll2 will work as long as you provide custom_die when you call the function roll2(). You can take advantage of this to roll different types of dice each time you call roll2.

Remember, we're rolling pairs of dice that have different number of sides:

```r
roll2(custom_die = 1:4) # 4-sided
```

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

```r
roll2(custom_die = 1:6) # 6-sided
```

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

```r
roll2(custom_die = 1:20) # 20-sided
```

```
## [1] 15
```

```r
roll2(c(1, 2, 3, 4, 4, 6)) # without the argument name, it works the same
```

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

Notice that roll2 will still give an error if you do not specify a value for the custom_die argument when you call roll2.

```r
roll2()
# Error in sample(bones, size = 2, replace = TRUE) :
#  argument "custom_die" is missing, with no default
```

As mentioned in the Error Message, we can fix the error by giving the custom_die argument a default value. To do this, set custom_die equal to a value when you define

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

Now you can provide a new value for custom_die if you like. If you do not provide any value for the custom_die argument, roll2() will use the default value that we defined above.

```
roll2() # using custom_die = 1:6
```

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

```
roll2(c(1, 2, 2, 4, 4, 5)) # mario party custom_die
```

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

Again, **return()** is not strictly needed in R. R will return the last value you calculate in the function by default, but **return()** makes it clearer.

```
roll3 <- function(custom_die = 1:6) {
  dice <- sample(custom_die, size = 2, replace = TRUE)
  sum(dice) # It is equivalent to just writing sum(dice)
}

roll3()
```

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

In R, the **return()** function can return only **one single object**. If we want to return multiple values in R, we can use a list (or other objects) that stores multiple values and return it as a single object.

```
roll4 <- function(custom_die = 1:6) {
  dice <- sample(custom_die, size = 2, replace = TRUE)
  l <- list(sum(dice), dice)
  return(l)
}
roll4()
```

```
## [[1]]
## [1] 7
##
## [[2]]
## [1] 3 4
```

```
# The first item returned is sum(dice).
# The second item returned is the list dice that stores 2 values.
```

You can give your functions **as many arguments as you like**. Just list their names, separated by commas, in the parentheses that follow function(). When the function is run, R will replace each argument name in the function body with the value that you provide for the argument.

- If you do not provide a value, R will replace the argument name with the argument's default value if you defined one.
- If you forget to provide a value for an argument that does not have any default value defined, R will stop execution and print an Error Message asking for it!

To summarize, **function()** helps you construct your own R functions.

1. Create a body of code for your function to run by writing code between the braces that follow **function()**.
2. Create arguments for your function to use by supplying their names in the parentheses that follow **function()**.
3. Give your function a name by saving its output to an R object.
4. Specify the return values if you want your function to return anything.

Once you have created your function, R will treat it like every other function in R, such as sum(), mean(), round(), etc. Think about how useful this is!

In the previous exericise, I asked you to sample the column TFR five times and calculate their averages. What if I asked you to do it 1000 times? Without functions, repeating the same chunk of code could be terrifying. As you learn to program in R, you will be able to create new, customized, reproducible tools for yourself whenever you like.

2.3 In-class exercises 3.2:

1. Write a function called func_square() that takes one argument of type integer and returns the square of the number.
2. Write a function that prints out just your **first name**, but returns your **first and last name** together. Think about what should be your arguments here. Use **print()** function to do the printing.
3. Write a function called **my_sampling()** that samples without replacement from 1 to 100 inclusive of a given size. The function should take one argument that specifies the size of the returning vector.
4. Write a function that given a dataframe will return its column names.
5. Write a function that given a vector and an integer will return whether the integer is inside the vector.
6. Write a function that randomly samples from the TFR column in the worldTFR dataset. The users should be able to modify the size of the sample, and whether to sample with or without replacement.
7. Use the function you created in Q6 to do random sampling with replacement seven times. Calculate the mean for each sample. Then compute the average of the means. Compare it to the true mean.

3 Control Statements

- imagine "traffic signals for a computer" In order to control the flow of execution of expressions in R, we make use of the **control structures**. These control structures are sometimes called loops in R. There are eight types of control structures in R (if/if-else, switch, for, while, nested loops, repeat and break, next, return), but we will only focus on **three here (if/if-else, for, while)**. Since all **while** loops can be re-written as **for** loops, we will mainly work on **for** loops and briefly introduce **while** loops.

Whenever possible, it is more efficient to use **built-in functions** in R rather than control structures. This facilitates the flow of execution to be controlled inside a function. Control structures define the flow of the program. The decision is then made after the variable is assessed.

