R PROGRAMMING

FOR DATA SCIENCE

R PROGRAMMING

As you make your way through the world of data science, learning R programming and other important skills, it's important to remember that data science isn't just a collection of tools. It requires a person to apply those tools in a smart way to produce results that are useful to people.

Data Science Podcast: Not So Standard Deviations

For regular discussions of the latest data science topics, you can go to the Not So Standard Deviations podcast that is co-hosted by myself and Dr. Hilary Parker, a Data Scientist at Stitch Fix. The goal of the podcast is to talk about important data science topics and to have a little fun doing it. We also commonly discuss differences between academia and industry as well as the craft of doing data analysis.
If you're an avid podcast listener, you can subscribe to the podcast through iTunes or any popular podcasting app

OVERVIEW AND HISTORY OF R

R is a programming language and software environment for statistical analysis, graphics representation and reporting. R was created by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand, and is currently developed by the R Development Core Team.

The core of R is an interpreted computer language which allows branching and looping as well as modular programming using functions. R allows integration with the procedures written in the C, C++, .Net, Python or FORTRAN languages for efficiency.

R is freely available under the GNU General Public License, and pre-compiled binary versions are provided for various operating systems like Linux, Windows and Mac.

R is free software distributed under a GNU-style copy left, and an official part of the GNU project called GNU S.

FEATURES OF R

As stated earlier, R is a programming language and software environment for statistical analysis, graphics representation and reporting. The following are the important features of R –

- R is a well-developed, simple and effective programming language which includes conditionals, loops, user defined recursive functions and input and output facilities.
- R has an effective data handling and storage facility,
- R provides a suite of operators for calculations on arrays, lists, vectors and matrices.
- R provides a large, coherent and integrated collection of tools for data analysis.
- R provides graphical facilities for data analysis and display either directly at the computer or printing at the papers.

As a conclusion, R is world's most widely used statistics programming language. It's the # 1 choice of data scientists and supported by a vibrant and talented community of contributors. R is

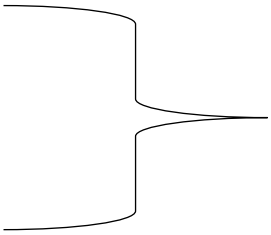
taught in universities and deployed in mission critical business applications. This tutorial will teach you R programming along with suitable examples in simple and easy steps.

R CONSOLE INPUT AND EVALUATION

```
X <- 1

X
[1] 1

Print(X)
[1] 1
```



This is how you assign values to variables. Then, print the values.

You can either write print(variable_name) or just Variable_name and press return to print the value stored in that variable.

The # is a comment. Anything written to the write of #(hash) is ignored by R. This is called a comment. Adding comment to the code helps the programmer identify what a program/set of codes are doing.

```
A <- 1:20
A
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

1:20 creates a vector and stores it in a variable 'A'.

'A' contains 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

=x=x=x=x=x=

R OBJECTS AND CLASSES

R has five basic or atomic classes:

- Character
- Numeric
- Integer
- Complex
- Logical

The most basic object is vector. A vector can only contain objects of same class.
But there is an exception, it is called list. It is represented as a vector but can contain objects of different classes.

To create an empty vector, we can use **vector()**.

NUMBERS

Numbers in R are generally treated as numeric objects (i.e. double precision real numbers). This means that even if you see a number like “1” or “2” in R, which you might think of as integers, they are likely represented behind the scenes as numeric objects (so something like “1.00” or “2.00”). This isn’t important most of the time…except when it is.

If you explicitly want an integer, you need to specify the L suffix. So entering 1 in R gives you a numeric object; entering 1L explicitly gives you an integer object.

There is also a special number Inf which represents infinity. This allows us to represent entities like 1 / 0. This way, Inf can be used in ordinary calculations; e.g. 1 / Inf is 0.

The value NaN represents an undefined value (“not a number”); e.g. 0 / 0; NaN can also be thought of as a missing value (more on that later)

ATTRIBUTES

R objects can have attributes, which are like metadata for the object. These metadata can be very useful in that they help to describe the object. For example, column names on a data frame help to tell us what data are contained in each of the columns. Some examples of R object attributes are:

- names, dimnames
- dimensions (e.g. matrices, arrays)
- class (e.g. integer, numeric)
- length
- other user-defined attributes/metadata

Attributes of an object (if any) can be accessed using the attributes() function. Not all R objects contain attributes, in which case the attributes() function returns NULL.

CREATING VECTORS

The c() function can be used to create vectors of objects by concatenating things together.

```
> x <- c(0.5, 0.6)          ## numeric
> x <- c(TRUE, FALSE)       ## logical
> x <- c(T, F)              ## logical
> x <- c("a", "b", "c")     ## character
> x <- 9:29                 ## integer
> x <- c(1+0i, 2+4i)        ## complex
```

Note that in the above example, T and F are short-hand ways to specify TRUE and FALSE. However, in general one should try to use the explicit TRUE and FALSE values when indicating logical values. The T and F values are primarily there for when you're feeling lazy. You can also use the vector() function to initialize vectors.

```
> x <- vector("numeric", length = 10)
> x
[1] 0 0 0 0 0 0 0 0 0 0
```

MIXING OBJECTS

There are occasions when different classes of R objects get mixed together. Sometimes this happens by accident but it can also happen on purpose.

