R Programming: Functions Writing Strategies

Week 2 Wednesday

Miles Chen

Department of Statistics

UCLA

```
library(knitr)
```

Related Reading

- Introduction to Scientific Programming and Simulation Using R
  - Chapter 5 - Programming with Functions
  - Must be on UCLA Network or connected to VPN to access:
  - https://www.taylorfrancis.com/books/mono/10.1201/9781420068740/introduction-scientific-programming-simulation-using-owen-jones-robert-maillardet-andrew-robinson
- Advanced R
  - Chapter 6: Functions
  - https://adv-r.hadley.nz/functions.html
  - (This chapter goes much more in-depth than I do.)

Section 1

Functions Basics

Function Basics

Functions in R have the form

```
name <- function(argument_1, argument_2, ...) {
    expression_1
    expression_2
    ...
    return(output)
}
```

Here, `argument_1` and `argument_2` are the names of variables supplied to the function. `expression_1`, `expression_2`, and `output` are expressions. `name` is the name of the function, and will be called to action with `name(x, y)`. The result of the function will be `output`, which can be assigned to an object.

Function Basics

If the function has only one line the curly braces can be left off (though it is not recommend). If `return()` is not used, the value returned is the last expression in the function.

Functions are objects themselves, which means we can work with them like any other object in R.

```
f <- function(x) x * x # no curly braces, x * x is the value returned
list(f) # a function can be put into a list
```

```
[[1]]
function (x)
x * x
```

```
typeof(f) # the type is 'closure'
```

```
[1] "closure"
```

```
mode(f) # the mode(f) is 'function'
```

```
[1] "function"
```

Return values

Explicit returns - uses `return()` to explicitly return a value

```
f <- function(x) {
    return(x * x)
}
```

Implicit - the value in the very last expression is returned. This is the recommended style.

```
f <- function(x) {
    x * x
}
```

Returning values

Functions can only return one object.

If we want a function to return more than one value, we must combine them into a single object using either vectors or lists.

```
f <- function(x) {
    c(x, x ^ 2, x ^ 3)
}
f(2)
```

```
[1] 2 4 8
```

Return multiple values with a list

```
f <- function(x) {
    list(x = x, square = x ^ 2, cube = x ^ 3)
}
f(2)
```

```
$x
[1] 2

$square
[1] 4

$cube
[1] 8
```

Argument names

When defining a function we are also defining names for the arguments. When we call the function to use it we use these names to enter arguments.

If you do not specify the argument names, R assumes you are putting them in order.

```
f <- function(x, y, z) {
  paste0("x = ", x, ", y = ", y, ", z = ", z)
}
f(1, 2, 3)
```

```
[1] "x = 1, y = 2, z = 3"
```

Argument names

When you call the function, you can specify the argument names and supply them in any order.

```
f(z = 1, x = 2, y = 3)
```

```
[1] "x = 2, y = 3, z = 1"
```

Argument names

If you only specify the names of one or two arguments, R does its best to match them. If you give the name of argument that is not part of the function definition, R gives an error "unused argument."

```
f(y = 2, 1, 3)
```

```
[1] "x = 1, y = 2, z = 3"
```

```
f(y = 2, 1, x = 3)
```

```
[1] "x = 3, y = 2, z = 1"
```

```
f(1, 2, 3, m = 1)
```

```
Error in `f()`:
! unused argument (m = 1)
```

Argument values are required

R requires a value for any argument that is used inside the function.

In the following code, I define a function with 4 arguments. Argument `w` however is not used anywhere in the code, while the other three arguments are. When R calls the function, it can operate without a value for `w`

```
f <- function(x, y, z, w) {
  paste0("x = ", x, ", y = ", y, ", z = ", z)
}
f(1, 2, 3, 4) # w is provided the value 4, but is not used anywhere
```

```
[1] "x = 1, y = 2, z = 3"
```

```
f(1, 2, 3) # w is not provided. it still runs
```

```
[1] "x = 1, y = 2, z = 3"
```

Argument values are required

```
f <- function(x, y, z, w) {
  paste0("x = ", x, ", y = ", y, ", z = ", z)
}
f(x = 1, y = 2, w = 4) # z is not provided. z is in the code. result: error
```

```
Error in `f()`:
! argument "z" is missing, with no default
```

Provide default values with the = sign

To avoid the error of not having a value, you can provide default values for function arguments.

```
f <- function(x = 1, y = 1, z = 1) {
  paste0("x = ", x, ", y = ", y, ", z = ", z)
}
f() # no values provided: uses all defaults
```

```
[1] "x = 1, y = 1, z = 1"
```

```
f(2) # 2 gets plugged in for the first argument
```

```
[1] "x = 2, y = 1, z = 1"
```

```
f(z = 3) # 3 gets plugged in for the argument z
```

```
[1] "x = 1, y = 1, z = 3"
```

Values created inside a function only exist inside the function

Values created inside a function only exist inside the function.

In general, you should not try to change values in the global environment from inside a function.

If you want to get a value 'out' of a function, it needs to be (part of) the returned object.

Values created inside a function only exist inside the function

```
x <- 10 # x is 10 in the global environment
f <- function(x){
  x <- x + 55 # We add 55 to x inside the function
  x # the function returns the new value of x
}
f(x) # we execute f(x). It returns 65
```

```
[1] 65
```

```
x # the value of x in the global environment is unchanged
```

```
[1] 10
```

To save the value of a function, you must assign it

```
x # x in the global environment is 10
```

```
[1] 10
```

```
f(x) # We plug x into the function and get 65
```

```
[1] 65
```

```
x # x is still 10 in the global environment
```

```
[1] 10
```

```
x <- f(x) # we assign the result of the function to x
x # x in the global environment is now 65
```

```
[1] 65
```

When to use functions

The goal of a function should be to create a small reusable piece of code that can be adapted for different values.

As you code, it is common to find yourself doing similar tasks multiple times. We will often copy and paste some code from one part of our project to another part and then make a few modifications.

Guideline: When to make a function

You should not copy and paste the same chunk of code (and alter variables) more than two times. By the time you have to copy and paste the code chunk a third time, you should write a function.

Advice

It is better to change code in one location than to try to change copies of the code everywhere.

Name your function to make it clear what the function does. Despite my frequent usage of them for examples in these slides, function names like `f` or `f2` are not good names.

Functions that return a `TRUE` or `FALSE` value should probably start with `is_` (for example, `is.character()` or `is_prime()`)

Functionality should be simple enough to be quickly understood.

Section 2

Function Writing Strategies

Writing a Bubble Sort Function

Bubble sort is a simple (but not very efficient) way to sort a vector.

From Wikipedia:

Starting from the beginning of the list, compare every adjacent pair, swap their position if they are not in the right order (the latter one is smaller than the former one).

After each iteration, one less element (the last one) is needed to be compared until there are no more elements left to be compared.

Bubble Sort

We start with the vector 5 1 8 2 4.

We want to sort this.

Our first pass:

(**5 1** 8 2 4) -> (**1 5** 8 2 4) Algorithm compares the first 2 elements, and swaps since $5 > 1$.

(1 **5 8** 2 4) -> (1 **5 8** 2 4), No swap because $5 < 8$

(1 5 **8 2** 4) -> (1 5 **2 8** 4), Swap because $8 > 2$

(1 5 2 **8 4**) -> (1 5 2 **4 8**), Swap because $8 > 4$

At this point, we know that the largest element is all the way to the right.

Second Pass

For the next pass, we perform the same comparisons. Because we know 8 is in the correct position from the last pass, we can stop after the third swap. I put *8* in *italics* to indicate its position is 'locked'

(**1** **5** 2 4 *8*) -> (**1** **5** 2 4 *8*), No swap because $1 < 5$

(1 **5** **2** 4 *8*) -> (1 **2** **5** 4 *8*), Swap because $5 > 2$

(1 2 **5** **4** *8*) -> (1 2 **4** **5** *8*), Swap because $5 > 4$

It stops this pass because it knows that *8* is already in the correct spot.

Third Pass

At this point, the vector is in order, but the algorithm does not know this.

The algorithm does know that the largest number (8) is in the right most position, and the second largest number (5) is in the second largest position. Again, I put *5* and *8* in *italics* to inidicate their positions are 'locked.'

It runs through the comparisons again, and stops after the second swap (because it knows the two largest values are sorted.)

(**1** **2** 4 *5* *8*) -> (**1** **2** 4 *5* *8*), No swap

(1 **2** **4** *5* *8*) -> (1 **2** **4** *5* *8*), No swap

Now it knows the three largest values are in the right spot.

Fourth and Final Pass

Now that we have performed 3 passes, the algorithm knows that the three largest values are in the correct spot.

