Modern R in a Corporate Environment

Original materials developed for RADARS

Brian Davis

2018-04-24

Contents

About

0.1 Useful Resources

- R Programming for Data Science by Roger D. Peng,
- Efficient R programming by Colin Gillespie & Robin Lovelace,
- Course on R debugging and robust programming by Laurent Gatto & Robert Stojnic,
- Mastering Software Development in R by Roger D. Peng, Sean Kross and Brooke Anderson,
- R for Data Science by Garrett Grolemund & Hadley Wickham
- Advanced R by Hadley Wickham
- R packages by Hadley Wickham,
- other resources linked from this material.

Chapter 1

Introduction

> Something that will make life easier in the long-run can be the most difficult thing to do today. For coders, prioritising the long term may involve an overhaul of current practice and the learning of a new skill.

1.1 Course Philosophy

> "The best programs are written so that computing machines can perform them quickly and so that human beings can understand them clearly. A programmer is ideally an essayist who works with traditional aesthetic and literary forms as well as mathematical concepts, to communicate the way that an algorithm works and to convince a reader that the results will be correct." Donald Knuth

1.1.1 Reproducible Research

Reproducible research is the idea that data analyses, and more generally, scientific claims, are published with their data and software code so that others may verify the findings and build upon them. There are two basic reasons to be concerned about making your research reproducible. The first is *to show evidence of the correctness of your results.* The second reason to aspire to reproducibility is *to enable others to make use of our methods and results.*

Modern challenges of reproducibility in research, particularly computational reproducibility, have produced a lot of discussion in papers, blogs and videos, some of which are listed here and here.

> Conclusions in experimental psychology often are the result of null hypothesis significance testing. Unfortunately, there is evidence ((from eight major psychology journals published between 1985 and 2013) that roughly half of all published empirical psychology articles contain at least one inconsistent p-value, and around one in eight articles contain a grossly inconsistent p-value that makes a non-significant result seem significant, or vice versa. statscheck and here

> "A key component of scientific communication is sufficient information for other researchers in the field to reproduce published findings. For computational and data-enabled research, this has often been interpreted to mean making available the raw data from which results were generated, the computer code that generated the findings, and any additional information needed such as workflows and input parameters. Many journals are revising author guidelines to include data and code availability. We chose a random sample of 204 scientific papers published in the journal **Science** after the implementation of their policy in February 2011. We found that were able to

> reproduce the findings for 26%." Proceedings of the National Academy of Sciences of the United States of America
>
> "Starting September 1 2016, JASA ACS will require code and data as a minimum standard for reproducibility of statistical scientific research." JASA

1.1.2 FDA Validation

> "Establishing documented evidence which provides a high degree of assurance that a specific process will consistently produce a product meeting its predetermined specifications and quality attributes." -Validation as defined by the FDA in **Validation of Systems for 21 CFR Part 11 Compliance**

1.1.3 The SAS Myth

Contray to what we hear the FDA does not require SAS to be used *EVER*. There are instances that you have to deliver data in XPORT format though which is open and implemented in many programming languages.

> "FDA does not require use of any specific software for statistical analyses, and statistical software is not explicitly discussed in Title 21 of the Code of Federal Regulations [e.g., in 21CFR part 11]. However, the software package(s) used for statistical analyses should be fully documented in the submission, including version and build identification. As noted in the FDA guidance, E9 Statistical Principles for Clinical Trials" FDA Statistical Software Clarifying Statement

Good write up with links to several FDA talks on the subject.

1.2 Prerequisites

- We will assume you have minimal experience and knowledge of R
- IT should have installed:
  - R version 3.5
  - RStudio version 1.1
  - MiTeX
  - RTools version 3.4
- We will install other dependencies throughout the course.

1.3 Content

It is impossible to become an expert in R in only one course even a multi-week one. Yet, this course aims at giving a wide understanding on many aspects of R as used in a corporate / production environment. It will roughly be based on R for Data Science. While this is an *excellent* resource it does not cover much of what we will need on a routine basis. Some external resources will be referred to in this book for you to be able to deepen what you would have learned in this course.

This is your course so if you feel we need to hit an area deeper, or add content based on a current need, let me know an we will work to adjust it.

The **rough** topic list of the course:

1. Good programming practices
2. Basics of R Programming
3. Importing Data

4. Tiyding Data
5. Visualizing Data
6. Functions
7. Strings
8. Dates and Time
9. Communicating Results

Making Code Production Ready:

10. Functions (part II)
11. Assertions
12. Unit tests
13. Documentation
14. Communicating Results (part II)

1.4 Structure

My current thoughts are to meet an hour a week and discuss a topic. We will not be going strictly through the R4DS, but will use it as our foundation into the topic at hand. Then give an assignment due for the next week which we go over the solutions. We will incorporate these assignments into a RADARS R package(s?) so we will have a collection of usefull reusable code for the future.

Open to other ideas as we go along. I'm going to try to keep the assignments related to our current work (maybe working through Site Investigator and/or Subscriber Reports) so we can work on the class during work hours.

Chapter 2

Good practices

> "Programs must be written for people to read, and only incidentally for machines to execute." Harold Abelson
>
> "Programming is the art of telling another human being what one wants the computer to do." Donald Knuth
>
> "Let us change our traditional attitude to the construction of programs. Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do." Donald Knuth
>
> "When you write a program, think of it primarily as a work of literature. You're trying to write something that human beings are going to read. Don't think of it primarily as something a computer is going to follow. The more effective you are at making your program readable, the more effective it's going to be: You'll understand it today, you'll understand it next week, and your successors who are going to maintain and modify it will understand it."

