

Modern R in a Corporate Environment

R course developed for the office

Brian Davis

2018-08-20

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 Golemund & 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 3.4.4
 - 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.

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 and 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

Making Code Production Ready:

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

1.4 Structure

My current thoughts are to meet an hour a week and discuss a topic. We will not be going strictly through the R4DS, but will use it as our foundation into the topic at hand. Then give 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, but it sure makes things easier to read. When I answer questions; first, I see if I 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 significantly more reliable, code must become more transparent. In particular, nested conditions and loops must be viewed with great suspicion. Complicated control flows confuse programmers. **Messy code often hides bugs.**”

— Bjarne Stroustrup

2.1.1 Comments

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



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

2.1.2 Naming

There are only two hard things in Computer Science: cache invalidation and naming things.

– Phil Karlton

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

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

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

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

2.1.3 Spacing

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

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

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

2.1.4 Indenting

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

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

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

# Bad
if (y < 0 && debug)
```

```
message("Y is negative")

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

2.1.5 Long lines

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

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

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

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

2.1.6 Other

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

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

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

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

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

```

  x + y
}

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

```

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

```

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

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

```

2.2 Coding practices

2.2.1 Variables

Create variables for values that are likely to change.

2.2.2 *Rule of Three*¹

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

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

The *Rule of Three* applies to look-up tables and such also. The key thing to think about is; if something changes how many touch points will there be? If it is 3 or more places it is time to abstract this code a bit.

2.2.3 Path names

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

DO NOT EVER USE `setwd()`

```

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

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

```

¹This 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 Exercises

1. See these RStudio Tips & Tricks or these and find one that looks interesting and **practice** it all week.
2. Create an R Project for this class.
3. Create the following directories in your project (tip sheet?)
 - Bonus points if you can do it from R and not RStudio or Windows Explorer
 - Double Bonus points if you can make it a function.
4. Read Chapters 1-3 of the Tidyverse Style Guide
5. Copy one of your R scripts into your R directory. (Bonus points if you can do it from R and not RStudio or Windows Explorer)
6. Apply the style guide to your code.
7. Apply the "Rule of 3"
 - Create variables as needed
 - Identify code that is used 3 or more times to make functions
 - Identify code that would be useful in 3 or more projects to integrate into a package.
8. Read how to make a great R reproducible example

Part II

Base R Basics

Chapter 3

R Basics

Here is a quick overview of the basics. Next we'll dive deep into R's basic data structures and then how to subset these data structures. 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.

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 constructs¹; objects, functions, and environments.

3.1 Assignment Operators

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

3.2 Objects

3.2.1 Vector

You create a vector with `c`. These have to be the same data type.

```
v <- c("my", "first", "vector")
v
#> [1] "my"      "first"    "vector"

# length of our vector
length(v)
#> [1] 3
```

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

There are several shortcut functions for common vector creation.

```
# create an ordered sequence
2:10
#> [1] 2 3 4 5 6 7 8 9 10
9:3
#> [1] 9 8 7 6 5 4 3

# generate regular sequences
seq(1, 20, by = 3)
#> [1] 1 4 7 10 13 16 19

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

# arguments are generally vectorized
rep(1:3, times = 3:1)
#> [1] 1 1 1 2 2 3

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

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

# another common mistake
n <- 6
1:n+1      # is (1:n) + 1, so 2:(n + 1)
#> [1] 2 3 4 5 6 7
1:(n+1)    # usually what is meant
#> [1] 1 2 3 4 5 6 7
seq_len(n+1) # a better way
#> [1] 1 2 3 4 5 6 7
```

3.2.2 Matrix

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

```
# fill in column order (default)
matrix(1:12, nrow = 3)
#>      [,1] [,2] [,3] [,4]
#> [1,]  1   4   7  10
#> [2,]  2   5   8  11
#> [3,]  3   6   9  12

# fill in row order
matrix(1:12, nrow = 3, byrow = TRUE)
#>      [,1] [,2] [,3] [,4]
#> [1,]  1   2   3   4
#> [2,]  5   6   7   8
```

```
#> [3,]    9   10   11   12

# can also specify the number of columns instead
matrix(1:12, ncol = 3)
#>      [,1] [,2] [,3]
#> [1,]    1    5    9
#> [2,]    2    6   10
#> [3,]    3    7   11
#> [4,]    4    8   12
```

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

```
m <- matrix(1:12, ncol = 3)
dim(m)
#> [1] 4 3
nrow(m)
#> [1] 4
ncol(m)
#> [1] 3
```

3.2.3 List

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

```
s <- c("aa", "bb", "cc", "dd", "ee")
b <- c(TRUE, FALSE, TRUE, FALSE, FALSE)
# x contains copies of n, s, b and our matrix from above
x <- list(n = c(2, 3, 5), s, b, 3, m)
x
#> $n
#> [1] 2 3 5
#>
#> [[2]]
#> [1] "aa" "bb" "cc" "dd" "ee"
#>
#> [[3]]
#> [1] TRUE FALSE TRUE FALSE FALSE
#>
#> [[4]]
#> [1] 3
#>
#> [[5]]
#>      [,1] [,2] [,3]
#> [1,]    1    5    9
#> [2,]    2    6   10
#> [3,]    3    7   11
#> [4,]    4    8   12

# length gives you length of the list not the elements in the list
length(x)
#> [1] 5
```

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

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

3.2.4 Data frame

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

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

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

# dimensions
dim(df)
#> [1] 3 4
nrow(df)
#> [1] 3
ncol(df)
#> [1] 4
length(df)
#> [1] 4
```

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

3.3 Comparison

```
v <- 1:12
v[v > 9]
#> [1] 10 11 12
```

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

```
x <- sqrt(2)
(y <- x^2)
#> [1] 2
y == 2
#> [1] FALSE
```



```
print(y, digits = 20)
#> [1] 2.000000000000000004
all.equal(y, 2)          ## equality with some tolerance
#> [1] TRUE
all.equal(y, 3)
#> [1] "Mean relative difference: 0.5"
isTRUE(all.equal(y, 3))  ## if you want a boolean, use isTRUE()
#> [1] FALSE
```

3.4 Logical and sets

```
x <- c(TRUE, FALSE)
df <- data.frame(expand.grid(x, x))
names(df) <- c("x", "y")
df$and <- df$x & df$y      # logical and
df$or  <- df$x | df$y      # logical or
df$notx <- !df$x           # negation
df$xor <- xor(df$x, df$y) # exclusive 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 double version returns a length-one vector. Use the double version in logical control structures (if, for, while, etc).

