Day 3 Lecture

Cecilia Y. Sui

01/12/2022

Day 3 Outline:

1. Functions & Arguments
2. Control Statements (if, if-else, for, while)
3. Sampling using loops

1. Functions

R comes with many functions that you can use to do sophisticated tasks like random sampling. For example, you can round a number with the **round()** function, or calculate its factorial with the **factorial()** function. Using a function is pretty simple: just write the name of the function and then the data you want the function to operate on in parentheses. In fact, you have been using functions in Day 2's problem set, including sum(), mean(), max() when analyzing the dataframes.

```
# The following two functions come with base R.
# You do not need to install and load new packages to use them.
round(3.1415)
```

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

```
factorial(3)
```

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

The data that you pass into the function is called the function's argument. The argument can be raw data, an R object, or even the results of another R function. In this last case, R will work **from the innermost function to the outermost**.

```
die <- 1:6
mean(1:6)
```

```
## [1] 3.5
```

```
mean(die)
```

```
## [1] 3.5
```

```
round(mean(die))
```

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

```
ceiling(mean(die))
```

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

```
floor(mean(die))
```

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

```
ceiling(3.01)
```

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

```
floor(3.99)
```

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

1.1 Sampling using the sample() function

Base R comes with the 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 if **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] 2 4
```

```
die <- 1:6
sample(x = die, size = 3)
```

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

```
# 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(die, size = 3)
```

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

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

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

Often, the name of the first argument is not very descriptive, and it is usually obvious what the first piece of data refers to anyways.

But how do you know which argument names to use? If you try to use a name that a function does not expect, you will likely get an error:

```
round(3.1415926535, numbers = 3)
# Error in round(3.1415, numbers = 3) : unused argument (numbers = 3)
```

If you are not sure which names to use with a function, you can look up the function's arguments with **args()**. To do this, simply plug in the name of the function into the parentheses. For example, you can see that the round function takes two arguments, one named x and one named digits:

```
args(round)
```

```
## function (x, digits = 0)
## NULL
```

```
args(ceiling)
```

```
## function (x)
## NULL
```

```
args(floor)
```

```
## function (x)
## NULL
```

Did you notice that args shows that the digits argument of round is already set to 0?

Frequently, an R function will take optional arguments like digits. These arguments are considered optional because they come with a **default value**. You can pass a new value to an optional argument if you want, and R will use the default value if you do not.

For example, round() will round your number to 0 digits past the decimal point by default. To override the default, you can always specify your own argument value. For example,

```
round(3.1415)
```

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

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

```
## [1] 3.14
```

When you first start to learn R, it is recommended that you write out the **names of all your arguments** when you do a function call with multiple arguments. This not only helps you to learn the functions and their arguments better and faster, but also helps others to understand your code more easily. Moreover, you might not always remember correctly how the arguments should be ordered when first using these functions. Writing out the argument names helps **prevent errors**.

After using R for a while, you might notice that it is often for R users to skip the names for the first one or two arguments, and only write out the names for the rest of arguments.

If you do not write out the names of your arguments, R will match your values to the arguments in your function by order. You can learn about the correct order by **running the args() function, hovering over the function call, or using the help pages** .

For example, in the following code, the first value, die, will be matched to the first argument of sample, which is named **x**. The next value, 1, will be matched to the next argument, **size**:

```
sample(die, 1)
```

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

As you provide more arguments, it becomes more likely that your order and R's order may **not align**. As a result, values may get passed to the wrong argument. Argument names prevent this. R will always match a value to its argument name, no matter where it appears in the order of arguments.

For example, if we reverse the order of arguments in the function call above, it is still equivalent.

```
sample(size = 1, x = die) # notice the order is reversed here
```

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

1.1.1 Sampling without Replacement

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

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

1.1.2 Sampling with Replacement

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

```
sample(x = die, size = 3, replace = TRUE)
```

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

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

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

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

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

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 = TRUE)
dice
```

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

```
sum(dice)
```

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

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] 2 2
```

```
dice
```

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

```
dice
```

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

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.

In-class exercises 3.1:

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

4. We are going to use the world total fertility rate dataset again to do some sampling here. First, load your dataset as worldTFR.
5. Sample without replacement of size 100 from the TFR column in the dataset. What is the mean of your sample? Compare it to the mean of the entire column.
6. Repeat the sampling in Q5 five times, and report the means for each sampling. Compare the means to the mean of the entire column TFR.
7. Take the average of all the means obtained from Q6. What do you get? Is the new average closer to the true average for the column TFR?

1.2 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] 5 6
```

```
sum(dice)
```

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

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.

1.2.1 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)
  return(sum(dice)) # return value
}
```

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] 10
```

```
roll()
```

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

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)
##   return(sum(dice)) # return value
## }
## <bytecode: 0x136308db0>
```

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

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

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
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.

1.2.2 Arguments

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

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

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

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

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

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

Now roll2 will work as long as you provide rcamp 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:

```
roll2(rcamp = 1:4) # 4-sided
```

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

```
roll2(rcamp = 1:6) # 6-sided
```

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

```
roll2(rcamp = 1:20) # 20-sided
```

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

```
roll2(1:20) # without the argument name, it works the same
```

```
## [1] 22
```

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

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

As mentioned in the Error Message, we can fix the error by giving the rcamp argument a default value. To do this, set rcamp equal to a value when you define roll2:

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

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

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

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

```
roll2(2:10) # using rcamp = 2:10
```

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

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

roll3()
```

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

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

```
## [[1]]
## [1] 10
##
## [[2]]
## [1] 4 6
```

```
# 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.

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.

2. Control Statements

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.

2.1 *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.

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

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

```r
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.

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

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

2.2 $if - else$ 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:

```r
a <- 10
b <- 88
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:

```r
a <- 5
b <- 5
if (a < b) {
  print("a is smaller than b")
} else if(a==b) {
  print("a is equal to b")
} else {
  print("a is greater than b")
}
```

```
## [1] "a is equal to 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

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

```
## [1] "a is less 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 if TRUE if the population is greater than 100 and FALSE otherwise.

```
df$PopOver100 <- ifelse(df$Population > 110, TRUE, FALSE)
```

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

```
# Q17.
resources$life <- as.factor(ifelse(resources$life_expectancy < 50, "low",
                           ifelse(resources$life_expectancy < 70, "medium",
                           ifelse(resources$life_expectancy < 100, "high"))))

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

2.3 *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:

```
# 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:

```
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
```

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

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

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

There is also a shortcut to get the ratio column.

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

2.4 *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:

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

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

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
```

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.