3.1 *if* Condition in R

The **if** and **if-else** conditions in R enforce **conditional execution** of the code. They are an important part of R's decision-making capability. It allows us to make a decision based on the result of a condition. The **if** statement contains a condition that evaluates to a logical output (TRUE = 1 / FALSE = 0).

- It runs the enclosed code block if the condition evaluates to TRUE.
- It skips the code block if the condition evaluates to FALSE.

```
# Syntax:
# This code block is not executable. Just for illustration.
if (test_expression) {
```

```
  statement
}
```

R is **NOT an indentation-based programming language**, so indentation or space characters here would not make a difference as we illustrated above in the functions section. You can put your braces at the end of the last line of your code inside the loop, or start a new line just for the closing braces.

Let's do some examples!

```
my_cond1 <- TRUE
# my_cond <- T
if (my_cond1) {
  print("My condition is TRUE")
}
```

```
## [1] "My condition is TRUE"
```

```
my_cond2 <- FALSE
if (my_cond2) {
  print("My condition is TRUE")
}


values <- 1:10
if (length(values) == 10) {
  print(values)
}
```

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

```
a <- 5
b <- 6
if (a < b) {
  print("a is smaller than b")
}
```

```
## [1] "a is smaller than b"
```

We will use a built-in dataset to illustrate how to recode variables using these control statements.

```
df <- longley
# You can use the help function to get more details on the dataset
# ?longley
# head(df)
```

For example, we can check to see if the year of the first row is 1947:

```
if (df$Year[1] == 1947) {
  print("Great, it's 1947!")
}
```

```
## [1] "Great, it's 1947!"
```

We can also check if there is any year where the number of unemployed exceeds 300.

```
if (any(df$Unemployed > 300)) {
  print("Yes")
}
```

```
## [1] "Yes"
```

3.2 *if-else* Condition in R

We use the **else** statement with the **if** statement to enact a choice between two alternatives. If the condition within the **if** statement evaluates to FALSE, it runs the code within the **else** statement.

For example:

```
a <- 88
b <- 100

if (a > b) {
  print("a is greater than b")
} else {
  print("b is greater than a")
}
```

```
## [1] "b is greater than a"
```

We can use the **else if** statement to select between multiple options. For example:

```
a <- 100
b <- 88
if (a < b) {
  print("a is smaller than b")
} else if (a == b) {
  print("a is equal to b")
} else { # a > b
  print("a is greater than b")
}
```

```
## [1] "a is greater than b"
```

There is also an $ifelse()$ function in R. It acts like the **if-else** structure. It is not very commonly used, but serves as a shortcut sometimes.

The following code is the syntax of the $ifelse()$ function in R:

```
# This code block is not executable. Just for illustration.
ifelse(condition, exp_if_true, exp_if_false)
```

- condition is the condition that evaluates to either TRUE or FALSE
- exp_if_true is the expression that is returned if the condition results in TRUE
- exp_if_false is the expression that is returned if the condition results in FALSE

```
a
```

```
## [1] 100
```

```
ifelse(a < 7, "a is less than 7", "a is greater than 7")
```

```
## [1] "a is greater than 7"
```

We can use the **ifelse()** function to modify values in our dataframe. For example, we can create a new column in the dataframe called PopOver100 where its value is TRUE if the population is greater than 100 and FALSE otherwise.

```
df$PopMean <- ifelse(df$Population > mean(df$Population), TRUE, FALSE)
# df$PopMean2 <- df$Population > mean(df$Population)
head(df)
```

You might remember Q17 and Q18 from Day2's problem set. You can solve them using this **ifelse()** function.

```r
# Q17.
resources <- read.csv("resources.csv")

resources$oil[is.na(resources$oil)] = mean(resources$oil, na.rm = TRUE)
resources$logGDPcp[is.na(resources$logGDPcp)] = mean(resources$logGDPcp, na.rm = TRUE)
resources$regime[is.na(resources$regime)] = mean(resources$regime, na.rm = TRUE)
resources$metal[is.na(resources$metal)] = mean(resources$metal, na.rm = TRUE)
resources$illit[is.na(resources$illit)] = mean(resources$illit, na.rm = TRUE)
resources$life[is.na(resources$life)] = mean(resources$life, na.rm = TRUE)
summary(resources)

colnames(resources)[7] <- "life_expectancy"

# categorical variables: represents categories or group
# examples: Colors: Red, Blue, Green. Sizes: Small, Medium, Large. Types of Animals: Dogs, Cats, Birds.

# as.factor create levels.
# without as.factor, "low" will be stored as character, with as factor "low" will be stored as vector o

resources$life <- as.factor(ifelse(resources$life_expectancy < 50, "low",
  ifelse(resources$life_expectancy < 70, "medium",
    ifelse(resources$life_expectancy < 100, "high")
  )
))
head(resources)

# Q18.
# resources$democracy <- ifelse(resources$regime >= 0, 1, 0)
```

