Modern R in a Corporate Environment

R course developed for the office

Brian Davis

2018-10-29

Contents

Welcome

> 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.

This is the course notes for our class. This course will teach you how to do data science with R. You'll learn the basics of R and then we'll go through R for Data Science by Garrett Grolemund & Hadley Wickham. You'll learn how to get your data into R, get it into the most useful structure, transform it, visualize it and communicate out your results. We'll mix in various topics from our current workload as well as some unique challenges of working in a corporate environment.

Most of these are the skills that allow data science to happen, and here you will find the best practices for doing each of these things with R. You'll learn how to use the grammar of graphics, literate programming, and reproducible research to save time and reduce errors.

We will build the tools to make our work easier and more streamlined together.

Part I

Preamble

Chapter 1

Introduction

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 Approach

What is Reproducible Research About?

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

Contrary 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 3.5.1
  - 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. Our aim is at gaining 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.

We will focus most of our attention to the *tidyverse* family of packages for data analysis. The *tidyverse* is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures.

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 / Exporting Data
4. Tidying Data
5. Visualizing Data
6. Functions
7. Strings
8. Dates and Time
9. Communicating Results
10. Iteration

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 some exercises due for the next week which we go over the solutions. We will incorporate these exercises into an R package(s?) so we will have a collection of useful 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 so we can work on the class during work hours. Bring what you are working on and we will see how we can fit it into the class.

Chapter 2

Good practices

> "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."
>
> – Donald Knuth

2.1 Coding style

Good coding style is like correct punctuation: you can manage without it, butitsuremakesthingseasiertoread. When I answer questions; first, I see if think 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).

> "To become ssignificantly more reliable, code must become more transparent. In particular, nested conditions and loops must be viewed with great suspicion. Complicated control flows confuse programmers. **Messy code often hides bugs**."
>
> — Bjarne Stroustrup

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: `#`.

Use commented lines of - to break up your file into easily readable chunks and to create a code outline in RStudio

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 auto complete.

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 indices)

```
# 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 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
```

2.1.4 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 }
```

2.1.5 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"
```

2.1.6 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.

```
# 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.

```
# 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 Three*[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 into a project 3 or more times *make a package*
- If 3 or more people will use the function *make a package*

The *Rule of Three* applies to look-up tables and such also. 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()`

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

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

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

```
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.

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

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 focused blog aggregation site with dozens of posts per day. Check it out.

2.6 Reading For Next Class

1. Read the chapter on Workflow: basics
2. Read the chapter on Workflow: scripts
3. Read the chapter on Workflow: projects
4. Read Chapters 1-3 of the Tidyverse Style Guide
5. See these RStudio Tips & Tricks or these and find one that looks interesting and **practice** it all week.
6. Read how to make a great R reproducible example

2.7 Exercises

1. Create an R Project for this class.
2. 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.
   - Hint: In the R console type `file` and scroll through the various functions which appear in the pop-up.
3. 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)

4. Apply the style guide to your code.

5. 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.

Part II

Base R Basics

Chapter 3

R Basics

Here is a quick overview of the basics. We will dive deep into R's basic data structures and then how to subset those data structures later in the course. 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 documentation for *topic*.
- `str` displays the structure of an R object in human readable form.

`glimpse` is a tidyverse equivalent to `str` but with nicer output for complicated data structures.

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 Programming for Data Science

In R there three basic constructs[1]; 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 Naming Rules

R has strict rules about what constitutes a valid name. A **syntactic** name must consist of letters[2], digits, `.` and `_`, and can't begin with `_`. Additionally, it can not be one of a list of **reserved words** like `TRUE`, `NULL`, `if`, and `function` (see the complete list in `?Reserved`). Names that don't follow these rules are called **non-syntactic** names, and if you try to use them, you'll get an error:

[1] Technically speaking functions and environments are objects which allows one to do things in R you can't do in many other languages.

[2] Surprisingly, what constitutes a letter is determined by your current locale. That means that the syntax of R code actually differs from computer to computer, and it's possible for a file that works on one computer to not even parse on another!