```
> y <- c(1.7, "a")        ## character
> y <- c(TRUE, 2)         ## numeric
> y <- c("a", TRUE)       ## character
```

In each case above, we are mixing objects of two different classes in a vector. But remember that the only rule about vectors says this is not allowed. When different objects are mixed in a vector, coercion occurs so that every element in the vector is of the same class.

In the example above, we see the effect of implicit coercion. What R tries to do is find a way to represent all of the objects in the vector in a reasonable fashion. Sometimes this does exactly what you want and…sometimes not. For example, combining a numeric object with a character object will create a character vector, because numbers can usually be easily represented as strings.

EXPLICIT COERCION

Objects can be explicitly coerced from one class to another using the as.* functions, if available.

```
> x <- 0:6 > class(x)
[1] "integer"

> as.numeric(x)
[1] 0 1 2 3 4 5 6

> as.logical(x)
[1] FALSE TRUE TRUE TRUE TRUE TRUE TRUE

> as.character(x)
[1] "0" "1" "2" "3" "4" "5" "6"
```

Sometimes, R can't figure out how to coerce an object and this can result in NAs being produced.

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

```
> as.numeric(x)
Warning: NAs introduced by coercion
[1] NA NA NA

> as.logical(x)
[1] NA NA NA

> as.complex(x)
Warning: NAs introduced by coercion
[1] NA NA NA
```

When nonsensical coercion takes place, you will usually get a warning from R.

MATRICES

Matrices are vectors with a dimension attribute. The dimension attribute is itself an integer vector of length 2 (number of rows, number of columns)

```
> m <- matrix(nrow = 2, ncol = 3)
> m
     [,1] [,2] [,3]
[1,]   NA   NA   NA
[2,]   NA   NA   NA

> dim(m) [1] 2 3
> attributes(m) $dim
[1] 2 3
```

Matrices are constructed column-wise, so entries can be thought of starting in the “upper left” corner and running down the columns.

```
> m <- matrix(1:6, nrow = 2, ncol = 3)
> m
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Matrices can also be created directly from vectors by adding a dimension attribute.

```
> m <- 1:10
> m [1] 1 2 3 4 5 6 7 8 9 10
> dim(m) <- c(2, 5)
> m
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

Matrices can be created by column-binding or row-binding with the cbind() and rbind() functions.

```
> x <- 1:3
> y <- 10:12
> cbind(x, y)
          x y
[1,]      1 10
[2,]      2 11
[3,]      3 12

> rbind(x, y)
          [,1] [,2] [,3]
X          1    2    3
Y          10  11   12
```

LISTS

Lists are a special type of vector that can contain elements of different classes. Lists are a very important data type in R and you should get to know them well. Lists, in combination with the various "apply" functions discussed later, make for a powerful combination. Lists can be explicitly created using the list() function, which takes an arbitrary number of arguments.

```
> x <- list(1, "a", TRUE, 1 + 4i)
> x
[[1]]
[1] 1

[[2]]
[1] "a"

[[3]]
[1] TRUE

[[4]]
[1] 1+4i
```

We can also create an empty list of a prespecified length with the vector() function

```
> x <- vector("list", length = 5)
```

```
> x
[[1]] NULL
[[2]] NULL
[[3]] NULL
[[4]] NULL
[[5]] NULL
```

FACTORS

Factors are used to represent categorical data and can be unordered or ordered. One can think of a factor as an integer vector where each integer has a label. Factors are important in statistical modeling and are treated specially by modelling functions like lm() and glm().

Using factors with labels is better than using integers because factors are self-describing. Having a variable that has values “Male” and “Female” is better than a variable that has values 1 and 2.

Factor objects can be created with the factor() function.

```
> x <- factor(c("yes", "yes", "no", "yes", "no"))
> x [1] yes yes no yes no
Levels: no yes

> table(x)
X
no yes
 2 3

> ## See the underlying representation of factor

> unclass(x)
[1] 2 2 1 2 1
attr(,"levels")
[1] "no" "yes"
```

Often factors will be automatically created for you when you read a dataset in using a function like read.table(). Those functions often default to creating factors when they encounter data that look like characters or strings.

The order of the levels of a factor can be set using the levels argument to factor(). This can be important in linear modelling because the first level is used as the baseline level.

MISSING VALUES

Missing values are denoted by NA or NaN for q undefined mathematical operations.

- is.na() is used to test objects if they are NA
- is.nan() is used to test for NaN
- NA values have a class also, so there are integer NA, character NA, etc.
- A NaN value is also NA but the converse is not true

```
> ## Create a vector with NAs in it
> x <- c(1, 2, NA, 10, 3)
 > ## Return a logical vector indicating which elements are NA
> is.na(x)
[1] FALSE FALSE TRUE FALSE FALSE

> ## Return a logical vector indicating which elements are NaN
> is.nan(x)
[1] FALSE FALSE FALSE FALSE FALSE

> ## Now create a vector with both NA and NaN values
> x <- c(1, 2, NaN, NA, 4)
> is.na(x)
[1] FALSE FALSE TRUE TRUE FALSE

> is.nan(x)
[1] FALSE FALSE TRUE FALSE FALSE
```

DATA FRAMES

Data frames are used to store tabular data in R. They are an important type of object in R and are used in a variety of statistical modeling applications. Hadley Wickham's package dplyr[5] has an optimized set of functions designed to work efficiently with data frames.

Data frames are represented as a special type of list where every element of the list has to have the same length. Each element of the list can be thought of as a column and the length of each element of the list is the number of rows.

Unlike matrices, data frames can store different classes of objects in each column. Matrices must have every element be the same class (e.g. all integers or all numeric).

In addition to column names, indicating the names of the variables or predictors, data frames have a special attribute called row.names which indicate information about each row of the data frame.

Data frames are usually created by reading in a dataset using the read.table() or read.csv(). However, data frames can also be created explicitly with the data.frame() function or they can be coerced from other types of objects like lists.

Data frames can be converted to a matrix by calling data.matrix(). While it might seem that the as.matrix() function should be used to coerce a data frame to a matrix, almost always, what you want is the result of data.matrix().