It does one final comparison and does not swap values.

(**1** **2** *4* *5* *8*) -> (**1** **2** *4* *5* *8*), No swap

Now we know all the values are in the correct order

(1 2 4 5 8)

Animated gif from Wikipedia

https://en.wikipedia.org/wiki/Bubble_sort#/media/File:Bubble-sort-example-300px.gif

Planning a function: Break down your tasks

Let's try to code the bubble sort algorithm as a function.

Big tasks can be broken down into smaller tasks

Let's break down the bubble sort algorithm into smaller tasks:

- Compare two adjacent values and swap them if they are not in order. This is broken down into:
  - Compare adjacent values
  - Swap two adjacent values
- Perform multiple sweeps on the vector, but each time we perform a sweep, we do one fewer comparison
  - Each sweep does comparisons/swaps for all positions that are not 'locked'
  - The following sweep will have one more position locked

Getting started

Advice: Do not immediately start writing a function.

Let's say you just want to see if you can swap the first two values in a vector.

```
x <- 5:1
x
```

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

First: what should the result be when you swap the first two values?

Answer: 4, 5, 3, 2, 1

Next: How do we do this with code?

Attempting a function

Here I attempt to

```
swap_first_two <- function(x) {
  x[1] <- x[2]
  x[2] <- x[1]
  x  # returns the last value
}
x <- 5:1
swap_first_two(x)
```

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

Chances are you will not code it 100% correctly the first time and you have to figure out what is wrong.

When a function runs, it creates its own environment/scope. Values created inside the function are not observable from the outside. This makes debugging very hard.

Writing a function: Stage your code in the global environment

Instead of trying to write a function immediately, see if you can get the basic idea of the algorithm working in the global environment. This way you can observe the values and variables directly.

Be sure you delete everything in the global environment first.

By keeping things in the global environment, we can monitor changes as they happen

```
# let's try to swap the first two values in the vector 5:1
rm(list = ls())
x <- 5:1
x[1] <- x[2]
x # x is in the global environment and I can check its value
```

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

```
x[2] <- x[1]
x # this step doesn't work because x[1] is no longer 5
```

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

Writing the code to perform a swap

```
# second attempt to swap the first two values
rm(list = ls()) # clear the global environment
x <- 5:1

temp <- x[1]
x[1] <- x[2]
x[2] <- temp

x  # it works!
```

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

Test the code again

```
rm(list = ls())
x <- 1:8

temp <- x[1]
x[1] <- x[2]
x[2] <- temp

x  # the first two values swapped
```

```
[1] 2 1 3 4 5 6 7 8
```

Building up your code: condition the swap

We have successfully swapped the first two values.

Now we want to do this only if the left value is larger than the right value. We still work in the global environment

```
rm(list = ls())
x <- 5:1 # should swap
if (x[1] > x[2]) {
  temp <- x[1]
  x[1] <- x[2]
  x[2] <- temp
}
x # indeed, it swapped
```

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

With each change, test the code

With every change, we test the code to make sure it does what we expect

```
rm(list = ls())
x <- 1:5 # should not swap
if (x[1] > x[2]) {
  temp <- x[1]
  x[1] <- x[2]
  x[2] <- temp
}
x # indeed, no swap
```

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

How we broke down the task

- Compare two adjacent values and swap them if they are not in order. This is broken down into:
  - Compare adjacent values
  - Swap two adjacent values
- Perform multiple sweeps on the vector, but each time we perform a sweep, we do one fewer comparison
  - **Each sweep does comparisons/swaps for all positions** that are not 'locked'
  - The following sweep will have one more position locked

Build up the code: perform a sweep

We now want to perform a sweep: do the swaps for all values in a vector. No positions are locked in the first sweep.

I'll try this with a for loop using `seq_len(length(x))`.