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

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



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

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

```
x <- c(FALSE, FALSE, FALSE, TRUE)
any(x)
#> [1] TRUE
all(x)
#> [1] FALSE
all(!x[1:3])
#> [1] TRUE
```

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

```

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

intersect(x, y) # in x and in y
#> [1] 3 4 5
union(x, y)     # different than c()
#> [1] 1 2 3 4 5 6 7
c(x,y)         # not a set operation
#> [1] 1 2 3 4 5 3 4 5 6 7
setdiff(x, y)   # in x but not in y
#> [1] 1 2
setdiff(y, x)   # in y but not in x
#> [1] 6 7
setequal(x, y)
#> [1] FALSE
z <- 5:1
setequal(x, z)
#> [1] TRUE

```

3.5 Control Structures

Control structures allow you to put some “logic” into your R code, rather than just always executing the same R code every time. Control structures allow you to respond to inputs or to features of the data and execute different R expressions accordingly.

Commonly used control structures are

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

3.5.1 if-else

The **if-else** combination is probably the most commonly used control structure in R (or perhaps any language). This structure allows you to test a condition and act on it depending on whether it’s true or false.

For starters, you can just use the **if** statement.

```

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

```

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

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

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

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



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.

3.5.2 for Loops

For loops are pretty much the only looping construct that you will need in R. While you may occasionally find a need for other types of loops, in my experience doing data analysis, I've found very few situations where a `for` loop wasn't sufficient.

In R, `for` loops take an iterator variable and assign it successive values from a sequence or vector. `for` loops are most commonly used for iterating over the elements of an object (list, vector, etc.)

The following three loops all have the similar behavior.

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

for(i in 1:length(x)) {
    ## Print out each element of 'x'
    print(x[i])
}
#> [1] "a"
#> [1] "b"
#> [1] "c"
#> [1] "d"
```

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

```
## Generate a sequence based on length of 'x'
for(i in seq_along(x)) {
    print(x[i])
}
#> [1] "a"
#> [1] "b"
#> [1] "c"
#> [1] "d"
```

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

```
for(letter in x) {
  print(letter)
}
#> [1] "a"
#> [1] "b"
#> [1] "c"
#> [1] "d"
```



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

3.5.3 while Loops

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

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



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

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

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

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

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

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

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

3.5.4 repeat Loops

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

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

```
x0 <- 1
tol <- 1e-8

repeat {
  x1 <- computeEstimate()

  if(abs(x1 - x0) < tol) { ## Close enough?
    break
  } else {
    x0 <- x1
  }
}
```



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

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

3.5.5 next, break

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

`next` is used to skip an iteration of a loop.

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

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

```
for(i in 1:100) {
  print(i)

  if(i > 20) {
```

```

        ## Stop loop after 20 iterations
        break
    }
}

```

3.5.6 Looping

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

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

```

# Two dimensional matrix
M <- matrix(sample(1:16), 4, 4)
M
#>      [,1] [,2] [,3] [,4]
#> [1,]  10   9   4  14
#> [2,]   8  12   6  16
#> [3,]   3   2  15   5
#> [4,]  11   7  13   1
# apply min to rows
apply(M, 1, min)
#> [1] 4 6 2 1
# apply max to columns
apply(M, 2, max)
#> [1] 11 12 15 16

```

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

```

x <- list(a = 1, b = 1:3, c = 10:25)
x
#> $a
#> [1] 1
#>
#> $b
#> [1] 1 2 3
#>
#> $c
#> [1] 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
lapply(x, FUN = length)
#> $a
#> [1] 1
#>
#> $b

```

```

#> [1] 3
#>
#> $c
#> [1] 16
sapply(x, FUN = length)
#> a b c
#> 1 3 16
vapply(x, FUN = length, FUN.VALUE = 0L)
#> a b c
#> 1 3 16

x <- 1:20
y <- factor(rep(letters[1:5], each = 4)) # a vector of the same length as x
tapply(x, y, sum)
#> a b c d e
#> 10 26 42 58 74

# Sums the 1st elements, the 2nd elements, etc.
mapply(sum, 1:5, 1:5, 1:5)
#> [1] 3 6 9 12 15

# find the mean of 10 random normal variables, 5 times
replicate(5, mean(rnorm(10)))
#> [1] -0.2258 0.4434 -0.0499 -0.3555 -0.1810

```

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

3.6 Vectorization & Recycling

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

The simplest example is when adding two vectors together.

```

x <- 1:3
y <- 11:13
z <- x + y
z
#> [1] 12 14 16

```

In most other languages you would have to do something like

```

z <- numeric(length(x))

for(i in seq_along(x)) {
  z[i] <- x[i] + y[i]
}
z
#> [1] 12 14 16

```

We saw a 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
```


3.6.1 Example

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

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

Monte-Carlo pseudo code:

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

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

  4*hits/N
}
N <- 1e6
system.time(pi_naive(N))
#>    user  system elapsed
#>  4.06    0.00    4.13
```

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

```
pi_vect <- function(N) {
  U1 <- runif(N)
  U2 <- runif(N)
  hits <- sum(U1^2 + U2^2 < 1)
  4*hits/N
}
system.time(pi_vect(N))
#>    user  system elapsed
#>  0.20    0.02    0.22
```

That is ~20x speed up.

3.7 Function Basics

To understand computations in R, two slogans are helpful:

- Everything that exists is an object.
 - Everything that happens is a function call.
- John Chambers

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

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.

3.7.1 Your First Function

All R functions have three parts:

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

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

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

- 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.)



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

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

```
myadd2 <- function(x = 3, y = 0){
  cat(paste0("x = ", x, "\n"))
  cat(paste0("y = ", y, "\n"))
  x + y
}
myadd2()              # use the defaults
#> x = 3
#> y = 0
#> [1] 3
myadd2(x = 1)
#> x = 1
#> y = 0
#> [1] 1
myadd2(y = 1)
#> x = 3
#> y = 1
#> [1] 4
myadd2(x = 1, y = 1)
#> x = 1
#> y = 1
#> [1] 2
```

By 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
```

3.7.2 Lazy Evaluation

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

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

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

f(2)      # this works
#> [1] 4
f(2, 1)   # this does too
#> [1] 4
```

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

3.7.3 The ... Argument

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

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

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

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

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

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

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

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

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

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

3.8 Environments & Scoping

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

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

```
x <- 10
x
#> [1] 10
```

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

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

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

```

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

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

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

```

What do the following return?

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

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

```

make.power <- function(n) {
  pow <- function(x) {
    x^n
  }
  pow
}

make.power(4)
#> function(x) {
#>   x^n
#> }
#> <environment: 0x00000000b4b6b10>
cube <- make.power(3)
square <- make.power(2)

x <- 1
n <- 2
pow(x=4)
#> Error in pow(x = 4): could not find function "pow"

```

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 can be achieved using vectorization.

Here is a formula for the Gregory-Leibniz series:

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

Here is the Gregory-Leibniz series in summation notation:

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

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
#>  2.12    0.00    2.14
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
}
```


Chapter 4

Base R Data Structures

4.1 Naming Rules

R has strict rules about what constitutes a valid name. A **syntactic** name must consist of letters¹, 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:

```
_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`.

4.2 Vectors

The most common data structure in R is the vector. R's vectors can be organised by their dimensionality (1d, 2d, or nd) and whether they're homogeneous or heterogeneous. This gives rise to the five data types most often used in data analysis:

	Homogeneous	Heterogeneous
1d	Atomic vector	List
2d	Matrix	Data frame
nd	Array	

Given an object, the best way to understand what data structures it is composed of is to use `str()`. `str()` is short for structure and it gives a compact, human readable description of any R data structure.

¹Surprisingly, 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!

Vectors have three common properties:

- Type, `typeof()`, what it is.
- Length, `length()`, how many elements it contains.
- Attributes, `attributes()`, additional arbitrary metadata.

They differ in the types of their elements: all elements of an atomic vector must be the same type, whereas the elements of a list can have different types.



`is.vector()` does not test if an object is a vector. Instead it returns TRUE only if the object is a vector with no attributes apart from names. Use `is.atomic(x) || is.list(x)` to test if an object is actually a vector.

4.2.1 Atomic Vectors

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

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

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

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

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

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

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

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

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

as.integer(x)
```

```
#> [1] 1 0
as.numeric(x)
#> [1] 1 0
as.character(x)
#> [1] "TRUE" "FALSE"
```

However, coercion is not associative.

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

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

What would you expect this to return?

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

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

You can test for an “atomic” types of data with: `is.logical`, `is.integer`, `is.double`, `is.numeric`², and `is.character`.

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

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

What would you expect these to return?

```
x <- 2

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

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

4.2.2 Lists

Lists are different from atomic vectors because their elements can be of any type, including other lists. Lists can contain complex objects so it’s not possible to pick one visual style that works for every list. You construct lists by using `list()` instead of `c()`:

```
x <- list(1:3, "a", c(TRUE, FALSE, TRUE), c(2.3, 5.9))
str(x)
#> List of 4
#> $ : int [1:3] 1 2 3
#> $ : chr "a"
```

²`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.

```
#> $ : logi [1:3] TRUE FALSE TRUE
#> $ : num [1:2] 2.3 5.9
```

Lists are sometimes called **recursive** vectors, because a list can contain other lists. This makes them fundamentally different from atomic vectors.

```
x <- list(list(list(list(1)))
str(x)
#> List of 1
#> $ :List of 1
#> ..$ :List of 1
#> .. ..$ :List of 1
#> .. .. ..$ : num 1
is.recursive(x)
#> [1] TRUE
```

`c()` will combine several lists into one. If given a combination of atomic vectors and lists, `c()` will coerce the vectors to lists before combining them. Compare the results of `list()` and `c()`:

```
l1 <- list(1, 2)
c1 <- c(3, 4)
x <- list(l1, c1)
y <- c(l1, c1)
str(x)
#> List of 2
#> $ :List of 2
#> ..$ : num 1
#> ..$ : num 2
#> $ : num [1:2] 3 4
str(y)
#> List of 4
#> $ : num 1
#> $ : num 2
#> $ : num 3
#> $ : num 4
```

The `typeof()` a list is `list`. You can test for a list with `is.list()` and coerce to a list with `as.list()`. You can turn a list into an atomic vector with `unlist()`. If the elements of a list have different types, `unlist()` uses the same coercion rules as `c()`.

Lists are used to build up many of the more complicated data structures in R. For example, both data frames (described in data frames) and linear models objects (as produced by `lm()`) are lists

4.2.3 NULL

Closely related to vectors is `NULL`, a singleton object often used to represent a vector of length 0. `NULL` is different than `NA`. For a good explanation of the differences see [this blog post](#).

4.2.4 Attributes

All objects can have arbitrary additional attributes, used to store metadata about the object. Attributes can be thought of as a named list³ (with unique names). Attributes can be accessed individually with `attr()`

³The reality is a little more complicated: attributes are actually stored in something called pairlists, which can you learn more about in Advanced R

or all at once (as a list) with `attributes()`.

```
a <- 1:3
attr(a, "createdBy") <- "Brian Davis"
attr(a, "version") <- 1.0
attr(a, "z") <- list(list())
a
#> [1] 1 2 3
#> attr("createdBy")
#> [1] "Brian Davis"
#> attr("version")
#> [1] 1
#> attr("z")
#> attr("z")[[1]]
#> list()
attributes(a)
#> $createdBy
#> [1] "Brian Davis"
#>
#> $version
#> [1] 1
#>
#> $z
#> $z[[1]]
#> list()
str(attributes(a))
#> List of 3
#> $ createdBy: chr "Brian Davis"
#> $ version : num 1
#> $ z       :List of 1
#> ..$ : list()
```

The `structure()` function returns a new object with modified attributes. Care must be taken with attributes since, by default, most attributes are lost when modifying a vector.

```
attributes(a[1])
#> NULL
attributes(sum(a))
#> NULL
```

The only attributes not lost are the three most important:

- Names, a character vector giving each element a name.
- Dimensions, used to turn vectors into matrices and arrays.
- Class, used to implement the S3 object system.

Each of these attributes has a specific accessor function to get and set values. When working with these attributes, use `names(x)`, `dim(x)`, and `class(x)`, not `attr(x, "names")`, `attr(x, "dim")`, and `attr(x, "class")`.

4.2.4.1 Names

You can name a vector in a couple⁴ ways:

⁴There are a couple less common ways. See Advanced R

- When creating it: `x <- c(a = 1, b = 2, c = 3)`.
- By modifying an existing vector in place: `x <- 1:3; names(x) <- c("a", "b", "c")`.

Named vectors are a great way to make an easy, human readable look up table. We will see this use case extensively when we get to data visualizations.

4.2.5 Factors

One important use of attributes is to define factors. A factor is a vector that can contains only predefined values, and is used to store categorical data. Factors are built on top of **integer vectors** using two attributes: the `class`, “factor”, which makes them behave differently from regular integer vectors, and the `levels`, which defines the set of allowed values. Factors can also have labels which effect how the factors are displayed. By default the labels are the same as the levels.

The order of the levels of a factor can be set using the `levels` argument to `factor()`. This can be important in linear modelling because the first level is used as the baseline level. This feature can also be used to customize order in plots that include factors, since by default factors are plotted in the order of their levels. Labels are also useful in plotting where you want the displayed text to be different than the underlying representation.

Factors are useful when you know the possible values a variable may take, even if you don’t see all values in a given data set. Using a factor instead of a character vector makes it obvious when some groups contain no observations:

```
gender_char <- c("m", "m", "m")
gender_factor <- factor(gender_char, levels = c("m", "f"))

gender_char
#> [1] "m" "m" "m"
table(gender_char)
#> gender_char
#> m
#> 3
gender_factor
#> [1] m m m
#> Levels: m f
table(gender_factor)
#> gender_factor
#> m f
#> 3 0
# See the underlying representation of a factor
unclass(gender_factor)
#> [1] 1 1 1
#> attr("levels")
#> [1] "m" "f"

gender_factor2 <- factor(gender_char, levels = c("m", "f"), labels = c("Male", "Female"))
gender_factor2
#> [1] Male Male Male
#> Levels: Male Female
table(gender_factor2)
#> gender_factor2
#> Male Female
#> 3 0
```

```
# See the underlying representation of a factor
unclass(gender_factor2)
#> [1] 1 1 1
#> attr("levels")
#> [1] "Male" "Female"
```

While factors look like (and often behave like) character vectors, they are actually **integers**. Be careful when treating them like strings. Some string methods (like `gsub()` and `grepl()`) will coerce factors to strings, while others (like `nchar()`) will throw an error, and still others (like `c()`) will use the underlying integer values. For this reason, it is best to explicitly convert factors to character vectors if you need string-like behavior.

Unfortunately, many base R functions (like `read.csv()` and `data.frame()`) automatically convert character vectors to factors. This is sub-optimal, because there's no way for those functions to know the set of all possible levels or their optimal order. Instead, use the argument `stringsAsFactors = FALSE` to suppress this behavior, and then manually convert character vectors to factors using your knowledge of the data only when you need the behavior of factors.

Factors tend to be most useful in data visualization and table creations where you want to report all categories but some categories may not be present in your data, or when you want to order the categories in something other than the default ordering. We will revisit factors and their usefulness later when we study the tidyverse and in particular the forcats package.

4.2.6 Matrices and arrays

Adding a `dim` attribute to an atomic vector allows it to behave like a multi-dimensional **array**. A special case of the array is the **matrix**, which has two dimensions. Matrices are used commonly as part of the mathematical machinery of statistics. Arrays are much rarer, but worth being aware of.

Matrices and arrays are created with `matrix()` and `array()`, or by using the assignment form of `dim()`:

```
# Two scalar arguments to specify rows and columns
a <- matrix(1:12, ncol = 3, nrow = 4)
a
#>      [,1] [,2] [,3]
#> [1,]    1    5    9
#> [2,]    2    6   10
#> [3,]    3    7   11
#> [4,]    4    8   12
# One vector argument to describe all dimensions
b <- array(1:12, c(2, 3, 2))
b
#> , , 1
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6
#> , , 2
#>      [,1] [,2] [,3]
#> [1,]    7    9   11
#> [2,]    8   10   12

# You can also modify an object in place by setting dim()
```

```

vec <- 1:12
vec
#> [1] 1 2 3 4 5 6 7 8 9 10 11 12
class(vec)
#> [1] "integer"

dim(vec) <- c(3, 4)
vec
#>      [,1] [,2] [,3] [,4]
#> [1,]    1    4    7   10
#> [2,]    2    5    8   11
#> [3,]    3    6    9   12
class(vec)
#> [1] "matrix"

dim(vec) <- c(3, 2, 2)
vec
#> , , 1
#>
#>      [,1] [,2]
#> [1,]    1    4
#> [2,]    2    5
#> [3,]    3    6
#>
#> , , 2
#>
#>      [,1] [,2]
#> [1,]    7   10
#> [2,]    8   11
#> [3,]    9   12
class(vec)
#> [1] "array"

```

`length()` and `names()` have high-dimensional generalizations:

- `length()` generalizes to `nrow()` and `ncol()` for matrices, and `dim()` for arrays.
- `names()` generalizes to `rownames()` and `colnames()` for matrices, and `dimnames()`, a list of character vectors, for arrays.

`c()` generalizes to `cbind()` and `rbind()` for matrices, and to `abind::abind()` for arrays. You can transpose a matrix with `t()`; the generalized equivalent for arrays is `aperm()`.

You can test if an object is a matrix or array using `is.matrix()` and `is.array()`, or by looking at the length of the `dim()`. `as.matrix()` and `as.array()` make it easy to turn an existing vector into a matrix or array.

Vectors are not the only 1-dimensional data structure. You can have matrices with a single row or single column, or arrays with a single dimension. They may print similarly, but will behave differently. The differences aren't too important, but it's useful to know they exist in case you get strange output from a function (`tapply()` is a frequent offender). As always, use `str()` to reveal the differences.

Matrices and arrays are most useful for mathematical calculations (particularly when fitting models); lists and data frames are a better fit for most other programming tasks in R.

4.2.7 Data Frames

A data frame is the most common way of storing data in R, and if used systematically makes data analysis easier. Under the hood, a data frame is a list of equal-length vectors. This makes it a 2-dimensional structure, so it shares properties of both the matrix and the list. This means that a data frame has `names()`, `colnames()`, and `rownames()`, although `names()` and `colnames()` are the same thing. The `length()` of a data frame is the length of the underlying list and so is the same as `ncol()`; `nrow()` gives the number of rows. You can subset a data frame like a 1d structure (where it behaves like a list), or a 2d structure (where it behaves like a matrix), we will discuss this further when we discuss subsetting.

4.2.7.1 Creation

You create a data frame using `data.frame()`, which takes named vectors as input:

```
df <- data.frame(x = 1:3, y = c("a", "b", "c"))
str(df)
#> 'data.frame':    3 obs. of  2 variables:
#> $ x: int  1 2 3
#> $ y: Factor w/ 3 levels "a","b","c": 1 2 3
```



Beware `data.frame()`'s default behavior which turns strings into factors. Use `stringsAsFactors = FALSE` to suppress this behavior.

```
df <- data.frame(
  x = 1:3,
  y = c("a", "b", "c"),
  stringsAsFactors = FALSE)
str(df)
#> 'data.frame':    3 obs. of  2 variables:
#> $ x: int  1 2 3
#> $ y: chr  "a" "b" "c"
```

Because a `data.frame` is an S3 class, its type reflects the underlying vector used to build it: the list.

```
typeof(df)
#> [1] "list"
```

4.2.7.2 Testing and coercion

Because a `data.frame` is an S3 class, its type reflects the underlying vector used to build it: the list. To check if an object is a data frame, use `is.data.frame()`:

```
is.data.frame(df)
#> [1] TRUE
```

You can coerce an object to a data frame with `as.data.frame()`:

- A vector will create a one-column data frame.
- A list will create one column for each element; it's an error if they're not all the same length.
- A matrix will create a data frame with the same number of columns and rows as the matrix.

The automatic coercion that causes the most problems is if you select a single column of a `data.frame`. R will coerce the column to an atomic vector, which generally is not what you want⁵.