2.1 Coding style

Good coding style is like correct punctuation: you can manage without it, butitsuremakesthingseasiertoread. When I answer questions; first, I read the title of the question to see if I can answer the question, secondly, I check the coding style of the question and if the code is too difficult to read, I just move on. Please make your code readable by following e.g. this coding style (most examples below come from this guide).

2.1.1 Comments

In code, use comments to explain the "why" not the "what" or "how". Each line of a comment should begin with the comment symbol and a single space: `#`.

2.1.2 Naming

> There are only two hard things in Computer Science: cache invalidation and naming things. – Phil Karlton

Names are not limited to 8 characters as in some other languages, however they are case sensitive. Be smart with your naming; be descriptive yet concise. Think about how your names will show up in autocomplete.

Throughout the course we will point out some standard naming conventions that are used in R (and other languages). (Ex. `i` and `j` as row and column indicies)

```
# Good
average_height <- mean((feet / 12) + inches)
plot(mtcars$disp, mtcars$mpg)

# Bad
ah<-mean(x/12+y)
plot(mtcars[, 3], mtcars[, 1])
```

2.1.3 Structure

Use commented lines of `-` to create a code outline.

2.1.4 Spacing

Put a space before and after `=` when naming arguments in function calls. Most infix operators (`==`, `+`, `-`, `<-`, etc.) are also surrounded by spaces, except those with relatively high precedence: `^`, `:`, `::`, and `:::`. Always put a space after a comma, and never before (just like in regular English).

```
# Good
average <- mean((feet / 12) + inches, na.rm = TRUE)
sqrt(x^2 + y^2)
x <- 1:10
base::sum

# Bad
average<-mean(feet/12+inches,na.rm=TRUE)
sqrt(x ^ 2 + y ^ 2)
x <- 1 : 10
base :: sum
```

2.1.5 Indenting

Curly braces, `{}`, define the the most important hierarchy of R code. To make this hierarchy easy to see, always indent the code inside `{}` by two spaces.

```
# Good
if (y < 0 && debug) {
  message("y is negative")
}

if (y == 0) {
  if (x > 0) {
    log(x)
  } else {
    message("x is negative or zero")
  }
} else {
  y ^ x
}
```

```
# Bad
if (y < 0 && debug)
message("Y is negative")

if (y == 0)
{
    if (x > 0) {
      log(x)
    } else {
  message("x is negative or zero")
    }
} else { y ^ x }
```

2.1.6 Long lines

Strive to limit your code to 80 characters per line. This fits comfortably on a printed page with a reasonably sized font. If you find yourself running out of room, this is a good indication that you should encapsulate some of the work into a separate function.

If a function call is too long to fit on a single line, use one line each for the function name, each argument, and the closing `)`. This makes the code easier to read and to change later.

```
# Good
do_something_very_complicated(
  something = "that",
  requires  = many,
  arguments = "some of which may be long"
)

# Bad
do_something_very_complicated("that", requires, many, arguments,
                              "some of which may be long"
```

2.1.7 Other

- Use `<-`, not `=`, for assignment. Keep `=` for parameters.

```
# Good
x <- 5
system.time(
  x <- rnorm(1e6)
)

# Bad
x = 5
system.time(
  x = rnorm(1e6)
)
```

- Don't put `;` at the end of a line, and don't use `;` to put multiple commands on one line.
- Only use `return()` for early returns. Otherwise rely on R to return the result of the last evaluated expression.

```r
# Good
add_two <- function(x, y) {
  x + y
}

# Bad
add_two <- function(x, y) {
  return(x + y)
}
```

- Use ", not ', for quoting text. The only exception is when the text already contains double quotes and no single quotes.

```r
# Good
"Text"
'Text with "quotes"'
'<a href="http://style.tidyverse.org">A link</a>'

# Bad
'Text'
'Text with "double" and \'single\' quotes'
```

2.2 Coding practices

2.2.1 Variables

Create variables for values that are likely to change.

2.2.2 *Rule of 3*[1]

Try not to copy code, or copy then modify the code, more than twice.

- If a change requires you to search/replace 3 or more times make a variable.
- If you copy a code chunk 3 or more times *make a function*
- If you copy a function 3 or more times *make your function more generic*
- If you copy a function 3 or more times into a project *make a package*
- If 3 or more people will use the function *make a package*
- If 3 or more projects will use the function *make a package*

Same thing goes for lookup tables and such. The key thing to think about is; if something changes how many touch points will there be? If it is 3 or more places it is time to abstract this code a bit.

2.2.3 Path names

It is better to use relative path names instead of hard coded ones. If you must read from (or write to) paths that are not in your project directory structure create a file name variable at the highest level you can (*always end with the /*) and then use relative paths.
DO NOT EVER USE `setwd()`

[1] This is sometimes called the DRY principle, or Don't Repeat Yourself.

```
# Good
raw_data <- read.csv("./data/mydatafile.csv")

input_file <- "./data/mydatafile.csv"
raw_data <- read.csv(input_file)

input_path <- "C:/Path/To/Some/other/project/directory/"
input_file <- paste0(input_path, "data/mydatafile.csv")
raw_data <- read.csv(input_file)

# Bad
setwd("C:/Path/To/Some/other/project/directory/data/")
raw_data <- read.csv("mydatafile.csv")
setwd("C:/Path/back/to/my/project/")
```

2.3 RStudio

Download the latest version of RStudio (> 1.1) and use it!