```
> x <- data.frame(foo = 1:4, bar = c(T, T, F, F))

> x
  foo bar
1   1  TRUE
2   2  TRUE
3   3 FALSE
4   4 FALSE

> nrow(x)
[1] 4

> ncol(x)
[1] 2
```

NAMES

R objects can have names, which is very useful for writing readable code and self-describing objects. Here is an example of assigning names to an integer vector.

```
> x <- 1:3
> names(x) NULL
> names(x) <- c("New York", "Seattle", "Los Angeles")
> x New York Seattle Los Angeles 1 2 3
> names(x)
[1] "New York" "Seattle" "Los Angeles"
```

Lists can also have names, which is often very useful.

```
> x <- list("Los Angeles" = 1, Boston = 2, London = 3)
> x
$`Los Angeles`
[1] 1

$Boston
[1] 2

$London
[1] 3
```

```
> names(x)
[1] "Los Angeles" "Boston" "London"
```

Matrices can have both column and row names

```
> m <- matrix(1:4, nrow = 2, ncol = 2)
> dimnames(m) <- list(c("a", "b"), c("c", "d"))
> m
  c d
a 1 3
b 2 4
```

Column names and row names can be set separately using the colnames() and rownames() functions.

```
> colnames(m) <- c("h", "f")
> rownames(m) <- c("x", "z")
> m
  h f
x 1 3
z 2 4
```

Note that for data frames, there is a separate function for setting the row names, the row.names() function. Also, data frames do not have column names, they just have names (like lists). So to set the column names of a data frame just use the names() function. Yes, I know its confusing. Here's a quick summary:

=x=x=x=x=x=

GETTING DATA IN AND OUT OF R

There are a few principal functions reading data into R.

- read.table, read.csv, for reading tabular data
- readLines, for reading lines of a text file
- source, for reading in R code files (inverse of dump)
- dget, for reading in R code files (inverse of dput)
- load, for reading in saved workspaces
- unserialize, for reading single R objects in binary form

There are analogous functions for writing data to files

- write.table, for writing tabular data to text files (i.e. CSV) or connections
- writeLines, for writing character data line-by-line to a file or connection
- dump, for dumping a textual representation of multiple R objects
- dput, for outputting a textual representation of an R object
- save, for saving an arbitrary number of R objects in binary format (possibly compressed) to a file.
- serialize, for converting an R object into a binary format for outputting to a connection.

read.table()

The read.table() function is one of the most commonly used functions for reading data. The help file for read.table() is worth reading in its entirety if only because the function gets used a lot (run ?read.table in R). I know, I know, everyone always says to read the help file, but this one is actually worth reading.

The read.table() function has a few important arguments:

- file, the name of a file, or a connection
- header, logical indicating if the file has a header line
- sep, a string indicating how the columns are separated
- colClasses, a character vector indicating the class of each column in the dataset
- nrows, the number of rows in the dataset. By default read.table() reads an entire file.
- comment.char, a character string indicating the comment character. This defalts to "#". If there are no commented lines in your file, it's worth setting this to be the empty string "".

- skip, the number of lines to skip from the beginning
- stringsAsFactors, should character variables be coded as factors? This defaults to TRUE because back in the old days, if you had data that were stored as strings, it was because those strings represented levels of a categorical variable. Now we have lots of data that is text data and they don't always represent categorical variables. So you may want to set this to be FALSE in those cases. If you always want this to be FALSE, you can set a global option via options(stringsAsFactors = FALSE).

```
> data <- read.table("foo.txt")
```

Reading in Larger Datasets with read.table()

With much larger datasets, there are a few things that you can do that will make your life easier and will prevent R from choking.

- Read the help page for read.table, which contains many hints
- Make a rough calculation of the memory required to store your dataset (see the next section for an example of how to do this). If the dataset is larger than the amount of RAM on your computer, you can probably stop right here.

- Set comment.char = "" if there are no commented lines in your file.
- Use the colClasses argument. Specifying this option instead of using the default can make 'read.table' run MUCH faster, often twice as fast. In order to use this option, you have to know the class of each column in your data frame. If all of the columns are "numeric", for example, then you can just set colClasses = "numeric". A quick an dirty way to figure out the classes of each column is the following:

```
> initial <- read.table("datatable.txt", nrows = 100)
> classes <- sapply(initial, class)
> tabAll <- read.table("datatable.txt", colClasses = classes)
```

- Set nrows. This doesn't make R run faster but it helps with memory usage. A mild overestimate is okay. You can use the Unix tool wc to calculate the number of lines in a file.

In general, when using R with larger datasets, it's also useful to know a few things about your system.

- How much memory is available on your system?
- What other applications are in use? Can you close any of them?
- Are there other users logged into the same system?
- What operating system ar you using? Some operating systems can limit the amount of memory a single process can access

Using Textual and Binary Formats for Storing Data

Using dput() and dump()

One way to pass data around is by deparsing the R object with dput() and reading it back in (parsing it) using dget().

```
> ## Create a data frame
> y <- data.frame(a = 1, b = "a")
> ## Print 'dput' output to console
> dput(y)
structure(list(a = 1, b = structure(1L, .Label = "a", class = "factor")), .Names = c("a", "b"), row.names =
c(NA, -1L), class = "data.frame")
```

Notice that the dput() output is in the form of R code and that it preserves metadata like the class of the object, the row names, and the column names.

The output of dput() can also be saved directly to a file

```
> ## Send 'dput' output to a file
> dput(y, file = "y.R")
> ## Read in 'dput' output from a file
> new.y <- dget("y.R")
> new.y
   a b
 1 1 a
```

Multiple objects can be deparsed at once using the dump function and read back in using source.

```
> x <- "foo"
> y <- data.frame(a = 1L, b = "a")
```

We can dump() R objects to a file by passing a character vector of their names.

```
> dump(c("x", "y"), file = "data.R") > rm(x, y)
```

The inverse of dump() is source().

```
> source("data.R")
> str(y)
'data.frame':     1 obs. of 2 variables:
 $ a: int 1
$ b: Factor w/ 1 level "a": 1
> x
[1] "foo"
```

Binary Formats

The complement to the textual format is the binary format, which is sometimes necessary to use for efficiency purposes, or because there's just no useful way to represent data in a textual manner. Also, with numeric data, one can often lose precision when converting to and from a textual format, so it's better to stick with a binary format. The key functions for converting R objects into a binary format are save(), save.image(), and serialize(). Individual R objects can be saved to a file using the save() function.

```
> a <- data.frame(x = rnorm(100), y = runif(100))
> b <- c(3, 4.4, 1 / 3)
> ## Save 'a' and 'b' to a file
> save(a, b, file = "mydata.rda")
> ## Load 'a' and 'b' into your workspace > load("mydata.rda")
```

If you have a lot of objects that you want to save to a file, you can save all objects in your workspace using the save.image() function.

```
> ## Save everything to a file
> save.image(file = "mydata.RData")
> ## load all objects in this file
> load("mydata.RData")
```

Notice that I've used the .rda extension when using save() and the .RData extension when using save.image(). This is just my personal preference; you can use whatever file extension you want. The save() and save.image() functions do not care. However, .rda and .RData are fairly common extensions and you may want to use them because they are recognized by other software.

The serialize() function is used to convert individual R objects into a binary format that can be communicated across an arbitrary connection. This may get sent to a file, but it could get sent over a network or other connection.

When you call serialize() on an R object, the output will be a raw vector coded in hexadecimal format.

```
> x <- list(1, 2, 3)
> serialize(x, NULL)

[1] 58 0a 00 00 00 02 00 03 03 02 00 02 03 00 00 00 00 13 00 00 00 03 00
[24] 00 00 0e 00 00 00 01 3f f0 00 00 00 00 00 00 00 00 00 0e 00 00 00 01
[47] 40 00 00 00 00 00 00 00 00 00 00 0e 00 00 00 01 40 08 00 00 00 00 00
[70] 00
```

If you want, this can be sent to a file, but in that case you are better off using something like save(). The benefit of the serialize() function is that it is the only way to perfectly represent an R object in an exportable format, without losing precision or any metadata. If that is what you need, then serialize() is the function for you.

Reading Lines of a Text File

Text files can be read line by line using the readLines() function. This function is useful for reading text files that may be unstructured or contain non-standard data.

```
> ## Open connection to gz-compressed text file
> con <- gzfile("words.gz")
> x <- readLines(con, 10)
> x
[1] "1080" "10-point" "10th" "11-point" "12-point" "16-point"
[7] "18-point" "1st" "2" "20-point"
```

For more structured text data like CSV files or tab-delimited files, there are other functions like read.csv() or read.table().

The above example used the gzfile() function which is used to create a connection to files compressed using the gzip algorithm. This approach is useful because it allows you to read from a file without having to uncompress the file first, which would be a waste of space and time.

There is a complementary function writeLines() that takes a character vector and writes each element of the vector one line at a time to a text file.

Reading From a URL Connection

The readLines() function can be useful for reading in lines of webpages. Since web pages are basically text files that are stored on a remote server, there is conceptually not much difference between a web page and a local text file. However, we need R to negotiate the communication between your computer and the web server. This is what the url() function can do for you, by creating a url connection to a web server

```
> ## Open a URL connection for reading
> con <- url("http://www.jhsph.edu", "r")
> ## Read the web page
> x <- readLines(con)
```

=x=x=x=x=x=

Subsetting R Objects

There are three operators that can be used to extract subsets of R objects.

- The [operator always returns an object of the same class as the original. It can be used to select multiple elements of an object
- The [[operator is used to extract elements of a list or a data frame. It can only be used to extract a single element and the class of the returned object will not necessarily be a list or data frame.
- The $ operator is used to extract elements of a list or data frame by literal name. Its semantics are similar to that of [[.

Subsetting a Vector

Vectors are basic objects in R and they can be subsetted using the [operator.

```
> x <- c("a", "b", "c", "c", "d", "a")
> x[1] ## Extract the first element
[1] "a"

> x[2] ## Extract the second element
[1] "b"
```

The [operator can be used to extract multiple elements of a vector by passing the operator an integer sequence. Here we extract the first four elements of the vector.

```
> x[1:4]
[1] "a" "b" "c" "c"
```

The sequence does not have to be in order; you can specify any arbitrary integer vector.

```
> x[c(1, 3, 4)]
[1] "a" "c" "c"
```

Subsetting a Matrix

Matrices can be subsetted in the usual way with (i,j) type indices. Here, we create simple 2×3 matrix with the matrix function.

```
> x <- matrix(1:6, 2, 3)
> x
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

We can access the (1, 2) or the (2, 1) element of this matrix using the appropriate indices.

```
> x[1, 2]
[1] 3

> x[2, 1]
[1] 2
```

By default, when a single element of a matrix is retrieved, it is returned as a vector of length 1 rather than a 1×1 matrix. Often, this is exactly what we want, but this behavior can be turned off by setting drop = FALSE.

```
> x <- matrix(1:6, 2, 3)
> x[1, 2]
[1] 3

> x[1, 2, drop = FALSE]

     [,1]
[1,]   3
```

```
> x <- matrix(1:6, 2, 3)
> x[1, ]
[1] 1 3 5

> x[1, , drop = FALSE]
      [,1] [,2] [,3]
[1,]    1    3    5
```

Subsetting Lists

Lists in R can be subsetted using all three of the operators mentioned above, and all three are used for different purposes.

```
> x <- list(foo = 1:4, bar = 0.6)
> x

$foo
[1] 1 2 3 4

$bar
[1] 0.6
```

The [[operator can be used to extract single elements from a list. Here we extract the first element of the list.

```
> x[[1]]
[1] 1 2 3 4
```

The [[operator can also use named indices so that you don't have to remember the exact ordering of every element of the list. You can also use the $ operator to extract elements by name.

```
> x[["bar"]]
[1] 0.6

> x$bar
[1] 0.6
```

Notice you don't need the quotes when you use the $ operator

```
> x <- list(foo = 1:4, bar = 0.6, baz = "hello")
> name <- "foo"
> ## computed index for "foo"
> x[[name]]
[1] 1 2 3 4

> ## element "name" doesnâ€™t exist! (but no error here)
> x$name NULL

> ## element "foo" does exist
> x$foo
[1] 1 2 3 4
```

Removing NA Values

A common task in data analysis is removing missing values (NAs).

```
> x <- c(1, 2, NA, 4, NA, 5)
> bad <- is.na(x)
> print(bad)
[1] FALSE FALSE TRUE FALSE TRUE FALSE

> x[!bad]
[1] 1 2 4 5
```

What if there are multiple R objects and you want to take the subset with no missing values in any of those objects?

```
> x <- c(1, 2, NA, 4, NA, 5)
> y <- c("a", "b", NA, "d", NA, "f")

> good <- complete.cases(x, y)
> good
[1] TRUE TRUE FALSE TRUE FALSE TRUE

> x[good]
[1] 1 2 4 5

> y[good]
[1] "a" "b" "d" "f"
```

=x=x=x=x=x=

VECTORIZED OPERATIONS

Many operations in R are vectorized, meaning that operations occur in parallel in certain R objects. This allows you to write code that is efficient, concise, and easier to read than in non-vectorized languages.

The simplest example is when adding two vectors together.

```
> x <- 1:4
> y <- 6:9
> z <- x + y
> z
[1] 7 9 11 13
```

Natural, right? Without vectorization, you'd have to do something like

```
z <- numeric(length(x))
for(i in seq_along(x)) {
        z[i] <- x[i] + y[i] }
z
[1] 7 9 11 13
```

Vectorized Matrix Operations

Matrix operations are also vectorized, making for nicly compact notation. This way, we can do element-by-element operations on matrices without having to loop over every element.

```
> x <- matrix(1:4, 2, 2)
> y <- matrix(rep(10, 4), 2, 2)
> ## element-wise multiplication

> x * y
      [,1] [,2]
[1,]   10   30
[2,]   20   40
```

=x=x=x=x=x=

DATES AND TIMES

Dates in R

Dates are represented by the Date class and can be coerced from a character string using the as.Date() function. This is a common way to end up with a Date object in R.

```
> ## Coerce a 'Date' object from character
> x <- as.Date("1970-01-01")
> x
[1] "1970-01-01"
```

Times in R

Times are represented by the POSIXct or the POSIXlt class. POSIXct is just a very large integer under the hood. It use a useful class when you want to store times in something like a data frame. POSIXlt is a list underneath and it stores a bunch of other useful information like the day of the week, day of the year, month, day of the month. This is useful when you need that kind of information.

There are a number of generic functions that work on dates and times to help you extract pieces of dates and/or times:

- weekdays: give the day of the week
- months: give the month name
- quarters: give the quarter number (“Q1”, “Q2”, “Q3”, or “Q4”

Times can be coerced from a character string using the as.POSIXlt or as.POSIXct function.

```
> x <- Sys.time()
> x
[1] "2016-12-13 10:16:34 EST"
> class(x) ## 'POSIXct' object
[1] "POSIXct" "POSIXt"
```

The POSIXlt object contains some useful metadata.

```
> p <- as.POSIXlt(x)
> names(unclass(p))
[1] "sec" "min" "hour" "mday" "mon" "year" "wday"
[8] "yday" "isdst" "zone" "gmtoff"
> p$wday ## day of the week
[1] 2
```

You can also use the POSIXct format.

```
> x <- Sys.time()
> x ## Already in 'POSIXct' format
[1] "2016-12-13 10:16:34 EST"
> unclass(x) ## Internal representation
[1] 1481642194

> x$sec ## Can't do this with 'POSIXct'!
Error in x$sec: $ operator is invalid for atomic vectors

> p <- as.POSIXlt(x)
> p$sec ## That's better
[1] 34.46357
```

Finally, there is the strptime() function in case your dates are written in a different format. strptime() takes a character vector that has dates and times and converts them into to a POSIXlt object.

```
> datestring <- c("January 10, 2012 10:40", "December 9, 2011 9:10")
> x <- strptime(datestring, "%B %d, %Y %H:%M")
> x [1] "2012-01-10 10:40:00 EST" "2011-12-09 09:10:00 EST"
> class(x)
[1] "POSIXlt" "POSIXt"



> ## My local time zone
> x <- as.POSIXct("2012-10-25 01:00:00")
> y <- as.POSIXct("2012-10-25 06:00:00", tz = "GMT")
> y-x

Time difference of 1 hours
```