```
(x1 <- df[, "y"])
#> [1] "a" "b" "c"
str(x1)
#> chr [1:3] "a" "b" "c"

(x2 <- df[, "y", drop = FALSE])
#>   y
#> 1 a
#> 2 b
#> 3 c
str(x2)
#> 'data.frame':   3 obs. of  1 variable:
#> $ y: chr "a" "b" "c"
```

4.2.7.3 Combining data frames

You can combine data frames using `cbind()` and `rbind()`:

```
cbind(df, data.frame(z = 3:1))
#>   x y z
#> 1 1 a 3
#> 2 2 b 2
#> 3 3 c 1
rbind(df, data.frame(x = 10, y = "z"))
#>   x y
#> 1  1 a
#> 2  2 b
#> 3  3 c
#> 4 10 z
```

When combining column-wise, the number of rows must match, but row names are ignored. When combining row-wise, both the number and names of columns must match.

It's a common mistake to try and create a data frame by `cbind()`ing vectors together. This is unlikely to do what you want because `cbind()` will create a matrix unless one of the arguments is already a data frame. Instead use `data.frame()` directly:

```
# This is always a mistake
bad <- data.frame(cbind(a = 1:2, b = c("a", "b")))
str(bad)
#> 'data.frame':   2 obs. of  2 variables:
#> $ a: Factor w/ 2 levels "1","2": 1 2
#> $ b: Factor w/ 2 levels "a","b": 1 2

good <- data.frame(a = 1:2, b = c("a", "b"))
str(good)
#> 'data.frame':   2 obs. of  2 variables:
#> $ a: int  1 2
#> $ b: Factor w/ 2 levels "a","b": 1 2
```

⁵We'll revisit this when we get into R for Data Science and discuss tibbles

4.2.7.4 List and matrix columns

Since a data frame is a list of vectors, it is possible for a data frame to have a column that is a list. This is a powerful technique because a list can contain any other R object. This means that you can have a column of data frames, or model objects, or even functions! We will see this again when we discuss tidy data.

```
df <- data.frame(x = 1:3)
df$y <- list(1:2, 1:3, 1:4)
df
#>   x      y
#> 1 1      1, 2
#> 2 2      1, 2, 3
#> 3 3 1, 2, 3, 4
```

However, when a list is given to `data.frame()`, it tries to put each item of the list into its own column, so this fails:

```
data.frame(x = 1:3, y = list(1:2, 1:3, 1:4))
#> Error in (function (... , row.names = NULL, check.rows = FALSE, check.names = TRUE, : arguments imply
```

A workaround is to use `I()`, which causes `data.frame()` to treat the list as one unit:

```
df1 <- data.frame(x = 1:3, y = I(list(1:2, 1:3, 1:4)))
str(df1)
#> 'data.frame':   3 obs. of  2 variables:
#> $ x: int  1 2 3
#> $ y:List of 3
#> ..$ : int  1 2
#> ..$ : int  1 2 3
#> ..$ : int  1 2 3 4
#> ..- attr(*, "class")= chr "AsIs"
```

`I()` adds the `AsIs` class to its input, but this can usually be safely ignored.

Similarly, it's also possible to have a column of a data frame that's a matrix or array, as long as the number of rows matches the data frame:

```
dfm <- data.frame(x = 1:3 * 10, y = I(matrix(1:9, nrow = 3)))
str(dfm)
#> 'data.frame':   3 obs. of  2 variables:
#> $ x: num  10 20 30
#> $ y: 'AsIs' int [1:3, 1:3] 1 2 3 4 5 6 7 8 9
```

Use list and array columns with caution. Many functions that work with data frames assume that all columns are atomic vectors, and the printed display can be confusing.

```
df1[2, ]
#>   x      y
#> 2 2 1, 2, 3
dfm[2, ]
#>   x y.1 y.2 y.3
#> 2 20  2  5  8
```


Chapter 5

Subsetting

R's subsetting operators are powerful and fast. Mastery of subsetting allows you to succinctly express complex operations in a way that few other languages can match. Subsetting can be hard to learn because you need to master a number of interrelated concepts:

- The three subsetting operators
- `[]` select multiple elements
- `[[`, and `$` select a single element
- The six types of subsetting.
 - **Positive integers** return elements at the specified positions
 - **Negative integers** omit elements at the specified positions
 - **Logical vectors** select elements where the corresponding logical value is `TRUE`
 - **Nothing** returns the original object.
 - **Zero** returns a zero-length object (This is not something you usually do on purpose)
 - **Character vectors** to return elements with matching names.
- Important differences in behavior for different objects (e.g., vectors, lists, factors, matrices, and data frames).
- The use of subsetting in conjunction with assignment.

It's easiest to learn how subsetting works for atomic vectors, and then how it generalizes to higher dimensions and other more complicated objects.

5.1 Selecting multiple elements

There is one accessor for selecting multiple elements `[]`.

5.1.1 Atomic vectors

Let's explore the different types of subsetting with a simple vector, `x`.

```
x <- c(2.1, 4.2, 3.3, 5.4)
```

Note that the number after the decimal point gives the original position in the vector.

There are five things that you can use to subset a vector.

- **Positive integers** return elements at the specified positions

```
x[c(3, 1)]
#> [1] 3.3 2.1

# order returns an index
x[order(x)]
#> [1] 2.1 3.3 4.2 5.4

# Duplicated indices yield duplicated values
x[c(1, 1)]
#> [1] 2.1 2.1

# Real numbers are silently truncated (not rounded) to integers
x[c(2.1, 2.9)]
#> [1] 4.2 4.2
```

- **Negative integers** omit elements at the specified positions

```
x[-c(3, 1)]
#> [1] 4.2 5.4
```

You can't mix positive and negative integers in a single subset.

```
x[c(-1, 2)]
#> Error in x[c(-1, 2)]: only 0's may be mixed with negative subscripts
```

- **Logical vectors** select elements where the corresponding logical value is `TRUE`. This is probably the most useful type of subsetting because you write the expression that creates the logical vector:

```
x[c(TRUE, TRUE, FALSE, FALSE)]
#> [1] 2.1 4.2

x[x > 3]
#> [1] 4.2 3.3 5.4
```

If the logical vector is shorter than the vector being subsetted, it will be *recycled* to be the same length.

```
x[c(TRUE, FALSE)]
#> [1] 2.1 3.3
# Equivalent to
x[c(TRUE, FALSE, TRUE, FALSE)]
#> [1] 2.1 3.3
```

A missing value in the index always yields a missing value in the output.

```
x[c(TRUE, TRUE, NA, FALSE)]
#> [1] 2.1 4.2 NA
```

- **Nothing** returns the original vector. This is not useful for vectors but is very useful for matrices, data frames, and arrays. It can also be useful in conjunction with assignment.

```
x[]
#> [1] 2.1 4.2 3.3 5.4
```

- **Zero** returns a zero-length vector. This is not something you usually do on purpose, but it can be helpful for generating test data and testing corner cases of functions.

```
x[0]
#> numeric(0)
```

If the vector is named, you can also use:

- **Character vectors** to return elements with matching names.

```
(y <- setNames(x, letters[1:4]))
#>   a   b   c   d
#> 2.1 4.2 3.3 5.4
# subsetting by name
y[c("d", "c", "a")]
#>   d   c   a
#> 5.4 3.3 2.1

# Like integer indices, you can repeat indices
y[c("a", "a", "a")]
#>   a   a   a
#> 2.1 2.1 2.1

# When subsetting with [ names are always matched exactly
z <- c(abc = 1, def = 2)
z[c("a", "d")]
#> <NA> <NA>
#>   NA   NA
```

5.1.2 Matrices and Arrays

You can subset higher-dimensional structures in three ways:

- With multiple vectors.
- With a single vector.
- With a matrix.

The most common way of subsetting matrices (2d) and arrays (>2d) is a simple generalization of 1d subsetting: you supply a 1d index for each dimension, separated by a comma. Blank subsetting is now useful because it lets you keep all rows or all columns.

```
a <- matrix(1:9, nrow = 3)
colnames(a) <- c("A", "B", "C")
a[1:2, ]
#>      A B C
#> [1,] 1 4 7
#> [2,] 2 5 8
a[c(TRUE, FALSE, TRUE), c("B", "A")]
#>      B A
#> [1,] 4 1
#> [2,] 6 3
a[2:3, -2]
#>      A C
#> [1,] 2 8
#> [2,] 3 9
```

By default, `[` will simplify the results to the lowest possible dimensionality. See below how to avoid this behavior.

Because matrices and arrays are implemented as vectors with special attributes, you can subset them with a single vector. In that case, they will behave like a vector. Arrays in R are stored in column-major order:

```
(vals <- outer(1:5, 1:5, FUN = "paste", sep = ","))
#>      [,1] [,2] [,3] [,4] [,5]
#> [1,] "1,1" "1,2" "1,3" "1,4" "1,5"
#> [2,] "2,1" "2,2" "2,3" "2,4" "2,5"
#> [3,] "3,1" "3,2" "3,3" "3,4" "3,5"
#> [4,] "4,1" "4,2" "4,3" "4,4" "4,5"
#> [5,] "5,1" "5,2" "5,3" "5,4" "5,5"
vals[c(4, 15)]
#> [1] "4,1" "5,3"
```

This behavior allows you to replace all missing values in one line.

```
# make a few values missing
vals[sample(1:25, 5)] <- NA_character_
vals
#>      [,1] [,2] [,3] [,4] [,5]
#> [1,] "1,1" "1,2" NA   "1,4" "1,5"
#> [2,] NA   "2,2" "2,3" "2,4" "2,5"
#> [3,] "3,1" "3,2" "3,3" "3,4" "3,5"
#> [4,] NA   NA   "4,3" "4,4" NA
#> [5,] "5,1" "5,2" "5,3" "5,4" "5,5"
# replace missing values with "missing"
vals[is.na(vals)] <- "missing"
vals
#>      [,1]      [,2]      [,3]      [,4] [,5]
#> [1,] "1,1"      "1,2"      "missing" "1,4" "1,5"
#> [2,] "missing" "2,2"      "2,3"      "2,4" "2,5"
#> [3,] "3,1"      "3,2"      "3,3"      "3,4" "3,5"
#> [4,] "missing" "missing" "4,3"      "4,4" "missing"
#> [5,] "5,1"      "5,2"      "5,3"      "5,4" "5,5"
```

You can also subset higher-dimensional data structures with an integer matrix (or, if named, a character matrix). Each row in the matrix specifies the location of one value, where each column corresponds to a dimension in the array being subsetted. This means that you use a 2 column matrix to subset a matrix, a 3 column matrix to subset a 3d array, and so on. The result is a vector of values:

```
vals <- outer(1:5, 1:5, FUN = "paste", sep = ",")
select <- matrix(ncol = 2, byrow = TRUE, c(
  1, 1,
  3, 1,
  2, 4
))
vals[select]
#> [1] "1,1" "3,1" "2,4"
```

5.1.3 Lists

Subsetting a list works in the same way as subsetting an atomic vector. Using `[` will always return a list; `[[` and `$`, as described below, let you pull out the components of the list.

```
x <- list(1:3, "a", c(TRUE, FALSE, TRUE), c(2.3, 5.9))
x
#> [[1]]
#> [1] 1 2 3
```



```

#>
#> [[2]]
#> [1] "a"
#>
#> [[3]]
#> [1] TRUE FALSE TRUE
#>
#> [[4]]
#> [1] 2.3 5.9
x[c(1,4)]
#> [[1]]
#> [1] 1 2 3
#>
#> [[2]]
#> [1] 2.3 5.9

```

5.1.4 Data Frames

Data frames possess the characteristics of both lists and matrices: if you subset with a single vector, they behave like lists; if you subset with two vectors, they behave like matrices.

```

df <- data.frame(x = 1:3, y = 3:1, z = letters[1:3])
df
#>   x y z
#> 1 1 3 a
#> 2 2 2 b
#> 3 3 1 c

# There are two ways to select columns from a data frame
# Like a list:
df[c("x", "z")]
#>   x z
#> 1 1 a
#> 2 2 b
#> 3 3 c
# Like a matrix
df[, c("x", "z")]
#>   x z
#> 1 1 a
#> 2 2 b
#> 3 3 c

# There's an important difference if you select a single
# column: matrix subsetting simplifies by default, list
# subsetting does not.
str(df["x"])
#> 'data.frame':   3 obs. of  1 variable:
#> $ x: int  1 2 3
str(df[, "x"])
#> int [1:3] 1 2 3

# for row subset like a matrix
df[df$x == 2, ]

```

```
#>   x y z
#> 2 2 2 b
df[c(1, 3), ]
#>   x y z
#> 1 1 3 a
#> 3 3 1 c
```

5.1.5 Preserving dimensionality

By default, any subsetting 2d data structures with a single number, single name, or a logical vector containing a single `TRUE` will simplify the returned output as described below. To preserve the original dimensionality, you must use `drop = FALSE`

- For matrices and arrays, any dimensions with length 1 will be dropped:

```
(a <- matrix(1:4, nrow = 2))
#>      [,1] [,2]
#> [1,]    1    3
#> [2,]    2    4
str(a[1, ])
#> int [1:2] 1 3

str(a[1, , drop = FALSE])
#> int [1, 1:2] 1 3
```

- Data frames with a single column will return just that column:

```
(df <- data.frame(a = 1:2, b = 1:2))
#>   a b
#> 1 1 1
#> 2 2 2
str(df[, "a"])
#> int [1:2] 1 2

str(df[, "a", drop = FALSE])
#> 'data.frame': 2 obs. of 1 variable:
#> $ a: int 1 2
```

The default `drop = TRUE` behavior is a common source of bugs in functions: you check your code with a data frame or matrix with multiple columns, and it works. Six months later you (or someone else) uses it with a single column data frame and it fails with a mystifying error. When writing functions, get in the habit of always using `drop = FALSE` when subsetting a 2d object.

Factor subsetting also has a `drop` argument, but the meaning is rather different. It controls whether or not levels are preserved (not the dimensionality), and it defaults to `FALSE` (levels are preserved, not simplified by default). If you find you are using `drop = TRUE` a lot it's often a sign that you should be using a character vector instead of a factor.

```
z <- factor(c("a", "b"))
z[1]
#> [1] a
#> Levels: a b
z[1, drop = TRUE]
#> [1] a
#> Levels: a
```

5.2 Selecting a single elements

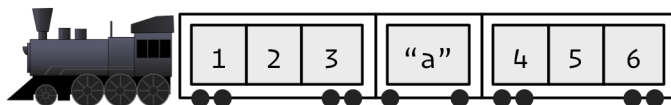
There are two other subsetting operators: `[[` and `$`. `[[` is used for extracting single values, and `$` is a useful shorthand for `[[` combined with character subsetting. `[[` is most important working with lists because subsetting a list with `[` always returns a smaller list. To help make this easier to understand we can use a metaphor:

“If list `x` is a train carrying objects, then `x[[5]]` is the object in car 5; `x[4:6]` is a train of cars 4-6.”

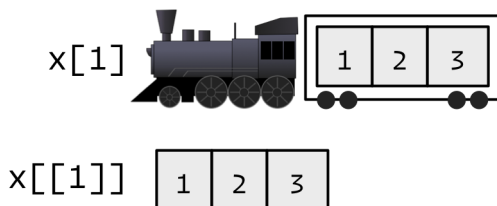
— @RLangTip, <https://twitter.com/RLangTip/status/268375867468681216>

Let’s make a simple list and draw it as a train:

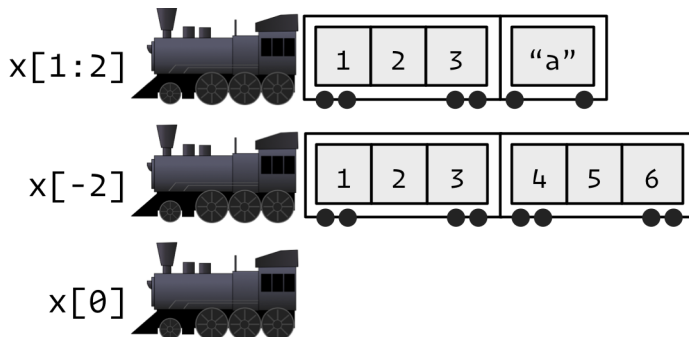
```
x <- list(1:3, "a", 4:6)
```



When extracting a single element, you have two options: you can create a smaller train, or you can extract the contents of a carriage. This is the difference between `[` and `[[`:



When extracting multiple elements (or zero!), you have to make a smaller train:



Because it can return only a single value, you must use `[[` with either a single positive integer or a string. Because data frames are lists of columns, you can use `[[` to extract a column from data frames: `mtcars[[1]]`, `mtcars[["cyl"]]`.

If you use a vector with `[[`, it will subset recursively:

```
(b <- list(a = list(b = list(c = list(d = 1))))
#> $a
#> $a$b
#> $a$b$c
#> $a$b$c$d
#> [1] 1
str(b)
```

```
#> List of 1
#> $ a:List of 1
#> ..$ b:List of 1
#> .. ..$ c:List of 1
#> .. .. ..$ d: num 1
b[[c("a", "b", "c", "d")]]
#> [1] 1

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

[[is crucial for working with lists, but I recommend using it whenever you want your code to clearly express that it's working with a single value. That frequently arises in for loops, i.e. instead of writing:

```
for (i in 2:length(x)) {
  out[i] <- fun(x[i], out[i - 1])
}
```

It's better to write:

```
for (i in 2:length(x)) {
  out[[i]] <- fun(x[[i]], out[[i - 1]])
}
```

5.2.1 \$

\$ is a shorthand operator: `x$y` is roughly equivalent to `x[["y"]]`. It's often used to access variables in a data frame, as in `mtcars$cyl` or `diamonds$carat`. One common mistake with \$ is to try and use it when you have the name of a column stored in a variable:

```
var <- "cyl"
# Doesn't work - mtcars$var translated to mtcars[["var"]]
mtcars$var
#> NULL

# Instead use [[
mtcars[[var]]
#> [1] 6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 4 4 4 4 8 8 8 8 4 4 4 8 6 8 4
```

There's one important difference between \$ and [[. \$ does partial matching:

```
x <- list(abc = 1, def = 2, ghi = c(4:6))
x$d
#> [1] 2
x[["d"]]
#> NULL
x[["def"]]
#> [1] 2
```

It is usually a good idea to **NOT** use partial matching. It tends to lead to hard to track down bugs and makes your code much less readable. With auto complete in RStudio it tends not to save any time or keystrokes.

5.2.2 Missing/out of bounds indices

TL;DR version use `purrr::pluck()`, which we will get to in *R for Data Science*

It's useful to understand what happens with `[]` and `[[` when you use an “invalid” index. The following tables summarize what happens when you subset a logical vector, list, and `NULL` with an out-of-bounds value (OOB), a missing value (i.e. `NA_integer_`), and a zero-length object (like `NULL` or `logical()`) with `[]` and `[[`. Each cell shows the result of subsetting the data structure named in the row by the type of index described in the column. I've only shown the results for logical vectors, but other atomic vectors behave similarly, returning elements of the same type.

	row[col]	Zero-length	OOB	Missing
Logical	<code>logical(0)</code>	<code>NA</code>		<code>NA</code>
List	<code>list()</code>	<code>list(NULL)</code>		<code>list(NULL)</code>
<code>NULL</code>	<code>NULL</code>	<code>NULL</code>		<code>NULL</code>

```
x <- c(TRUE, FALSE, TRUE)
x[NULL]
#> logical(0)
x[10]
#> [1] NA
x[NA_real_]
#> [1] NA

y <- list(abc = 1, def = 2, ghi = c(4:6))
y[NULL]
#> named list()
y[10]
#> $<NA>
#> NULL
y[NA_real_]
#> $<NA>
#> NULL

NULL[NULL]
#> NULL
NULL[1]
#> NULL
NULL[NA_real_]
#> NULL
```