I replace *1 with i* and *2 with i + 1*

```
rm(list = ls())
x <- 5:1

if (x[1] > x[2]) {
  temp <- x[1]
  x[1] <- x[2]
  x[2] <- temp
}
```

```
rm(list = ls())
x <- 5:1
for (i in seq_len(length(x))) {
  if (x[i] > x[i + 1]) {
    temp <- x[i]
    x[i] <- x[i + 1]
    x[i + 1] <- temp
  }
}
```

Build up the code: perform a sweep

Our code now looks like this, but it produces an error

```
rm(list = ls())
x <- 5:1
for (i in seq_len(length(x))) {
  if (x[i] > x[i + 1]) {
    temp <- x[i]
    x[i] <- x[i + 1]
    x[i + 1] <- temp
  }
}
```

```
Error in `if (x[i] > x[i + 1]) ...`:
! missing value where TRUE/FALSE needed
```

Finding the error

Because our code is in the global environment, I can search for the error. The error says `Error in if (x[i] > x[i + 1]) { : missing value where TRUE/FALSE needed`. So We look at the values `i`, `x[i]` and `x[i + 1]`

```
i
```

```
[1] 5
```

```
x[i]
```

```
[1] 5
```

```
x[i + 1]
```

```
[1] NA
```

We see that `i` is 5 and `i+1` is 6 and `x[6]` is a missing value.

Fixing our code

If my vector is length 5, I only need to do 4 potential swaps. So my sequence length is not `length(x)` but rather `(length(x) - 1)`

```
rm(list = ls())
x <- 5:1
for (i in seq_len(length(x) - 1)) {
  if (x[i] > x[i + 1]) {
    temp <- x[i]
    x[i] <- x[i + 1]
    x[i + 1] <- temp
  }
}
x # it has moved the largest value all the way to the right
```

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

Testing the code (still in the global environment)

We try more cases to make sure the code works properly

```
rm(list = ls())
x <- c(1, 2, 5, 8, 3, 6) # a new case
for (i in seq_len(length(x) - 1)) {
  if (x[i] > x[i + 1]) {
    temp <- x[i]
    x[i] <- x[i + 1]
    x[i + 1] <- temp
  }
}
x # it has moved the largest value all the way to the right
```

```
[1] 1 2 5 3 6 8
```

Another test of the code (still in the global environment)

```
rm(list = ls())
x <- sample(10) # another test
x
```

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

```
for (i in seq_len(length(x) - 1)) {
  if (x[i] > x[i + 1]) {
    temp <- x[i]
    x[i] <- x[i + 1]
    x[i + 1] <- temp
  }
}
x # still works
```

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

Broken down tasks again

We have successfully written code that performs a 'sweep.' Each sweep moves the largest value all the way to the end of the vector.

Let's see what still needs to be done:

- Compare two adjacent values and swap them if they are not in order. This is broken down into:
  - Compare adjacent values
  - Swap two adjacent values
- **Perform multiple sweeps on the vector**, but each time we perform a sweep, we do one fewer comparison
  - Each sweep does comparisons/swaps for all positions that are not 'locked'
  - **The following sweep will have one more position locked**

Multiple Sweeps

My current code does one sweep and moves the largest value all the way to the right. What I can do is simply repeat the code. When I repeat it, the second largest value is moved to the second to last position.

```
rm(list = ls())
x <- 5:1
for (i in seq_len(length(x) - 1)) {
    if (x[i] > x[i + 1]) {
      temp <- x[i]
      x[i] <- x[i + 1]
      x[i + 1] <- temp
    }
  }
x
```

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

```
for (i in seq_len(length(x) - 1)) {
    if (x[i] > x[i + 1]) {
      temp <- x[i]
      x[i] <- x[i + 1]
      x[i + 1] <- temp
    }
  }
x
```

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

A loop of loops

Now we want to perform multiple sweeps. I put my current code into another for loop. With a vector of length 5, I need to do a total of 4 sweeps.

I can simply put the loop that performs a sweep into another loop that will perform the sweep `length(x) - 1` times.

Multiple Sweeps

```
rm(list = ls())
x <- 5:1
for (s in seq_len(length(x) - 1)) { # outer loop performs multiple sweeps
  for (i in seq_len(length(x) - 1)) { # inner loop performs a single sweep
    if (x[i] > x[i + 1]) {
      temp <- x[i]
      x[i] <- x[i + 1]
      x[i + 1] <- temp
    }
  }
}
x
```

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

Adding a tiny bit of efficiency

Our code can sort a vector.

However, there is a bit of inefficiency. As we perform multiple sweeps, each sweep works through the entire vector. This is unnecessary.

- The first time we perform a sweep, we should move through the entire vector `(length(x) - 1)`.
- The second time we perform a sweep, we do not need to complete the entire vector. We know the final value is already in position. We only need to move through the `length(x) - 2`
- The third time we perform a sweep, we only need to move through `length(x) - 3`

Locking in values

```
rm(list = ls())
x <- 5:1
locked <- 0 # begin with no values locked
for (s in seq_len(length(x) - 1)) {
  for (i in seq_len(length(x) - 1 - locked)) { # length of each sweep adjusted
    if (x[i] > x[i + 1]) {
      temp <- x[i]
      x[i] <- x[i + 1]
      x[i + 1] <- temp
    }
  }
  locked <- locked + 1 # with each sweep, one more value is locked
}
x
```

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

Note on the loop

The inner for loop is done for `(i in seq_len(length(x) - 1 - locked))`

This is a good example of where using `seq_len()` is superior to doing `1:(length(x) - 1 - locked)`

The number of swaps to perform in each iteration is `length(x) - 1 - locked`

For the final iteration, 4 values become locked, and `length(x) - 1 - locked` will equal 0.

`1:0` will actually be a vector and will cause the for loop to still run.

`seq_len(0)` will be a length 0 vector and the code in the loop will not run and the code will stop.

Let's test it out some more

```
rm(list = ls())
x <- sample(40) # Randomly sorted vector
locked <- 0
for (s in seq_len(length(x) - 1)) {
  for (i in seq_len(length(x) - 1 - locked)) {
    if (x[i] > x[i + 1]) {
      temp <- x[i]
      x[i] <- x[i + 1]
      x[i + 1] <- temp
    }
  }
  locked <- locked + 1
}
x
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
[26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
```

Now we make it a function

```
rm(list = ls())
bubble_sort <- function(x){
  locked <- 0
  for (s in seq_len(length(x) - 1)) {
    for (i in seq_len(length(x) - 1 - locked)) {
      if (x[i] > x[i + 1]) {
        temp <- x[i]
        x[i] <- x[i + 1]
        x[i + 1] <- temp
      }
    }
    locked <- locked + 1
  }
}
```

Test our function

```
x <- sample(100)
bubble_sort(x)
```

Wait, nothing happened

Check our function

Silly me! I forgot to have a return object

```
rm(list = ls())
bubble_sort <- function(x){
  locked <- 0
  for (s in seq_len(length(x) - 1)) {
    for (i in seq_len(length(x) - 1 - locked)) {
      if (x[i] > x[i + 1]) {
        temp <- x[i]
        x[i] <- x[i + 1]
        x[i + 1] <- temp
      }
    }
    locked <- locked + 1 # with each sweep, one more value is locked
  }
  x
}
```

Test again

```
x <- sample(100)
bubble_sort(x) # it works!
```

```
  [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
 [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
 [37]  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
 [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
 [73]  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
 [91]  91  92  93  94  95  96  97  98  99 100
```

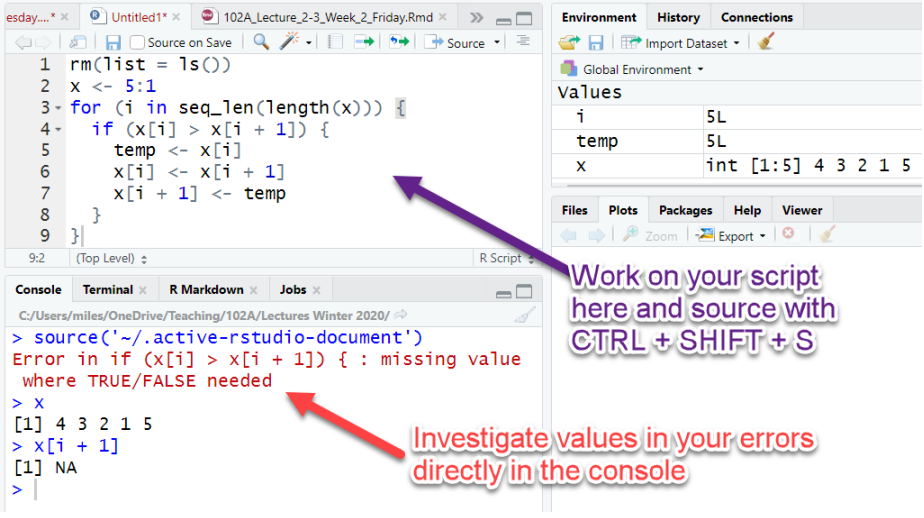
Recap

1. Break down the task into smaller steps.
2. Don't start building your function right away.
3. Work in the global environment.
4. Write your code.
5. When something breaks, check the values in the global environment
6. Clear your environment each time you change something
7. Add to your code incrementally
8. Test each change thoroughly before moving on

Tips on efficiency

- When drafting your code, work with a script file
- Put `rm(list = ls())` at the top of your script so it clears the global environment each time you source it
- Make changes and source them with **Ctrl/Cmd + Shift + S**
- Investigate errors directly in the console

Efficiency tips



Getting started with homework 2

One task is to draw the game board. We want to keep the code flexible to allow for different columns and rows.

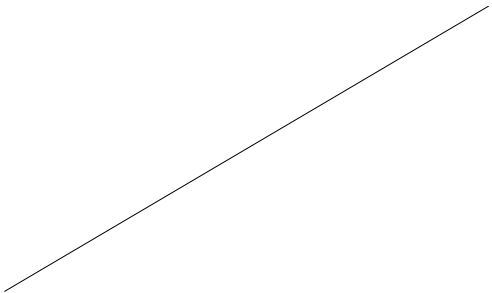
Let's start by working in the global environment and creating a new blank plot.

```
rm(list = ls())
plot.new()
```

Add a single line

Let's see what happens if I want to draw a line segment. Our board is 10 x 10

```
rm(list = ls())
plot.new()
segments(x0 = 0, y0 = 0, x1 = 10, y1 = 10)
```



Fix a single line

We don't want a diagonal line.

```
rm(list = ls())
plot.new()
segments(x0 = 0, y0 = 0, x1 = 10, y1 = 0)
```

Two lines

```
rm(list = ls())
plot.new()
segments(x0 = 0, y0 = 0, x1 = 10, y1 = 0)
segments(x0 = 0, y0 = 1, x1 = 10, y1 = 1)
```

Three lines

```
rm(list = ls())
plot.new()
segments(x0 = 0, y0 = 0, x1 = 10, y1 = 0)
segments(x0 = 0, y0 = 1, x1 = 10, y1 = 1)
segments(x0 = 0, y0 = 2, x1 = 10, y1 = 2)
```

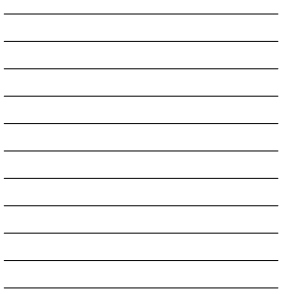
Fix plot limits and window settings

We can use `plot.window()` and the arguments to set the limits and aspect ratio.

```
rm(list = ls())
plot.new()
plot.window(xlim = c(0, 10), ylim = c(0, 10), asp = 1)
segments(x0 = 0, y0 = 0, x1 = 10, y1 = 0)
segments(x0 = 0, y0 = 1, x1 = 10, y1 = 1)
segments(x0 = 0, y0 = 2, x1 = 10, y1 = 2)
```

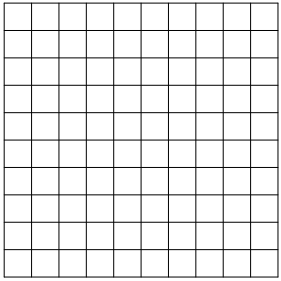
A loop to make 10 lines

```
rm(list = ls())
plot.new()
plot.window(xlim = c(0, 10), ylim = c(0, 10), asp = 1)
for(r in 0:10){
  segments(x0 = 0, y0 = r, x1 = 10, y1 = r)
}
```



Add vertical lines

```
rm(list = ls())
plot.new()
plot.window(xlim = c(0, 10), ylim = c(0, 10), asp = 1)
for(r in 0:10){
  segments(x0 = 0, y0 = r, x1 = 10, y1 = r)
}
for(c in 0:10){
  segments(x0 = c, y0 = 0, x1 = c, y1 = 10)
}
```



Continue building up

Continue building up the code, such as adding code to place text values.

I recommend working in the global environment with actual values.

In the beginning, it's probably a good idea to clear the environment each time you run your script.

After you get it working completely in the global environment, then begin to make adjustments to make it a function with variables.

One potential source of problems is that the number or rows and the number of columns is both 10. This can become confusing when you turn your code into a function where you want the number of rows and columns to be variables. It may be helpful to use `row = 9` and `col = 10` in order to differentiate the two values. When it is time to replace the numbers with variables, you can easily find all instances of 9 and replace it with `row` and all instances of 10 and replace it with the variable `col`.