=X=X=X=X=X=

CONTROL STRUCTURES

Control structures in R allow you to control the flow of execution of a series of R expressions. Basically, control structures allow you to put some "logic" into your R code, rather than just always executing the same R code every time. Control structures allow you to respond to inputs or to features of the data and execute different R expressions accordingly. Commonly used control structures are

- if and else: testing a condition and acting on it
- for: execute a loop a fixed number of times
- while: execute a loop while a condition is true
- repeat: execute an infinite loop (must break out of it to stop)
- break: break the execution of a loop
- next: skip an interation of a loop

if-else

The if-else combination is probably the most commonly used control structure in R (or perhaps any language). This structure allows you to test a condition and act on it depending on whether it's true or false. For starters, you can just use the if statement.

```
if() {
 ## do something
}
```

```
if() {

## do something

}

else {

## do something else

}
```

for Loops

For loops are pretty much the only looping construct that you will need in R. While you may occasionally find a need for other types of loops, in my experience doing data analysis, I've found very few situations where a for loop wasn't sufficient.
In R, for loops take an interator variable and assign it successive values from a sequence or vector. For loops are most commonly used for iterating over the elements of an object (list, vector, etc.)

```
> for(i in 1:10) {
+ print(i)
+ }
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

This loop takes the i variable and in each iteration of the loop gives it values 1, 2, 3, …, 10, executes the code within the curly braces, and then the loop exits.

while Loops

While loops begin by testing a condition. If it is true, then they execute the loop body. Once the loop body is executed, the condition is tested again, and so forth, until the condition is false, after which the loop exits.

```
> count <- 0
> while(count < 10) {
+ print(count)
+ count <- count + 1
+ }
[1] 0
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
```

repeat Loops

repeat initiates an infinite loop right from the start. These are not commonly used in statistical or data analysis applications but they do have their uses. The only way to exit a repeat loop is to call break.

One possible paradigm might be in an iterative algorith where you may be searching for a solution and you don't want to stop until you're close enough to the solution. In this kind of situation, you often don't know in advance how many iterations it's going to take to get "close enough" to the solution.

```
x0 <- 1
tol <- 1e-8 repeat {
x1 <- computeEstimate()
if(abs(x1 - x0) < tol) {
          ## Close enough?
          break }
else {
          x0 <- x1
 }
}
```

next, break

next is used to skip an iteration of a loop.

```
for(i in 1:100) {
if(i <= 20) {
          ## Skip the first 20 iterations next
          }
 ## Do something here
 }
```

break is used to exit a loop immediately, regardless of what iteration the loop may be on.

```
for(i in 1:100) {
print(i)
if(i > 20) {
          ## Stop loop after 20 iterations break
          }
}
```

=x=x=x=x=x=

FUNCTIONS

The writing of a function allows a developer to create an interface to the code, that is explicitly specified with a set of parameters. This interface provides an abstraction of the code to potential users. This abstraction simplifies the users' lives because it relieves them from having to know every detail of how the code operates

Functions in R

Functions in R are "first class objects", which means that they can be treated much like any other R object. Importantly,

- Functions can be passed as arguments to other functions. This is very handy for the various apply functions, like lapply() and sapply().
- Functions can be nested, so that you can define a function inside of another function.

```
> f <- function() {
+ ## This is an empty function
+ }
> ## Functions have their own class
> class(f)
[1] "function"
> ## Execute this function
> f() NULL
```

=x=x=x=x=x=

SCOPING RULES OF R

Scoping Rules

The scoping rules for R are the main feature that make it different from the original S language (in case you care about that). This may seem like an esoteric aspect of R, but it's one of its more interesting and useful features.

The scoping rules of a language determine how a value is associated with a free variable in a function. R uses lexical scoping[2] or static scoping. An alternative to lexical scoping is dynamic scoping which is implemented by some languages.

Lexical scoping turns out to be particularly useful for simplifying statistical computations Related to the scoping rules is how R uses the search list to bind a value to a symbol Consider the following function.

```
> f <- function(x, y) {
+ x^2
+ y / z
+ }
```

Lexical vs. Dynamic Scoping

We can use the following example to demonstrate the difference between lexical and dynamic scoping rules.

```
 > y <- 10
> f <- function(x) {
+ y <- 2
+ y^2
+ g(x)
+ }
> g <- function(x) {
+ x*y
+ }
------------------
> g <- function(x) {
+ a <- 3
+ x+a
+y
+ ## 'y' is a free variable
+ }
> g(2)
Error in g(2): object 'y' not found
> y <- 3
> g(2) [1] 8
```

LOOP FUNCTIONS

Writing for and while loops is useful when programming but not particularly easy when working interactively on the command line. Multi-line expressions with curly braces are just not that easy to sort through when working on the command line.

R has some functions which implement looping in a compact form to make your life easier.

- lapply(): Loop over a list and evaluate a function on each element
- sapply(): Same as lapply but try to simplify the result
- apply(): Apply a function over the margins of an array
- tapply(): Apply a function over subsets of a vector
- mapply(): Multivariate version of lapply

lapply()

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").

This function takes three arguments: (1) a list X; (2) a function (or the name of a function) FUN; (3) other arguments via its ... argument. If X is not a list, it will be coerced to a list using as.list(). The body of the lapply() function can be seen here.

```
> lapply function (X, FUN, ...) {
FUN <- match.fun(FUN)
if (!is.vector(X) || is.object(X))
    X <- as.list(X) .Internal(lapply(X, FUN))
}
```

It's important to remember that lapply() always returns a list, regardless of the class of the input.

Here's an example of applying the mean() function to all elements of a list. If the original list has names, the the names will be preserved in the output.

