Modern R in a Corporate Environment

R course developed for the office

Brian Davis

2019-01-11

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)

```r
# 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).

```r
# 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.

```r
# 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)
```

```r
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.

```r
# 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.

```r
# 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) {
```

```r
  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 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()`

```r
# 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)
```

```r
# 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:

```r
(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.

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

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

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

However, coercion is not associative.

```r
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?

```r
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`.

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

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

What would you expect these to return?

```r
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.

```r
# 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.

```r
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

```r
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.

```r
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

```r
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).

```r
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`

```r
x <- c(FALSE, FALSE, FALSE, TRUE)
any(x)
#> [1] TRUE
all(x)
#> [1] FALSE
```

```r
all(!x[1:3])
#> [1] TRUE
```

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

```r
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.

```r
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. ma-
> trices, 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. $\hat{pi} = 4 * AreaEstimate$

```r
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.

```r
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
#>    0.10    0.00    0.18
pi_est_vect
#> [1] 3.14
```

The speed up from vectorization is impressive.

```r
floor(t1/t2)
#>    user  system elapsed
#>      37     NaN      21
```

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
#>    2.14    0.00    2.26
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

Tidyverse

Chapter 4

Data Import & Export

```r
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

```r
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

```r
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

```r
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
- replace_na(): 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.

5.7   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.

5.9   Reading For Next Class

1. Read Tidy Data paper published in the Journal of Statistical Software, http://www.jstatsoft.org/v59/i10/paper.

5.10   Exercises

1. Read and Try the exercises in Data transformation. There are quite a few useful functions and use cases we did not cover. It also does a very good job of how does one go from a question we want answered to quickly seeing the answer in a useful form.

2. We barely scratched the surface of some of the useful dplyr functions for data transformations. Review the dplyr-transformation cheatsheet and see the help file for interesting functions.

Chapter 6

Tidy Data

```
library(tidyverse)
```

> "Happy families are all alike; every unhappy family is unhappy in its own way." —- Leo Tolstoy

> "Tidy datasets are all alike, but every messy dataset is messy in its own way." —- Hadley Wickham

In this chapter, you will learn a consistent way to organize your data in R, an organisation called **tidy data**. Getting your data into this format requires some upfront work, but that work pays off in the long term. Once you have tidy data and the tidy tools provided by packages in the tidyverse, you will spend much less time munging data from one representation to another, allowing you to spend more time on the analytic questions at hand.

This chapter will give you a practical introduction to tidy data and the accompanying tools in the **tidyr** package. If you'd like to learn more about the underlying theory, you might enjoy the *Tidy Data* paper published in the Journal of Statistical Software, http://www.jstatsoft.org/v59/i10/paper.

6.1 Tidy data

There are three interrelated rules which make a dataset tidy:

1. Each variable must have its own column.
2. Each observation must have its own row.
3. Each value must have its own cell.

Figure 6.1 shows the rules visually.

These three rules are interrelated because it's impossible to only satisfy two of the three.

Why ensure that your data is tidy? There are two main advantages:

1. There's a general advantage to picking one consistent way of storing data. If you have a consistent data structure, it's easier to learn the tools that work with it because they have an underlying uniformity.

2. There's a specific advantage to placing variables in columns because it allows R's vectorized nature to shine. As you learned in mutate and summary functions, most built-in R functions work with vectors of values. That makes transforming tidy data feel particularly natural.

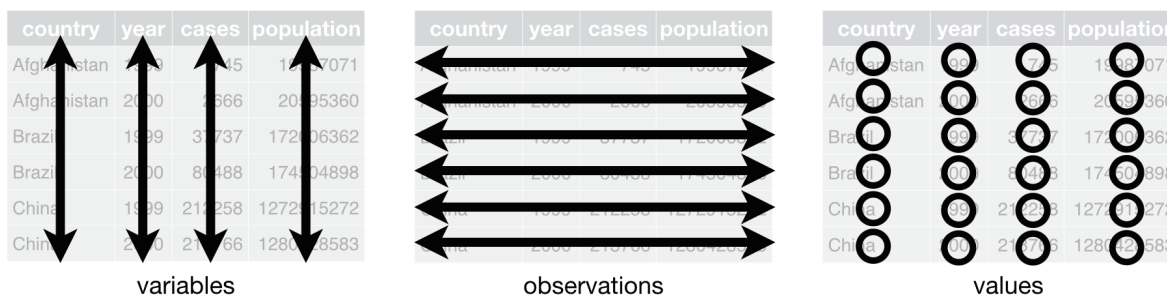dplyr, ggplot2, and all the other packages in the tidyverse are designed to work with tidy data.

Figure 6.1: Following three rules makes a dataset tidy: variables are in columns, observations are in rows, and values are in cells.

6.2   Example

You can represent the same underlying data in multiple ways. The example below shows the same data organised in four different ways. Each dataset shows the same values of four variables *country*, *year*, *population*, and *cases*, but each dataset organizes the values in a different way.

```
table1
#> # A tibble: 6 x 4
#>   country      year   cases population
#>   <chr>       <int>   <int>      <int>
#> 1 Afghanistan  1999     745   19987071
#> 2 Afghanistan  2000    2666   20595360
#> 3 Brazil       1999   37737  172006362
#> 4 Brazil       2000   80488  174504898
#> 5 China        1999  212258 1272915272
#> 6 China        2000  213766 1280428583
table2
#> # A tibble: 12 x 4
#>   country      year type            count
#>   <chr>       <int> <chr>           <int>
#> 1 Afghanistan  1999 cases             745
#> 2 Afghanistan  1999 population   19987071
#> 3 Afghanistan  2000 cases            2666
#> 4 Afghanistan  2000 population   20595360
#> 5 Brazil       1999 cases           37737
#> 6 Brazil       1999 population  172006362
#> # ... with 6 more rows
table3
#> # A tibble: 6 x 3
#>   country      year rate
#> * <chr>       <int> <chr>
#> 1 Afghanistan  1999 745/19987071
#> 2 Afghanistan  2000 2666/20595360
#> 3 Brazil       1999 37737/172006362
#> 4 Brazil       2000 80488/174504898
#> 5 China        1999 212258/1272915272
#> 6 China        2000 213766/1280428583
```

```
# Spread across two tibbles
table4a  # cases
#> # A tibble: 3 x 3
#>   country     `1999` `2000`
#> * <chr>        <int>  <int>
#> 1 Afghanistan    745   2666
#> 2 Brazil       37737  80488
#> 3 China       212258 213766
table4b  # population
#> # A tibble: 3 x 3
#>   country          `1999`      `2000`
#> * <chr>             <int>       <int>
#> 1 Afghanistan    19987071    20595360
#> 2 Brazil        172006362   174504898
#> 3 China        1272915272  1280428583
```

These are all representations of the same underlying data, but they are not equally easy to use. One dataset, the tidy dataset, will be much easier to work with inside the tidyverse.

Which dataset is tidy? Why aren't the other datasets considered tidy?

> How would you calculate the rate per 10,000 population for each data set? How would you compute the cases per year? How would you visualize the changes over time?

6.3 Spreading and gathering

The principles of tidy data seem so obvious that you might wonder if you'll ever encounter a dataset that isn't tidy. Unfortunately, however, most data that you will encounter will be untidy. There are two main reasons:

1. Most people aren't familiar with the principles of tidy data, and it's hard to derive them yourself unless you spend a *lot* of time working with data.

2. Data is often organised to facilitate some use other than analysis. For example, data is often organised to make entry as easy as possible.

3. How to efficiently store, analyze, and present data are *usually* three different data formats.

This means for most real analyses, you'll need to do some tidying / untidying. The first step is always to figure out what the variables and observations are. Sometimes this is easy; other times it can be a bit more difficult. The second step is to resolve one of two common problems:

1. One variable might be spread across multiple columns.

2. One observation might be scattered across multiple rows.

Typically a dataset will only suffer from one of these problems; it'll only suffer from both if you're really unlucky! To fix these problems, you'll need the two most important functions in tidyr: `gather()` and `spread()`.

6.3.1 Gathering

A common problem is a dataset where some of the column names are not names of variables, but *values* of a variable. Take `table4a`: the column names `1999` and `2000` represent values of the `year` variable, and each
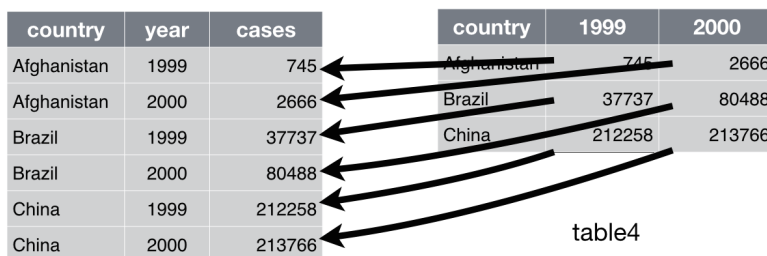
Figure 6.2: Gathering 'table4' into a tidy form.

row represents two observations, not one.

```
table4a
#> # A tibble: 3 x 3
#>   country      `1999` `2000`
#> * <chr>         <int>  <int>
#> 1 Afghanistan     745   2666
#> 2 Brazil        37737  80488
#> 3 China        212258 213766
```

To tidy a dataset like this, we need to **gather** those columns into a new pair of variables. To describe that operation we need three parameters:

- The name of the variable whose values form the column names. This is the `key`, and here it is `year`.

- The name of the variable whose values are spread over the cells. That is the `value`, and here it's the number of `cases`.

- The set of columns that represent values, not variables. In this example, those are the columns `1999` and `2000`.

Together those parameters generate the call to `gather()`:

```
table4a %>%
  gather(key = "year", value = "cases", `1999`, `2000`)
#> # A tibble: 6 x 3
#>   country      year    cases
#>   <chr>        <chr>   <int>
#> 1 Afghanistan 1999      745
#> 2 Brazil      1999    37737
#> 3 China       1999   212258
#> 4 Afghanistan 2000     2666
#> 5 Brazil      2000    80488
#> 6 China       2000   213766
```

Note that "1999" and "2000" are non-syntactic names (because they don't start with a letter) so we have to surround them in backticks. To refresh your memory of the other ways to select columns, see select.

In the final result, the gathered columns are dropped, and we get new `key` and `value` columns. Otherwise, the relationships between the original variables are preserved. Visually, this is shown in Figure 6.2. We can

use `gather()` to tidy `table4b` in a similar fashion. The only difference is the variable stored in the cell values.

To combine the tidied versions of `table4a` and `table4b` into a single tibble, we need to use `dplyr::left_join()`, which you'll learn about in [relational data].

```
tidy4a <- table4a %>%
  gather(key = "year", value = "cases", `1999`, `2000`)
tidy4b <- table4b %>%
  gather(key = "year", value = "population", `1999`, `2000`)
left_join(tidy4a, tidy4b)
#> Joining, by = c("country", "year")
#> # A tibble: 6 x 4
#>   country      year   cases population
#>   <chr>        <chr>  <int>      <int>
#> 1 Afghanistan 1999     745   19987071
#> 2 Brazil      1999   37737  172006362
#> 3 China       1999  212258 1272915272
#> 4 Afghanistan 2000    2666   20595360
#> 5 Brazil      2000   80488  174504898
#> 6 China       2000  213766 1280428583
```

6.3.2 Spreading

Spreading is the opposite of gathering. You use it when an observation is scattered across multiple rows. For example, take `table2`: an observation is a country in a year, but each observation is spread across two rows.

```
table2
#> # A tibble: 12 x 4
#>   country      year type            count
#>   <chr>       <int> <chr>           <int>
#> 1 Afghanistan  1999 cases             745
#> 2 Afghanistan  1999 population   19987071
#> 3 Afghanistan  2000 cases            2666
#> 4 Afghanistan  2000 population   20595360
#> 5 Brazil       1999 cases           37737
#> 6 Brazil       1999 population  172006362
#> # ... with 6 more rows
```

To tidy this up, we first analyse the representation in similar way to `gather()`. This time, however, we only need two parameters:

* The column that contains variable names, the `key` column. Here, it's `type`.

* The column that contains values from multiple variables, the `value` column. Here it's `count`.

Once we've figured that out, we can use `spread()`, as shown programmatically below, and visually in Figure 6.3.

```
table2 %>%
    spread(key = type, value = count)
#> # A tibble: 6 x 4
#>   country      year  cases population
#>   <chr>       <int>  <int>      <int>
#> 1 Afghanistan  1999    745   19987071
#> 2 Afghanistan  2000   2666   20595360
```
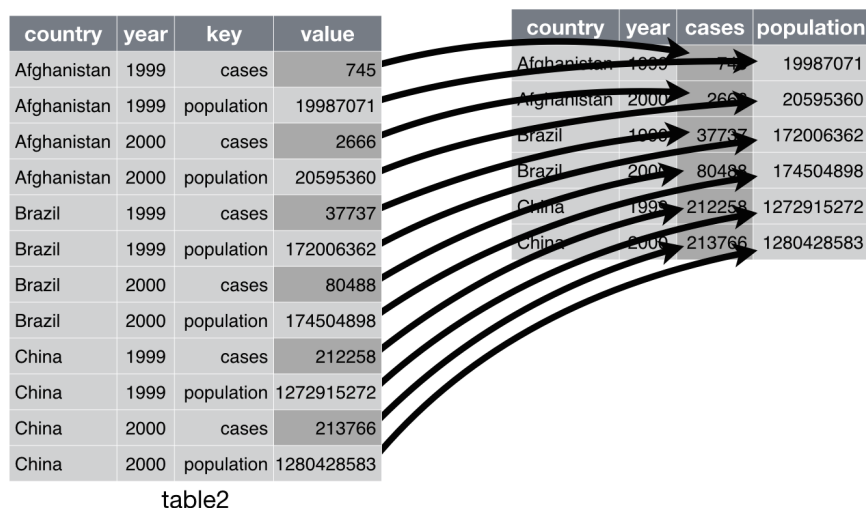
Figure 6.3: Spreading 'table2' makes it tidy

```
#> 3 Brazil        1999   37737   172006362
#> 4 Brazil        2000   80488   174504898
#> 5 China         1999  212258  1272915272
#> 6 China         2000  213766  1280428583
```

> As you might have guessed from the common `key` and `value` arguments, `spread()` and `gather()` are complements. `gather()` makes wide tables narrower and longer; `spread()` makes long tables shorter and wider.

6.4   Separating and uniting

So far you've learned how to tidy `table2` and `table4`, but not `table3`. `table3` has a different problem: we have one column (`rate`) that contains two variables (`cases` and `population`). To fix this problem, we'll need the `separate()` function. You'll also learn about the complement of `separate()`: `unite()`, which you use if a single variable is spread across multiple columns.

6.4.1   Separate

`separate()` pulls apart one column into multiple columns, by splitting wherever a separator character appears. Take `table3`:

```
table3
#> # A tibble: 6 x 3
#>   country      year rate
#> * <chr>       <int> <chr>
#> 1 Afghanistan  1999 745/19987071
#> 2 Afghanistan  2000 2666/20595360
#> 3 Brazil       1999 37737/172006362
```

country	year	rate		country	year	cases	population
Afghanistan	1999	**745** / 19987071		Afghanistan	1999	**745**	19987071
Afghanistan	2000	**2666** / 20595360		Afghanistan	2000	**2666**	20595360
Brazil	1999	**37737** / 172006362		Brazil	1999	**37737**	172006362
Brazil	2000	**80488** / 174504898		Brazil	2000	**80488**	174504898
China	1999	**212258** / 1272915272		China	1999	**212258**	1272915272
China	2000	**213766** / 1280428583		China	2000	**213766**	1280428583

table3

Figure 6.4: Separating 'table3' makes it tidy

```
#> 4 Brazil        2000 80488/174504898
#> 5 China         1999 212258/1272915272
#> 6 China         2000 213766/1280428583
```

The `rate` column contains both `cases` and `population` variables, and we need to split it into two variables. `separate()` takes the name of the column to separate, and the names of the columns to separate into, as shown in Figure 6.4 and the code below.

```
table3 %>%
  separate(rate, into = c("cases", "population"))
#> # A tibble: 6 x 4
#>   country       year cases   population
#> * <chr>        <int> <chr>   <chr>
#> 1 Afghanistan   1999 745     19987071
#> 2 Afghanistan   2000 2666    20595360
#> 3 Brazil        1999 37737   172006362
#> 4 Brazil        2000 80488   174504898
#> 5 China         1999 212258  1272915272
#> 6 China         2000 213766  1280428583
```

> By default, `separate()` will split values wherever it sees a non-alphanumeric character (i.e. a character that isn't a number or letter). For example, in the code above, `separate()` split the values of `rate` at the forward slash characters. If you wish to use a specific character to separate a column, you can pass the character to the `sep` argument of `separate()`. Formally, `sep` is a regular expression, which we learn more about in [strings].

Look carefully at the column types: you'll notice that `cases` and `population` are character columns. This is the default behavior in `separate()`: it leaves the type of the column as is. Here, however, it's not very useful as those really are numbers. We can ask `separate()` to try and convert to better types using `convert = TRUE`:

```
table3 %>%
  separate(rate, into = c("cases", "population"), convert = TRUE)
#> # A tibble: 6 x 4
#>   country       year  cases population
#> * <chr>        <int>  <int>      <int>
```
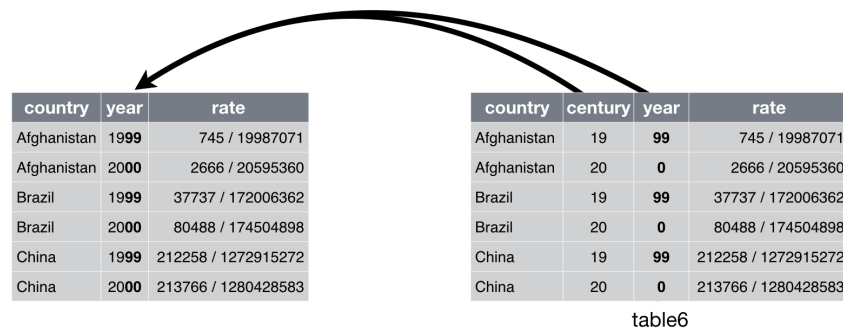
country	year	rate
Afghanistan	1999	745 / 19987071
Afghanistan	2000	2666 / 20595360
Brazil	1999	37737 / 172006362
Brazil	2000	80488 / 174504898
China	1999	212258 / 1272915272
China	2000	213766 / 1280428583

country	century	year	rate
Afghanistan	19	99	745 / 19987071
Afghanistan	20	0	2666 / 20595360
Brazil	19	99	37737 / 172006362
Brazil	20	0	80488 / 174504898
China	19	99	212258 / 1272915272
China	20	0	213766 / 1280428583

table6

Figure 6.5: Uniting 'table5' makes it tidy

```
#> 1 Afghanistan  1999     745    19987071
#> 2 Afghanistan  2000    2666    20595360
#> 3 Brazil       1999   37737   172006362
#> 4 Brazil       2000   80488   174504898
#> 5 China        1999  212258  1272915272
#> 6 China        2000  213766  1280428583
```

You can also pass a vector of integers to `sep`. `separate()` will interpret the integers as positions to split at. Positive values start at 1 on the far-left of the strings; negative value start at -1 on the far-right of the strings. When using integers to separate strings, the length of `sep` should be one less than the number of names in `into`.

You can use this arrangement to separate the last two digits of each year. This make this data less tidy, but is useful in other cases, as you'll see in a little bit.

```
table3 %>%
  separate(year, into = c("century", "year"), sep = 2)
#> # A tibble: 6 x 4
#>   country     century year  rate
#> * <chr>       <chr>   <chr> <chr>
#> 1 Afghanistan 19      99    745/19987071
#> 2 Afghanistan 20      00    2666/20595360
#> 3 Brazil      19      99    37737/172006362
#> 4 Brazil      20      00    80488/174504898
#> 5 China       19      99    212258/1272915272
#> 6 China       20      00    213766/1280428583
```

6.4.2   Unite

`unite()` is the inverse of `separate()`: it combines multiple columns into a single column. You'll need it much less frequently than `separate()`, but it's still a useful tool to have in your back pocket.

We can use `unite()` to rejoin the *century* and *year* columns that we created in the last example. That data is saved as `tidyr::table5`. `unite()` takes a data frame, the name of the new variable to create, and a set of columns to combine, again specified in `dplyr::select()` style:

```
table5 %>%
  unite(new, century, year)
#> # A tibble: 6 x 3
```

```
#>   country    new    rate
#>   <chr>      <chr> <chr>
#> 1 Afghanistan 19_99 745/19987071
#> 2 Afghanistan 20_00 2666/20595360
#> 3 Brazil      19_99 37737/172006362
#> 4 Brazil      20_00 80488/174504898
#> 5 China       19_99 212258/1272915272
#> 6 China       20_00 213766/1280428583
```

In this case we also need to use the `sep` argument. The default will place an underscore (`_`) between the values from different columns. Here we don't want any separator so we use `""`:

```
table5 %>%
  unite(new, century, year, sep = "")
#> # A tibble: 6 x 3
#>   country    new    rate
#>   <chr>      <chr> <chr>
#> 1 Afghanistan 1999  745/19987071
#> 2 Afghanistan 2000  2666/20595360
#> 3 Brazil      1999  37737/172006362
#> 4 Brazil      2000  80488/174504898
#> 5 China       1999  212258/1272915272
#> 6 China       2000  213766/1280428583
```

6.5   Missing values

Changing the representation of a dataset brings up an important subtlety of missing values. Surprisingly, a value can be missing in one of two possible ways:

- **Explicitly**, i.e. flagged with `NA`.
- **Implicitly**, i.e. simply not present in the data.

Let's illustrate this idea with a very simple data set:

```
mydata <- tibble(
  year   = c(2015, 2015, 2015, 2015, 2016, 2016, 2016),
  qtr    = c(   1,    2,    3,    4,    2,    3,    4),
  rate   = c(1.88, 0.59, 0.35,   NA, 0.92, 0.17, 2.66)
)

mydata
#> # A tibble: 7 x 3
#>    year   qtr  rate
#>   <dbl> <dbl> <dbl>
#> 1  2015     1  1.88
#> 2  2015     2  0.59
#> 3  2015     3  0.35
#> 4  2015     4 NA
#> 5  2016     2  0.92
#> 6  2016     3  0.17
#> # ... with 1 more row
```

There are two missing values in this dataset:

- The rate for the fourth quarter of 2015 is explicitly missing, because the cell where its value should be instead contains `NA`.

- The rate for the first quarter of 2016 is implicitly missing, because it simply does not appear in the dataset.

The way that a dataset is represented can make implicit values explicit. For example, we can make the implicit missing value explicit by putting years in the columns:

```
mydata %>%
  spread(year, rate)
#> # A tibble: 4 x 3
#>     qtr `2015` `2016`
#>   <dbl>  <dbl>  <dbl>
#> 1     1   1.88     NA
#> 2     2   0.59   0.92
#> 3     3   0.35   0.17
#> 4     4     NA   2.66
```

Because these explicit missing values may not be important in other representations of the data, you can set `na.rm = TRUE` in `gather()` to turn explicit missing values implicit:

```
mydata %>%
  spread(year, rate) %>%
  gather(year, rate, `2015`:`2016`, na.rm = TRUE)
#> # A tibble: 6 x 3
#>     qtr year   rate
#> * <dbl> <chr> <dbl>
#> 1     1 2015   1.88
#> 2     2 2015   0.59
#> 3     3 2015   0.35
#> 4     2 2016   0.92
#> 5     3 2016   0.17
#> 6     4 2016   2.66
```

Another important tool for making missing values explicit in tidy data is `complete()`:

```
mydata %>%
  complete(year, qtr)
#> # A tibble: 8 x 3
#>    year   qtr   rate
#>   <dbl> <dbl>  <dbl>
#> 1  2015     1   1.88
#> 2  2015     2   0.59
#> 3  2015     3   0.35
#> 4  2015     4     NA
#> 5  2016     1     NA
#> 6  2016     2   0.92
#> # ... with 2 more rows
```

`complete()` takes a set of columns, and finds all unique combinations. It then ensures the original dataset contains all those values, filling in explicit `NA`s where necessary.

It is also possible that you have such incomplete data that `complete()` does not have all the data needed for complete cases. Imagine in the above example if the year 2016 data wat instead complete missing but you had 2017 data. In this case a different technique would need to be used.

> A common solution to this problem is to use `crossing()` to make the compete cases then "join" the data set to the complete cases.

There's one other important tool that you should know for working with missing values. Sometimes when a data source has primarily been used for data entry (tables from *Excel*, *pdf*, and *Word*), missing values indicate that the previous value should be carried forward:

```
treatment <- tribble(
  ~ person,           ~ treatment, ~response,
  "Derrick Whitmore", 1,           7,
  NA,                 2,           10,
  NA,                 3,           9,
  "Katherine Burke",  1,           4
)

treatment
#> # A tibble: 4 x 3
#>   person          treatment response
#>   <chr>               <dbl>    <dbl>
#> 1 Derrick Whitmore        1        7
#> 2 <NA>                    2       10
#> 3 <NA>                    3        9
#> 4 Katherine Burke         1        4
```

You can fill in these missing values with `fill()`. It takes a set of columns where you want missing values to be replaced by the most recent non-missing value (sometimes called last observation carried forward).

```
treatment %>%
  fill(person)
#> # A tibble: 4 x 3
#>   person          treatment response
#>   <chr>               <dbl>    <dbl>
#> 1 Derrick Whitmore        1        7
#> 2 Derrick Whitmore        2       10
#> 3 Derrick Whitmore        3        9
#> 4 Katherine Burke         1        4
```

6.6 Non-tidy data

Before we continue on to other topics, it's worth talking briefly about non-tidy data. Earlier in the chapter, I used the pejorative term "messy" to refer to non-tidy data. That's an oversimplification: there are lots of useful and well-founded data structures that are not tidy data. There are two main reasons to use other data structures:

- Alternative representations may have substantial performance or space advantages.

- Specialized fields have evolved their own conventions for storing data that may be quite different to the conventions of tidy data.

Either of these reasons means you'll need something other than a tibble (or data frame). If your data does fit naturally into a rectangular structure composed of observations and variables, I think tidy data should be your default choice. But there are good reasons to use other structures; tidy data is not the only way.

If you'd like to learn more about non-tidy data, I'd highly recommend this thoughtful blog post by Jeff Leek: http://simplystatistics.org/2016/02/17/non-tidy-data/

Part IV

Beyond Basics

Chapter 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

7.1 Introduction to Functions

Functions are an central part of robust R programming and we will spend a significant amount of time writing functions. Think of functions in the mathematical sense will make the properties much more apparent than any other framework.

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.

- 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

Advice 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.

> Software testing is important, but, in part because it is frustrating and boring, many of us avoid it. 'testthat' is a testing framework for R that is easy learn and use, and integrates with your existing 'workflow'.

7.1.1   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.

```r
myadd <- function(x, y) {
  message(paste0("x = ", x, "\n"))
  message(paste0("y = ", y, "\n"))
  x + y
}
```

- 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 environment. (We'll discuss environments more in the next section.)

When calling a function you can pass the parameters **in order**, **by name**, or a combination.

```r
myadd(1, 3)              # arguments by position
#> x = 1
#> y = 3
#> [1] 4
myadd(x = 1, y = 3)      # arguments by name
#> x = 1
#>
#> y = 3
#> [1] 4
myadd(y = 3, x = 1)      # name order doesn't matter
#> x = 1
```

```
#>
#> y = 3
#> [1] 4
myadd(y = 3, 1)          # combination
#> x = 1
#>
#> y = 3
#> [1] 4
```

> 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. Convention is to pass arguments in the order the function defines them, and to use the arguments names if the function takes more than 2 or 3 arguments.

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 default 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
```

7.1.2 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. It's common to write a function that does not use an argument and not notice it simply because R never throws an error.

7.1.3 The Dot-dot-dot (. . .) 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 two most common cases for using . . . in a function are:

1. The number of arguments passed to the function cannot be known in advance.
2. Extending another function and you don't want to copy the entire argument list of the original function.

Number of arguments passed to the function cannot be known in advance.

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()`, `cat()`, `sum()`, and `mean()`.

```
args(paste)
#> function (..., sep = " ", collapse = NULL)
#> NULL
args(cat)
#> function (..., file = "", sep = " ", fill = FALSE, labels = NULL,
#>     append = FALSE)
#> NULL
args(sum)
#> function (..., na.rm = FALSE)
#> NULL
args(mean)
```

```
#> function (x, ...)
#> 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 `...`. Similarly with `sum()`, and `mean()`.

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.

Extending another 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 `plot` default is `type = "p"`).

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

Sometimes you will combine both in one function.

```
commas <- function(...) {
  paste(..., sep = "", collapse = ", ")
}

commas(letters[1:10])
#> [1] "a, b, c, d, e, f, g, h, i, j"
```

7.2   Environments & Scoping

An **environment** is a collection of (symbol, value) pairs, i.e. `x <- 10`, `x` is a symbol and `10` 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.

```r
x <- 0

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

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

h <- function() {
  y <- 1
  c(x, y)
}
```

What do the following return?

- `f()`
- `g()`
- `h()`
- `g(h())`
- `f(g())`
- `g(f())`

7.3 "First class objects"

Since functions **ARE** objects you can pass functions as arguments and return functions as results.

```r
my_summary <- function(x, funs = c(mean, sd), ...) {
  lapply(funs, function(f) f(x, na.rm = TRUE))
}

y <- 1:10
my_summary(y)
#> [[1]]
#> [1] 5.5
#>
#> [[2]]
#> [1] 3.03
my_summary(y, c(mean, median, sd, IQR, mad))
#> [[1]]
#> [1] 5.5
#>
#> [[2]]
```

```
#> [1] 5.5
#>
#> [[3]]
#> [1] 3.03
#>
#> [[4]]
#> [1] 4.5
#>
#> [[5]]
#> [1] 3.71
```

Unlike most languages you can define a function within a function and / or return a function. This nested function only lives inside the parent function.

```r
make.power <- function(n) {
  # [TBD] checks on n
  pow <- function(x) {
      # [TBD] checks on x
    x^n
  }
  pow
}

make.power(4)  # returns a function
#> function(x) {
#>       # [TBD] checks on x
#>     x^n
#>   }
#> <environment: 0x0000000007bf8c40>
pow(x=4)       # Note: `pow` does not exist outside of the `make.power` function
#> Error in pow(x = 4): could not find function "pow"

cube <- make.power(3)
as.list(environment(cube))
#> $pow
#> function (x)
#> {
#>     x^n
#> }
#> <bytecode: 0x00000000078959f8>
#> <environment: 0x00000000086c7f18>
#>
#> $n
#> [1] 3
cube(2)
#> [1] 8

square <- make.power(2)
squareroot <- make.power(.5)


