



**Calhoun: The NPS Institutional Archive**  
**DSpace Repository**

---

Faculty and Researchers

Faculty and Researchers' Publications

---

2014-10

## Lab #4: Writing Functions

Fricker, Ronald D. Jr

---

<http://hdl.handle.net/10945/66312>

---

This publication is a work of the U.S. Government as defined in Title 17, United States Code, Section 101. Copyright protection is not available for this work in the United States.

*Downloaded from NPS Archive: Calhoun*



Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

**Dudley Knox Library / Naval Postgraduate School**  
**411 Dyer Road / 1 University Circle**  
**Monterey, California USA 93943**

<http://www.nps.edu/library>



**NAVAL  
POSTGRADUATE  
SCHOOL**

---

**LAB #4: WRITING FUNCTIONS**

---

**Probability (OA3101)**

## Lab: Writing Functions

**Goal:** Introduce students to how to use R scripts and logs and how to write R functions.



**Lab type:** Interactive lab demonstration.

**Time allotted:** 50 minutes.

**Reading Assignment:** *The R Book*, chapters 2.15 – 2.17

*A note on this week's reading assignment: The text, particularly section 2.15, assumes you are familiar with many concepts that you will not learn until OA3102. Don't despair! When you read the chapter, focus on the concepts related to writing functions in R and don't try to figure out the details of the examples themselves. That is, spend your time understanding how the functions in the red text work, not what the specific calculations of the functions mean.*

### DEMONSTRATION

1. Keeping track of your work in R.
  - a. Script files. A great way (in my opinion) of keeping track of your work is to save it in the format of an R script. This is basically a mini-program that you can re-run any time you want.
    - i. Benefits of using a script file:
      - R scripts are also a great way to document what you have done in a particular analysis (so you can repeat it later, for example). Along with the R Editor, R scripts can also make it easier to work with the R command line.
      - With a script file you can always reproduce your work and your results. That may not seem important now with your coursework, but it will be when you start doing your own research (say with your thesis).
      - Also, it's a great way to keep code that you can refer back to later or reuse. I do this all the time.
    - ii. An R script is basically just a text file that ends with a ".R" extension.
    - iii. Creating and opening a script file in R Studio:
      - In the upper left corner, click on the  icon and select "R Script." This will open up a blank window in the upper left quadrant of your screen.
      - Now you can type commands in this window, that you can execute either by highlighting the text and entering Ctrl + Enter or by clicking the  button.

- Most importantly, you can save your script file by clicking on the save button. This will save your script file with the name you give it and the .R extension.
- Then, next time you want to use it, just right click on it and use “Open with” and select R Studio.
  - Alternatively, you can associate the .R extension with R Studio so you can just double click on the file to launch R Studio and open the file simultaneously.
  - Of, if you’re like me and prefer to use plain ol’ R GUI, you just need to double click on the file to launch the R GUI and open the file in it (since the .R extension is usually associated with it by default).
- iv. Creating and using a script file in the R GUI. After opening it in R (PCs: File > Open script; Macs: File > Open Document), you can execute the commands right from the Editor window:
  - PCs: Put the cursor on a line and hit Ctrl+R. Note that the cursor automatically moves down to the next line so, if you want to execute a sequence of commands, you can just keep hitting Ctrl+R. Alternatively, you can highlight part of a line or a bunch of lines and select Edit > Run line or selection, or if you want to run everything in the script, choose Edit > Run all.
  - Macs: Put the cursor on a line and hit Command+Enter. Note that, unlike on the PC, the cursor *does not* automatically move down to the next line. Alternatively, you can highlight part of a line or a bunch of lines and hit Command+Enter.
- v. So, if you're going to follow along in the lab, download the R script called "Lab 4 Script.R" from Sakai and either open it up in R Studio or load it into your R Editor window.
- b. Log files. While using a script can help you keep a record of the commands you run or the functions you've written, it does not keep track of the details of what happened during a particular session.
  - i. From the R Console:
    - In Windows-based PCs running the R Console, the easiest way to save what you've done is to use the "Save to file..." option under the File pull down menu in the R Console. This will save *whatever is in your console window* to a text file. This is usually sufficient for most sessions, but note that for long sessions or if you print out large datasets, earlier material may have exceeded the R Console's window and be lost.
    - On Macs running the R Console, use the "Save As..." option under the File pull down menu.
  - ii. In R Studio, right now you can't do this (other than copy and pasting from the Console screen to a text file).

- In reality, you'll probably first want to save your Console output a lot, but eventually you'll find this a cumbersome way to work because saving all the console output is just too much and too confusing.
- What works better for many more experienced users is to make well-organized and well-documented script files. From them, you can always recreate any output you want.

## 2. About Writing Functions in R.

- a. In R, you can automate any calculation that you want to repeat with a function. Think of a function as a little program or subroutine.
- b. You can recognize a function in R because the function name must be followed with parenthesis. Thus, for example, the command `ls()` is a function that lists the things in your working directory. The empty parentheses just mean that you are not passing any arguments to the function.
  - i. Note that if you do not type the parenthesis, then R outputs the function's code.
  - ii. For example, just type `ls` and see what you get.