```
> x <- list(a = 1:5, b = rnorm(10))
> lapply(x, mean)

 $a
[1] 3

$b
[1] 0.1322028
```

The lapply() function and its friends make heavy use of anonymous functions. Anonymous functions are like members of Project Mayhem²—they have no names. These are functions are generated "on the fly" as you are using lapply(). Once the call to lapply() is finished, the function disappears and does not appear in the workspace.

```
> x <- list(a = matrix(1:4, 2, 2), b = matrix(1:6, 3, 2))
> x

$a
     [,1] [,2]
[1,]    1    3
[2,]    2    4

$b     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

sapply()

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 it can't figure things out, a list is returned Here's the result of calling lapply().

```
> x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d = rnorm(100, 5))
> lapply(x, mean)
$a
[1] 2.5

$b
[1] -0.251483

$c
[1] 1.481246

$d
[1] 4.968715
```

Notice that lapply() returns a list (as usual), but that each element of the list has length 1. Here's the result of calling sapply() on the same list

```
> sapply(x, mean)
A           b           c           d
2.500000   -0.251483    1.481246    4.968715
```

Because the result of lapply() was a list where each element had length 1, sapply() collapsed the output into a numeric vector, which is often more useful than a list.

split()

The split() function takes a vector or other objects and splits it into groups determined by a factor or list of factors.

The arguments to split() are

> str(split) function (x, f, drop = FALSE, ...)

where

- x is a vector (or list) or data frame
- f is a factor (or coerced to one) or a list of factors
- drop indicates whether empty factors levels should be dropped

The combination of split() and a function like lapply() or sapply() is a common paradigm in R.

SPLITTING A DATA FRAME

```
> library(datasets)
> head(airquality)
```

	Ozone	Solar.R	Wind	Temp	Month	Day
1	41	190	7.4	67	5	1
2	36	118	8.0	72	5	2
3	12	149	12.6	74	5	3
4	18	313	11.5	62	5	4
5	NA	NA	14.3	56	5	5
6	28	NA	14.9	66	5	6

We can split the airquality data frame by the Month variable so that we have separate sub-data frames for each month.

> s <- split(airquality, airquality$Month) > str(s)

List of 5

$ 5:'data.frame': 31 obs. of 6 variables:

..$ Ozone : int [1:31] 41 36 12 18 NA 28 23 19 8 NA ...
..$ Solar.R: int [1:31] 190 118 149 313 NA NA 299 99 19 194 ...
..$ Wind : num [1:31] 7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 20.1 8.6 ...
..$ Temp : int [1:31] 67 72 74 62 56 66 65 59 61 69 ...
..$ Month : int [1:31] 5 5 5 5 5 5 5 5 5 5 ...
..$ Day : int [1:31] 1 2 3 4 5 6 7 8 9 10 ...

$ 6:'data.frame': 30 obs. of 6 variables:

..$ Ozone : int [1:30] NA NA NA NA NA NA 29 NA 71 39 ...
..$ Solar.R: int [1:30] 286 287 242 186 220 264 127 273 291 323 ...
..$ Wind : num [1:30] 8.6 9.7 16.1 9.2 8.6 14.3 9.7 6.9 13.8 11.5 ...
..$ Temp : int [1:30] 78 74 67 84 85 79 82 87 90 87 ...
..$ Month : int [1:30] 6 6 6 6 6 6 6 6 6 6 ...
..$ Day : int [1:30] 1 2 3 4 5 6 7 8 9 10 ...

$ 7:'data.frame': 31 obs. of 6 variables:

..$ Ozone : int [1:31] 135 49 32 NA 64 40 77 97 97 85 ...
..$ Solar.R: int [1:31] 269 248 236 101 175 314 276 267 272 175 ...
..$ Wind : num [1:31] 4.1 9.2 9.2 10.9 4.6 10.9 5.1 6.3 5.7 7.4 ...
..$ Temp : int [1:31] 84 85 81 84 83 83 88 92 92 89 ...
..$ Month : int [1:31] 7 7 7 7 7 7 7 7 7 7 ...
..$ Day : int [1:31] 1 2 3 4 5 6 7 8 9 10 ...

$ 8:'data.frame': 31 obs. of 6 variables:

..$ Ozone : int [1:31] 39 9 16 78 35 66 122 89 110 NA ...
..$ Solar.R: int [1:31] 83 24 77 NA NA NA 255 229 207 222 ...
..$ Wind : num [1:31] 6.9 13.8 7.4 6.9 7.4 4.6 4 10.3 8 8.6 ...
..$ Temp : int [1:31] 81 81 82 86 85 87 89 90 90 92 ...
..$ Month : int [1:31] 8 8 8 8 8 8 8 8 8 8 ...
..$ Day : int [1:31] 1 2 3 4 5 6 7 8 9 10 ...

$ 9:'data.frame': 30 obs. of 6 variables:

..$ Ozone : int [1:30] 96 78 73 91 47 32 20 23 21 24 ...
..$ Solar.R: int [1:30] 167 197 183 189 95 92 252 220 230 259 ...
..$ Wind : num [1:30] 6.9 5.1 2.8 4.6 7.4 15.5 10.9 10.3 10.9 9.7 ...
..$ Temp : int [1:30] 91 92 93 93 87 84 80 78 75 73 ...
..$ Month : int [1:30] 9 9 9 9 9 9 9 9 9 9 ...
..$ Day : int [1:30] 1 2 3 4 5 6 7 8 9 10 ...

tapply()

tapply() is used to apply a function over subsets of a vector. It can be thought of as a combination of split() and sapply() for vectors only. I've been told that the "t" in tapply() refers to "table", but that is unconfirmed.