square(8)
#> [1] 64
squareroot(9)
```

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

7.4 Exercises

1. Create function that takes a numeric year-quater (ex 20183) and returns the quarter n-quarters before / after it. Example two quarters previous to 20183 is 20181.
2. Come up with 5 functions (you don't have program them) that will operate on your data. (Ex. Create a demographics table)
3. Create a `read_*` function that

- reads in the data file
- converts all columns to the appropriate data types
- "Tidy" your data (if appropriate)
  The first argument to your read function should be the file name. Are there additional parameters that are needed? Think beyond the Subscriber Report, what are other things you typically do upon first reading in a data file.

Chapter 8

R Markdown

8.1 Introduction

What is R Markdown

R Markdown provides an unified authoring framework for data science, combining your code, its results, and your prose commentary. R Markdown documents are fully reproducible and support dozens of static and dynamic output formats like PDFs, Word files, slideshows, and more.. For a comprehensive resource on R Markdown see R Markdown: The Definitive Guide

R Markdown integrates a number of R packages and external tools. This means that help is, by-and-large, not available through ?. Instead, as you work through this chapter, and use R Markdown in the future, keep these resources close to hand:

- R Markdown Cheat Sheet: Help > Cheatsheets > R Markdown Cheat Sheet,
- R Markdown Reference Guide: Help > Cheatsheets > R Markdown Reference Guide.

Both cheatsheets are also available at http://rstudio.com/cheatsheets.

The real point of R Markdown is that it lets you include your code, have the code run automatically when your document is rendered, and seamlessly include the results of that code in your document.

8.1.1 Code languages

While there is an **"R"** in R Markdown this is not limited to just the R programming language. It is a generic markup language that can knit any code that can be run from a command line into a document. Some of the more popular languages are:

- R
- Python
- SQL
- C/C++
- Bash
- Rcpp
- Stan
- JavaScript
- CSS
- Julia
- SAS (see SASmarkdown package)

You can intermix languages within the same document, and in some cases pass data between languages.

> You do not have to include any programming language. There are many books, websites, and wikis written in R Markown which contain no code.

8.2   Markdown Basics

Format the text in your R Markdown file with Pandoc's Markdown, a set of markup annotations for plain text files. When you render your file, Pandoc transforms the marked up text into formatted text in your final file format.

Notice that the file contains three types of content:

1. Text mixed with simple text formatting.
2. Code chunks and the corresponding output.
3. An (optional) YAML header controlling the "whole document" settings.

8.2.1   Headers

The character **#** at the beginning of a line means that the rest of the line is interpreted as a section header. The number of **#**s at the beginning of the line indicates whether it is treated as a section, sub-section, sub-sub-section, etc. of the document.

```
# Heading Level 1
## Heading Level 2
### Heading Level 3
```

In this document the chapter title **R Markdown** is preceded by a single **#**, but **Markdown Basics** at the start of this paragraph was preceded by **##** and the current section **Headers** is preceded by **###** in the text file.

8.2.2   Paragraph Breaks and Forced Line Breaks

```
To insert a break between paragraphs, include a single completely blank line.
```

```
To force a line break, put two blank
spaces at the end of a line.
```

To insert a break between paragraphs, include a single completely blank line.

To force a line break, put two blank
spaces at the end of a line.

```
If you don't put the two blank
spaces at the end of a line
they will run together.
```

If you don't put the two blank spaces at the end of a line they will run together.

8.2.3   Italics and Boldface

```
Text to be _italicized_ goes inside _a single set of underscores_ or *asterisks*.  Text to be **boldfac
```

Text to be *italicized* goes inside *a single set of underscores* or *asterisks.* Text to be **boldfaced** goes inside a **double set of underscores** or **asterisks**.

8.2.4 Bullet Points

```
* This is a list marked where items are marked with bullet points.
* Each item in the list should start with a `*` (asterisk) character, or a single dash (`-`) and then ha
* Each item should also be on a new line.
    + Indent lines with 4 spaces and begin them with `+` for sub-bullets.
    + Sub-sub-bullet aren't really a thing in R Markdown.
```

- This is a list marked where items are marked with bullet points.
- Each item in the list should start with a * (asterisk) character, or a single dash (-) and then have a space.
- Each item should also be on a new line.
    - Indent lines with 4 spaces and begin them with + for sub-bullets.
    - Sub-sub-bullet are not't really a thing in R Markdown.

8.2.5 Numbered Lists

```
1. Lines which begin with a numeral (0-9), followed by a period, will usually be interpreted as items i
1. R Markdown handles the numbering in what it renders automatically, so the actual number doesn't matte
1. This can be handy when you lose count or don't update the numbers yourself when editing. (Look caref
    a. Sub-lists of numbered lists, with letters for sub-items, are a thing.
    b. They are however a fragile thing, which you'd better not push too hard.
```

1. Lines which begin with a numeral (0–9), followed by a period, will usually be interpreted as items in a numbered list.
2. R Markdown handles the numbering in what it renders automatically, so the actual number doesn't matter.
3. This can be handy when you lose count or don't update the numbers yourself when editing. (Look carefully at the .Rmd file for this item.)
    a. Sub-lists of numbered lists, with letters for sub-items, are a thing.
    b. They are however a fragile thing, which you'd better not push too hard.

8.3 Math

R Markdown uses standard LaTeX to render complex mathematical formulas and derivations, and have them displayed very nicely. Like code, the math can either be inline or set off (displays).

Inline math is marked off with a pair of dollar signs (\$), `$\frac{a+b}{b} = 1 + \frac{a}{b}$` $\frac{a+b}{b} = 1 + \frac{a}{b}$

Mathematical displays are marked off with \[and\], as in

`\[ \frac{a+b}{b} = 1 + \frac{a}{b} \]`

$$\frac{a+b}{b} = 1 + \frac{a}{b}$$

8.4   Code Chunks

Think of a chunk like a function. A chunk should be relatively self-contained, and focused around a single task. You can quickly insert chunks like these into your file with

- the keyboard shortcut **Ctrl + Alt + I** (OS X: **Cmd + Option + I**)
- typing the chunk delimiters ```` ```{r} ```` and ```` ``` ````.
- the *Add Chunk* command in the editor toolbar

When you render your .Rmd file, R Markdown will run each code chunk and embed the results beneath the code chunk in your final report.

There are three main sections to a the code chunk header:

1. the programming language engine to run the code
2. the code chunk name (optional but very useful)
3. the code chunk options.

8.4.1   Chunk Names

Chunks can be given an optional name: ```` ```{r by-name} ````. This has three advantages:

1. You can more easily navigate to specific chunks using the drop-down code navigator in the bottom-left of the script editor:
2. Graphics produced by the chunks will have useful names that make them easier to use elsewhere.
3. You can set up networks of cached chunks to avoid re-performing expensive computations on every run. More on that below.

It's a good idea to name code chunks that produce figures, even if you don't routinely label other chunks. The chunk label is used to generate the file name of the graphic on disk, so naming your chunks makes it much easier to pick out plots and reuse in other circumstances (i.e. if you want to quickly drop a single plot into an email)

> There is one chunk name wit special behaviour: `setup`. In notebook mode, the chunk named `setup` will be run automatically once, before any other code is run.

8.4.2   Chunk Options

Chunk output can be customized with **options**, arguments supplied to chunk header. Knitr provides almost 60 options that you can use to customize your code chunks. Here we'll cover the most important chunk options that you'll use frequently. You can see the full list at http://yihui.name/knitr/options/.

The most important set of options controls if your code block is executed and what results are inserted in the finished report:

- `include = FALSE` prevents code and results from appearing in the finished file. Use this to run code and results used by other chunks (i.e. setup code).

- `echo = FALSE` prevents code, but not the results from appearing in the finished file. Use this when writing reports.

- `message = FALSE` or `warning = FALSE` prevents messages or warnings from appearing in the finished file.

- `results = 'hide'` hides printed output; `fig.show = 'hide'` hides plots.

- `error = TRUE` causes the render to continue even if code returns an error. Use this for debugging.

To set global options that apply to every chunk in your file, call `knitr::opts_chunk$set` in a code chunk. Knitr will treat each option that you pass to `knitr::opts_chunk$set` as a global default that can be overwritten in individual chunk headers If you are doing a report where you would never show code include `knitr::opts_chunk$set(echo = FALSE)` in the `setup` code block.

```
\```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = FALSE)
\```
```

8.5   Inline Code

Code output can also be seamlessly incorporated into the text, using inline code. This is code not set off on a line by itself, but beginning with `` `r `` and ending with `` ` ``. Using inline code is how this document knows that the mtcars data set contains 32 rows, and that the mean mpg of the 6 cylinder cars is 19.7.

R Markdown will always

- display the results of inline code, but not the code
- apply relevant text formatting to the results

As a result, inline output is indistinguishable from the surrounding text. Inline expressions do not take knitr options.

8.6   Plots

Inserting a graph into RMarkdown is easy, the more energy-demanding aspect might be adjusting the formatting.

```
\```{r mtcarsplot1, fig.cap = "Default Plot"}
ggplot(data = mpg, mapping = aes(x = class, y = hwy)) +
  geom_boxplot() +
  coord_flip()
\```
```

```
ggplot(data = mpg, mapping = aes(x = class, y = hwy)) +
  geom_boxplot() +
  coord_flip()
```

By default, RMarkdown will place graphs by maximizing their height, while keeping them within the margins of the page and maintaining aspect ratio. If you have a particularly tall figure, this can mean a really huge graph. To manually set the figure dimensions in the code chunk options use `fig.height` and `fig.width`.

You can also set the alignment with `fig.align` and either `left`, `right` or `center`. Also, by default the file type of the plot is determined by the output file type. You can override these defaults by using the `dev` option in the code chunk options. See R Markdown: The Definitive Guide for more details.

8.7   Tables

By default, R Markdown displays data frames and matrices as they would be in the R terminal (in a monospaced font). This generally is not what you want. If you prefer that data be displayed with additional formatting there are a number of packages that can make publication ready tables.

- kable + kableExtra
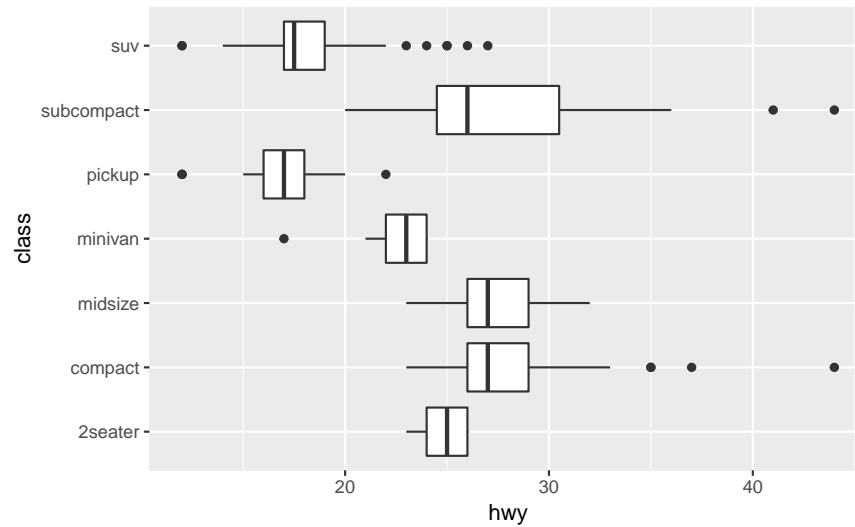
Figure 8.1: Default Plot

Table 8.1: kable table

	mpg	cyl	disp	hp	drat	wt
Mazda RX4	21.0	6	160	110	3.90	2.62
Mazda RX4 Wag	21.0	6	160	110	3.90	2.88
Datsun 710	22.8	4	108	93	3.85	2.32
Hornet 4 Drive	21.4	6	258	110	3.08	3.21
Hornet Sportabout	18.7	8	360	175	3.15	3.44

- xtable
- stargazer
- pander
- tables
- ascii

Which package you should use depends on what your output format is and which features you want. All these packages have vignettes which show how to use the major features.

```r
library(kableExtra)
dt <- mtcars[1:5, 1:6]
kable(dt, caption = "kable table", booktabs = TRUE)

kable(mtcars, longtable = TRUE, booktabs = TRUE,
      caption = "Kable Longtable") %>%
  add_header_above(c(" ","Group 1" = 5, "Group 2" = 6)) %>%
  kable_styling(latex_options = c("repeat_header"))
```

Table 8.2: Kable Longtable

	Group 1					Group 2					
	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160.0	110	3.90	2.62	16.5	0	1	4	4
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.88	17.0	0	1	4	4

Table 8.2: Kable Longtable *(continued)*

	Group 1					Group 2					
	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Datsun 710	22.8	4	108.0	93	3.85	2.32	18.6	1	1	4	1
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.21	19.4	1	0	3	1
Hornet Sportabout	18.7	8	360.0	175	3.15	3.44	17.0	0	0	3	2
Valiant	18.1	6	225.0	105	2.76	3.46	20.2	1	0	3	1
Duster 360	14.3	8	360.0	245	3.21	3.57	15.8	0	0	3	4
Merc 240D	24.4	4	146.7	62	3.69	3.19	20.0	1	0	4	2
Merc 230	22.8	4	140.8	95	3.92	3.15	22.9	1	0	4	2
Merc 280	19.2	6	167.6	123	3.92	3.44	18.3	1	0	4	4
Merc 280C	17.8	6	167.6	123	3.92	3.44	18.9	1	0	4	4
Merc 450SE	16.4	8	275.8	180	3.07	4.07	17.4	0	0	3	3
Merc 450SL	17.3	8	275.8	180	3.07	3.73	17.6	0	0	3	3
Merc 450SLC	15.2	8	275.8	180	3.07	3.78	18.0	0	0	3	3
Cadillac Fleetwood	10.4	8	472.0	205	2.93	5.25	18.0	0	0	3	4
Lincoln Continental	10.4	8	460.0	215	3.00	5.42	17.8	0	0	3	4
Chrysler Imperial	14.7	8	440.0	230	3.23	5.34	17.4	0	0	3	4
Fiat 128	32.4	4	78.7	66	4.08	2.20	19.5	1	1	4	1
Honda Civic	30.4	4	75.7	52	4.93	1.61	18.5	1	1	4	2
Toyota Corolla	33.9	4	71.1	65	4.22	1.83	19.9	1	1	4	1
Toyota Corona	21.5	4	120.1	97	3.70	2.46	20.0	1	0	3	1
Dodge Challenger	15.5	8	318.0	150	2.76	3.52	16.9	0	0	3	2
AMC Javelin	15.2	8	304.0	150	3.15	3.44	17.3	0	0	3	2
Camaro Z28	13.3	8	350.0	245	3.73	3.84	15.4	0	0	3	4
Pontiac Firebird	19.2	8	400.0	175	3.08	3.85	17.1	0	0	3	2
Fiat X1-9	27.3	4	79.0	66	4.08	1.94	18.9	1	1	4	1
Porsche 914-2	26.0	4	120.3	91	4.43	2.14	16.7	0	1	5	2
Lotus Europa	30.4	4	95.1	113	3.77	1.51	16.9	1	1	5	2
Ford Pantera L	15.8	8	351.0	264	4.22	3.17	14.5	0	1	5	4
Ferrari Dino	19.7	6	145.0	175	3.62	2.77	15.5	0	1	5	6
Maserati Bora	15.0	8	301.0	335	3.54	3.57	14.6	0	1	5	8
Volvo 142E	21.4	4	121.0	109	4.11	2.78	18.6	1	1	4	2

8.8 YAML

At the top of any RMarkdown script is a YAML header section enclosed by `---`. By default this includes a title, author, date and the file type you want to output to. Many other options are available for different functions and formatting, see here for .html options and here for .pdf options. Rules in the header section will alter the whole document. Have a flick through quickly to familiarize yourself with the sorts of things you can alter by adding an option to the YAML header.

8.8.1 Table of Contents

```
---
title: "My Report"
output:
```

```
  pdf_document:
    toc: true
    toc_depth: 2
    number_sections: TRUE
---
```

8.8.2   Parameterized reports

One of the many benefits of working with R Markdown is that you can reproduce analysis at the click of a button. This makes it very easy to update any work and alter any input parameters within the report. Parameterized reports extend this one step further, and allow users to specify one or more parameters to customize the analysis. This is useful if you want to create a report template that can be reused across multiple similar scenarios.

Declaring parameters

Parameters are specified using the params field within the YAML section. We can specify one or more parameters with each item on a new line. As an example:

```
---
title: "My Report"
output: pdf_document
params:
  year: 2018
  region: Europe
  data: file.csv
---
```

All standard R types that can be included as parameters, including `character`, `numeric`, `integer`, and `logical` types. We can also use R objects by including `!r` before R expressions. For example, we could include the current date with the following R code:

```
---
title: "My Report"
output: html_document
params:
  date: !r Sys.Date()
---
```

Using parameters

You can access the parameters within the knitting environment and the R console in RStudio. The values are contained within a read-only list called `params`. In the previous example, the parameters can be accessed as follows:

```
params$year
params$region
```

```
mydata <- read_csv(params$data)
```

Knitting with parameters

- Using the **Knit** button within RStudio to run with the default values.
- Using the **Knit with Parameters** button within RStudio to fill in the parameters interactively.
- Knit with custom parameters

```
render_report = function(file, region, year) {
  rmarkdown::render(
    "MyDocument.Rmd", params = list(
```

```
      region = region,
      year   = year,
      data   = file
    ),
    output_file = paste0("Report-", region, "-", year, ".pdf")
  )
}
```

8.8.3 Output Types

There are many types of output with more being added every day. lists just the ones

- Documents
    - `pdf_document` makes a PDF with LaTeX
    - `word_document` for Microsoft Word documents (`.docx`).
    - `odt_document` for OpenDocument Text documents (`.odt`).
    - `rtf_document` for Rich Text Format (`.rtf`) documents.
    - `md_document` for a Markdown document.
    - `github_document`: this is a tailored version of `md_document` designed for sharing on GitHub.
- Presentations
    - `ioslides_presentation` - HTML presentation with ioslides
    - `slidy_presentation` - HTML presentation with W3C Slidy
    - `beamer_presentation` - PDF presentation with LaTeX Beamer.
    - `revealjs::revealjs_presentation` - HTML presentation with reveal.js.
    - **rmdshower**, https://github.com/MangoTheCat/rmdshower, provides a wrapper around the **shower**, https://github.com/shower/shower, presentation engine
    - `powerpoint_presentation` - Create MS PowerPoint presentations.
- Notebooks
    - `html_notebook` is a variation on a `html_document`.
- Dashboards
    - `flexdashboard::flex_dashboard`, http://rmarkdown.rstudio.com/flexdashboard/ provides simple tools for creating sidebars, tabsets, value boxes, and gauges.
- Interactivity
    - **shiny**, a package that allows you to create interactivity using R code, not JavaScript.
    - **htmlwidgets**, produce interactive HTML visualizations
    - **dygraphs**, http://rstudio.github.io/dygraphs/, for interactive time series visualizations.
    - **DT**, http://rstudio.github.io/DT/, for interactive tables.
    - **threejs**, https://github.com/bwlewis/rthreejs for interactive 3d plots.
    - **DiagrammeR**, http://rich-iannone.github.io/DiagrammeR/ for diagrams (like flow charts and simple node-link diagrams).
- Other
    - **blogdown** create a simple website.
    - **bookdown**, https://github.com/rstudio/bookdown,makes it easy to write books, like this one
    - **prettydoc**, https://github.com/yixuan/prettydoc/, provides lightweight document formats with a range of attractive themes.
    - **rticles**, https://github.com/rstudio/rticles, compiles a selection of formats tailored for specific scientific journals.

Part V

Appendix

Chapter 9

Appendix A

9.1 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.