```
_abc <- 1
#> Error: unexpected input in "_"

if <- 10
#> Error: unexpected assignment in "if <-"
```

While `TRUE` and `FALSE` are reserved words `T` and `F` are not. However, you can use `T` and `F` as logical. If someone assigns either of those a different value you will get a **very** hard to track down bug. Always spell out the `TRUE` and `FALSE`.

3.3 Objects

3.3.1 Vector

You create a vector with `c`. These have to be the same data type (See next section).

```
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, 20, by = 3)
#> [1]  1  4  7 10 13 16 19

# replicate a number n times
rep(3, times = 4)
#> [1] 3 3 3 3

# arguments are generally vectorized
rep(1:3, times = 3:1)
#> [1] 1 1 1 2 2 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) # a better way
#> [1] 1 2 3 4 5 6 7
```

3.3.2 Atomic Vectors

There are many "atomic" types of data: `logical`, `integer`, `double` and `character` (in this order, see below). There are also `raw` and `complex` but they are rarely used.

You can't mix types in an atomic vector (you can in a list). Coercion will automatically occur if you mix types:

```
(a <- FALSE)
#> [1] FALSE
typeof(a)
#> [1] "logical"

(b <- 1:10)
#>  [1]  1  2  3  4  5  6  7  8  9 10
typeof(b)
#> [1] "integer"
c(a, b)         ## FALSE is coerced to integer 0
#>  [1]  0  1  2  3  4  5  6  7  8  9 10

(c <- 10.5)
#> [1] 10.5
typeof(c)
#> [1] "double"
(d <- c(b, c))  ## coerced to double
#>  [1]  1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0 10.0 10.5

c(d, "a")       ## coerced to character
#>  [1] "1"    "2"    "3"    "4"    "5"    "6"    "7"    "8"    "9"    "10"
#> [11] "10.5" "a"

50 < "7"
#> [1] TRUE
```

You can force coercion with `as.logical`, `as.integer`, `as.double`, `as.numeric`, and `as.character`. Most of the time the coercion rules are straight forward, but not always.

```
x <- c(TRUE, FALSE)
typeof(x)
#> [1] "logical"

as.integer(x)
#> [1] 1 0
as.numeric(x)
#> [1] 1 0
```

```
as.character(x)
#> [1] "TRUE"  "FALSE"
```

However, coercion is not associative.

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

x2 <- as.integer(x)
x3 <- as.numeric(x2)
as.character(x3)
#> [1] "1" "0"
```

What would you expect this to return?

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

as.integer(as.character(x))
```

You can test for an "atomic" types of data with: `is.logical`, `is.integer`, `is.double`, `is.numeric`[3], and `is.character`.

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

is.logical(x)
#> [1] TRUE
is.integer(x)
#> [1] FALSE
```

What would you expect these to return?

```
x <- 2

is.integer(x)
is.numeric(x)
is.double(x)
```

Missing values are specified with `NA`, which is a logical vector of length 1. `NA` will always be coerced to the correct type if used inside `c()`, or you can create `NA`s of a specific type with `NA_real_` (a double vector), `NA_integer_` and `NA_character_`.

3.3.3 Matrix

Matrices are 2D vectors, with all elements of the same type. Generally 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]
```

[3] `is.numeric()` is a general test for the "numberliness" of a vector and returns TRUE for both integer and double vectors. It is not a specific test for double vectors, which are often called numeric.

```
#> [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
```

3.3.4 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
```

Table 3.1: Logical Operators

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

We'll discuss lists in more detail later in the course.

3.3.5 Data frame

A data frame is a list with each vector of the same length. This is the main data structure used and is analogous 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 greater detail later in the course.

3.4 Comparision

```
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
```

3.5 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).

```
df$x && df$y  # only and the first elements
#> [1] TRUE
df$x || df$y  # only or the first elements
#> [1] TRUE
```

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

`=` is used for assignment while `==` is used for comparison. A common bug is to use `=` instead of `==` inside a control structure.

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
```

3.6 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 iteration of a loop

3.6.1 `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
}
```

There is also an `ifelse` function which is vectorized version. It is essentially an `if-else` wrapped in a `for` loop so that the condition, and action, is performed on each element in a vector.

3.6.2 `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 iterator 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"
```

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 probably want to break up the loops by using functions (discussed later).

We will discus looping and the other control structures in more detail when we get to the section on iterators.

3.7 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 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 recycles 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
```

3.7.1 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. Essentially, 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 pseudo code:

1. Initialize `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
(t1 <- system.time(pi_est_naive <- pi_naive(N)))
#>    user  system elapsed
#>    3.72    0.00    3.82
pi_est_naive
#> [1] 3.14
```

That's a long run time (and bad estimate). Let's vectorize it.

```
pi_vect <- function(N) {
  U1 <- runif(N)
  U2 <- runif(N)
  hits <- sum(U1^2 + U2^2 < 1)
  4*hits/N
}
(t2 <- system.time(pi_est_vect <- pi_vect(N)))
#>    user  system elapsed
#>    0.09    0.03    0.20
pi_est_vect
#> [1] 3.14
```

The speed up from vectorization is impressive.

```
floor(t1/t2)
#>    user  system elapsed
#>      41       0      19
```

3.8 Reading For Next Class

1. Read the chapter on Tibbles
2. Try the exercises at the end of the chapter.
   - Problem 2: Create a tibble (or convert the data frame) and compare. Also compare `str` or `glimpse` on the objects.
   - Problem 3: Try to extract the column "mpg" from the mtcars data frame (convert to a tibble and compare) using `var <- "mpg"`. This illustrates a common source of confusion for people coming from SAS.

- Problem 4: non-syntactic names usually occur when importing data from various file types. Knowing how to use / correct them is very useful. Don't worry about creating the plot.

3.9 Exercises

1. Browse this vocabulary list and read the help file for functions that interest you.
2. Re-run the three cases in the For loop section with `x <- NULL`
3. Vectorization / function practice.

We'll calculate pi using the Gregory-Leibniz series. Mathematicians will be quick to point out that this is a poor way to calculate pi, since the series converges very slowly. But our goal is not calculating pi, our goal is examining the performance benefit that be be achieved using vectorization.

Here is a formula for the Gregory-Leibniz series:

$$1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \cdots = \frac{\pi}{4} \tag{3.1}$$

Here is the Gregory-Leibniz series in summation notation:

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2 \cdot n + 1} = \frac{\pi}{4} \tag{3.2}$$

The straightforward implementation using an R loop would look like this:

```
GL_naive <- function (limit) {
  p <- 0
  for (n in 0:limit) {
    p <- (-1)^n/(2 * n + 1) + p
    }
  4*p
}

N <- 1e7
system.time(pi_est <- GL_naive(N))
#>    user  system elapsed
#>    2.15    0.00    2.18
pi_est
#> [1] 3.14
```

Your task is to vectorize this function. Do not use any looping or apply functions. This one is a bit tricky. Hint: It may be easier to think about it in terms of the series notation and not the summation notation.

```
GL_vect <- function(limit) {
  # your code here
  # use only base functions and no looping mechanisms
}
```

Part III

R4DS

Chapter 4

Data Import

```
library(tidyverse)
```

Since R is a "glue" language. You can read in from just about any standard data source. We will only cover the most common types, but you can also read from pdfs (package pdftools), web scraping (package rvest and httr), twitter (package twitteR), Facebook (package Rfacebook), and many many more.

4.1 Text Files

4.1.1 readr

One of the most common data sources are text files. Usually these come with a delimiter, such a commas, semicolons, or tabs. The **readr** package is part of the core tidyverse.

Compared to Base R `read` functions, **readr** are:

- They are typically much faster (~10x)
- Long running reads automatically get a progress bar
- Default to **tibbles** not data frames
- Does not convert characters to factors
- More reproducible. (Base R read functions inherit properties from the OS)

If you're looking for raw speed, try `data.table::fread()`. It doesn't fit quite so well into the tidyverse, but it can be quite a bit faster.

- `read_csv()` reads comma delimited files
- `read_csv2()` reads semicolon separated files (common in countries where , is used as the decimal place)
- `read_tsv()` reads tab delimited files
- `read_delim()` reads in files with any delimiter
- `read_fwf()` reads fixed width files

All the `read_*` functions follow the same basic structure. The first argument is the file to read in, followed by the other parameters.

Files ending in .gz, .bz2, .xz, or .zip will be automatically uncompressed. Files starting with http://, https://, ftp://, or ftps:// will be automatically downloaded. Remote gz files can also be

automatically downloaded and decompressed.

Some useful parameters are: - `col_types` for specifying the data types for each column. - `skip = n` to skip the first `n` lines - `comment = "#"` to drop all lines that start with `#` - `locale` locale controls defaults that vary from place to place

4.1.2 base R

The tidyverse packages, and **readr** make some simplifying assumptions. The equivalent base R functions are:

- `read.csv()` reads comma delimited files
- `read.csv2()` reads semicolon separated files
- `read.tsv()` reads tab delimited files
- `read.delim()` reads in files with any delimiter
- `read.fwf()` reads fixed width file

4.2 SAS Files

```
library(haven)
```

The **haven** library, which is part of the tidyverse but not part of the core tidyverse package, must be loaded explicitly. **haven** is the most robust option for reading SAS data files. Reading supports both sas7bdat files and the accompanying sas7bdat files that SAS uses to record value labels.

`read_sas()` reads `.sas7bdat` + `sas7bcat` files `read_xpt()` reads SAS transport files

The **haven** package can also read in:

- SPSS files with `read_sav()` or `read_por()`
- Stata files with `read_dta()`

SAS has the notion of a "labelled" variable (so do Stata and SPSS). These are similar to factors, but:

- Integer, numeric and character vectors can be labelled.
- Not every value must be associated with a label.

Factors, by contrast, are always integers and every integer value must be associated with a label.

Haven provides a labelled class to model these objects. It doesn't implement any common methods, but instead focuses of ways to turn a labelled variable into standard R variable:

- as_factor(): turns labelled integers into factors. Any values that don't have a label associated with them will become a missing value. (NOTE: there's no way to make as.factor() work with labelled variables, so you'll need to use this new function.)
- zap_labels(): turns any labelled values into missing values. This deals with the common pattern where you have a continuous variable that has missing values indicated by sentinel values.

There are other packages that can read SAS data files, namely `sas7bdat` and `foreign`. `sas7bdat` is no longer being maintained for the last several years and is not recommended for production use. While `foreign` only reads SAS XPORT format.

4.3 Excel

```
library(readxl)
```

The **readxl** package makes it easy to get data out of Excel and into R. It is designed to work with tabular data. **readxl** supports both the legacy .xls format and the modern xml-based .xlsx format. The **readxl** library, which is part of the tidyverse but not part of the core tidyverse package, must be loaded explicitly.

There are two main functions in the **readxl** package.

- `excel_sheets()` lists all the sheets in an excel spreadsheet.
- `read_excel()` reads in xls and xlsx files based on the file extension

If you want to prevent `read_excel()` from guessing which spreadsheet type you have you can use `read_xls()` or `read_xlsx()` directly.

There are several other packages which also can read excel files.

- `openxlsx` - can read but is tricky to extract data, but shines in writing Excel files.
- `xlsx` requires Java, usually cannot get corporate IT to install it on Windows.
- `XLConnect` requires Java, usually cannot get corporate IT to install it on Windows.
- `gdata` required Perl, usually cannot get corporate IT to install it on Windows machines.
- `xlsReadWrite` - Does not support .xlsx files

4.4 Databases

Here we have to use two (or three) packages. The **DBI** package is used to make the network connection to the database. The connection string should look familiar if you have ever made a connection to a database from another program. As database vendors have slightly different interfaces and connection types. You will have to use the package for your particular database backend. Some common ones include:

- `RSQLite::SQLite()` for SQLite
- `RMySQL::MySQL()` for MySQL
- `RPostgreSQL::PostgreSQL()` for PostgreSQL
- `odbc::odbc()` for Microsoft SQL Server
- `bigrquery::bigquery()` for BigQuery

```
con <- dbConnect(odbc::odbc(),                # for a Mircosoft server
                 dsn      = "my_dsn",
                 server   = "our_awesome_server",
                 database = "cool_db")
```

To interact with a database you usually use SQL, the Structured Query Language. SQL is over 40 years old, and is used by pretty much every database in existence.

This leads to two methods to extract data from a database which boil down to:

- Pull the entire table into a data frame with `dbReadTable()`
- Write SQL for you specific dialect and pull into a data frame with `dbGetQuery()`

Another popular package for connecting to databases is **RODBC**. It tends to be a bit slower than **DBI**.

Alternatively, you can use the **dbplyr** and the connection to the database to auto generate SQL queries using standard dplyr syntax.

The goal of **dbplyr** is to automatically generate SQL for you so that you're not forced to use it. However, SQL is a very large language and **dbplyr** doesn't do everything. It focuses on SELECT statements, the SQL you write most often as an analysis. See `vignette(("dbplyr"))` for a in depth discussion.

4.5 Reading For Next Class

1. Read Chapter on Pipes.

4.6 Exercises

Chapter 5

Data Transformation

```
library(tidyverse)
```

5.1 Tibbles

Tibbles **are** data frames, but they tweak some older behaviors to make life a little easier. R is an old language, and some things that were useful 10 or 20 years ago now get in your way. It's difficult to change base R without breaking existing code, so most innovation occurs in packages.

See Chapter 7 in R for Data Science and `vignette("tibble")` for a more complete description of tibbles.

Key advantages of tibbles:

- Never changes the types of the inputs (e.g. it never converts strings to factors!).
- Never changes the names of variables.
- Never creates row names.
- Refined printing to the console:
  - only prints the first 10 rows
  - shows the data type of each column
  - highlights missing values
  - aligns numeric data
- Enhanced subsetting

When creating a tibble:

- it will **ONLY** recycle inputs of length 1
- you can refer to variables you just created

```
mytibble <- tibble(
  x = 1:5,
  y = 1,
  z = x ^ 2 + y
)

# versus

mydf <- data.frame(
```

```
  x = 1:5,
  y = 1
)

mydf$z <- mydf$x^2 + mydf$y

mytibble
#> # A tibble: 5 x 3
#>       x     y     z
#>   <int> <dbl> <dbl>
#> 1     1     1     2
#> 2     2     1     5
#> 3     3     1    10
#> 4     4     1    17
#> 5     5     1    26
mydf
#>   x y  z
#> 1 1 1  2
#> 2 2 1  5
#> 3 3 1 10
#> 4 4 1 17
#> 5 5 1 26
```

For the most part you can use data.frames and tibbles interchangeably. However, some older functions in base R do not work properly with tibbles. This is because of the `[` subset operator. As we saw in subsetting, if you return a single column with `[` on a data.frame you will get a vector back instead of a single column data.frame. Tibbles always return tibbles and never a vector.

You can convert a data frame to a tibble with `as_tibble()`, while you can coerce a tibble to a data frame with `as.data.fame()`.

dplyr functions never modify their inputs, so if you want to save the result, you'll need to use the assignment operator, `<-`

5.2 Filter rows with `filter()`

`filter()` allows you to subset observations based on their values. The first argument is the name of the data frame. The second and subsequent arguments are the expressions that filter the data frame.

5.3 Arrange rows with `arrange()`

`arrange()` works similarly to `filter()` except that instead of selecting rows, it changes their order. It takes a data frame and a set of column names (or more complicated expressions) to order by. If you provide more than one column name, each additional column will be used to break ties in the values of preceding columns.

Notes:

- Use `desc()` to re-order by a column in descending order:
- Missing values are always sorted at the end.

5.4 Select columns with `select()`

It's not uncommon to get datasets with hundreds or even thousands of variables. In this case, the first challenge is often narrowing in on the variables you're actually interested in. `select()` allows you to rapidly zoom in on a useful subset using operations based on the names of the variables.

There are a number of helper functions you can use within select():

- everything(): all variables or everything else not already selected / deselected.
- starts_with(): start with a prefix.
- ends_with(): ends with a prefix
- contains(): contains a literal string
- matches(): match a regular expression.
- num_range(): a numerical range like x1, x2, x3.
- one_of(): variable in a character vector

Closely related to `select()` is `rename()` for renaming columns.

5.5 Add new variables with `mutate()`

Besides selecting sets of existing columns, it's often useful to add new columns that are functions of existing columns. `mutate()` always adds new columns at the end of your dataset so we'll start by creating a narrower dataset so we can see the new variables. Remember that when you're in RStudio, the easiest way to see all the columns is `View()`.

Because we have are using tibbles you can refer to columns you just created.

If you only want to keep the new variables, use `transmute()`.

There are a number of helper functions you can use within mutate():

- na_if(): convert values to NA
- coalesce(): convert NA to a value.
- if_else(): vectorised if
- recode(): recode values
- case_when(): A general vectorised if. Equivalent of the SQL CASE WHEN statement.

5.6 Pipes

See Chapter 14 in R for Data Science.

Typically it takes a series of operations to go from raw data to meaningful analysis. There are (at least) four ways to to do this:

- Save each intermediate step as a new object.
- Overwrite the original object many times.
- Compose functions.
- Use pipes

Each have the place and utility. None are perfect for every situation. We'll use piping frequently from now on because it considerably improves the readability of code.

5.6.1 Intermediate steps

Simplest approach is to create a new object at each step. This method tends to clutter the code with unimportant, and uninformative names. Also, each object eats up memory, which becomes an issue as data set sizes grow large.

Rule of thumb: If there is a natural new name it may be an appropriate method. If the natural name is to increment a counter on the end of the original variable a different method is generally preferable.

5.6.2 Overwrite the original

Instead of creating intermediate objects at each step, we could overwrite the original object.

This is less typing (and less thinking), so you're less likely to make mistakes. However, there are two problems:

- Debugging is painful: if you make a mistake you'll need to re-run the complete pipeline from the beginning.
- The repetition of the object being transformed (we've written foo_foo six times!) obscures what's changing on each line.

A common method is to combine the above methods and use an temporary variable name for the intermediate steps such a "tmp"" or ".".

5.6.3 Function composition

Another approach is to abandon assignment and just string the function calls together. Here the disadvantage is that you have to read from inside-out, from right-to-left, and that the arguments end up spread far apart (evocatively called the dagwood sandwich problem). In short, this code is hard for a human to consume.

This method is generally not appropriate if you need to nest more than two functions together. However, it is appropriate in many cases. (e.g. sum(is.na(x)) to find the number of missing values in x)

5.6.4 Use the pipe operator %>%

Finally, we can use the pipe. This is my favorite form, because it focuses on verbs (what we are doing), not nouns (what we are doing it on). You can read this series of function compositions like it's a set of imperative actions.

The pipe works by performing a "lexical transformation". Behind the scenes, `magrittr` reassembles the code in the pipe to a form that works by overwriting an intermediate object. This means that pipes won't work for two classes of functions:

1. Functions that use the current environment. (e.g. `assign`, `with`,`get` or `load`)
2. Functions that use lazy evaluation. In R, function arguments are only computed when the function uses them, not prior to calling the function. The pipe computes each element in turn, so you can't rely on this behavior. (e.g. `tryCatch`, `try`, `suppressMessages`)

The pipe is a powerful tool, but it's not the only tool at your disposal, and it doesn't solve every problem! Pipes are most useful for rewriting a fairly short linear sequence of operations. I think you should reach for another tool when:

- Your pipes are longer than (say) ten steps. In that case, create intermediate objects with meaningful names. That will make debugging easier, because you can more easily check the intermediate results, and it makes it easier to understand your code, because the variable names can help communicate intent.
- You have multiple inputs or outputs. If there isn't one primary object being transformed, but two or more objects being combined together, don't use the pipe.
- You are starting to think about a directed graph with a complex dependency structure. Pipes are fundamentally linear and expressing complex relationships with them will typically yield confusing code.

5.7 Grouped summaries with `summarise()`

The last key verb is `summarise()`. It collapses a data frame to a single row. `summarise()` is not terribly useful unless we pair it with `group_by()`. This changes the unit of analysis from the complete dataset to individual groups. Then, when you use the dplyr verbs on a grouped data frame they'll be automatically applied "by group".

Together `group_by()` and `summarise()` provide one of the tools that you'll use most commonly when working with dplyr: grouped summaries.

If you need to remove grouping, and return to operations on ungrouped data, use ungroup()

5.8 Grouped mutates

Grouping is most useful in conjunction with `summarise()`, but you can also do convenient operations with `mutate()` and `filter()`

- Find the best/worst members of each group.
- Find all groups bigger/smaller than a threshold.
- Standardize to compute per group metrics.

Part IV

Appendix

Chapter 6

Appendix A

6.1 E-Books

- R Programming for Data Science by Roger D. Peng,
- Efficient R programming by Colin Gillespie & Robin Lovelace,
- Mastering Software Development in R by Roger D. Peng, Sean Kross, and Brooke Anderson
- 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.