A Crash Course in R

David Kahle
Department of Statistical Science
Baylor University
www.kahle.io

Contents

What This Document is About

The purpose of this document is to give you a crash course in R. Since it's a crash course, I don't intend in this document to teach you anything about programming per se: I assume you know what variables are, what control flows are, what functions are and so on, just in another language. Where R does something that's unexpected and remarkable coming from some other common language (C, C++, Python, etc.), and it's easy to showcase quickly, I do so. Otherwise, I try to keep this as brief as possible.

If you do have any questions, or if you think I should add something to this document, please email me!

RStudio

Whether you're just getting started with R or you're a package developer, I strongly recommend you use RStudio. You can download it for free from www.rstudio.com/products/rstudio/download/.

In programming jargon, RStudio is an integrated development environment (IDE) for the R language. At a basic level, what that means is that RStudio is a program that runs R but contains a lot of added functionality that makes writing R and R-related code easier. For example, it has a ton of keyboard shortcuts that do various things that you may want to do in R, ranging from basic things like inserting the pipe operator (Cmd-Shift-M on a Mac) to reflowing a package's **roxygen2** documentation (Cmd-Shift-/). But that just scratches the surface, RStudio enhances R in many, many ways.

In addition to the program RStudio, the company RStudio produces high-quality R related instructional content for free. It provides free webinars roughly every month or so discussing R-related topics, ranging from basic to advanced, and it posts them to its website free of charge: www.rstudio.com/resources/webinars/. If you're just getting into R, you may want to checkout the RStudio Essentials videos on that page after you read through this document.

RStudio also produces a number of high quality cheat sheets available from www.rstudio.com/resources/cheatsheets/. Those can be handy to have around depending on what you're working on.

Installing and Loading Packages in R

Installing packages: `install.packages()` and `install_github()`

When you download R, it comes with its basic computing abilities and handful of packages. These include **base**, **MASS**, **stats**, **utils**, and so on. There are currently two main ways to install packages that others have written: `install.packages()` and `install_github()`.

`install.packages()` is a function that comes with R. To install a package, you pass the name of the package you want to download into `install.packages()`. For example, if you want to download the **devtools** package, you type

```
install.packages("devtools")
```

If you want to install several at once, you do something like

```
install.packages(c("devtools", "dplyr", "tidyr"))
```

Since R is installing code from one of its repositories, you have to be connected to the internet to run `install.packages()`. It may ask for what mirror you want to download it from – that's the server that it's getting the package from. R's central server is cran.r-project.org, where CRAN stands for Comprehensive R Archive Network. The "mirrors" are simply copies of cran.r-project.org, which update roughly daily (it varies by mirror). There are several good mirrors. For example, cran.rstudio.com and cran.revolutionanalytics.com are both popular and support HTTPS, but the packages you get from them are identical to those from CRAN.

`install_github()` is a function in the **devtools** package that installs packages off of Github. Github is a social coding website that allows developers to keep good records of their code and collaborate on projects. In many cases, developers also make their code public ("open source") so that others can take it and add onto it or fix little bugs. For example, you can find R itself on Github, although it is not developed there. You can also find packages, such as **ggplot2** or **mpoly**, there. To install them directly from Github, you'd use code like

```r
devtools::install_github("dkahle/mpoly")
```

where `dkahle` is the username of the developer of the **mpoly** package. If you know the name of the package you want but don't know the developer, just Google them. For example, "mpoly github".

Loading packages

To use the code in a package you have to load it. The typical way to do this is with the `library()` function, like this:

```r
library(mpoly)
```

Alternatively, you sometimes see people use the `require()` function. They basically do the same thing with slightly different return values.

When you load a package, all of the functions in that package become available to you. While that may sound like a good thing, sometimes it can be a nuisance. For example, if two packages have functions in them that are the same name, you can get tripped up as to which one will be used. To fix this, you can refer to a function in a specific package by using the double colon operator `::`, like this:

```r
mpoly::permutations(3)
#      [,1] [,2] [,3]
# [1,]    1    2    3
# [2,]    1    3    2
# [3,]    2    1    3
# [4,]    2    3    1
# [5,]    3    1    2
# [6,]    3    2    1
```

The `::` operator is also commonly used to use functions in packages that have not been loaded.

Note: you only have to install a package once, but you have to load it every time you open R.

Help!

There are many ways to look for help in R. For example if you want to know about the function `lm()` you can access the documentation with either of the following:

```r
?lm
help("lm")
```

If you want to search the help for something, there is the `help.search()` function. For example:

```r
help.search("generalized linear models")
```

For even more help, the function `findFn()` in the **sos** package can be used to conduct an exhaustive search using search.r-project.org. You use it essentially in the same way you use `help.search()`, for example `sos::findFn("eastic net")`. To read up on how it works, see *The R Journal*'s article on it.

R as a Calculator

R has a lot of basic math functionality built into it. For example, it has all of the ordinary arithmetic operations `+`, `-`, `*`, `/`, and `^` in addition to those you typically see in programming languages `%%` (modulus) and `%/%` (integer division). Matrix multiplication is `%*%`, see below for how to create matrices.

In addition to the arithmetic operations, there are also many built-in mathematical functions, such as the trigonometric functions `sin()`, `cos()`, `tan()`, and their inverses `asin()`, `acos()` and `atan()`; the exponential functions `exp()` and `log()`; and special functions such as `beta()` and `gamma()`, their logs `lbeta()` and `lgamma()`. Polygamma functions included as well, see `?gamma`. The factorial is `factorial()`, and its log is `lfactorial()`. The binomial coefficient is `choose()`; `lchoose()` also exists. `sin()`, `abs()`, and `sqrt()` do the standard things.

Infinity is `Inf`, and negative infinity is `-Inf`. `NaN` governs not a number, which is to be distinguished from `NA` (a missing value), or `NULL`, the null object.

Variables and Assignment

The assignment operator in R is `<-` For example:

```
x <- 5
x + 1
# [1] 6
```

You can assign essentially anything to a variable. While `=` works for assignment, the two are not interchangeable, so it is generally recommended to use `<-`.

Notice that the resulting value of `x` in the above is not printed out by default. To print the assigned value to the screen, put parentheses around the assignment:

```
(x <- 5)
# [1] 5
```

This is particularly useful in situations where the assigned value is not clear; typically because the right hand side is some expression that has to be evaluated. For example, the `rnorm()` function generates pseudo-random numbers from the normal distribution. If you set `x <- rnorm(5)`, `x` will contain 5 randomly sampled numbers, but you won't know what they are unless you print `x` out. Wrapping the code up in parentheses fixes this problem:

```
(x <- rnorm(5))
# [1]  1.01253808 -0.77984445  2.04036375 -1.72415299 -0.01389306
```

You can list all the variables in the global environment with `ls()`:

```
ls()
# [1] "x"
```

You remove variables with the `rm()` function.

```
x
# [1]  1.01253808 -0.77984445  2.04036375 -1.72415299 -0.01389306
rm(x)
x
# Error in eval(expr, envir, enclos): object 'x' not found
```

To remove all the variables in your global environment, you can use the following nuclear `rm(list = ls())`.

Occasionally you may come across the global assignment operator `<<-`. In a context where lexical scoping matters, the global assignment operator will make the assignment in the nearest environment where the variable is defined or the global environment, whichever is closest. If it makes it to that point, it will assign it there. It is usually used inside functions to perpetuate variables. For example:

```
f <- function() { a <- 1 }
f()
a
# Error in eval(expr, envir, enclos): object 'a' not found
f <- function() { a <<- 1 }
f()
a
# [1] 1
```

Data Structures in R

R has effectively four basic data types: logicals, integers, doubles, and characters. Logicals are either `TRUE` or `FALSE`, integers are made by typing a `L` after the desired number (e.g. `5L` or `-3L`), and characters are made with either single or double quotes so that `'a'` is the same as `"a"`. Complex types are also available, but since you don't use them much I won't cover them here.

Note: To check what kind of data structure you're dealing with, use the `str()` function.

```
str(TRUE)
#  logi TRUE
str(1L)
#  int 1
str(1.0) # num = numeric = double
#  num 1
str("a")
#  chr "a"
```

Vectors

Vectors are sequences of the basic types. The basic way to make vectors is to use the `c()` function:

```
c(1, 2, 3)
# [1] 1 2 3
c("a", "b", "c")
# [1] "a" "b" "c"
```

Vectors that are sequences of integers can be easily constructed with the colon `:` operator:

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

Sequences of other numbers are made with the `seq()` function, which can create the vectors using a step size or a total length:

```
seq(0, 1, .1) # = seq(0, 1, by = .1)
#  [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
seq(0, 1, length.out = 5)
# [1] 0.00 0.25 0.50 0.75 1.00
```

You can also create vectors using the `rep()` function, which repeats its arguments. You use it in two ways:

```
rep(c("a", "b"), 5)
#  [1] "a" "b" "a" "b" "a" "b" "a" "b" "a" "b"
rep(c("a", "b"), each = 5)
#  [1] "a" "a" "a" "a" "a" "b" "b" "b" "b" "b"
```

All vectors have length that can be accessed with `length()`. Since R is a vectorized language, stand-alone numbers or characters are implicitly length-one vectors:

```
length(2)
# [1] 1
length(1:10)
# [1] 10
```

Occasionally you'll come across a length zero object. These look like `logical(0)`, `integer(0)`, `numeric(0)` (`numeric` is often an alias for `double`), and `character(0)`.

If you want to label the elements of a vector, you can just pass the names into the `c()` function like this:

```
c(first = 1, second = 2, third = 3)
#  first second  third
#      1      2      3
```

These are called *named vectors*; we'll come back to them when we talk about extraction. Alternatively, you can assign names to a vector after the fact using `names() <-`, like this:

```
x <- 1:3
names(x) <- c("a", "b", "c")
x
# a b c
# 1 2 3
```

Matrices and Arrays

Matrices and arrays are usually created using the `matrix()` or `array()` functions:

```
matrix(1:6, nrow = 2, ncol = 3)
#      [,1] [,2] [,3]
# [1,]    1    3    5
# [2,]    2    4    6
```

Notice that matrices are populated by column by default.

Here's an example of an array:

```
array(1:(1*2*3), dim = c(1, 2, 3))
# , , 1
#
#      [,1] [,2]
# [1,]    1    2
#
# , , 2
#
#      [,1] [,2]
# [1,]    3    4
#
# , , 3
#
#      [,1] [,2]
# [1,]    5    6
```

Matrices and arrays both have dimensions accessed with `dim()`. For matrices, the functions `nrow()` and `ncol()` give you the number of rows and columns.

To make a diagonal matrix with a given diagonal, use `diag()`. For example:

```
diag(1:3)
#      [,1] [,2] [,3]
# [1,]    1    0    0
# [2,]    0    2    0
# [3,]    0    0    3
```

The identity matrix is made by simply passing an integer into `diag()`. For example, `diag(c(1,1,1)) = diag(3)`.

Naming matrices is easiest using `colnames() <-` and `rownames() <-`, assigning them after the fact. For example:

```
A <- diag(1:3)
rownames(A) <- c("a", "b", "c")
colnames(A) <- c("A", "B", "C")
A
#   A B C
# a 1 0 0
# b 0 2 0
# c 0 0 3
```

Data Frames

As described above, vectors and matrices must have the same types of elements. If you try to combine elements of different types, the elements get changed ("coerced") into analogues of a higher type following the scale logical < integer < double < character. For example, look at what happens when you combine a double and a character:

```
c(1.1, "a")
# [1] "1.1" "a"
str(c(1.1, "a"))
#  chr [1:2] "1.1" "a"
```

```
identical(1.1, "1.1")
# [1] FALSE
```

Typically, when considering spread-sheet like data, the rows represent different individuals or objects of which there are n, and the columns represent different measurements. The problem with using a matrix for this data structure is that the types of different measurements change dependent on what kind of variables you have. For example, in a dataset containing one column of addresses (a character vector) and valuations (a numeric vector), a matrix would "promote" all the valuations to characters. The problem with that is that arithmetic operations aren't defined for character vectors. Enter the data frame.

A data frame is like a matrix, but its columns can have different types. You usually make a data frame by passing vectors into the `data.frame()` function:

```
data.frame(1:2, c("a","b"))
#   X1.2 c..a....b..
# 1    1           a
# 2    2           b
```

To make a data frame with sensible column names, enter the names before the vectors with an equal sign dividing them:

```
data.frame(ints = 1:2, chars = c("a","b"))
#   ints chars
# 1    1     a
# 2    2     b
```

You can also use the `names() <-` syntax to assign or re-assign column names to a data frame that already exists:

```
df <- data.frame(1:2, c("a","b"))
names(df) <- c("ints", "chars")
df
#   ints chars
# 1    1     a
# 2    2     b
```

Lists

In R, lists are vectors whose elements (1) can be any kinds of objects and (2) don't have to be the same kind of object. What we have previously referred to as vectors are properly called atomic vectors, because their elements are one of the atomic types.

You make a list using the `list()` function:

```
list(TRUE, 1:3, rnorm(3), data.frame(a = 1:2, b = 3:4))
# [[1]]
# [1] TRUE
#
# [[2]]
# [1] 1 2 3
#
# [[3]]
# [1] -1.35849398  0.03148329  0.21554016
```

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

Lists can be given names in the same way vectors and data frames can be given names, by assigning them in `list()` or using `names() <-`:

```
(l <- list(a = 1:3, b = c("a","b","c")))
# $a
# [1] 1 2 3
#
# $b
# [1] "a" "b" "c"
names(l) <- c("c", "d")
l
# $c
# [1] 1 2 3
#
# $d
# [1] "a" "b" "c"
```

Notice that the list in this last example looks very much like the data frame `data.frame(a = 1:3, b = c("a","b","c"))`. In fact, under the hood data frames are lists whose elements are vectors of the same length. This will have important consequences when we talk about extracting parts of data frames, where we will see that they can be referenced like lists or like matrices.

Factors and Ordered Factors

Another kind of data structure that is seen frequently in data analysis is the factor, which is used to represent categorical and ordinal variables. Factors are special types of vectors whose elements must be one of a finite collection of potential elements. They are made with the `factor()` function. For example, we may represent a series of coin flips as follows:

```
factor(c("H", "H", "T", "H", "T"), levels = c("H","T"))
# [1] H H T H T
# Levels: H T
```

When each of the levels is observed, you don't have to put in the `levels = c("H","T")` part in. R will look at the unique values, computed with the `unique()` function, and `sort()` them to compute the levels. For example:

```
factor(c("H", "H", "T", "H", "T"))
# [1] H H T H T
# Levels: H T
```

Ordinal variables have a natural ordering to their levels. They are made using `factor()`, too, but you need to set `ordered = TRUE`:

```r
factor(
  c("Thr", "Thr", "Fri", "Thr", "Wed", "Wed", "Mon", "Tue"),
  levels = c("Mon", "Tue", "Wed", "Thr", "Fri"),
  ordered = TRUE
)
# [1] Thr Thr Fri Thr Wed Wed Mon Tue
# Levels: Mon < Tue < Wed < Thr < Fri
```

A useful function in the context of factors is `table()`, which computes frequency distributions:

```r
table(factor(c("H", "H", "T", "H", "T")))
#
# H T
# 3 2
```

Tibbles

A newer data structure that has been gaining popularity recently is the tabled data frame, also called a "tibble", made available through the R package tibble. Although they are much more flexible than data frames, you should think of tibbles as data frames that print more nicely.

They are also the default data structures in many of the RStudio packages, such as **dplyr**, **tidyr**, **readr**, and so on.

You can create a tibble using the `data_frame()` function or, if you already have a data frame and you want to convert it into a tibble, you can use `tbl_df()`. Example:

```r
library(tibble)
data_frame(ints = 1:2, chars = c("a","b"))
# Source: local data frame [2 x 2]
#
#    ints chars
#   <int> <chr>
# 1     1     a
# 2     2     b
```

Coercion

Coercion refers to taking a data structure of one type and changing it into a data structure of another type. R has a systematic suite of functions that do this: `as.logical()`, `as.integer()`, `as.numeric()`, `as.character()`, `as.matrix()`, `as.data.frame()`, `as.list()`. And generally speaking, package developers obey the pattern, so you get things like `as_data_frame()` to coerce something into a tibble. (This is, incidentally, the same as `tbl_df()`.)

In some circumstances R automatically coerces objects. For example, we saw that at the beginning of the discussion on data frames when you try to make a vector with a number `1.1` and a character `"a"`, it coerced `1.1` to `"1.1"`. However, automatic coercion happens in other circumstances, too. For example, arithmetic between non-like types involves coercion:

```r
TRUE + 3
# [1] 4
```

This coercion is also called “promotion”, because the lower type is always coerced into the higher type, following the scheme logical < integer < double < character. That’s why in the `c(1.1, "a")` example I used the `identical()` function to test if the number `1.1` is equal to the character `"1.1"`. Ask yourself: why does the following return `TRUE`?

```
1.1 == "1.1"
# [1] TRUE
```

Extracting Parts of Data Structures

Extracting parts of data structures is done with the `[`, `[[`, and `$` operators.

Extracting From Vectors and Matrices

`[` is the basic subsetting operator; it works on essentially all objects. Here’s a basic example:

```
x <- c(1.1, 2.1, 3.1, 4.1, 5.1)
x[3]
# [1] 3.1
```

You can put essentially four kinds of things in the bracket `[`:

1. a vector of positive integers, in which case those elements are extracted,
2. a vector of negative integers, in which those elements are removed,
3. a logical of vector of the same length as the vector in question, in which the `TRUE`’s are extracted, or
4. (named vectors only) a character vector, in which case the elements are extracted.

Examples:

```
x[c(1,3,5)]
# [1] 1.1 3.1 5.1
x[-c(1,3,5)]
# [1] 2.1 4.1
x[c(TRUE, FALSE, TRUE, FALSE, TRUE)]
# [1] 1.1 3.1 5.1
names(x) <- c("a", "b", "c", "d", "e")
x[c("a","c","e")]
#   a   c   e
# 1.1 3.1 5.1
```

Extracting with logicals is most often done implicitly via an expression with a relational operator `>`, `>=`, `<`, `<=`, `==`, or `!=`, for example:

```
x > 2.5
#     a     b     c     d     e
# FALSE FALSE  TRUE  TRUE  TRUE
x[x > 2.5]
#   c   d   e
# 3.1 4.1 5.1
```

Extracting elements of matrices and arrays works in precisely the same way, but commas are used inside the bracket to separate dimensions. Blanks are interpreted as all positive integer values in the range of that index:

```
A
#   A B C
# a 1 0 0
# b 0 2 0
# c 0 0 3
A[1,1]
# [1] 1
A[1, ] # first row
# A B C
# 1 0 0
A[ ,2:3] # second and third columns
#   B C
# a 0 0
# b 2 0
# c 0 3
A["b", c(FALSE, TRUE, TRUE)]
# B C
# 2 0
```

Notice that in the first row subsetting above, the result is a vector, not a matrix. When the result of subsetting a matrix is one row or one column, R simplifies the result to a vector and does not preserve the matrix structure. To fix this, you can use `drop = FALSE` inside the brackets:

```
A[1, , drop = FALSE]
#   A B C
# a 1 0 0
```

Extracting From Lists and Data Frames

When you use `[` on a list, you get a list back following the exact same rules as above. However, when dealing with lists it is common to want to extract the elements themselves, that's what `[[` and `$` are for.

Here's a typical situation. You have a named list

```
(l <- list(a = c(TRUE, FALSE, TRUE), b = 1:3, c = c("c", "b", "a")))
# $a
# [1]  TRUE FALSE  TRUE
#
# $b
# [1] 1 2 3
#
# $c
# [1] "c" "b" "a"
```

and you want to extract the values in the `"b"` element. You try

```
l[2]
# $b
# [1] 1 2 3
```

and it looks like it works, until you try to do something with it, like take its mean:

```
mean(l[2])
# Warning in mean.default(l[2]): argument is not numeric or logical:
# returning NA
# [1] NA
```

The problem is that subsetting a list with `[` *always* returns a list, and the `mean()` function expects a numeric vector. You didn't want a list of length 1, you wanted the contents of that element. To get that, you use `[[`:

```
mean(l[[2]])
# [1] 2
```

`[[` extracts the contents of only one element of a list, so you need to refer to it specifically by index or name. The name looks like `l[["b"]]`. Since that's a lot of typing, the developers of R put in the `$` operator, which does the same in a far cleaner way:

```
l$b
# [1] 1 2 3
```

Since data frames are lists, they are subset in precisely the same way. Since the columns of a dataset are variables, it is very common to see things like `data$height` to extract all the heights of individuals in a dataset.

However, since data frames also look and feel like matrices, the double-indexed `[` notation (e.g. `[1,2]`) works on them just as it does on matrices. For example:

```
(df <- data.frame(name = c("bob", "jim", "sally"), height = c(5.2, 5.8, 5.5)))
#    name height
# 1   bob    5.2
# 2   jim    5.8
# 3 sally    5.5
df$height
# [1] 5.2 5.8 5.5
df[2:3, ]
#    name height
# 2   jim    5.8
# 3 sally    5.5
```

Matrix Algebra

Matrix multiplication is done using the `%*%` operator. When using `%*%`, vectors are treated as either row or column vectors as-needed to make the operation work.

`det()` computes the determinant of a matrix, and `t()` transposes it.

`solve()` computes the inverse of a matrix:

```
(A <- diag(c(2,4,8)))
#      [,1] [,2] [,3]
# [1,]    2    0    0
# [2,]    0    4    0
# [3,]    0    0    8
```

```
solve(A)
#      [,1] [,2]  [,3]
# [1,]  0.5 0.00 0.000
# [2,]  0.0 0.25 0.000
# [3,]  0.0 0.00 0.125
```

It is also used to solve matrix equations $Ax = b$:

```
b <- c(8,4,2)
solve(A, b)
# [1] 4.00 1.00 0.25
```

The QR decomposition is given by `qr()`, but you need to use `qr.Q()` and `qr.R()` on it's output to get the matrices you want:

```
(qr_A <- qr(A))
# $qr
#      [,1] [,2] [,3]
# [1,]   -2    0    0
# [2,]    0   -4    0
# [3,]    0    0    8
#
# $rank
# [1] 3
#
# $qraux
# [1] 2 2 8
#
# $pivot
# [1] 1 2 3
#
# attr(,"class")
# [1] "qr"
(Q <- qr.Q(qr_A))
#      [,1] [,2] [,3]
# [1,]   -1    0    0
# [2,]    0   -1    0
# [3,]    0    0    1
(R <- qr.R(qr_A))
#      [,1] [,2] [,3]
# [1,]   -2    0    0
# [2,]    0   -4    0
# [3,]    0    0    8
Q %*% R
#      [,1] [,2] [,3]
# [1,]    2    0    0
# [2,]    0    4    0
# [3,]    0    0    8
```

There actually a lot of QR related functionality in R because that's how it solves linear regression problems. See `?qr` for details.

The Cholesky decomposition is

```r
# make a positive (semi)definite matrix
preA <- matrix(c(6,1,8,2,5,4,7,3,9), nrow = 3, ncol = 3)
(A <- t(preA) %*% preA)
#      [,1] [,2] [,3]
# [1,]  101   49  117
# [2,]   49   45   65
# [3,]  117   65  139

# compute cholesky decomposition and verify
(chol_A <- chol(A))
#          [,1]     [,2]       [,3]
# [1,] 10.04988 4.875682 11.6419351
# [2,]  0.00000 4.607355  1.7879289
# [3,]  0.00000 0.000000  0.5183211
t(chol_A) %*% chol_A
#      [,1] [,2] [,3]
# [1,]  101   49  117
# [2,]   49   45   65
# [3,]  117   65  139
```

Eigenvalues and eigenvectors can be computed with `eigen()`, which returns a named list whose elements can be extracted like any other named list:

```r
(e_stuff <- eigen(A))
# $values
# [1] 268.1359748  16.7356668   0.1283584
#
# $vectors
#            [,1]        [,2]       [,3]
# [1,] -0.6041180  0.42924944  0.6714063
# [2,] -0.3422941 -0.90061996  0.2678030
# [3,] -0.7196362  0.06803385 -0.6910103
e_stuff$values
# [1] 268.1359748  16.7356668   0.1283584
```

Note: The assignment operator `<-` does not support multiassignment. Consequently, you can't replace the above code with something like `list(lambda, V) <- eigen(A)`.

The singular value decomposition is computed with `svd()`; it's a named list, too:

```r
svd(A)
# $d
# [1] 268.1359748  16.7356668   0.1283584
#
# $u
#            [,1]        [,2]       [,3]
# [1,] -0.6041180  0.42924944 -0.6714063
# [2,] -0.3422941 -0.90061996 -0.2678030
# [3,] -0.7196362  0.06803385  0.6910103
#
# $v
#            [,1]        [,2]       [,3]
# [1,] -0.6041180  0.42924944 -0.6714063
# [2,] -0.3422941 -0.90061996 -0.2678030
# [3,] -0.7196362  0.06803385  0.6910103
```

You can use this to compute the square root of a matrix as follows:

```
# compute the svd and extract components
svd_A <- svd(A)
D <- diag(svd_A$d)
U <- svd_A$u
V <- svd_A$v

# check the svd, this is A
U %*% D %*% t(V)
#      [,1] [,2] [,3]
# [1,]  101   49  117
# [2,]   49   45   65
# [3,]  117   65  139

# and here's the square root matrix
R <- U %*% sqrt(D) %*% t(V)
R %*% R
#      [,1] [,2] [,3]
# [1,]  101   49  117
# [2,]   49   45   65
# [3,]  117   65  139
```

It can also be computed with the eigen decomposition:

```
# compute eigen decomposition
D <- diag(eigen(A)$values)
V <- eigen(A)$vectors
V %*% D %*% solve(V)
#      [,1] [,2] [,3]
# [1,]  101   49  117
# [2,]   49   45   65
# [3,]  117   65  139

# compute root matrix
R <- V %*% sqrt(D) %*% solve(V)
R %*% R
#      [,1] [,2] [,3]
# [1,]  101   49  117
# [2,]   49   45   65
# [3,]  117   65  139
```

Vectorized Operations

Essentially every basic operation in R is vectorized. This holds for the arithmetic operations `+`, `-`, `*`, `/`, `^`, `%%` and `%/%`. For example:

```
1 + 1:3
# [1] 2 3 4
1:3 + 7:9
# [1]  8 10 12
1 + diag(3)
#      [,1] [,2] [,3]
# [1,]    2    1    1
# [2,]    1    2    1
# [3,]    1    1    2
5*(1:5)
# [1]  5 10 15 20 25
```

It also works for the relational operators `>`, `>=`, `<`, `<=`, `==`, and `!=`:

```
(0:8 %% 2) == 0
# [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
c("a", "b", "c") == c("a", "g", "c")
# [1]  TRUE FALSE  TRUE
```

These are very important capabilities in R. When possible, you should vectorize operations, because the resulting code is clear and *fast*. In most cases, vectorized code is evaluated using algorithms written in C, so they are much much faster than their for loop analogues.

The Boolean operators `&` and `|`, which we meet next, are also vectorized.

Conditionals: If Statements and Switch

Every programming language has if statements, which execute a chunk of code if a certain condition is met. In R, they look like this:

```
if (1 == 1) {
  print("yes!")
}
# [1] "yes!"
```

If the if statement is very simple, you can put it inline as `if (1 == 1) print("yes!")`.

If else statements look like this:

```
if (1 == 2) {
  print("yes!")
} else {
  print("no!")
}
# [1] "no!"
```

And you can chain them like this:

```
if (1 == 2) {
  print("yes!")
} else if (2 == 3) {
  print("maybe!")
```

```r
} else {
  print("no.")
}
# [1] "no."
```

Boolean Operators

The Boolean operators "and" and "or" are `&` and `|` in R. For example:

```r
x <- 5
0 <= x & x <= 10
# [1] TRUE
```

Note that this can *not* be shortened to `0 <= x <= 10`.

As noted in the section above, `&` and `|` are vectorized. We can use this, for example, to make truth tables:

```r
c(T, T, F, F) & c(T, F, T, F)
# [1]  TRUE FALSE FALSE FALSE
c(T, T, F, F) | c(T, F, T, F)
# [1]  TRUE  TRUE  TRUE FALSE
```

The Short Circuiting Boolean Operators `&&` and `||`

The input to an if statement is typically an expression that evaluates to *a single* `TRUE` or `FALSE` (e.g. `1 == 1`). If you want to check more than one condition and tie them together with ands or ors, you can do so more efficiently with `&&` and `||`. `&&` and `||` are short circuiting and and or that are ideal for if statements because they always return only one logical value.

Concretely, there are two differences between `&` and `&&` (or `|` and `||`). First, the first is vectorized and the second is not. Second, `&&` (`||`) is smart in its evaluation scheme. For example, if you type `(1 == 2) && (1 == 1)`, it will first evaluate `1 == 2` and, seeing that it evaluates to `FALSE`, immediately evaluates to `FALSE` without even evaluating `1 == 1`. The same goes for `||`.

`any()` and `all()`

`any()` is a function that takes in a logical vector and evaluates to `TRUE` if any of the values is true. `all()` accepts a logical vector and returns `TRUE` if all the elements are true. For example:

```r
all( (c(2, 4, 6) %% 2) == 0 )
# [1] TRUE
```

`switch()`

Most languages also have some kind of switch or cases statement. These evaluate an expression and then return a value conditional on what the expression evaluates to. In R, this is done with `switch()`.

```r
x <- "b"
switch(x,
  "a" = 1,
  "b" = 2,
  "c" = 3
)
# [1] 2
```

Note: `switch()` works slightly differently if the result from the expression is numeric (integer or double). See `?switch` for details.

Loops

There are three types of loops in R: for loops, while loops, and repeat statements.

For Loops

For loops in R look like this:

```
for (k in 1:5) {
  print(k)
}
# [1] 1
# [1] 2
# [1] 3
# [1] 4
# [1] 5
```

The index variable (`x` above) can have any valid name in the R language, and the index set `1:5` can be any kind of vector. For example:

```
for (letter in c("a", "b","c")) {
  if(letter == "b") next
  print(toupper(letter))
}
# [1] "A"
# [1] "C"
```

(Note the use of the `next` above.)

Like many scripting languages, there's no incrementing step in the statement of the for loop. In particular, you don't see things like `i++`, because the `++` operator doesn't exist in R.

For loops are often frowned upon by R programmers because they are perceived to be slow. This really depends on what the for loop is doing, but as a general rule looping in R will be significantly slower than looping in some lower level language such as C, C++, or Java, and in some cases much, much slower. As a consequence, it is quite rare to see R programmers nesting for loops. Typically R programmers write in a functional style using functions described in the section on functionals.

While Loops

While loops look like this:

```
a <- 0
while (a < 5) {
  print(a)
  a <- a + 1
}
# [1] 0
# [1] 1
# [1] 2
# [1] 3
# [1] 4
```

Nothing new here, given what we've seen in for loops.

Repeat Statements

It's rare to see repeat statements in R, but they do exist. They just run the same chunk of code over and over until they meet a `break`:

```
repeat{
  x <- rnorm(1)
  print(x)
  if(x > 1.5) break
}
# [1] -0.6264538
# [1] 0.1836433
# [1] -0.8356286
# [1] 1.595281
```

Note that unlike other functions, repeat uses the curly braces `{}`, not the parentheses `()`.

`replicate()`

In addition to the for, while, and repeat looping constructs, R provides another looping functionality with the `replicate()` function. `replicate()` accepts a chunk of code and the number of times you want it run (in the opposite order), and it returns the results of each run.

```
replicate(5, {
  x <- rnorm(1)
  x^2
})
# [1] 0.1085754 0.6731684 0.2375871 0.5451234 0.3315242
```

This is very useful for simulation purposes. For example, the central limit theorem (CLT) states that if you take samples from almost any distribution and average them, the distribution of the mean is at worst approximately normal. Using samples from the Poisson distribution as an example, we can sample from the distribution of the average of 30 Poisson(20) observations using `replicate()`:

```
N <- 5   # the number of desired samples
n <- 30  # the size of the dataset
replicate(N, mean(rpois(n, 20)))
# [1] 19.86667 20.30000 20.93333 17.66667 20.23333
```

To assess the CLT approximation, we could change `N` to `1e4` (i.e. `10000`) and then plot a histogram (`hist()`) or make a normal plot (`qqnorm()`). Since this crash course doesn't cover graphics, however, we'll leave it there.

Functions

Making a basic function in R is done like this:

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

For a really short function, you can make it inline as `f <- function(x) x^2`.

They are used like this:

```
f(2)
# [1] 4
```

If you want a function of several variables, you do this:

```
g <- function (x, y) {
  x + y
}
g(1, 2)
# [1] 3
```

You can default values of functions as follows:

```
h <- function (x = 5) {
  x + 2
}
h(1)
# [1] 3
h()
# [1] 7
```

Here are some key notes on functions:

1. You don't have to tell the function what kinds of things the arguments (inputs) are.
2. By default, the last thing evaluated (the last line run) is what's returned by the function. You can short-circuit this behavior with `return()` so that the function stops in the middle of its body; however, don't get into the habit of putting a `return()` on the last line of the body. You don't need it there.
3. You don't have to specify the type of the returned value when you define the function.

Note also that things that happen within functions don't persist outside them. We already saw this in the discussion on the global assignment operator `<<-` in the section on assignment; however, it bears repeating. Notice the following behavior:

```
z
# Error in eval(expr, envir, enclos): object 'z' not found
f <- function () {
  z <- 100
  z
}
f()
# [1] 100
z
# Error in eval(expr, envir, enclos): object 'z' not found
```

Although `z` was defined inside the function, its value does not persist after the application of the function. In general, any changes to variables made inside a function won't persist after the function has executed.

The Pipe Operator `%>%`

In the past few years, many packages in R have adopted the use of the pipe operator, which in R looks like `%>%`, implemented in the **magrittr** package. The pipe operator simply changes the way that you can specify the evaluation of a function. For example, instead of writing `f(2)` as above, the pipe operator allows you to write `2 %>% f()`; or, more simply, `2 %>% f`. In other words, with the pipe operator `f(x)` is written `x %>% f`. If the function in question has more than one argument, like `g()` above, the pipe operator puts the left hand side into the first argument of the function. For example, `g(1, 2)` is written `1 %>% g(2)`. There are more details and uses of `%>%`, but that's the basic idea.

Why is the pipe operator useful? Essentially for one reason: it eases situations with function composition. For example, if you have functions `f()`, `g()`, and `h()`, we can evaluate `h(g(f(x)))` by writing `x %>% f %>% g %>% h`. In real data analyses, we often take a dataset, manipulate it with various transformations, and then compute on it in some way. This often involves applying many functions do the dataset and is done across several lines, as follows:

```
data %>%
  transformation_one(other_arguments) %>%
  transformation_two(other_arguments) %>%
  computation(other_arguments)
```

The leading packages implementing this paradigm are **dplyr**, **tidyr**, **purrr**, **ggplot2**/**ggvis**, and others created by RStudio folks. The first two in particular are very, very useful for practical data analysis tasks.

Functionals

In R, functions are first-class citizens. What that means is that you can do things like pass functions into other functions, return functions from functions ("closures"), store them in lists, and use them anonymously. There are many functions that do this in R; they generally follow the form of `*apply()`, like `lapply()`, `sapply()`, `vapply()`, `tapply()`, `mapply()`, and so on. Each of these are functionals, functions that take as input functions (and maybe other data structures as well) and return data structures.

In essence, each of the `*apply()` functions above do the same thing: apply a function to a bunch of things. Consider the following example using `lapply()`:

```
f <- function (x) x^2
lapply(1:3, f)
# [[1]]
# [1] 1
#
# [[2]]
# [1] 4
#
# [[3]]
# [1] 9
```

`lapply()` takes two arguments – a vector and a function – and it evaluates the function at every element of the vector. The vector is oftentimes a list, so think of `lapply()` like "list-apply".

`lapply()` take in a list `X` and a function `f`, and returns a list `Y` whose elements are `f` applied to the elements of `X`.

`sapply()` works similarly but then tries to simplify the list into a simpler data structure, often a vector. For example, here's the same statement as before using `sapply()` instead of `lapply()`:

```
sapply(1:3, f)
# [1] 1 4 9
```

The return value is an atomic vector, not a list.

Not all of the `*apply()` functions are equally important. `lapply()` and `sapply()` are the heavy hitters – they are typically used in place of what otherwise would be a for loop. After those, `Map()` is probably the most important.

`Map()` is a multivariate version of `lapply()` used when you have function of several variables and several lists/vectors. For example:

```
g <- function (x, y) x + y
Map(g, 1:3, 4:6)
# [[1]]
# [1] 5
#
# [[2]]
# [1] 7
#
# [[3]]
# [1] 9
```

`mapply()` is the corresponding multivariate version of `sapply()`:

```
mapply(g, 1:3, 4:6)
# [1] 5 7 9
```

In any of the above cases, we could have used anonymous functions, where we simply define the function on the fly in the functional. For example:

```
sapply(1:5, function(x) log(1 + x))
# [1] 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595
```

`apply()`, by itself, is a function that acts as a functional over arrays. Consequently, it takes in an array (that includes matrices), an index or indices, and a function, and applies the function over the array for all combinations of that index/those indices. For example, I can compute the sum of the rows of a matrix as follows:

```r
preA
#      [,1] [,2] [,3]
# [1,]    6    2    7
# [2,]    1    5    3
# [3,]    8    4    9
apply(preA, 1, sum) # row sums
# [1] 15  9 21
apply(preA, 2, sum) # col sums
# [1] 15 11 19
```

For some common operations, like those above, there are built-in functions such as `rowSums()` or `colMeans()`:

```r
rowSums(preA)
# [1] 15  9 21
colMeans(preA)
# [1] 5.000000 3.666667 6.333333
```

The **matrixstats** package provides tons of these that are very efficient.

Statistics

Statistics Functions

Since R is designed for statistics, it's got tons of statistics functions built into it. We saw `sum()` above, there's also `prod()`. `cumsum()` and `cumprod()` also exist.

For measures of center, there's `mean()`, `median()`, but not mode. The `mode()` function does something else. Google anything else you want.

For measures of dispersion, there's `sd()`, `var()`, `range()` (or rather `diff(range())`), `IQR()`, `mad()`, and so on. Google anything else you want.

For multivariate data, there's `cor()`, `corr()`, `cov()`

Percentiles are computed with the `quantile()` function. For example:

```r
quantile(1:10, .25)
#  25%
# 3.25
```

And it's vectorized across the probabilities, so you can do

```r
quantile(1:10, c(.025, .975)) # the 95% percentile interval
#  2.5% 97.5%
# 1.225 9.775
```

Distribution Functions

Most of the major probability distributions are built into R. Moreover, four important functions concerning them follow a pattern that's easy to remember.

Let's look at the functions related to the normal distribution. We have

1. `rnorm()` generates pseudo-random numbers from the distribution,

2. `dnorm()` is the probability density function (PDF) of the normal distributions. For discrete distributions, `d*()` provides the probability mass function (PMF),
3. `pnorm()` is the cumulative distribution function (CDF) of the normal distribution, and
4. `qnorm()` is the quantile function (the inverse of the CDF) of the normal distribution.

They are each vectorized in many ways.

These exist for many common distributions, you just need to know the tag line. Here we have

Continuous Distribution	Tag Line	Discrete Distribution	Tag Line
Normal	`norm`	Binomial	`binom`
Uniform	`unif`	Poisson	`pois`
Gamma	`gamma`	Geometric	`geom`
Chi-squared	`chisq`	Negative binomial	`nbinom`
t	`t`	Multinomial	`multinom`
F	`f`	Hypergeometric	`hyper`
Beta	`beta`	.	.
Exponential	`exp`	.	.
Log-normal	`lnorm`	.	.

So, if you want to generate 10 samples from the negative binomial distribution with parameters $n = 10$ and $p = 2$, you type

```r
rnbinom(10, 10, .2)
#  [1] 35 32 52 23 20 34 74 43 36 43
```

If you want the probabilities that a coin flips heads x times out of 10 flips of a fair coin, you type

```r
round(dbinom(0:10, 10, .5), 2)
#  [1] 0.00 0.01 0.04 0.12 0.21 0.25 0.21 0.12 0.04 0.01 0.00
```

If you want the probability that a $N(10, 3)$ random variable is between 6 and 8, you type

```r
pnorm(8, 10, 3) - pnorm(6, 10, 3)
# [1] 0.1612813
```

Or, if you want the 99th percentile of the exponential(5) distribution:

```r
qexp(.99, 5)
# [1] 0.921034
```

All of these functions combined form a really potent platform for statistical analysis. Be sure to familiarize yourself with them!

Linear Models

Linear models are a massive topic in R, so putting them as a subsection in a crash course is kind of silly. Nevertheless, here's the basics:

In R, linear models are fit with the `lm()` function. Generalized linear models are fit with the `glm()` function. Both accept (1) a formula defining the structure of the model and (2) a dataset that contains the data the model refers to.

Let's do a simple example. The code below generates a little fake dataset for us to play with. It has two predictors (features), X_1, a continuous predictor ranging from 0 to 5, and X_2, a binary predictor. The functional relationship between X_1, X_2 and Y is

$$Y = 20 - 5X_1 + X_1^2 + 4X_2.$$

```
library(dplyr, warn.conflicts = FALSE)
n <- 100
data <- data_frame(x1 = runif(n, 0, 5), x2 = sample(0:1, n, T)) %>%
  mutate(y = 20 - 5*x1 + x1^2 + 4*x2 + rnorm(n, sd = .5))
data
# Source: local data frame [100 x 3]
#
#           x1    x2        y
#        <dbl> <int>    <dbl>
# 1  1.3275433     1 18.81447
# 2  1.8606195     0 14.17987
# 3  2.8642668     0 13.42723
# 4  4.5410389     1 21.99485
# 5  1.0084097     1 19.64755
# 6  4.4919484     0 18.60150
# 7  4.7233763     0 19.05176
# 8  3.3039890     0 14.85149
# 9  3.1455702     1 18.35885
# 10 0.3089314     1 23.39187
# ..       ...   ...      ...
```

To fit the linear model $Y = \beta_0 + \beta_1 X_1 + \beta_2 X_1^2 + \beta_3 X_2$ to the data, we use `lm()` as follows:

```
(mod <- lm(y ~ x1 + I(x1^2) + x2, data = data))
#
# Call:
# lm(formula = y ~ x1 + I(x1^2) + x2, data = data)
#
# Coefficients:
# (Intercept)           x1      I(x1^2)           x2
#      20.122       -5.101        1.018        3.937
```

The output of `lm()` is a giant named list containing all sorts of aspects of the linear model.

```
str(mod, max.level = 1, give.attr = FALSE)
# List of 12
#  $ coefficients : Named num [1:4] 20.12 -5.1 1.02 3.94
#  $ residuals    : Named num [1:100] -0.2672 0.0235 -0.4381 0.1029 -0.3029 ...
#  $ effects      : Named num [1:100] -176.712 -0.661 18.904 -19.438 -0.307 ...
#  $ rank         : int 4
#  $ fitted.values: Named num [1:100] 19.1 14.2 13.9 21.9 20 ...
#  $ assign       : int [1:4] 0 1 2 3
#  $ qr           :List of 5
#  $ df.residual  : int 96
#  $ xlevels      : Named list()
#  $ call         : language lm(formula = y ~ x1 + I(x1^2) + x2, data = data)
#  $ terms        :Classes 'terms', 'formula' length 3 y ~ x1 + I(x1^2) + x2
#  $ model        :'data.frame':    100 obs. of  4 variables:
```

summary() is a function that works on many types of objects in R to summarize them. It is particularly helpful for models:

```
summary(mod)
#
# Call:
# lm(formula = y ~ x1 + I(x1^2) + x2, data = data)
#
# Residuals:
#      Min       1Q   Median       3Q      Max
# -0.90868 -0.29600 -0.09003  0.29318  1.21384
#
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
# (Intercept) 20.12203    0.18945  106.21   <2e-16 ***
# x1          -5.10072    0.15575  -32.75   <2e-16 ***
# I(x1^2)      1.01818    0.02983   34.14   <2e-16 ***
# x2           3.93664    0.09812   40.12   <2e-16 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 0.4845 on 96 degrees of freedom
# Multiple R-squared:  0.9703,  Adjusted R-squared:  0.9693
# F-statistic:  1045 on 3 and 96 DF,  p-value: < 2.2e-16
```

Logistic and Poisson regressions are done with the glm() function, which essentially has the same syntax with another argument describing the conditional distribution of the response, such as family = gaussian() for ordinary linear regression, family = binomial() for logistic regression, and family = poisson() for Poisson regression. See ?glm for details.

Basic Applied Math

Quadrature (numerical integration)

Numerical integration of a function is done with the integrate() function:

```
f <- function (x) sqrt(1-x^2)
integrate(f, lower = -1, upper = 1)
# 1.570796 with absolute error < 1e-09
pi/2 # the theoretical value
# [1] 1.570796
```

The output value of integrate, like so many other functions in R, is a named list.

```
int <- integrate(f, lower = -1, upper = 1)
str(int)
# List of 5
#  $ value       : num 1.57
#  $ abs.error   : num 1e-09
#  $ subdivisions: int 10
#  $ message     : chr "OK"
#  $ call        : language integrate(f = f, lower = -1, upper = 1)
#  - attr(*, "class")= chr "integrate"
int$value
# [1] 1.570796
```

Optimization

R contains a number of routines for optimization. The general-purpose optimization function that comes with R is `optim()`. By default, it performs minimization. `optim()` accepts two arguments: a function and a starting point for the optimization routine (in the opposite order). Note that the function *must* have a vector argument, like `f(x)` where `x` is a vector, *not* `f(x, y)`. This is illustrated in the code below.

```
g <- function (v) {
  x <- v[1]; y <- v[2]
  (x-2)^2 + (y-2)^2
}
optim(c(0, 0), g)
# $par
# [1] 2.000160 2.000153
#
# $value
# [1] 4.876462e-08
#
# $counts
# function gradient
#       65       NA
#
# $convergence
# [1] 0
#
# $message
# NULL
```

Like `integrate()`, the output of `optim()` is a named list. The `par` element contains the argument to the function that minimizes it (the argmin) and the element `value` contains the function value at that point (the min). `counts` tells you how many times the function was evaluated (and the gradient function, if you supplied it) and `convergence` contains codes that indicate how the algorithm performed. For example, `0` means it converged, `1` means it reached the maximum allowed iterations without yet converging, and so on. See `?optim` for the list of all the codes as well as a full run down on the optimization methods available and your control over them.

Working Directories and Reading in Data

R considers itself to be running from a directory (folder) on your computer. When you save something (like a data file or a graphic), it saves to that directory. If you ask R to load a file and only tell it the file name (and not the folder it's in), it'll look to that directory.

The directory R thinks it's in is called the working directory. You can figure out what your working directory is by typing `getwd()`

```
getwd()
# [1] "/Users/david_kahle/Dropbox/Baylor University/Courses/R Crash"
```

You can change it with `setwd()`. `setwd()` accepts a character that contains a path. For example, I could change my working directory to my home directory (`/Users/david_kahle`) by typing `setwd("/Users/david_kahle")`.

If you're having a hard time with setting your path, use the `file.choose()` function. You don't need to give it any arguments, just type `file.choose()` and R will open a window for you to navigate around your computer normally and select a file. After you select the file, R will tell you what it's full path is.

R has a lot of functionality for working with files on your computer since that's typically where it reads datasets from. You can list the files in your working directory with

```
list.files()
#  [1] "cache"            "cars.csv"         "figure"
#  [4] "md"               "R Crash.Rproj"    "R-Crash_cache"
#  [7] "R-Crash_files"    "R-Crash-md.md"    "R-Crash-md.Rmd"
# [10] "R-Crash-pdf.pdf" "R-Crash-pdf.Rmd" "R-Crash.html"
# [13] "R-Crash.Rmd"
```

Loading data into R is a whole topic in and of itself, so we'll only talk the basics here. Most data processed in R comes in the form of spreadsheet-like data stored in a `.csv`, `.txt`, or similar file. R's basic functions to load these kinds of files is `read.csv()`, `read.table()`, and the like. However, I find the functions in the **readr** package, `read_csv()`, `read_table()`, and the like, to be more helpful. They're also faster.

Notice in my files listed above that I have a dataset called `cars.csv`. Here is how I would load that in R.

```
library(readr)
data <- read_csv("cars.csv")
data
# Source: local data frame [50 x 2]
#
#    speed  dist
#    <int> <int>
# 1      4     2
# 2      4    10
# 3      7     4
# 4      7    22
# 5      8    16
# 6      9    10
# 7     10    18
# 8     10    26
# 9     10    34
# 10    11    17
# ..   ...   ...
```

Styling and Where to Go Next

In any kind of computer programming, style matters, even if it does not affect any aspect concerning the evaluation of the code. As a general style guide, I prefer Hadley Wickham's discussion on the topic.

In fact, Hadley's book Advanced R is an authoritative reference for where to go next with your R training. If you want to get into the details of the stuff above, and learn much more, check that book out.

If you're more of a data science type, check out Garrett and Hadley's book on R for Data Science. More than simply a cookbook of R code that completes data science tasks, Garrett and Hadley are great at giving you the right way to think about data science and carry it out in R. I highly recommend anything they write.