> str(tapply) function (X, INDEX, FUN = NULL, ..., simplify = TRUE)

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?

apply()

The apply() function is used to a evaluate a function (often an anonymous one) over the margins of an array. It is most often used to apply a function to the rows or columns of a matrix (which is just a 2-dimensional array). However, it can be used with general arrays, for example, to take the average of an array of matrices. Using apply() is not really faster than writing a loop, but it works in one line and is highly compact.

> str(apply)
function (X, MARGIN, FUN, ...)

The arguments to apply() are

- X is an array
- MARGIN is an integer vector indicating which margins should be "retained".
- FUN is a function to be applied
- ... is for other arguments to be passed to FUN

Col/Row Sums and Means

For the special case of column/row sums and column/row means of matrices, we have some useful shortcuts.

- rowSums = apply(x, 1, sum)
- rowMeans = apply(x, 1, mean)
- colSums = apply(x, 2, sum)
- colMeans = apply(x, 2, mean)

mapply()

The mapply() function is a multivariate apply of sorts which applies a function in parallel over a set of arguments. Recall that lapply() and friends only iterate over a single R object. What if you want to iterate over multiple R objects in parallel? This is what mapply() is for.

```
> str(mapply)
function (FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE)
```

The arguments to mapply() are

- FUN is a function to apply
- ... contains R objects to apply over
- MoreArgs is a list of other arguments to FUN.
- SIMPLIFY indicates whether the result should be simplified

The mapply() function has a different argument order from lapply() because the function to apply comes first rather than the object to iterate over. The R objects over which we apply the function are given in the ... argument because we can apply over an arbitrary number of R objects.

```
list(rep(1, 4), rep(2, 3), rep(3, 2), rep(4, 1))
```

With mapply(), instead we can do

```
> mapply(rep, 1:4, 4:1)
[[1]]
[1] 1 1 1 1

[[2]]
[1] 2 2 2

[[3]]
[1] 3 3

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

=x=x=x=x=x=

DEBUGGING

R provides a number of tools to help you with debugging your code. The primary tools for debugging functions in R are

• traceback(): prints out the function call stack after an error occurs; does nothing if there's no error

• debug(): flags a function for "debug" mode which allows you to step through execution of a function one line at a time

• browser(): suspends the execution of a function wherever it is called and puts the function in debug mode

• trace(): allows you to insert debugging code into a function a specific places
• recover(): allows you to modify the error behavior so that you can browse the function call stack

=x=x=x=x=x=

PROFILING R CODE

The basic principles of optimizing your code are:

- Design first, then optimize
- Remember: Premature optimization is the root of all evil
- Measure (collect data), don't guess.
- If you're going to be scientist, you need to apply the same principles here!

Using system.time()

They system.time() function takes an arbitrary R expression as input (can be wrapped in curly braces) and returns the amount of time taken to evaluate the expression. The system.time() function computes the time (in seconds) needed to execute an expression and if there's an error, gives the time until the error occurred. The function returns an object of class proc_time which contains two useful bits of information:

- user time: time charged to the CPU(s) for this expression
- elapsed time: "wall clock" time, the amount of time that passes for you as you're sitting there

```
## Elapsed time > user time
system.time(readLines("http://www.jhsph.edu"))
   user  system elapsed
  0.004   0.002   0.431


## Elapsed time < user time
> hilbert <- function(n) {
+ i <- 1:n
+ 1 / outer(i - 1, i, "+")
+ }

> x <- hilbert(1000)
> system.time(svd(x))

   user  system elapsed
  1.035   0.255   0.462
```

THE R PROFILER

Using system.time()

It allows you to test certain functions or code blocks to see if they are taking excessive amounts of time. However, this approach assumes that you already know where the problem is and can call system.time() on it that piece of code. What if you don't know where to start?

> Rprof() ## Turn on the profiler

> Rprof(NULL) ## Turn off the profiler

Using summaryRprof()

The summaryRprof() function tabulates the R profiler output and calculates how much time is spent in which function. There are two methods for normalizing the data.

- "by.total" divides the time spend in each function by the total run time
- "by.self" does the same as "by.total" but first subtracts out time spent in functions above the current function in the call stack. I personally find this output to be much more useful.

=x=x=x=x=x=

GENERATING RANDOM NUMBERS

R comes with a set of pseudo-random number generators that allow you to simulate from wellknown probability distributions like the Normal, Poisson, and binomial. Some example functions for probability distributions in R

- rnorm: generate random Normal variates with a given mean and standard deviation
- dnorm: evaluate the Normal probability density (with a given mean/SD) at a point (or vector of points)

- pnorm: evaluate the cumulative distribution function for a Normal distribution
- rpois: generate random Poisson variates with a given rate

For each probability distribution there are typically four functions available that start with a "r", "d", "p", and "q". The "r" function is the one that actually simulates randon numbers from that distribution. The other functions are prefixed with a.

- d for density
- r for random number generation
- p for cumulative distribution
- q for quantile function (inverse cumulative distribution)

```
dnorm(x, mean = 0, sd = 1, log = FALSE)
pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
rnorm(n, mean = 0, sd = 1)
```

Setting the random number seed

When simulating any random numbers it is essential to set the random number seed. Setting the random number seed with set.seed() ensures reproducibility of the sequence of random numbers. For example, I can generate 5 Normal random numbers with rnorm().

```
> set.seed(1)
> rnorm(5)
[1] -0.6264538 0.1836433 -0.8356286 1.5952808 0.3295078
```

Note that if I call rnorm() again I will of course get a different set of 5 random numbers.