Learn more about new features of RStudio v1.1 there.

RStudio features:

- everything you can expect from a good IDE
- keyboard shortcuts I use frequently
  1. *Ctrl + Space* (auto-completion, better than *Tab*)
  2. *Ctrl + Up* (command history & search)
  3. *Ctrl + Enter* (execute line of code)
  4. *Ctrl + Shift + A* (reformat code)
  5. *Ctrl + Shift + C* (comment/uncomment selected lines)
  6. *Ctrl + Shift + /* (reflow comments)
  7. *Ctrl + Shift + O* (View code outline)
  8. *Ctrl + Shift + B* (build package, website or book)
  9. *Ctrl + Shift + M* (pipe)
  10. *Alt + Shift + K* to see all shortcuts…
- Panels (everything is integrated, including **Git** and a terminal)
- Interactive data importation from files and connections (see this webinar)
- Use code diagnostics:
- **R Projects**:
  - **Meaningful structure** in one folder
  - The working directory automatically switches to the project's folder
  - File tab displays the associated files and folders in the project
  - History of R commands and open files
  - Any settings associated with the project, such as Git settings, are loaded. Note that a *set-up.R* or even a *.Rprofile* file in the project's root directory enable project-specific settings to be loaded each time people work on the project.

> The only two things that make @JennyBryan . Instead use projects + here::here() #rstats pic.twitter.com/GwxnHePL4n
>
> — Hadley Wickham (@hadleywickham) December 11 2017

Read more at https://www.tidyverse.org/articles/2017/12/workflow-vs-script/ and also see chapter *Efficient set-up* of book *Efficient R programming*.

2.4 Getting help

2.4.1 Help yourself, learn how to debug

A basic solution is to print everything, but it usually does not work well on complex problems. A convenient solution to see all the variables' states in your code is to place some `browser()` anywhere you want to check the variables' states.

Learn more with this book chapter, this other book chapter, this webinar and this RStudio article.

2.4.2 External help

Can't remember useful functions? Use cheat sheets.

You can search for specific R stuff on https://rseek.org/. You should also read documentations carefully. If you're using a package, search for vignettes and a GitHub repository.

You can also use Stack Overflow. The most common use of Stack Overflow is when you have an error or a question, you google it, and most of the times the first links are Q/A on Stack Overflow.

You can ask questions on Stack Overflow (using the tag `r`). You need to make a great R reproducible example if you want your question to be answered. Most of the times, while making this reproducible example, you will find the answer to your problem.

If you're confident enough in your R skills, you can go to the next step and answer questions on Stack Overflow. It's a good way to increase your skills, or just to procrastinate while writing a scientific manuscript.

2.5 Keeping up to date

With over 10,000 packages on CRAN it is hard to keep up with the constantly changing landscape. R-Bloggers is an R forcused blog aggregator with dozens of posts per day. Checkit out.

Join the R-help mailing list. Sign up to get the daily digest and scan it for questions that interest you.

2.6 Assignement

1. See these Rstudio Tips & Tricks or these and find one that looks interesting and **practice** it all week.
2. Create an R Project for this class.
3. Create the following directories in your project (tip sheet?)
   - Bonus points if you can do it from R and not RStudio or Windows Explorer
   - Double Bonus points if you can make it a function.
4. Read Chapters 1-3 of the Tidyverse Style Guide
5. Copy one of your R scripts into your R directory. (Bonus points if you can do it from R and not RStudio or Windows Explorer)
6. Apply the style guide to your code.

7. Apply the "Rule of 3"
   - Create variables as needed

- Identify code that is used 3 or more times to make functions
- Identify code that would be useful in 3 or more projects to integrate into a package.

8. Read how to make a great R reproducible example

Chapter 3

R Basics

With over 10,000 packages on CRAN we can't cover everything. In general there are several ways, or packages, to accomplish a given task.

Here is a quick look at some of the basics. Next we'll dive deep into R's basic data structures and how to subset them in subsequent chapters. This will give us a good overview of base R and the background needed to dive into **R for Data Science**.

The three most important functions in R `?`, `??`, and `str`:

- `?topic` provides access to the documentation for *topic*.
- `??topic` searches the documention for *topic*.
- `str` displays the structure of an R object in human readable form.

See this vocabulary list for a good starting point on the basics functions in base R and some important libraries.

A book to learn the basics is R Programmig for Data Science

In R there three basic constructs; objects, functions, and environments:

3.1 Assignment Operators

We saw this is Coding Style. Use `<-` for assignment and use `=` for parameters. While you can use `=` for assignment it is generally considered bad practice.

3.2 Objects

3.2.1 Vector

You create a vector with `c`.

```
v <- c("my", "first", "vector")
v
```

```
#> [1] "my"     "first"  "vector"
```

```
# length of our vector
length(v)
```

```
#> [1] 3
```

There are several shortcut functions for common vector creation.

```
# create an ordered sequence
2:10
```

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

```
9:3
```

```
#> [1] 9 8 7 6 5 4 3
```

```
# generate regular sequences
seq(1, 5, by = 3)
```

```
#> [1] 1 4
```

```
# replicate a number n times
rep(3, times = 4)
```

```
#> [1] 3 3 3 3
```

```
# common mistake using 1:length(n) in loops
# but if n = 0
1:0
```

```
#> [1] 1 0
```

```
# use seq_len(n) instead and the loop won't execute
seq_len(0)
```

```
#> integer(0)
```

```
# another common mistake
n <- 6
1:n+1        # is (1:n) + 1, so 2:(n + 1)
```

```
#> [1] 2 3 4 5 6 7
```

```
1:(n+1)      # usually what is meant
```

```
#> [1] 1 2 3 4 5 6 7
```

```
seq_len(n+1) # another way
```

```
#> [1] 1 2 3 4 5 6 7
```

3.2.2 Matrix

Matrices are 2D vectors, with all elements of the same type. Genearlly used for mathematics.

```
# fill in column order (default)
matrix(1:12, nrow = 3)
```

```
#>      [,1] [,2] [,3] [,4]
#> [1,]    1    4    7   10
#> [2,]    2    5    8   11
#> [3,]    3    6    9   12
```

```
# fill in row order
matrix(1:12, nrow = 3, byrow = TRUE)
```

```
#>      [,1] [,2] [,3] [,4]
#> [1,]    1    2    3    4
#> [2,]    5    6    7    8
#> [3,]    9   10   11   12
```

```
# can also specify the number of columns instead
matrix(1:12, ncol = 3)
```

```
#>      [,1] [,2] [,3]
#> [1,]    1    5    9
#> [2,]    2    6   10
#> [3,]    3    7   11
#> [4,]    4    8   12
```

You find the dimensions of a matrix with `nrow`, `ncol`, and `dim`

```
m <- matrix(1:12, ncol = 3)
dim(m)
```

```
#> [1] 4 3
```

```
nrow(m)
```

```
#> [1] 4
```

```
ncol(m)
```

```
#> [1] 3
```

3.2.3 List

A list is a generic vector containing other objects. These do **NOT** have to be the same type or the same length.

```
s <- c("aa", "bb", "cc", "dd", "ee")
b <- c(TRUE, FALSE, TRUE, FALSE, FALSE)
# x contains copies of n, s, b and our matrix from above
x <- list(n = c(2, 3, 5) , s, b, 3, m)
x
```

```
#> $n
#> [1] 2 3 5
#>
#> [[2]]
#> [1] "aa" "bb" "cc" "dd" "ee"
#>
#> [[3]]
#> [1]  TRUE FALSE  TRUE FALSE FALSE
#>
#> [[4]]
#> [1] 3
#>
#> [[5]]
#>      [,1] [,2] [,3]
#> [1,]    1    5    9
#> [2,]    2    6   10
#> [3,]    3    7   11
#> [4,]    4    8   12
```

```
# length gives you length of the list not the elements in the list
length(x)
```

```
#> [1] 5
```

We'll discuss lists in detail in the next chapter.

3.2.4 Data frame

A data frame is a list with each vector of the same length. This is the main data structure used and is analagous to a data set in SAS. While these **look** like matrices they behave very different.

```
df = data.frame(n = c(2, 3, 5),
                s = c("aa", "bb", "cc"),
                b = c(TRUE, FALSE, TRUE),
                y = v
                )       # df is a data frame
df
```

```
#>   n  s     b      y
#> 1 2 aa  TRUE     my
#> 2 3 bb FALSE  first
#> 3 5 cc  TRUE vector
```

```
# dimensions
dim(df)
```

```
#> [1] 3 4
```

```
nrow(df)
```

```
#> [1] 3
```

```
ncol(df)
```

```
#> [1] 4
```

```
length(df)
```

```
#> [1] 4
```

We'll discuss data frames in great detail in the next chapter.

3.3 Comparision

Logical Operators include:

Operator	Description
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
==	exactly equal to
!=	not equal to

```
v <- 1:12
v[v > 9]
```

```
#> [1] 10 11 12
```

Equality can be tricky to test for since real numbers can't be expressed exactly in computers.

```
x <- sqrt(2)
(y <- x^2)
```

```
#> [1] 2
```

```
y == 2
```

```
#> [1] FALSE
```

```
print(y, digits = 20)
```

```
#> [1] 2.0000000000000004
```

```
all.equal(y, 2)          ## equality with some tolerance
```

```
#> [1] TRUE
```

```
all.equal(y, 3)
```

```
#> [1] "Mean relative difference: 0.5"
```

```
isTRUE(all.equal(y, 3))  ## if you want a boolean, use isTRUE()
```

```
#> [1] FALSE
```

3.4 Logical and sets

```
x <- c(TRUE, FALSE)
df <- data.frame(expand.grid(x, x))
names(df) <- c("x", "y")
df$and  <- df$x & df$y     # logical and
df$or   <- df$x | df$y     # logical or
df$notx <- !df$x           # negation
df$xor  <- xor(df$x, df$y) # exlusive or
df
```

```
#>       x     y   and    or  notx   xor
#> 1  TRUE  TRUE  TRUE  TRUE FALSE FALSE
#> 2 FALSE  TRUE FALSE  TRUE  TRUE  TRUE
#> 3  TRUE FALSE FALSE  TRUE FALSE  TRUE
#> 4 FALSE FALSE FALSE FALSE  TRUE FALSE
```