- c. The syntax for creating a function is:

```
new.function <- function(argument1, argument2,...)
{
  lines of code using the arguments
}
```

where you then run the function with

```
new.function(argument1 value, argument2 value,...)
```

- d. Here's a simple function:

```
square <- function(x)
{
  x^2
}
```

And here's an example of using the function:

```
> square(4)
[1] 16
```

- e. When creating a function, you can specify the default values for the arguments by including them in the definition of the arguments. For example:

```
cube <- function(x=2)
{
  x^3
}
```

And here's an example of using the function with and without the default:

```
> cube()
[1] 8
> cube(10)
```

```
[1] 1000
```

- f. Note that, barring any other instructions in the function, it automatically returns the results of the last line. For example:

```
square <- function(x)
{
  x^3 #Computed but not returned
  x^2
}
> square(10)
[1] 100
```

- g. Note that variables defined within a function are only available within the function. For example:

```
trivial.function <- function()
{
  x <- 99
  x
}
> trivial.function()
[1] 99
> x
Error: object "x" not found
```

- h. The great thing about R functions is that they can call other functions. For example, if we wanted to create a function to calculate the average of a vector of numbers, you could create the function:

```
average <- function(x)
{
  sum(x)/length(x)
}
```

This function uses two other functions that already exist in R: the sum function adds up the numbers in its argument and the length function returns the number of items in its argument. For example:

```
y <- c(3,3,2,4,5,5)
```

```
sum(y)
[1] 22
```

```
length(y)
[1] 6
```

```
sum(y)/length(y)
[1] 3.666667
```

```
average(y)
[1] 3.666667
```

Of course, R already has a built in function for calculating averages called `mean()`:

```
mean(y)
[1] 3.666667
```

- i. In Chapter 2 we learned about unions and intersections, so let's write a more complicated function that takes two sets and returns either the union or the intersection of the sets depending on what we tell it to do.

```
set.function <- function(set.A, set.B, union.or.intersection)
{
  if(union.or.intersection == "union")
  {
    all <- c(as.vector(set.A), as.vector(set.B))
    unique(all)
  }
  else
  {
    x <- as.vector(set.A)
    y <- as.vector(set.B)
    location <- match(x, y, 0L)
    unique(y[location])
  }
}
```

Let's test it out:

```
> set.function(1:4, 3:6, "union")
[1] 1 2 3 4 5 6

> set.function(1:4, 3:6, "intersection")
[1] 3 4

> x <- c("a", "b", "c")
> y <- c("a")

> set.function(x, y, "union")
[1] "a" "b" "c"

> set.function(x, y, "intersection")
[1] "a"
```

- j. Now, the code for the intersection part of `set.function` does some stuff we've never seen before. Besides reading the help pages, one way to get some insight

into what it does is to interactively run each line on a toy example and see what they do. Let's do that here:

```
set.A <- 1:10
set.A
[1] 1 2 3 4 5 6 7 8 9 10

set.B <- 5:15
set.B
[1] 5 6 7 8 9 10 11 12 13 14 15

x <- as.vector(set.A)
x
[1] 1 2 3 4 5 6 7 8 9 10

y <- as.vector(set.B)
y
[1] 5 6 7 8 9 10 11 12 13 14 15

location <- match(x, y, 0L)
location
[1] 0 0 0 0 1 2 3 4 5 6

y[location]
[1] 5 6 7 8 9 10

unique(y[location])
[1] 5 6 7 8 9 10
```

### 3. A Bit About Executing Functions.

- a. Remember, if you just type in the name of a function without parentheses after them, you're telling R to print out the function code. For example, type in `average` and see that you get back the code for the function we created earlier.
- b. If you are using a function with more than one argument, you have to be a bit careful about how you specify the values of the arguments. Let's illustrate with a simple function:

```
power.func <- function(x=2,y=3)
{
  x^y
}
```



- Now, if you execute the function by just putting numbers in for the arguments, then you must put the numbers exactly in the order of the arguments. In this case, you have to put the number for  $x$  first and  $y$  second. For example,

`power . func (3 , 5)`

is not the same as

`power . func (5 , 3)`

- Alternatively, you can use the variable names in the argument and then you can put them in any order you want. For example,

`power . func (x=3 , y=5)`

is the same as

`power . func (y=5 , x=3)`

Name: \_\_\_\_\_

### LAB #4 HOMEWORK

1. Read the assignment in *The R Book*. As you read, try out some of the commands and examples in R. Remember, *don't worry about the particular details of what the functions mean*. You'll learn most of this in OA3102. Right now, just concentrate on the R syntax and how to construct functions.
2. Write four functions and demonstrate that they work.

(a) First, create a function called `permute` that, when given values for arguments `n` and `r`, calculates  $P_r^n = \frac{n!}{(n-r)!}$ . In writing the function, it will probably be helpful to use the built-in function `factorial()`.

R code and example output:

(b) Now, create a function called `combine` that, when given values for arguments `n` and `r`, calculates  $C_r^n = \frac{n!}{r!(n-r)!}$ . In writing the function, it will probably be helpful to use the built-in function `factorial()`.

R code and example output:

- (c) Demonstrate that the results of your `combine` function match the results you get from the built-in R function `choose(n, r)`.

R code and example output showing they match:

- (d) Finally, create a function that combines the previous two into one. Call it `pandc` that, when given values for arguments `n` and `r`, returns either  $P_r^n$  or  $C_r^n$  depending on the value of a third argument called `which` that can have value either `c` or `p`.

R code and example output: