Creating Efficient & Readable Code in R

*"To iterate is human, to recurse divine."* - L. Peter Deutsch

Don't repeat yourself (DRY) is a software development principle aimed at reducing repetition. Formulated by Andy Hunt and Dave Thomas in their book [The Pragmatic Programmer](http://www.amazon.com/Pragmatic-Programmer-Journeyman-Master/dp/020161622X/ref=sr_1_1?s=books&ie=UTF8&qid=1456066112&sr=1-1&keywords=the+pragmatic+programmer), the DRY principle states that "every piece of knowledge must have a single, unambiguous, authoritative representation within a system." This principle has been widely adopted to imply that you should not duplicate code. Although the principle was meant to be far grander than that[[1]](#footnote-22), there's plenty of merit behind this slight misinterpretation.

Removing duplication is an important part of writing efficient code and reducing potential errors. First, reduced duplication of code can improve computing time and reduces the amount of code writing required. Second, less duplication results in less creation and saving of unnecessary objects. Inefficient code invariably creates copies of objects you have little interest in other than to feed into some future line of code; this wrecks havoc on properly managing your objects as it basically results in a global environment charlie foxtrot! Less duplication also results in less editing. When changes to code are required, duplicated code becomes tedious to edit and invariably mistakes or fat-fingering occur in the cut-and-paste editing process which just lengthens the editing that much more.

Furthermore, its important to have readable code. Clarity in your code creates clarity in your data analysis process. This is important as data analysis is a collaborative process so your code will likely need to be read and interpreted by others. Plus, invariably there will come a time where you will need to go back to an old analysis so your code also needs to be clear to your future-self.

This section covers the process of creating efficient and readable code. First, I cover the basics of [writing your own functions(#functions) so that you can reduce code duplication and automate generalized tasks to be applied recursively. I then cover [loop control statements](#control_structures) which allow you to perform repetititve code processes with different intentions and allow these automated expressions to naturally respond to features of your data. Lastly, I demonstrate how you can [simplify your code](#pipe) to make it more readable and clear. Combined, these tools will move you forward in writing efficient, simple, *and* readable code.

# Functions

R is a functional programming language, meaning that everything you do is basically built on functions. However, moving beyond simply *using* pre-built functions to *writing* your own functions is when your capabilities really start to take off and your code development/writing takes on a new level of efficiency. Functions allow you to reduce code duplication by automating a generalized task to be applied recursively. Whenever you catch yourself repeating a function or copy and pasteing code there is a good change that you should write a function to eliminate the redundancies.

Unfortunately, due to their abstractness, grasping the idea of writing functions (let alone writing them well) can take some time. However, in this chapter I will provide you with the basic knowledge of how functions operate in R to get you started on the right path. To do this, I cover the general [components of functions](#function_components), specifying function [arguments](#function_arguments), [scoping](#function_scoping) and [evaluation](#function_lazy) rules, [managing function outputs](#function_outputs), handling [invalid parameters](#function_invalid), and [saving & sourcing functions](#function_saving) for reuse. This will provide you the with the required knowledge to start building your own functions. Lastly, I offer some [additional resources](#functions_add_resource) that will help you learn more about functions in R.

## Function Components

With the exception of [primitive functions](https://cran.r-project.org/doc/manuals/r-release/R-ints.html#g_t_002eInternal-vs-_002ePrimitive) all R functions have three parts:

* body(): the code inside the function
* formals(): the list of arguments used to call the function
* environment(): the mapping of the location(s) of the function's variables

For example, let's build a function that calculates the present value (PV) of a single future sum. The equation for a single sum PV is:

where FV is future value, r is the interest rate, and n is the number of periods. In the function that follows the body of the function includes the equation

and then rounding the output to two decimals. The formals (or arguments) required for the function include FV, r, and n. And the environment shows that function operates in the global environment.

PV <- function(FV, r, n) {  
 PV <- FV / (1 + r)^n  
 round(PV, 2)  
}  
  
body(PV)  
## {  
## PV <- FV / (1 + r)^n  
## round(PV, 2)  
## }  
  
formals(PV)  
## $FV  
##   
##   
## $r  
##   
##   
## $n  
  
environment(PV)  
## <environment: R\_GlobalEnv>

## Arguments

To perform the PV() function we can call the arguments in different ways.

# using argument names  
PV(FV = 1000, r = .08, n = 5)  
## [1] 680.58  
  
# same as above but without using names (aka "positional matching")  
PV(1000, .08, 5)  
## [1] 680.58  
  
# if using names you can change the order  
PV(r = .08, FV = 1000, n = 5)  
## [1] 680.58  
  
# if not using names you must insert arguments in proper order  
# in this e.g. the function assumes FV = .08, r = 1000, and n = 5  
PV(.08, 1000, 5)  
## [1] 0

Note that when building a function you can also set default values for arguments. In our original PV() we did not provide any default values so if we do not supply all the argument parameters an error will be returned. However, if we set default values then the function will use the stated default if any parameters are missing:

# missing the n argument  
PV(1000, .08)  
## Error in PV(1000, 0.08): argument "n" is missing, with no default  
  
# creating default argument values  
PV <- function(FV = 1000, r = .08, n = 5) {  
 PV <- FV / (1 + r)^n  
 round(PV, 2)  
}  
  
# function will use default n value  
PV(1000, .08)  
## [1] 680.58  
  
# specifying a different n value  
PV(1000, .08, 3)  
## [1] 793.83

## Scoping Rules

Scoping refers to the set of rules a programming language uses to lookup the value to variables and/or symbols. The following illustrates the basic concept behind the lexical scoping rules that R follows.

A function will first look inside the function to identify all the variables being called. If all variables exist then their is no additional search required to identify variables.

PV1 <- function() {  
 FV <- 1000   
 r <- .08  
 n <- 5  
 FV / (1 + r)^n  
}  
  
PV1()  
## [1] 680.5832

However, if a variable does not exist within the function, R will look one level up to see if the variable exists.

# the FV variable is outside the function environment  
FV <- 1000   
  
PV2 <- function() {  
 r <- .08  
 n <- 5  
 FV / (1 + r)^n  
}  
  
PV2()  
## [1] 680.5832

This same concept applies if you have functions embeded within functions:

FV <- 1000   
  
PV3 <- function() {  
 r <- .08  
 n <- 5  
 denominator <- function() {  
 (1 + r)^n  
 }  
 FV/denominator()  
}  
  
PV3()  
## [1] 680.5832

This also applies for functions in which some arguments are called but not all variables used in the body are identified as arguments:

# n is specified within the function  
PV4 <- function(FV, r) {  
 n <- 5  
 FV / (1 + r)^n  
}  
  
PV4(1000, .08)  
## [1] 680.5832  
  
# n is specified within the function and  
# r is specified outside the function  
r <- 0.08  
  
PV5 <- function(FV) {  
 n <- 5  
 FV / (1 + r)^n  
}  
  
PV5(1000)  
## [1] 680.5832

## Lazy Evaluation

R functions perform "lazy" evaluation in which arguments are only evaluated if required in the body of the function.

# the y argument is not used so not including it causes  
# no harm  
lazy <- function(x, y){  
 x \* 2  
}  
lazy(4)  
## [1] 8  
  
# however, if both arguments are required in the body  
# an error will result if an argument is missing  
lazy2 <- function(x, y){  
 (x + y) \* 2  
}  
lazy2(4)  
## Error in lazy2(4): argument "y" is missing, with no default

## Returning Multiple Outputs from a Function

If a function performs multiple tasks and therefore has multiple results to report then we have to include the c() function inside the function to display all the results. If you do not include the c() function then the function output will only return the last expression:

bad <- function(x, y) {  
 2 \* x + y  
 x + 2 \* y  
 2 \* x + 2 \* y  
 x / y  
}  
bad(1, 2)  
## [1] 0.5  
  
good <- function(x, y) {  
 output1 <- 2 \* x + y  
 output2 <- x + 2 \* y  
 output3 <- 2 \* x + 2 \* y  
 output4 <- x / y  
 c(output1, output2, output3, output4)  
}  
good(1, 2)  
## [1] 4.0 5.0 6.0 0.5

Furthermore, when we have a function which performs multiple tasks (i.e. computes multiple computations) then it is often useful to save the results in a list.

good\_list <- function(x, y) {  
 output1 <- 2 \* x + y  
 output2 <- x + 2 \* y  
 output3 <- 2 \* x + 2 \* y  
 output4 <- x / y  
 c(list(Output1 = output1, Output2 = output2,   
 Output3 = output3, Output4 = output4))  
}  
good\_list(1, 2)  
## $Output1  
## [1] 4  
##   
## $Output2  
## [1] 5  
##   
## $Output3  
## [1] 6  
##   
## $Output4  
## [1] 0.5

## Dealing with Invalid Parameters

For functions that will be used again, and especially for those used by someone other than the creator of the function, it is good to check the validity of arguments within the function. One way to do this is to use the stop() function. The following uses an if() statement to check if the class of each argument is numeric. If one or more arguments are not numeric then the stop() function will be triggered to provide a meaningful message to the user.

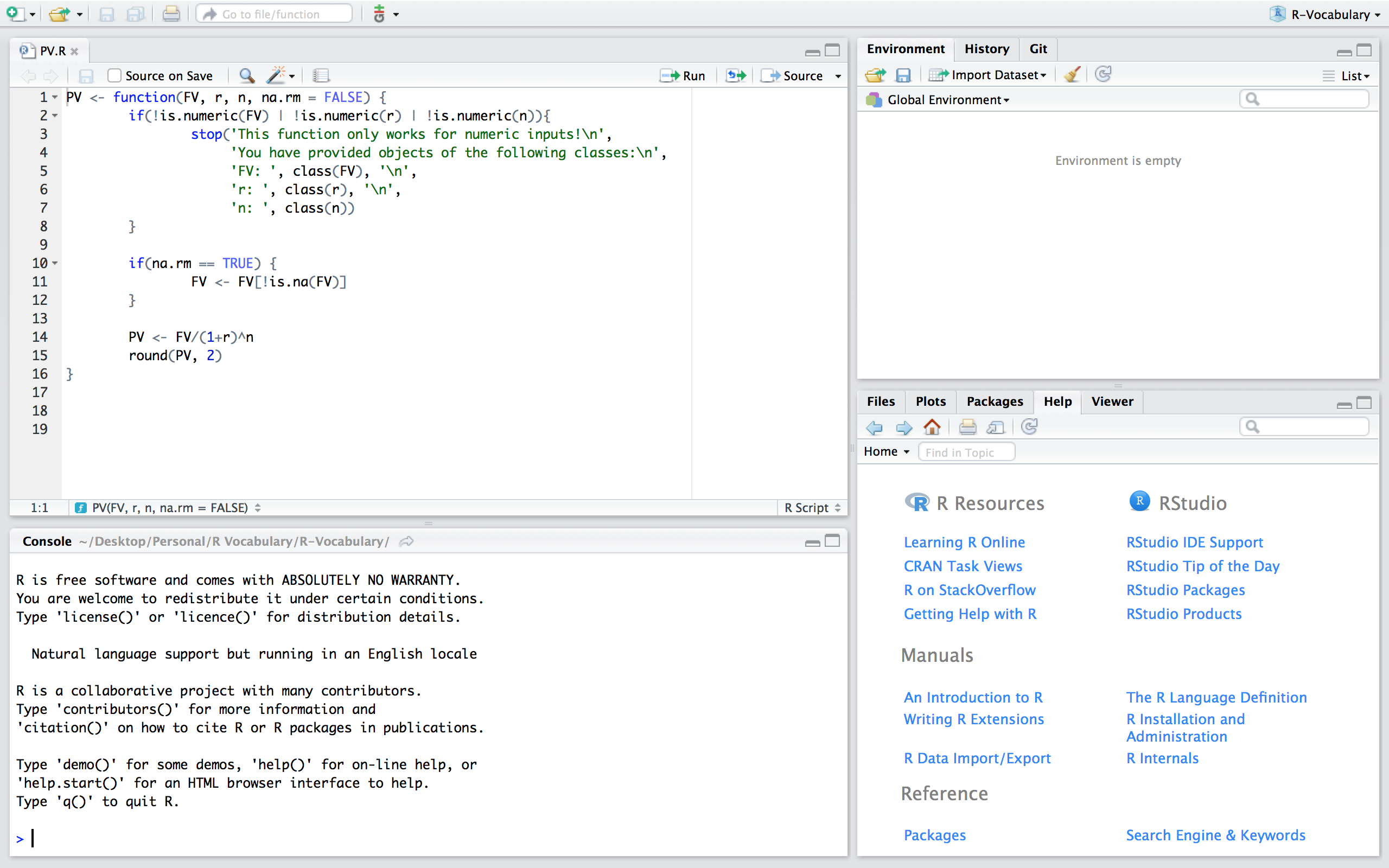
PV <- function(FV, r, n) {  
 if(!is.numeric(FV) | !is.numeric(r) | !is.numeric(n)){  
 stop('This function only works for numeric inputs!\n',   
 'You have provided objects of the following classes:\n',   
 'FV: ', class(FV), '\n',  
 'r: ', class(r), '\n',  
 'n: ', class(n))  
 }  
   
 PV <- FV / (1 + r)^n  
 round(PV, 2)  
}  
  
PV("1000", 0.08, "5")  
## Error in PV("1000", 0.08, "5"): This function only works for numeric inputs!  
## You have provided objects of the following classes:  
## FV: character  
## r: numeric  
## n: character

Another concern is dealing with missing or NA values. Lets say you wanted to perform the PV() function on a vector of potential future values. The function as is will output NA in place of any missing values in the FV input vector. If you want to remove the missing values then you can incorporate the na.rm parameter in the function arguments along with an if statement to remove missing values if na.rm = TRUE.

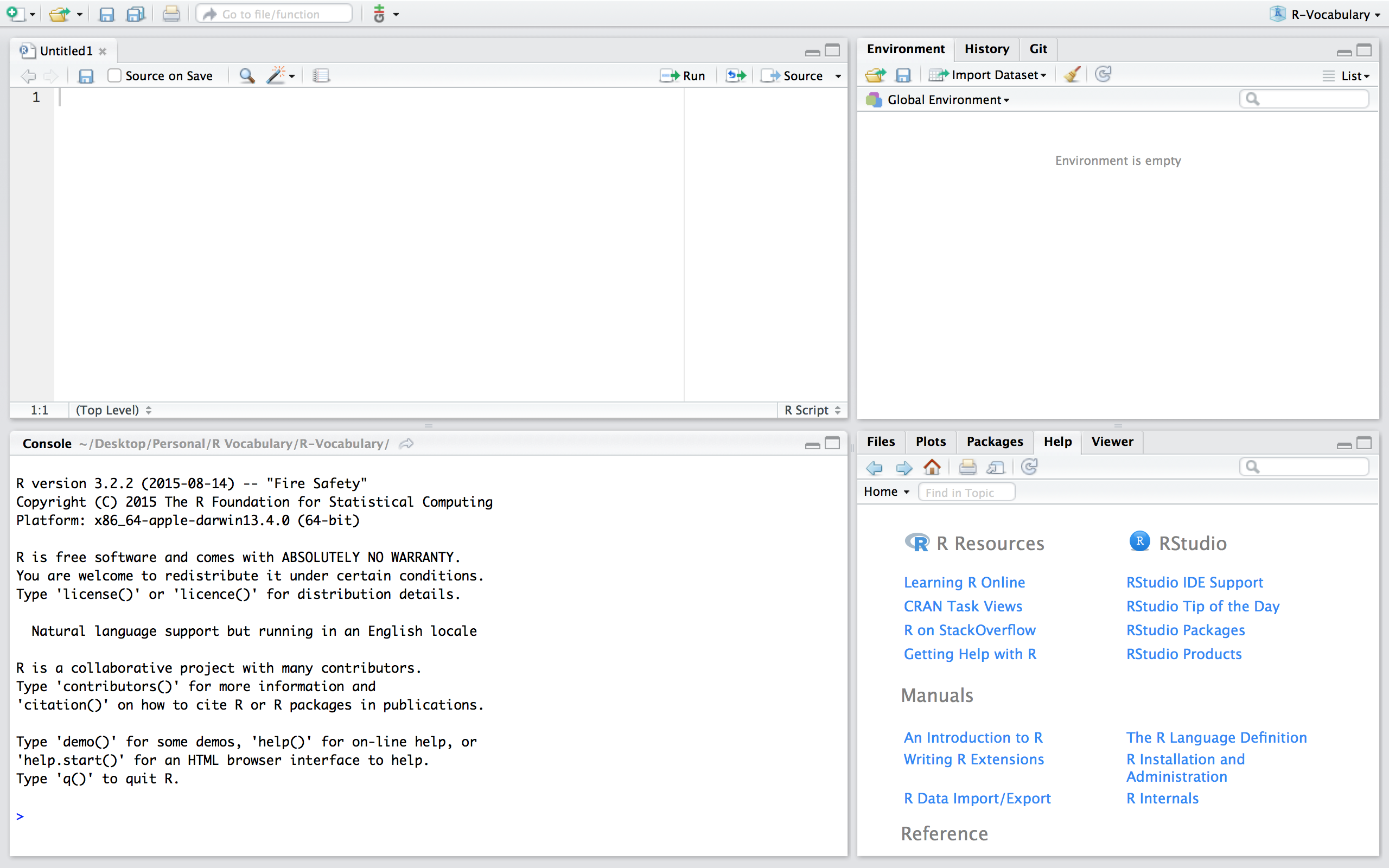
# vector of future value inputs  
fv <- c(800, 900, NA, 1100, NA)  
  
# original PV() function will return NAs  
PV(fv, .08, 5)  
## [1] 544.47 612.52 NA 748.64 NA  
  
# add na.rm argument  
PV <- function(FV, r, n, na.rm = FALSE) {  
 if(!is.numeric(FV) | !is.numeric(r) | !is.numeric(n)){  
 stop('This function only works for numeric inputs!\n',   
 'You have provided objects of the following classes:\n',   
 'FV: ', class(FV), '\n',  
 'r: ', class(r), '\n',  
 'n: ', class(n))  
 }  
   
 if(na.rm == TRUE) {  
 FV <- FV[!is.na(FV)]  
 }  
   
 PV <- FV / (1 + r)^n  
 round(PV, 2)  
}  
  
# setting na.rm = TRUE argument eliminates NA outputs  
PV(fv, 0.08, 5, na.rm = TRUE)  
## [1] 544.47 612.52 748.64

## Saving and Sourcing Functions

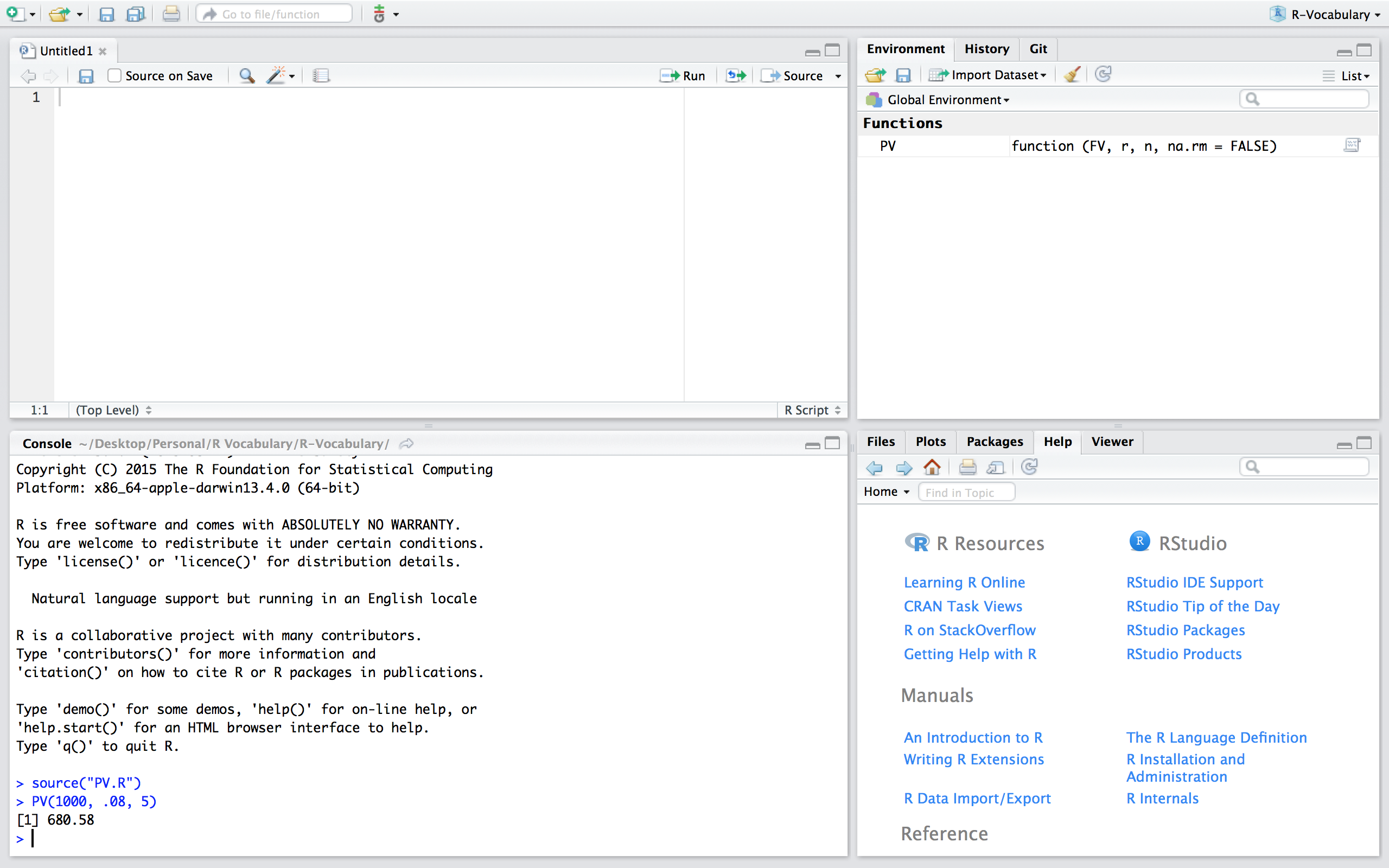
If you want to save a function to be used at other times and within other scripts there are two main ways to do this. One way is to build a package which I do not cover in this book but is discussed in more details [here](http://r-pkgs.had.co.nz/). Another option, and the one discussed here, is to save the function in a script. For example, we can save a script that contains the PV() function and save this script as PV.R.



Now, if we are working in a fresh script you'll see that we have no objects and functions in our working environment:



If we want to use the PV function in this new script we can simply read in the function by sourcing the script using source("PV.R"). Now, you'll notice that we have the PV() function in our global environment and can use it as normal. Note that if you are working in a different directory then where the PV.R file is located you'll need to include the proper command to access the relevant directory.



## Additional Resources

Functions are a fundamental building block of R and writing functions is a core activity of an R programmer. It represents the key step of the transition from a mere "user" to a developer who creates new functionality for R. As a result, its important to turn your existing, informal knowledge of functions into a rigorous understanding of what functions are and how they work. A few additional resources that can help you get to the next step of understanding functions include:

* [Hadley Wickham's Advanced R book](http://adv-r.had.co.nz/Functions.html)
* [Roger Peng's R Programming for Data Science book](https://leanpub.com/rprogramming)
* [DataCamp's Intermediate R course](https://www.datacamp.com/courses/intermediate-r?utm_source=functions_r_tutorial_post&utm_medium=blog&utm_campaign=functions_r_tutorial_post)
* [Coursera's R Programming course](https://www.coursera.org/course/rprog)

# Loop Control Statements

Looping is similiar to creating functions in that they are merely a means to automate a certain multi-step process by organizing sequences of R expressions. R consists of several loop control statements which allow you to perform repetititve code processes with different intentions and allow these automated expressions to naturally respond to features of your data. Consequently, learning these loop control statements will go a long ways in reducing code redundancy and becoming a more efficient data wrangler.

This chapter starts by covering the [basic control statements](#loop_functions) in R, which includes if, else, along with the for, while, and repeat loop control structures. In addition, I cover break and next which allow you to further control flow within the aforementioned control statements. Next I cover a set of vectorized functions known as the [apply family](#apply_family) of functions which minimize your need to explicitly create loops. I then provide some [additional "loop-like" functions](#other_loops) that are helpful in everyday data analysis followed by a list of [additional resources](#functions_add_resource) to learn more about control structures in R.

## Basic control statements (i.e. if, for, while, etc.)

### if Statement

The conditional if statement is used to test an expression. If the test\_expression is TRUE, the statement gets executed. But if it's FALSE, nothing happens.

# syntax of if statement  
if (test\_expression) {  
 statement  
}

The following is an example that tests if any values in a vector are negative. Notice there are two ways to write this if statement; since the body of the statement is only one line you can write it with or without curly braces. I recommend getting in the habit of using curly braces, that way if you build onto if statements with additional functions in the body or add an else statement later you will not run into issues with unexpected code procedures.

x <- c(8, 3, -2, 5)  
  
# without curly braces  
if(any(x < 0)) print("x contains negative numbers")  
## [1] "x contains negative numbers"  
  
# with curly braces produces same result  
if(any(x < 0)){  
 print("x contains negative numbers")  
}  
## [1] "x contains negative numbers"  
  
# an if statement in which the test expression is FALSE  
# does not produce any output  
y <- c(8, 3, 2, 5)  
  
if(any(y < 0)){  
 print("y contains negative numbers")  
}

### if...else Statement

The conditional if...else statement is used to test an expression similar to the if statement. However, rather than nothing happening if the test\_expression is FALSE, the else part of the function will be evaluated.

# syntax of if...else statement  
if (test\_expression) {  
 statement 1  
} else {  
 statement 2  
}

The following extends the previous example illustrated for the if statement in which the if statement tests if any values in a vector are negative; if TRUE it produces one output and if FALSE it produces the else output.

# this test results in statement 1 being executed  
x <- c(8, 3, -2, 5)  
  
if(any(x < 0)){  
 print("x contains negative numbers")  
} else{  
 print("x contains all positive numbers")  
}  
## [1] "x contains negative numbers"  
  
# this test results in statement 2 (or the else statement) being executed  
y <- c(8, 3, 2, 5)  
  
if(any(y < 0)){  
 print("y contains negative numbers")  
} else{  
 print("y contains all positive numbers")  
}  
## [1] "y contains all positive numbers"

Simple if...else statements, as above, in which only one line of code is being executed in the statements can be written in a simplified alternative manner. These alternatives are only recommended for very short if...else code:

x <- c(8, 3, 2, 5)  
  
# alternative 1  
if(any(x < 0)) print("x contains negative numbers") else print("x contains all positive numbers")  
## [1] "x contains all positive numbers"  
  
# alternative 2 using the ifelse function  
ifelse(any(x < 0), "x contains negative numbers", "x contains all positive numbers")  
## [1] "x contains all positive numbers"

We can also nest as many if...else statements as required (or desired). For example:

# this test results in statement 1 being executed  
x <- 7  
  
if(x >= 10){  
 print("x exceeds acceptable tolerance levels")  
} else if(x >= 0 & x < 10){  
 print("x is within acceptable tolerance levels")  
} else {  
 print("x is negative")  
}  
## [1] "x is within acceptable tolerance levels"

### for Loop

The for loop is used to execute repetitive code statements for a particular number of times. The general syntax is provided below where i is the counter and as i assumes each sequential value defined (1 through 100 in this example) the code in the body will be performed for that ith value.

{linenos=off}

# syntax of for loop  
for(i in 1:100) {  
 <do stuff here with i>  
}

An important lesson to learn is that R is not efficient at *growing* data objects. As a result, it is more efficient to create an empty data object and *fill* it with the for loop outputs. For example, if you want to create a vector in which 5 values are randomly drawn from a poisson distribution with mean 5, it is less efficient to perform the first example in the following code chunk than to perform the second example. Although this inefficiency is not noticed in this small example, when you perform larger repetitions it will become noticable so you might as well get in the habit of *filling* rather than *growing*.

# not advised  
for(i in 5){  
 x <- rpois(5, lambda = 5)  
 print(x)  
}  
## [1] 11 5 8 8 7  
  
# advised  
x <- vector(mode = "numeric", length = 5)  
  
for(i in 5){  
 x <- rpois(5, lambda = 5)  
 print(x)  
}  
## [1] 5 8 9 5 4

Another example in which we create an empty matrix with 5 rows and 5 columns. The for loop then iterates over each column (note how *i* takes on the values 1 through the number of columns in the my.mat matrix) and takes a random draw of 5 values from a poisson distribution with mean *i* in column *i*:

my.mat <- matrix(NA, nrow = 5, ncol = 5)  
  
for(i in 1:ncol(my.mat)){  
 my.mat[, i] <- rpois(5, lambda = i)  
}  
my.mat  
## [,1] [,2] [,3] [,4] [,5]  
## [1,] 0 2 1 7 1  
## [2,] 1 2 2 3 9  
## [3,] 2 1 5 6 6  
## [4,] 2 1 5 2 10  
## [5,] 0 2 2 2 4

### while Loop

While loops begin by testing a condition. If it is true, then they execute the statement. Once the statement is executed, the condition is tested again, and so forth, until the condition is false, after which the loop exits. It's considered a best practice to include a counter object to keep track of total iterations

# syntax of while loop  
counter <- 1  
  
while(test\_expression) {  
 statement  
 counter <- counter + 1  
}

while loops can potentially result in infinite loops if not written properly; therefore, you must use them with care. To provide a simple example to illustrate how similiar for and while loops are:

counter <- 1  
  
while(counter <= 10) {  
 print(counter)  
 counter <- counter + 1  
}  
  
# this for loop provides the same output  
counter <- vector(mode = "numeric", length = 10)  
  
for(i in 1:length(counter)) {  
 print(i)  
}

The primary difference between a for loop and a while loop is: a for loop is used when the number of iterations a code should be run is known where a while loop is used when the number of iterations is not known. For instance, the following takes value x and adds or subtracts 1 from the value randomly until x exceeds the values in the test expression. The output illustrates that the code runs 14 times until x exceeded the threshold with the value 9.

counter <- 1  
x <- 5  
set.seed(3)  
  
while(x >= 3 && x <= 8 ) {  
 coin <- rbinom(1, 1, 0.5)  
   
 if(coin == 1) { ## random walk  
 x <- x + 1  
 } else {  
 x <- x - 1  
 }  
 cat("On iteration", counter, ", x =", x, '\n')  
 counter <- counter + 1  
}  
## On iteration 1 , x = 4   
## On iteration 2 , x = 5   
## On iteration 3 , x = 4   
## On iteration 4 , x = 3   
## On iteration 5 , x = 4   
## On iteration 6 , x = 5   
## On iteration 7 , x = 4   
## On iteration 8 , x = 3   
## On iteration 9 , x = 4   
## On iteration 10 , x = 5   
## On iteration 11 , x = 6   
## On iteration 12 , x = 7   
## On iteration 13 , x = 8   
## On iteration 14 , x = 9

### repeat Loop

A repeat loop is used to iterate over a block of code multiple number of times. There is test expression in a repeat loop to end or exit the loop. Rather, we must put a condition statement explicitly inside the body of the loop and use the break function to exit the loop. Failing to do so will result into an infinite loop.

# syntax of repeat loop  
counter <- 1  
  
repeat {  
 statement  
   
 if(test\_expression){  
 break  
 }  
 counter <- counter + 1  
}

For example ,say we want to randomly draw values from a uniform distribution between 1 and 25. Furthermore, we want to continue to draw values randomly until our sample contains at least each integer value between 1 and 25; however, we do not care if we've drawn a particular value multiple times. The following code repeats the random draws of values between 1 and 25 (in which we round). We then include an if statement to check if all values between 1 and 25 are present in our sample. If so, we use the [break](#break) statement to exit the loop. If not, we add to our counter and let the loop repeat until the conditional if statement is found to be true. We can then check the counter object to assess how many iterations were required to reach our conditional requirement.

counter <- 1  
x <- NULL  
  
repeat {  
 x <- c(x, round(runif(1, min = 1, max = 25)))  
   
 if(all(1:25 %in% x)){  
 break  
 }  
   
 counter <- counter + 1  
}  
  
counter  
## [1] 75

### break Function to Exit a Loop

The break function is used to exit a loop immediately, regardless of what iteration the loop may be on. break functions are typically embedded in an if statement in which a condition is assessed, if TRUE break out of the loop, if FALSE continue on with the loop. In a nested looping situation, where there is a loop inside another loop, this statement exits from the innermost loop that is being evaluated.

x <- 1:5  
  
for (i in x) {  
 if (i == 3){  
 break  
 }  
 print(i)  
}  
## [1] 1  
## [1] 2

### next Function to Skip an Iteration in a Loop

The next statement is useful when we want to skip the current iteration of a loop without terminating it. On encountering next, the R parser skips further evaluation and starts next iteration of the loop.

x <- 1:5  
  
for (i in x) {  
 if (i == 3){  
 next  
 }  
 print(i)  
}  
## [1] 1  
## [1] 2  
## [1] 4  
## [1] 5

## Apply family

The apply family consists of vectorized functions which minimize your need to explicitly create loops. These functions will apply a specified function to a data object and there primary difference is in the object class in which the function is applied to (list vs. matrix, etc) and the object class that will be returned from the function. The following presents the most common forms of apply functions that I use for data analysis but realize that additional functions exist (mapply, rapply, & vapply) which are not covered here.

### apply() for Matrices and Data Frames

The apply() function is most often used to apply a function to the rows or columns (margins) of matrices or data frames. However, it can be used with general arrays, for example, to take the average of an array of matrices. Using apply() is not faster than using a loop function, but it is highly compact and can be written in one line.

The syntax for apply() is as follows where

* x is the matrix, dataframe or array
* MARGIN is a vector giving the subscripts which the function will be applied over. E.g., for a matrix 1 indicates rows, 2 indicates columns, c(1, 2) indicates rows and columns.
* FUN is the function to be applied
* ... is for any other arguments to be passed to the function

# syntax of apply function  
apply(x, MARGIN, FUN, ...)

To provide examples let's use the mtcars data set provided in R:

# show first few rows of mtcars  
head(mtcars)  
## mpg cyl disp hp drat wt qsec vs am gear carb  
## Mazda RX4 21.0 6 160 110 3.90 2.620 16.46 0 1 4 4  
## Mazda RX4 Wag 21.0 6 160 110 3.90 2.875 17.02 0 1 4 4  
## Datsun 710 22.8 4 108 93 3.85 2.320 18.61 1 1 4 1  
## Hornet 4 Drive 21.4 6 258 110 3.08 3.215 19.44 1 0 3 1  
## Hornet Sportabout 18.7 8 360 175 3.15 3.440 17.02 0 0 3 2  
## Valiant 18.1 6 225 105 2.76 3.460 20.22 1 0 3 1  
  
# get the mean of each column   
apply(mtcars, 2, mean)  
## mpg cyl disp hp drat wt   
## 20.090625 6.187500 230.721875 146.687500 3.596563 3.217250   
## qsec vs am gear carb   
## 17.848750 0.437500 0.406250 3.687500 2.812500  
  
# get the sum of each row (not really relevant for this data  
# but it illustrates the capability)  
apply(mtcars, 1, sum)  
## Mazda RX4 Mazda RX4 Wag Datsun 710   
## 328.980 329.795 259.580   
## Hornet 4 Drive Hornet Sportabout Valiant   
## 426.135 590.310 385.540   
## Duster 360 Merc 240D Merc 230   
## 656.920 270.980 299.570   
## Merc 280 Merc 280C Merc 450SE   
## 350.460 349.660 510.740   
## Merc 450SL Merc 450SLC Cadillac Fleetwood   
## 511.500 509.850 728.560   
## Lincoln Continental Chrysler Imperial Fiat 128   
## 726.644 725.695 213.850   
## Honda Civic Toyota Corolla Toyota Corona   
## 195.165 206.955 273.775   
## Dodge Challenger AMC Javelin Camaro Z28   
## 519.650 506.085 646.280   
## Pontiac Firebird Fiat X1-9 Porsche 914-2   
## 631.175 208.215 272.570   
## Lotus Europa Ford Pantera L Ferrari Dino   
## 273.683 670.690 379.590   
## Maserati Bora Volvo 142E   
## 694.710 288.890  
  
# get column quantiles (notice the quantile percents as row names)  
apply(mtcars, 2, quantile, probs = c(0.10, 0.25, 0.50, 0.75, 0.90))  
## mpg cyl disp hp drat wt qsec vs am gear carb  
## 10% 14.340 4 80.610 66.0 3.007 1.95550 15.5340 0 0 3 1  
## 25% 15.425 4 120.825 96.5 3.080 2.58125 16.8925 0 0 3 2  
## 50% 19.200 6 196.300 123.0 3.695 3.32500 17.7100 0 0 4 2  
## 75% 22.800 8 326.000 180.0 3.920 3.61000 18.9000 1 1 4 4  
## 90% 30.090 8 396.000 243.5 4.209 4.04750 19.9900 1 1 5 4

### lapply() for Lists...Output as a List

The lapply() function does the following simple series of operations:

1. it loops over a list, iterating over each element in that list
2. it applies a function to each element of the list (a function that you specify)
3. and returns a list (the l is for "list").

The syntax for lapply() is as follows where

* x is the list
* FUN is the function to be applied
* ... is for any other arguments to be passed to the function

# syntax of lapply function  
lapply(x, FUN, ...)

To provide examples we'll generate a list of four items:

data <- list(item1 = 1:4, item2 = rnorm(10),   
 item3 = rnorm(20, 1), item4 = rnorm(100, 5))  
  
# get the mean of each list item   
lapply(data, mean)  
## $item1  
## [1] 2.5  
##   
## $item2  
## [1] 0.5529324  
##   
## $item3  
## [1] 1.193884  
##   
## $item4  
## [1] 5.013019

The above provides a simple example where each list item is simply a vector of numeric values. However, consider the case where you have a list that contains data frames and you would like to loop through each list item and perform a function to the data frame. In this case we can embed an apply function within an lapply function.

For example, the following creates a list for R's built in beaver data sets. The lapply function loops through each of the two list items and uses apply to calculate the mean of the columns in both list items. Note that I wrap the apply function with round to provide an easier to read output.

# list of R's built in beaver data  
beaver\_data <- list(beaver1 = beaver1, beaver2 = beaver2)  
  
# get the mean of each list item   
lapply(beaver\_data, function(x) round(apply(x, 2, mean), 2))  
## $beaver1  
## day time temp activ   
## 346.20 1312.02 36.86 0.05   
##   
## $beaver2  
## day time temp activ   
## 307.13 1446.20 37.60 0.62

### sapply() for Lists...Output Simplified

The sapply() function behaves similarly to lapply(); the only real difference is in the return value. sapply() will try to simplify the result of lapply() if possible. Essentially, sapply() calls lapply() on its input and then applies the following algorithm:

* If the result is a list where every element is length 1, then a vector is returned
* If the result is a list where every element is a vector of the same length (> 1), a matrix is returned.
* If neither of the above simplifications can be performed then a list is returned

To illustrate the differences we can use the previous example using a list with the beaver data and compare the sapply and lapply outputs:

# list of R's built in beaver data  
beaver\_data <- list(beaver1 = beaver1, beaver2 = beaver2)  
  
# get the mean of each list item and return as a list  
lapply(beaver\_data, function(x) round(apply(x, 2, mean), 2))  
## $beaver1  
## day time temp activ   
## 346.20 1312.02 36.86 0.05   
##   
## $beaver2  
## day time temp activ   
## 307.13 1446.20 37.60 0.62  
  
# get the mean of each list item and simply the output  
sapply(beaver\_data, function(x) round(apply(x, 2, mean), 2))  
## beaver1 beaver2  
## day 346.20 307.13  
## time 1312.02 1446.20  
## temp 36.86 37.60  
## activ 0.05 0.62

### tapply() for Vectors

tapply() is used to apply a function over subsets of a vector. It is primarily used when we have the following circumstances:

1. A dataset that can be broken up into groups (via categorical variables - aka factors)
2. We desire to break the dataset up into groups
3. Within each group, we want to apply a function

The arguments to tapply() are as follows:

* x is a vector
* INDEX is a factor or a list of factors (or else they are coerced to factors)
* FUN is a function to be applied
* ... contains other arguments to be passed FUN
* simplify, should we simplify the result?

# syntax of tapply function  
tapply(x, INDEX, FUN, ..., simplify = TRUE)

To provide an example we'll use the built in mtcars dataset and calculate the mean of the mpg variable grouped by the cyl variable.

# show first few rows of mtcars  
head(mtcars)  
## mpg cyl disp hp drat wt qsec vs am gear carb  
## Mazda RX4 21.0 6 160 110 3.90 2.620 16.46 0 1 4 4  
## Mazda RX4 Wag 21.0 6 160 110 3.90 2.875 17.02 0 1 4 4  
## Datsun 710 22.8 4 108 93 3.85 2.320 18.61 1 1 4 1  
## Hornet 4 Drive 21.4 6 258 110 3.08 3.215 19.44 1 0 3 1  
## Hornet Sportabout 18.7 8 360 175 3.15 3.440 17.02 0 0 3 2  
## Valiant 18.1 6 225 105 2.76 3.460 20.22 1 0 3 1  
  
# get the mean of the mpg column grouped by cylinders   
tapply(mtcars$mpg, mtcars$cyl, mean)  
## 4 6 8   
## 26.66364 19.74286 15.10000

Now let's say you want to calculate the mean for *each* column in the mtcars dataset grouped by the cylinder categorical variable. To do this you can embed the tapply function within the apply function.

# get the mean of all columns grouped by cylinders   
apply(mtcars, 2, function(x) tapply(x, mtcars$cyl, mean))  
## mpg cyl disp hp drat wt qsec vs  
## 4 26.66364 4 105.1364 82.63636 4.070909 2.285727 19.13727 0.9090909  
## 6 19.74286 6 183.3143 122.28571 3.585714 3.117143 17.97714 0.5714286  
## 8 15.10000 8 353.1000 209.21429 3.229286 3.999214 16.77214 0.0000000  
## am gear carb  
## 4 0.7272727 4.090909 1.545455  
## 6 0.4285714 3.857143 3.428571  
## 8 0.1428571 3.285714 3.500000

Note that this type of summarization can also be done using the dplyr package with clearer syntax. This is covered in the *Transforming Your Data with dplyr* section.

## Other useful "loop-like" functions

In addition to the [apply family](#apply_family) which provide vectorized functions that minimize your need to explicitly create loops, there are also a few commonly applied apply functions that have been further simplified. These include the calculation of column and row sums, means, medians, standard deviations, variances, and summary quantiles across the entire data set.

The most common apply functions that have been include calculating the sums and means of columns and rows. For instance, to calculate the sum of columns across a data frame or matrix you could do the following:

apply(mtcars, 2, sum)  
## mpg cyl disp hp drat wt qsec vs   
## 642.900 198.000 7383.100 4694.000 115.090 102.952 571.160 14.000   
## am gear carb   
## 13.000 118.000 90.000

However, you can perform the same function with the shorter colSums() function and it performs faster:

colSums(mtcars)  
## mpg cyl disp hp drat wt qsec vs   
## 642.900 198.000 7383.100 4694.000 115.090 102.952 571.160 14.000   
## am gear carb   
## 13.000 118.000 90.000

To illustrate the speed difference we can compare the performance of using the apply() function versus the colSums() function on a matrix with 100 million values (10K x 10K). You can see that the speed of colSums() is significantly faster.

# develop a 10,000 x 10,000 matrix  
mat = matrix(sample(1:10, size=100000000, replace=TRUE), nrow=10000)  
  
system.time(apply(mat, 2, sum))  
## user system elapsed   
## 1.544 0.329 1.879  
  
system.time(colSums(mat))  
## user system elapsed   
## 0.126 0.000 0.127

Base R provides the following simplified apply functions:

* colSums (x, na.rm = FALSE)
* rowSums (x, na.rm = FALSE)
* colMeans(x, na.rm = FALSE)
* rowMeans(x, na.rm = FALSE)

In addition, the following functions are provided through the specified packages:

* [miscTools package](https://cran.r-project.org/web/packages/mixtools/index.html) (note that these functions will work on data frames)
  + colMedians()
  + rowMedians()
* [matrixStats package](https://cran.r-project.org/web/packages/matrixStats/index.html) (note that these functions only operate on matrices)
  + colMedians() & rowMedians()
  + colSds() & rowSds()
  + colVar() & rowVar()
  + colRanges() & rowRanges()
  + colQuantiles() & rowQuantiles()
  + along with several additional summary statistic functions

In addition, the summary() function will provide relevant summary statistics over each column of data frames and matrices. Note in the the example that follows that for the first four columns of the iris data set the summary statistics include min, med, mean, max, and 1st & 3rd quantiles. Whereas the last column (Species) only provides the total count since this is a factor variable.

summary(iris)  
## Sepal.Length Sepal.Width Petal.Length Petal.Width   
## Min. :4.300 Min. :2.000 Min. :1.000 Min. :0.100   
## 1st Qu.:5.100 1st Qu.:2.800 1st Qu.:1.600 1st Qu.:0.300   
## Median :5.800 Median :3.000 Median :4.350 Median :1.300   
## Mean :5.843 Mean :3.057 Mean :3.758 Mean :1.199   
## 3rd Qu.:6.400 3rd Qu.:3.300 3rd Qu.:5.100 3rd Qu.:1.800   
## Max. :7.900 Max. :4.400 Max. :6.900 Max. :2.500   
## Species   
## setosa :50   
## versicolor:50   
## virginica :50   
##   
##   
##

## Additional Resources

This provides an introduction to control statements in R. However, the following provides additional resources to learn more:

* [Tutorial on loops by DataCamp](https://www.datacamp.com/community/tutorials/tutorial-on-loops-in-r)
* Roger Peng's [R Programming for Data Science](https://leanpub.com/rprogramming)
* Hadley Wickham's [Advanced R](http://adv-r.had.co.nz/)

# Simplify Your Code with %>%

Removing duplication is an important principle to keep in mind with your code; however, equally important is to keep your code efficient and readable. Efficiency is often accomplished by leveraging functions and control statements in your code. However, efficiency also includes eliminating the creation and saving of unnecessary objects that often result when you are trying to make your code more readable, clear, and explicit. Consequently, writing code that is simple, readable, *and* efficient is often considered contradictory. For this reason, the magrittr package is a powerful tool to have in your data wrangling toolkit.

The [magrittr](https://cran.r-project.org/web/packages/magrittr/index.html) package was created by [Stefan Milton Bache](https://twitter.com/stefanbache) and, in Stefan's words, has two primary aims: "to decrease development time and to improve readability and maintainability of code." Hence, it aims to increase efficiency and improve readability; and in the process it greatly simplifies your code. The following covers the basics of the magrittr toolkit.

## Pipe (%>%) Operator

The principal function provided by the magrittr package is %>%, or what's called the "pipe" operator. This operator will forward a value, or the result of an expression, into the next function call/expression. For instance a function to filter data can be written as:

C> filter(data, variable == numeric\_value)

C> or

C> data %>% filter(variable == numeric\_value)

Both functions complete the same task and the benefit of using %>% may not be immediately evident; however, when you desire to perform multiple functions its advantage becomes obvious. For instance, if we want to filter some data, group it by categories, summarize it, and then order the summarized results we could write it out three different ways. Don't worry, you'll learn how to operate these specific functions in the next section.

\_\_\_\_Nested Option\_\_\_\_:

library(magrittr)  
library(dplyr)  
  
arrange(  
 summarize(  
 group\_by(  
 filter(mtcars, carb > 1),  
 cyl  
 ),  
 Avg\_mpg = mean(mpg)  
 ),  
 desc(Avg\_mpg)  
 )  
## Source: local data frame [3 x 2]  
##   
## cyl Avg\_mpg  
## (dbl) (dbl)  
## 1 4 25.90  
## 2 6 19.74  
## 3 8 15.10

This first option is considered a "nested" option such that functions are nested within one another. Historically, this has been the traditional way of integrating code; however, it becomes extremely difficult to read what exactly the code is doing and it also becomes easier to make mistakes when making updates to your code. Although not in violation of the DRY principle, it definitely violates the basic principle of readability and clarity, which makes communication of your analysis more difficult. To make things more readable, people often move to the following approach...

\_\_\_\_Multiple Object Option\_\_\_\_:

a <- filter(mtcars, carb > 1)  
b <- group\_by(a, cyl)  
c <- summarise(b, Avg\_mpg = mean(mpg))  
d <- arrange(c, desc(Avg\_mpg))  
print(d)  
## Source: local data frame [3 x 2]  
##   
## cyl Avg\_mpg  
## (dbl) (dbl)  
## 1 4 25.90  
## 2 6 19.74  
## 3 8 15.10

This second option helps in making the data wrangling steps more explicit and obvious but definitely violates the DRY principle. By sequencing multiple functions in this way you are likely saving multiple outputs that are not very informative to you or others; rather, the only reason you save them is to insert them into the next function to eventually get the final output you desire. This inevitably creates unnecessary copies and wrecks havoc on properly managing your objects...basically it results in a global environment charlie foxtrot! To provide the same readability (or even better), we can use %>% to string these arguments together without unnecessary object creation...

\_\_\_\_%>% Option\_\_\_\_:

mtcars %>%  
 filter(carb > 1) %>%  
 group\_by(cyl) %>%  
 summarise(Avg\_mpg = mean(mpg)) %>%  
 arrange(desc(Avg\_mpg))  
## Source: local data frame [3 x 2]  
##   
## cyl Avg\_mpg  
## (dbl) (dbl)  
## 1 4 25.90  
## 2 6 19.74  
## 3 8 15.10

This final option which integrates %>% operators makes for more efficient *and* legible code. Its efficient in that it doesn't save unncessary objects (as in option 2) and performs as effectively (as both option 1 & 2) but makes your code more readable in the process. Its legible in that you can read this as you would read normal prose (we read the %>% as *"and then"*)- "take mtcars *and then* filter *and then* group by *and then* summarize *and then* arrange."

And since R is a functional programming language, meaning that everything you do is basically built on functions, you can use the pipe operator to feed into just about any argument call. For example, we can pipe into a linear regression function and then get the summary of the regression parameters. Note in this case I insert "data = ." into the lm() function. When using the %>% operator the default is the argument that you are forwarding will go in as the **first** argument of the function that follows the %>%. However, in some functions the argument you are forwarding does not go into the default first position. In these cases, you place "." to signal which argument you want the forwarded expression to go to.

mtcars %>%  
 filter(carb > 1) %>%  
 lm(mpg ~ cyl + hp, data = .) %>%  
 summary()  
##   
## Call:  
## lm(formula = mpg ~ cyl + hp, data = .)  
##   
## Residuals:  
## Min 1Q Median 3Q Max   
## -4.6163 -1.4162 -0.1506 1.6181 5.2021   
##   
## Coefficients:  
## Estimate Std. Error t value Pr(>|t|)   
## (Intercept) 35.67647 2.28382 15.621 2.16e-13 \*\*\*  
## cyl -2.22014 0.52619 -4.219 0.000353 \*\*\*  
## hp -0.01414 0.01323 -1.069 0.296633   
## ---  
## Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1  
##   
## Residual standard error: 2.689 on 22 degrees of freedom  
## Multiple R-squared: 0.7601, Adjusted R-squared: 0.7383   
## F-statistic: 34.85 on 2 and 22 DF, p-value: 1.516e-07

You can also use %>% to feed into plots:

library(ggplot2)  
  
mtcars %>%  
 filter(carb > 1) %>%  
 qplot(x = wt, y = mpg, data = .)

{width=60%} Piping into a Plot

You will also find that the %>% operator is now being built into packages to make programming much easier. For instance, in the [section that follows](#shape_transform) where I illustrate how to reshape and transform your data with the dplyr and tidyr packages, you will see that the %>% operator is already built into these packages. It is also built into the ggvis and dygraphs packages (visualization packages), the httr package (which we covered in the [data scraping chapter](#scrape)), and a growing number of newer packages.

## Additional Functions

In addition to the %>% operator, magrittr provides several additional functions which make operations such as addition, multiplication, logical operators, re-naming, etc. more pleasant when composing chains using the %>% operator. Some examples follow but you can see the current list of the available aliased functions by typing ?magrittr::add in your console.

# subset with extract  
mtcars %>%  
 extract(, 1:4) %>%  
 head  
## mpg cyl disp hp  
## Mazda RX4 21.0 6 160 110  
## Mazda RX4 Wag 21.0 6 160 110  
## Datsun 710 22.8 4 108 93  
## Hornet 4 Drive 21.4 6 258 110  
## Hornet Sportabout 18.7 8 360 175  
## Valiant 18.1 6 225 105  
  
# add, subtract, multiply, divide and other operations are available  
mtcars %>%   
 extract(, "mpg") %>%   
 multiply\_by(5)  
## [1] 105.0 105.0 114.0 107.0 93.5 90.5 71.5 122.0 114.0 96.0 89.0  
## [12] 82.0 86.5 76.0 52.0 52.0 73.5 162.0 152.0 169.5 107.5 77.5  
## [23] 76.0 66.5 96.0 136.5 130.0 152.0 79.0 98.5 75.0 107.0  
  
# logical assessments and filters are available  
mtcars %>%   
 extract(, "cyl") %>%   
 equals(4)  
## [1] FALSE FALSE TRUE FALSE FALSE FALSE FALSE TRUE TRUE FALSE FALSE  
## [12] FALSE FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE FALSE  
## [23] FALSE FALSE FALSE TRUE TRUE TRUE FALSE FALSE FALSE TRUE  
  
# renaming columns and rows is available  
mtcars %>%  
 head %>%  
 set\_colnames(paste("Col", 1:11, sep = ""))  
## Col1 Col2 Col3 Col4 Col5 Col6 Col7 Col8 Col9 Col10  
## Mazda RX4 21.0 6 160 110 3.90 2.620 16.46 0 1 4  
## Mazda RX4 Wag 21.0 6 160 110 3.90 2.875 17.02 0 1 4  
## Datsun 710 22.8 4 108 93 3.85 2.320 18.61 1 1 4  
## Hornet 4 Drive 21.4 6 258 110 3.08 3.215 19.44 1 0 3  
## Hornet Sportabout 18.7 8 360 175 3.15 3.440 17.02 0 0 3  
## Valiant 18.1 6 225 105 2.76 3.460 20.22 1 0 3  
## Col11  
## Mazda RX4 4  
## Mazda RX4 Wag 4  
## Datsun 710 1  
## Hornet 4 Drive 1  
## Hornet Sportabout 2  
## Valiant 1

## Additional Pipe Operators

magrittr also offers some alternative pipe operators. Some functions, such as plotting functions, will cause the string of piped arguments to terminate. The tee (%T>%) operator allows you to continue piping functions that normally cause termination.

# normal piping terminates with the plot() function resulting in  
# NULL results for the summary() function  
mtcars %>%  
 filter(carb > 1) %>%  
 extract(, 1:4) %>%  
 plot() %>%  
 summary()

{width=60%} Regular Pipe Operator Terminates String of Functions at a Plot

## Length Class Mode   
## 0 NULL NULL

# inserting %T>% allows you to plot and perform the functions that   
# follow the plotting function  
mtcars %>%  
 filter(carb > 1) %>%  
 extract(, 1:4) %T>%  
 plot() %>%  
 summary()

{width=60%} Tee Operator Allows You to Pipe Through a Plot

## mpg cyl disp hp   
## Min. :10.40 Min. :4.00 Min. : 75.7 Min. : 52.0   
## 1st Qu.:15.20 1st Qu.:6.00 1st Qu.:146.7 1st Qu.:110.0   
## Median :17.80 Median :8.00 Median :275.8 Median :175.0   
## Mean :18.62 Mean :6.64 Mean :257.7 Mean :163.7   
## 3rd Qu.:21.00 3rd Qu.:8.00 3rd Qu.:351.0 3rd Qu.:205.0   
## Max. :30.40 Max. :8.00 Max. :472.0 Max. :335.0

The compound assignment %<>% operator is used to update a value by first piping it into one or more expressions, and then assigning the result. For instance, let's say you want to transform the mpg variable in the mtcars data frame to a square root measurement. Using %<>% will perform the functions to the right of %<>% and save the changes these functions perform to the variable or data frame called to the left of %<>%.

# note that mpg is in its typical measurement  
head(mtcars)  
## mpg cyl disp hp drat wt qsec vs am gear carb  
## Mazda RX4 21.0 6 160 110 3.90 2.620 16.46 0 1 4 4  
## Mazda RX4 Wag 21.0 6 160 110 3.90 2.875 17.02 0 1 4 4  
## Datsun 710 22.8 4 108 93 3.85 2.320 18.61 1 1 4 1  
## Hornet 4 Drive 21.4 6 258 110 3.08 3.215 19.44 1 0 3 1  
## Hornet Sportabout 18.7 8 360 175 3.15 3.440 17.02 0 0 3 2  
## Valiant 18.1 6 225 105 2.76 3.460 20.22 1 0 3 1  
  
# we can square root mpg and save this change using %<>%  
mtcars$mpg %<>% sqrt  
  
head(mtcars)  
## mpg cyl disp hp drat wt qsec vs am gear carb  
## Mazda RX4 4.582576 6 160 110 3.90 2.620 16.46 0 1 4 4  
## Mazda RX4 Wag 4.582576 6 160 110 3.90 2.875 17.02 0 1 4 4  
## Datsun 710 4.774935 4 108 93 3.85 2.320 18.61 1 1 4 1  
## Hornet 4 Drive 4.626013 6 258 110 3.08 3.215 19.44 1 0 3 1  
## Hornet Sportabout 4.324350 8 360 175 3.15 3.440 17.02 0 0 3 2  
## Valiant 4.254409 6 225 105 2.76 3.460 20.22 1 0 3 1

Some functions (e.g. lm, aggregate, cor) have a data argument, which allows the direct use of names inside the data as part of the call. The exposition (%$%) operator is useful when you want to pipe a dataframe, which may contain many columns, into a function that is only applied to some of the columns. For example, the correlation (cor) function only requires an x and y argument so if you pipe the mtcars data into the cor function using %>% you will get an error because cor doesn't know how to handle mtcars. However, using %$% allows you to say "take this dataframe and then perform cor() on these specified columns within mtcars."

# regular piping results in an error  
mtcars %>%  
 subset(vs == 0) %>%  
 cor(mpg, wt)  
## Error in pmatch(use, c("all.obs", "complete.obs", "pairwise.complete.obs", : object 'wt' not found  
  
# using %$% allows you to specify variables of interest  
mtcars %>%  
 subset(vs == 0) %$%  
 cor(mpg, wt)  
## [1] -0.830671

## Additional Resources

The magrittr package and its pipe operators are a great tool for making your code simple, efficient, and readable. There are limitations, or at least suggestions, on when and how you should use the operators. Garrett Grolemund and Hadley Wickham offer some advice on the proper use of pipe operators in their [R for Data Science](http://r4ds.had.co.nz/) book. However, the %>% has greatly transformed our ability to write "simplified" code in R. As the pipe gains in popularity you will likely find it in more future packages and being familiar will likely result in better communication of your code.

Some additional resources regarding magrittr and the pipe operators you may find useful:

* The magrittr vignette (vignette("magrittr")) in your console) provides additional examples of using pipe operators and functions provided by magrittr.
* A [blog post](http://www.r-bloggers.com/simpler-r-coding-with-pipes-the-present-and-future-of-the-magrittr-package/) by Stefan Milton Bache regarding the past, present and future of magrittr
* [magrittr questions](http://stackoverflow.com/questions/tagged/magrittr) on Stack Overflow
* The [ensurer](https://cran.r-project.org/web/packages/ensurer/vignettes/ensurer.html) package, also written by [Stefan Milton Bache](https://twitter.com/stefanbache), provides a useful way of verifying and validating data outputs in a sequence of pipe operators.

1. According to [Dave Thomas](http://www.artima.com/intv/dry.html), "DRY says that every piece of system knowledge should have one authoritative, unambiguous representation. Every piece of knowledge in the development of something should have a single representation. A system's knowledge is far broader than just its code. It refers to database schemas, test plans, the build system, even documentation." [↑](#footnote-ref-22)