R has two versions of the logical operators `&` and `&&` (`|` and `||`). The single version is the vectorized version while the the double version returns a length-one vector. Use the double version in logical control structures (if, for, while, etc).

```
# TRUE/FALSE and each element
TRUE & c(TRUE, FALSE)
```

```
#> [1]  TRUE FALSE
```

```
FALSE & c(TRUE, FALSE)
```

```
#> [1] FALSE FALSE
```

```
# TRUE/FALSE and first element
TRUE && c(TRUE, FALSE)
```

```
#> [1] TRUE
```

```
FALSE && c(TRUE, FALSE)
```

```
#> [1] FALSE
```

```
# TRUE/FALSE or each element
TRUE | c(TRUE, FALSE)
```

```
#> [1] TRUE TRUE
```

```
FALSE | c(TRUE, FALSE)
```

```
#> [1]  TRUE FALSE
```

```
# TRUE/FALSE or first element
TRUE || c(TRUE, FALSE)
```

```
#> [1] TRUE
```

```
FALSE || c(TRUE, FALSE)
```

```
#> [1] TRUE
```

This is a common source of bugs in control structures (if, for, while, etc) where you must have a single TRUE / FALSE.

Also, note that `=` is used for assignment and not comparison `==`.

It also has useful helpers `any` and `all`

```
x <- c(FALSE, FALSE, FALSE, TRUE)
any(x)
```

```
#> [1] TRUE
```

```
all(x)
```

```
#> [1] FALSE
```

```
all(!x[1:3])
```

```
#> [1] TRUE
```

And also some useful set operations intersect, union, setdiff, setequal

```
x <- 1:5
y <- 3:7

intersect(x, y) # in x and in y
```

```
#> [1] 3 4 5
```

```
union(x, y)     # different than c()
```

```
#> [1] 1 2 3 4 5 6 7
```

```
c(x,y)          # not a set operation
```

```
#>  [1] 1 2 3 4 5 3 4 5 6 7
```

```
setdiff(x, y)   # in x but not in y
```

```
#> [1] 1 2
```

```
setdiff(y, x)   # in y but not in x
```

```
#> [1] 6 7
```

```
setequal(x, y)
```

```
#> [1] FALSE
```

```
z <- 5:1
setequal(x, z)
```

```
#> [1] TRUE
```

3.5 Control Structures

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

3.5.1 `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(<condition>) {
        # do something
}
# Continue with rest of code
```

The above code does nothing if the condition is false. If you have an action you want to execute when the condition is false, then you need an `else` clause.

```
if(<condition>) {
        # do something
}
else {
        # do something else
}
```

You can have a series of tests by following the initial `if` with any number of `else if`s.

```
if(<condition1>) {
        # do something
} else if(<condition2>)  {
        # do something different
} else {
        # do something else different
}
```

3.5.2 `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.)

The following three loops all have the similar behavior.

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

for(i in 1:length(x)) {
        ## Print out each element of 'x'
        print(x[i])
}
```

```
#> [1] "a"
#> [1] "b"
#> [1] "c"
#> [1] "d"
```

The `seq_along()` function is commonly used in conjunction with for loops in order to generate an integer sequence based on the length of an object (in this case, the object `x`).

```
## Generate a sequence based on length of 'x'
for(i in seq_along(x)) {
        print(x[i])
}
```

```
#> [1] "a"
#> [1] "b"
#> [1] "c"
#> [1] "d"
```

It is not necessary to use an index-type variable.

```
for(letter in x) {
        print(letter)
```

```
}
```

```
#> [1] "a"
#> [1] "b"
#> [1] "c"
#> [1] "d"
```

Try these examples above but with `x <- NULL` and notice the difference in behavior.

Nested loops are commonly needed for multidimensional or hierarchical data structures (e.g. matrices, lists). Be careful with nesting though. Nesting beyond 2 to 3 levels often makes it difficult to read/understand the code. If you find yourself in need of a large number of nested loops, you may want to break up the loops by using functions (discussed later).

3.5.3 `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
```

While loops can potentially result in infinite loops if not written properly. Use with care!

Sometimes there will be more than one condition in the test.

```
z <- 5
set.seed(1)

while(z >= 3 && z <= 10) {
        coin <- rbinom(1, 1, 0.5)

        if(coin == 1) {  ## random walk
                z <- z + 1
        } else {
                z <- z - 1
        }
}
print(z)
```

```
#> [1] 2
```

Conditions are always evaluated from left to right. For example, in the above code, if `z` were less than 3, the second test would not have been evaluated.

3.5.4 `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
        }
}
```

Note that the above code will not run if the `computeEstimate()` function is not defined (I just made it up for the purposes of this demonstration).

The loop above is a bit dangerous because there's no guarantee it will ever stop. You could get in a situation where the values of `x0` and `x1` oscillate back and forth and never converge. Better to set a hard limit on the number of iterations by using a `for` loop and then report whether convergence was achieved or not.

3.5.5 `next`, `break`

While not used very often it's nice to know about these.

`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
      }
}
```

3.5.6 Looping

For loops are so common that that R has some functions which implement looping in a compact form to make your life easier. For a more in depth look see this

- `apply` is generic: applies a function to a matrix's rows or columns (or, more generally, to dimensions of an array)
- `lapply` is a list apply which acts on a list or vector and returns a list.
- `sapply` is a simple lapply but defaults to returning a vector (or matrix) if possible.
- `vapply` is a verified apply. This is a sapply with the return object type prespecified.
- `rapply` is a recursive apply for nested lists, i.e. lists within lists
- `tapply` is a tagged apply where the tags identify the subsets to apply a function
- `mapply` is a multivariabe apply for functions that have multiple arguments.
- `Map` is a wrapper to mapply with SIMPLIFY = FALSE, so it is guaranteed to return a list.
- `replicate` is a wrapper around sapply for repeated evaluation of an expression

```
# Two dimensional matrix
M <- matrix(seq(1,16), 4, 4)
M
```

```
#>      [,1] [,2] [,3] [,4]
#> [1,]    1    5    9   13
#> [2,]    2    6   10   14
#> [3,]    3    7   11   15
#> [4,]    4    8   12   16
```

```
# apply min to rows
apply(M, 1, min)
```

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

```
# apply max to columns
apply(M, 2, max)
```

```
#> [1]  4  8 12 16
```

If you want row/column means or sums for a 2D matrix, be sure to investigate the highly optimized, lightning-quick `colMeans`, `rowMeans`, `colSums`, `rowSums`.

```
x <- list(a = 1, b = 1:3, c = 10:25)
x
```

```
#> $a
#> [1] 1
#>
#> $b
#> [1] 1 2 3
#>
#> $c
#>  [1] 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
```

```
lapply(x, FUN = length)
```

```
#> $a
#> [1] 1
#>
#> $b
#> [1] 3
#>
```

```
#> $c
#> [1] 16
```

```
sapply(x, FUN = length)
```

```
#>  a  b  c
#>  1  3 16
```

```
vapply(x, FUN = length, FUN.VALUE = 0L)
```

```
#>  a  b  c
#>  1  3 16
```

```
x <- 1:20
y <- factor(rep(letters[1:5], each = 4)) # a vector of the same length as x
tapply(x, y, sum)
```

```
#>  a  b  c  d  e
#> 10 26 42 58 74
```

```
# Sums the 1st elements, the 2nd elements, etc.
mapply(sum, 1:5, 1:5, 1:5)
```

```
#> [1]  3  6  9 12 15
```

```
# find the mean of 10 random normal variables, 5 times
replicate(5, mean(rnorm(10)))
```

```
#> [1]  0.009560203 -0.082577193  0.032326586  0.192819739 -0.432339116
```

`tapply` is in a simalar spirit to a common data analysis paradigm called split-apply-combine where we split our data set based on a group, apply a function or code to it, and combine the results back together. We will revisit this paradigm in greater detail when we get to *R for Data Science*.

3.6 Vectorization & Recycling

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 <- 1:3
y <- 11:13
z <- x + y
z
```

```
#> [1] 12 14 16
```

In most other languages you would have to do something like

```
z <- numeric(length(x))

for(i in seq_along(x)) {
      z[i] <- x[i] + y[i]
}
z
```

```
#> [1] 12 14 16
```

We saw a a form of vectorization above in the logical operators.

```
x
```

```
#> [1] 1 2 3
```

```
x > 2
```

```
#> [1] FALSE FALSE  TRUE
```

```
x[x > 2]
```

```
#> [1] 3
```

Matrix operations are also vectorized, making for nice 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)
x
```

```
#>      [,1] [,2]
#> [1,]    1    3
#> [2,]    2    4
```

```
y
```

```
#>      [,1] [,2]
#> [1,]   10   10
#> [2,]   10   10
```

```
x * y  # element-wise multiplication
```

```
#>      [,1] [,2]
#> [1,]   10   30
#> [2,]   20   40
```

```
x / y  # element-wise division
```

```
#>      [,1] [,2]
#> [1,]  0.1  0.3
#> [2,]  0.2  0.4
```

```
x %*% y  # true matrix multiplication
```

```
#>      [,1] [,2]
#> [1,]   40   40
#> [2,]   60   60
```

R also recyles arguments.

```
x <- 1:10
z <- x + .1  # add .1 to each element
z
```

```
#>  [1]  1.1  2.1  3.1  4.1  5.1  6.1  7.1  8.1  9.1 10.1
```

While you usually either want the same length vector or a length one vector. You are not limited to just these options.

```
x <- 1:10
y <- x + c(.1, .2)
y
```

```
#>  [1]  1.1  2.2  3.1  4.2  5.1  6.2  7.1  8.2  9.1 10.2
```

```
z <- x + c(.1, .2, .3)
```

```
#> Warning in x + c(0.1, 0.2, 0.3): longer object length is not a multiple of shorter object length
```

```
z
```

```
#>  [1]  1.1  2.2  3.3  4.1  5.2  6.3  7.1  8.2  9.3 10.1
```

3.6.1 Example

One (not so good) way to estimate `pi` is through Monte-Carlo simulation.

Suppose we wish to estimate the value of `pi` using a Monte-Carlo method. Esentially, we throw darts at the unit square and count the number of darts that fall within the unit circle. We'll only deal with quadrant one. Thus the $Area = \frac{\pi}{4}$

Monte-Carlo psuedo code:

1. Initalize `hits = 0`
2. **for i in 1:N**
3. Generate two random numbers, U_1 and U_2, between 0 and 1
4. If $U_1^2 + U_2^2 < 1$, then `hits = hits + 1`
5. **end for**
6. Area estimate = `hits / N`
7. $\hat{pi} = 4 * AreaEstimate$

```
pi_naive <- function(N) {
  hits <- 0
  for(i in seq_len(N)) {
    U1 <- runif(1)
    U2 <- runif(1)
    if ((U1^2 + U2^2) < 1) {
      hits <- hits + 1
    }
  }

  4*hits/N
}
N <- 1e6
system.time(pi_naive(N))
```

```
#>    user  system elapsed
#>    3.94    0.00    4.19
```

That's a long runtime (and bad estimate). Let's vecrtorize it.

```
pi_vect <- function(N) {
  U1 <- runif(N)
  U2 <- runif(N)
  hits <- sum(U1^2 + U2^2 < 1)
  4*hits/N
}
system.time(pi_vect(N))
```

```
#>    user  system elapsed
#>    0.17    0.01    0.20
```

3.7 Function Basics

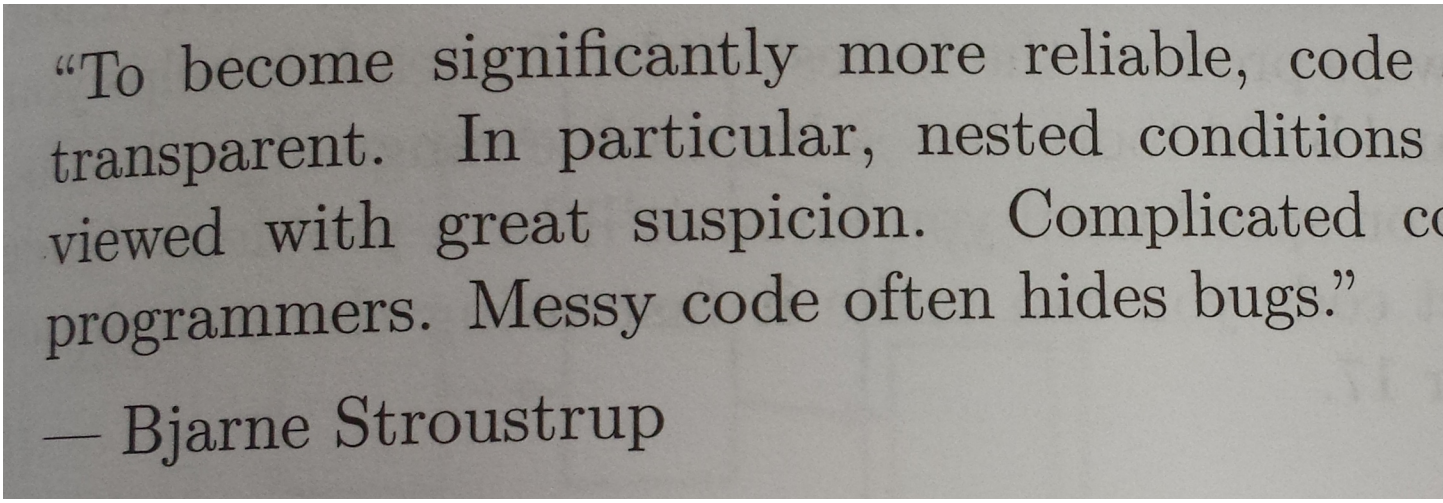
> To understand computations in R, two slogans are helpful:
> - Everything that exists is an object.
> - Everything that happens is a function call.
> John Chambers

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

If you're familiar with common language like C, these features might appear a bit strange. However, they are really important in R and can be useful for data analysis.

Messy code hides bugs



- Functions are a means of **abstraction**. A concept/computation is encapsulated/isolated from the rest with a function.
- Functions should **do one thing**, and do it well (compute, or plot, or save, … not all in one go).
- **Side effects**: your functions should not have any (unless, of course, that is the main point of that function - plotting, write to disk, …). Functions shouldn't make any changes in any environment. The only return their output.
- **Do not use global variables**. Everything the function needs is being passed as an argument. Function must be **self-contained**.
- Function streamline code and process

From the R Inferno:

Make your functions as simple as possible. Simple has many advantages:

- Simple functions are likely to be human efficient: they will be easy to understand and to modify.
- Simple functions are likely to be computer efficient.
- Simple functions are less likely to be buggy, and bugs will be easier to fix.
- (Perhaps ironically) simple functions may be more general—thinking about the heart of the matter often broadens the application.

Functions can be

1. Correct.
2. An error occurs that is clearly identified.
3. An obscure error occurs.
4. An incorrect value is returned.

We like category 1. Category 2 is the right behavior if the inputs do not make sense, but not if the inputs are sensible. Category 3 is an unpleasant place for your users, and possibly for you if the users have access to you. Category 4 is by far the worst place to be - the user has no reason to believe that anything is wrong. Steer clear of category 4.

Finally, functions are

- Easier to debug
- Easier to profile
- Easier to parallelise

Functions are an central part of robust R programming and we will spend a significant amount of time writing functions.

3.7.1 Your First Function

All R functions have three parts:

- the `body()`, the code inside the function.
- the `formals()`, the list of arguments which controls how you can call the function.
- the `environment()`, the "map" of the location of the function's variables.

When you print a function in R, it shows you these three important components. If the environment isn't displayed, it means that the function was created in the global environment.