3.3 *for* Loop in R

"The **for** loop in R, repeats through sequences to perform repeated tasks. They work with an iterable variable to go through a sequence." The following code is the syntax of **for** loops in R:

```r
# This code is not executable. Just for illustration.
for (variable in sequence) {
  code_to_repeat
}
```

- variable is the iterative variable,
- sequence is the sequence which we need to loop through,
- code_to_repeat is the code that runs every iteration.

Here is an example:

```r
vec <- c(2, 7)
vec
```

```
## [1] 2 7
```

```r
vec2 <- c(10:17)
vec2
```

```
## [1] 10 11 12 13 14 15 16 17
```

```r
for (i in vec) {
  print(vec2[i])
```

```
}
```

```
## [1] 11
## [1] 16
```

With the **for** loops, we can now iterate through the entire dataframe.

For example, if you want to list all the years:

```
dim(df)
```

```
## [1] 16  7
```

```
n1 <- dim(df)[1]
n1
```

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

```
# n2 <- dim(df)[2]
for (i in 1:16) {
  print(df$Year[i])
}
```

```
## [1] 1947
## [1] 1948
## [1] 1949
## [1] 1950
## [1] 1951
## [1] 1952
## [1] 1953
## [1] 1954
## [1] 1955
## [1] 1956
## [1] 1957
## [1] 1958
## [1] 1959
## [1] 1960
## [1] 1961
## [1] 1962
```

You can also recode the variables using **for** loops and the **if-else** statement. We can start with listing out the years where the number of unemployed exceeds 300.

```
for (i in 1:dim(df)[1]) {
  if (df$Unemployed[i] > 300) {
    print(df$Year[i])
  }
}
```

```
## [1] 1949
## [1] 1950
## [1] 1954
## [1] 1958
## [1] 1959
## [1] 1960
## [1] 1961
## [1] 1962
```

```r
# For good practice, you can assign dim(df)[1] to a new variable.
# For example:
L <- dim(df)[1]
```

Let's create a new column called ratio, and initialize it with NAs.

```r
df$ratio <- NA
```

Now we want to recode the ratio variable, where we divide Employed over the sum of Unemployed and Employed.

```r
# L <- dim(df)[1] # 16
for (i in 1:L) {
  df$ratio[i] <- df$Employed[i] / (df$Unemployed[i] + df$Employed[i])
}
head(df)
```

```
##      GNP.deflator     GNP Unemployed Armed.Forces Population Year Employed
## 1947         83.0 234.289      235.6        159.0    107.608 1947   60.323
## 1948         88.5 259.426      232.5        145.6    108.632 1948   61.122
## 1949         88.2 258.054      368.2        161.6    109.773 1949   60.171
## 1950         89.5 284.599      335.1        165.0    110.929 1950   61.187
## 1951         96.2 328.975      209.9        309.9    112.075 1951   63.221
## 1952         98.1 346.999      193.2        359.4    113.270 1952   63.639
##          ratio
## 1947 0.2038469
## 1948 0.2081656
## 1949 0.1404647
## 1950 0.1544007
## 1951 0.2314762
## 1952 0.2477778
```

There is also a shortcut to get the ratio column.

```r
df$ratio2 <- df$Employed / (df$Unemployed + df$Employed)
```

3.4 *while* Loop in R

while loops work similarly to **for** loops. The following code is the syntax of a **while** loop.

```r
while (test_expression) {
  code_to_repeat
}
```

- test_expression is evaluated to a logical output (TRUE / FALSE)
- If TRUE, code_to_repeat will be executed. If FALSE, R skips the code_to_repeat.

However, different from the test_expression in the **if-else** statement, in **while** loops, the test_expression will be evaluated again when the execution of code_to_repeat is completed. The process **repeats** each time **until test_expression evaluates to FALSE**. When test_expression evaluates to FALSE, the loop exits.

A trivial example:

```r
while (FALSE) {
  print("F")
}
```

The code inside the **while** loop is never executed, because the test_expression is always FALSE.

If we flip the FALSE to TRUE:

```
# Don't run please
while (TRUE) {
  print("T")
}
```

This creates an endless **while** loop. It will keep running and printing "T"s in your console.

You should **NEVER** set your test_expression to the logical TRUE directly when writing programs. This could lead R to abort your session.

Here is another example:

```
i <- 1 # i is initialized to 1
while (i <= 6) {
  print(i)
  i <- i + 1 # increment i
}
```

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

Remember: **all for loops can be written as while loops, and vice-versa.**

Thus, when making a choice between the two, we should always consider which one is more appropriate for the application. In general, you should use a **for** loop whenever you already known how many times the loop should be executed (the number of iterations). If the execution of the loop body is based on certain condition, you should pick **while** loops instead. You will encounter **for** loops a lot more than **while** loops in QPM I.

Just to illustrate the equivalency:

```
vec <- c(1:10)
for (i in vec) {
  print(vec[i])
}
```

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

```
vec <- c(1:10)
i <- 1
while (i <= length(vec)) {
  print(vec[i])
  i <- i + 1
}
```

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

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

3.5 In-class exercises 3.3:

1. Write a function that samples with replacement from a given range with a given size. The user should also be able to choose whether to do sampling with or without replacement.
2. Write a **for** loop that does random sampling of size 1 without replacement, 10 times from the range 1 to 100. Remember to print your randomly sampled number for each iteration. Use the function you created in Q1.
3. Convert your **for** loop into a **while** loop.
4. Write a **for** loop that does random sampling of size 100 with replacement, 10 times from 1 to 10000. Calculate the mean for each iteration and store the means in a new vector called "samples". This time you do not need to print the sampled numbers for each iteration. Simply print out the vector that contains all the means. Use the function you created in Q1.

```
# To create a vector to store your values:
length <- 10
samples <- rep(NA, length)
```

5. Calculate the average of the means obtained in Q4.

4 Bootstrap and Simulation

4.1 Bootstrap

We will learn mode details about bootstrap in QPM1. Here we briefly touch the code!

First, we can bootstrap using `sample()` and `for` loop.

```
# Bootstrap the mean of GDPpc
worldTFR <- read.csv("./worldTFR.csv")
worldTFR <- na.omit(worldTFR)

B <- 100 # number of bootstrap
bootstrapped <- numeric(B) # Create numeric vector size of B to store results
set.seed(63130)
for (b in 1:B) {
  idx <- sample(1:dim(worldTFR)[1], replace=TRUE) # take random index with replacement
  temp <- worldTFR[idx, "GDPpc"] # subset the data and select the column
  bootstrapped[b] <- mean(temp) # Store bth mean
}

print(mean(bootstrapped))
```

```
## [1] 12728.24
```

Alternatively, we can use `boot` package.

```r
# install.packages("boot")
library(boot)

# Define the function first
mean_fn <- function(data, indices) {
  return( mean(data$GDPpc[indices]) )
}

bootstrapped2 <- boot(worldTFR, statistic = mean_fn, R = 100)
print(bootstrapped2)
```

```
##
## ORDINARY NONPARAMETRIC BOOTSTRAP
##
##
## Call:
## boot(data = worldTFR, statistic = mean_fn, R = 100)
##
##
## Bootstrap Statistics :
##     original   bias    std. error
## t1* 12657.75 31.17215    265.6022
```

`boot` package also provides useful `boot.ci()` function, which calculates different types of confidence intervals.

```r
boot.ci(bootstrapped2)
```

```
## Warning in boot.ci(bootstrapped2): bootstrap variances needed for studentized
## intervals
```

```
## BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
## Based on 100 bootstrap replicates
##
## CALL :
## boot.ci(boot.out = bootstrapped2)
##
## Intervals :
## Level      Normal              Basic
## 95%   (12106, 13147 )   (12107, 13188 )
##
## Level     Percentile            BCa
## 95%   (12128, 13209 )   (12025, 13132 )
## Calculations and Intervals on Original Scale
## Some basic intervals may be unstable
## Some percentile intervals may be unstable
## Some BCa intervals may be unstable
```

4.2 In-class exercises:

1. Bootstrap the mean of `MtoF04` in `worldTFR`.

4.3 Simulation

Very similar to the simulation studies; we can utilize `for` loop like we did. Another base R friendly way is using a function `replicate()`

```r
# Do it 1,000 times!
set.seed(63130)
results <- replicate(1000, roll())
print(mean(results))
```

```
## [1] 6.939
```

In the real research life, you may need to define a right function for you for bootstrap and simulations!