```
myadd <- function(x, y) {
  cat(paste0("x = ", x, "\n"))
  cat(paste0("y = ", y, "\n"))
  x + y
}
myadd(1, 3)             # call by position
```

```
#> x = 1
#> y = 3
```

```
#> [1] 4
```

```
myadd(x = 1, y = 3)     # call by name
```

```
#> x = 1
#> y = 3
```

```
#> [1] 4
```

```
myadd(y = 3, x = 1)     # name order doesn't matter
```

```
#> x = 1
#> y = 3
```

```
#> [1] 4
```

- The body of the function is everything between the `{ }`. Note this does the computation **AND** returns the result.
- `x` and `y` are the arguments to the function.

- the environment this function lives in is the global environement. (We'll discuss environments more in the next section.)

Even though it's legal, I don't recommend messing around with the order of the arguments too much, since it can lead to some confusion.

You can also specify default values for your arguments. Default values *should* be the values most often used. `rnorm` uses the default of `mean = 0` and `sd = 1`. We usually want to sample from the standard normal distribution, but we are not forced to.

```
myadd2 <- function(x = 3, y = 0){
  cat(paste0("x = ", x, "\n"))
  cat(paste0("y = ", y, "\n"))
  x + y
}
myadd2()              # use the defaults
```

```
#> x = 3
#> y = 0
```

```
#> [1] 3
```

```
myadd2(x = 1)
```

```
#> x = 1
#> y = 0
```

```
#> [1] 1
```

```
myadd2(y = 1)
```

```
#> x = 3
#> y = 1
```

```
#> [1] 4
```

```
myadd2(x = 1, y = 1)
```

```
#> x = 1
#> y = 1
```

```
#> [1] 2
```

By defualt the last line of the function is returned. Thus, there is no reason to explicitly call `return`, unless you are returning from the function early. Inside functions use `stop` to return error messages, `warning` to return warning messages, and `message` to print a message to the console.

```
f <- function(age) {
  if (age < 0) {
    stop("age must be a positive number")
  }

  if (age < 18) {
    warning("Check your data.  We only care about adults.")
  }

  message(paste0("Your person is ", age, " years old"))
  invisible()
}

f(-10)
```

```
#> Error in f(-10): age must be a positive number
```

```
f(10)
```

```
#> Warning in f(10): Check your data. We only care about adults.
```

```
#> Your person is 10 years old
```

```
f(30)
```

```
#> Your person is 30 years old
```

3.7.2 Lazy Evaluation

R is lazy. Arguments to functions are evaluated *lazily*, that is they are evaluated only as needed in the body of the function.

In this example, the function `f()` has two arguments: `a` and `b`.

```
f <- function(a, b) {
  a^2
}

f(2)     # this works
```

```
#> [1] 4
```

```
f(2, 1)  # this does too
```

```
#> [1] 4
```

This function never actually uses the argument `b`, so calling `f(2)` or `f(2, 1)` will not produce an error because the 2 gets positionally matched to a. This behavior can be good or bad. It's common to write a function that doesn't use an argument and not notice it simply because R never throws an error.

3.7.3 The ... Argument

There is a special argument in R known as the `...` argument, which indicate a variable number of arguments that are usually passed on to other functions. The `...` argument is often used when extending another function and you don't want to copy the entire argument list of the original function

For example, a custom plotting function may want to make use of the default `plot()` function along with its entire argument list. The function below changes the default for the `type` argument to the value `type = "l"` (the original default was `type = "p"`).

```
myplot <- function(x, y, type = "l", ...) {
        plot(x, y, type = type, ...)         ## Pass '...' to 'plot' function
}
```

The `...` argument is also necessary when the number of arguments passed to the function cannot be known in advance. This is clear in functions like `paste()` and `cat()`.

```
args(paste)
```

```
#> function (..., sep = " ", collapse = NULL)
#> NULL
```

```
args(cat)
```

```
#> function (..., file = "", sep = " ", fill = FALSE, labels = NULL,
#>     append = FALSE)
#> NULL
```

Because both `paste()` and `cat()` print out text to the console by combining multiple character vectors together, it is impossible for those functions to know in advance how many character vectors will be passed to the function by the user. So the first argument to either function is `...`.

One catch with `...` is that any arguments that appear *after* `...` on the argument list must be named explicitly and cannot be partially matched or matched positionally.

Take a look at the arguments to the `paste()` function.

```
args(paste)
```

```
#> function (..., sep = " ", collapse = NULL)
#> NULL
```

With the `paste()` function, the arguments `sep` and `collapse` must be named explicitly and in full if the default values are not going to be used.

3.8 Environments & Scoping

An **environment** is a collection of (symbol, value) pairs, i.e. `x` is a symbol and `3.14` might be its value. Every environment has a parent environment and it is possible for an environment to have multiple "children". The only environment without a parent is the empty environment.

Scoping is the set of rules that govern how R looks up the value of a symbol. In the example below, scoping is the set of rules that R applies to go from the symbol `x` to its value `10`:

```
x <- 10
x
```

```
#> [1] 10
```

R has two types of scoping: lexical scoping, implemented automatically at the language level, and dynamic scoping, used in select functions to save typing during interactive analysis. We discuss lexical scoping here because it is intimately tied to function creation. Dynamic scoping is an advanced topic and is discussed in Advanced R.

How do we associate a value to a free variable? There is a search process that occurs that goes as follows:

If the value of a symbol is not found in the environment in which a function was defined, then the search is continued in the parent environment. The search continues up the sequence of parent environments until we hit the top-level environment; this usually the global environment (workspace) or the namespace of a package. After the top-level environment, the search continues down the search list until we hit the empty environment. If a value for a given symbol cannot be found once the empty environment is arrived at, then an error is thrown.

```
x <- 0
f <- function(x = -1) {
  x <- 1
  y <- 2
  c(x, y)
}

g <- function(x = -1) {
  y <- 1
  c(x, y)
```