With `[]`, it doesn't matter whether the OOB index is a position or a name, but it does for `[[`:

	row[[col]]	Zero-length	OOB (int)	OOB (chr)	Missing
Atomic		Error	Error	Error	Error
List		Error	Error	<code>NULL</code>	<code>NULL</code>
<code>NULL</code>		<code>NULL</code>	<code>NULL</code>	<code>NULL</code>	<code>NULL</code>

```
x
#> [1] TRUE FALSE TRUE
x[[NULL]]
#> Error in x[[NULL]]: attempt to select less than one element in get1index
```

```

x[[10]]
#> Error in x[[10]]: subscript out of bounds
x[["x"]]
#> Error in x[["x"]]: subscript out of bounds
x[[NA_real_]]
#> Error in x[[NA_real_]]: subscript out of bounds

y
#> $abc
#> [1] 1
#>
#> $def
#> [1] 2
#>
#> $ghi
#> [1] 4 5 6
y[[NULL]]
#> Error in y[[NULL]]: attempt to select less than one element in get1index
y[[10]]
#> Error in y[[10]]: subscript out of bounds
y[["x"]]
#> NULL
y[[NA_real_]]
#> NULL

NULL[[NULL]]
#> NULL
NULL[[1]]
#> NULL
NULL[["x"]]
#> NULL
NULL[[NA_real_]]
#> NULL

```

If the input vector is named, then the names of OOB, missing, or NULL components will be "<NA>".

5.3 Subsetting and assignment

All subsetting operators can be combined with assignment to modify selected values of the input vector.

```

x <- 1:5
x[c(1, 2)] <- 2:3
x
#> [1] 2 3 3 4 5

# The length of the LHS needs to match the RHS
x[-1] <- 4:1
x
#> [1] 2 4 3 2 1

# Duplicated indices go unchecked and may be problematic
x[c(1, 1)] <- 2:3
x

```

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

# You can't combine integer indices with NA
x[c(1, NA)] <- c(1, 2)
#> Error in x[c(1, NA)] <- c(1, 2): NAs are not allowed in subscripted assignments
# But you can combine logical indices with NA
# (where they are treated as FALSE).
x[c(T, F, NA)] <- 1
x
#> [1] 1 4 3 1 1

# This is mostly useful when conditionally modifying vectors
df <- data.frame(a = c(1, 10, NA))
df$a[df$a < 5] <- 0
df$a
#> [1] 0 10 NA
```

Subsetting with nothing can be useful in conjunction with assignment because it will preserve the original object class and structure. Compare the following two expressions. In the first, `mtcars` will remain as a data frame. In the second, `mtcars` will become a list.

```
mtcars[] <- lapply(mtcars, as.integer)
mtcars <- lapply(mtcars, as.integer)
```

With lists, you can use `[[+ assignment + NULL` to remove components from a list. To add a literal `NULL` to a list, use `[` and `list(NULL)`:

```
x <- list(a = 1, b = 2)
x[["b"]] <- NULL
str(x)
#> List of 1
#> $ a: num 1

y <- list(a = 1)
y["b"] <- list(NULL)
str(y)
#> List of 2
#> $ a: num 1
#> $ b: NULL
```

5.4 Applications

The basic principles described above give rise to a wide variety of useful applications. Some of the most important are described below. Many of these basic techniques are wrapped up into more concise functions (e.g., `subset()`, `merge()`, `dplyr::arrange()`), but it is useful to understand how they are implemented with basic subsetting. This will allow you to adapt to new situations that are not dealt with by existing functions.

5.4.1 Lookup tables (character subsetting)

Character matching provides a powerful way to make look-up tables. Say you want to convert abbreviations:

```
x <- c("m", "f", "u", "f", "f", "m", "m")
lookup <- c(m = "Male", f = "Female", u = NA)
lookup[x]
#>      m      f      u      f      f      m      m
#>  "Male" "Female"  NA "Female" "Female"  "Male"  "Male"

unname(lookup[x])
#> [1] "Male" "Female" NA      "Female" "Female" "Male"  "Male"
```

If you don't want names in the result, use `unname()` to remove them.

5.4.2 Ordering (integer subsetting)

`order()` takes a vector as input and returns an integer vector describing how the subsetted vector should be ordered:

```
x <- c("b", "c", "a")
order(x)
#> [1] 3 1 2
x[order(x)]
#> [1] "a" "b" "c"
```

To break ties, you can supply additional variables to `order()`, and you can change from ascending to descending order using `decreasing = TRUE`. By default, any missing values will be put at the end of the vector; however, you can remove them with `na.last = NA` or put at the front with `na.last = FALSE`.

For two or more dimensions, `order()` and integer subsetting makes it easy to order either the rows or columns of an object:

```
(df <- data.frame(x = rep(1:3, each = 2), y = 6:1, z = letters[1:6]))
#>   x y z
#> 1 1 6 a
#> 2 1 5 b
#> 3 2 4 c
#> 4 2 3 d
#> 5 3 2 e
#> 6 3 1 f
# Randomly reorder df
df2 <- df[sample(nrow(df)), 3:1]
df2
#>   z y x
#> 3 c 4 2
#> 2 b 5 1
#> 5 e 2 3
#> 6 f 1 3
#> 4 d 3 2
#> 1 a 6 1

df2[order(df2$x), ]
#>   z y x
#> 2 b 5 1
#> 1 a 6 1
#> 3 c 4 2
#> 4 d 3 2
#> 5 e 2 3
```



```
#> 6 f 1 3
df2[, order(names(df2))]
#>   x y z
#> 3 2 4 c
#> 2 1 5 b
#> 5 3 2 e
#> 6 3 1 f
#> 4 2 3 d
#> 1 1 6 a
```

You can sort vectors directly with `sort()`, or use `dplyr::arrange()` or similar to sort a data frame.

5.4.3 Selecting rows based on a condition (logical subsetting)

Because it allows you to easily combine conditions from multiple columns, logical subsetting is probably the most commonly used technique for extracting rows out of a data frame.

```
mtcars[mtcars$gear == 5, ]
#>      mpg cyl  disp  hp drat   wt  qsec vs am gear carb
#> 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

mtcars[mtcars$gear == 5 & mtcars$cyl == 4, ]
#>      mpg cyl  disp  hp drat   wt  qsec vs am gear carb
#> 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
```

Remember to use the vector boolean operators `&` and `|`, not the short-circuiting scalar operators `&&` and `||` which are more useful inside if statements. Don't forget De Morgan's laws, which can be useful to simplify negations:

- `!(X & Y)` is the same as `!X | !Y`
- `!(X | Y)` is the same as `!X & !Y`

For example, `!(X & !(Y | Z))` simplifies to `!X | !(Y|Z)`, and then to `!X | Y | Z`.

`subset()` is a specialized shorthand function for subsetting data frames, and saves some typing because you don't need to repeat the name of the data frame..

```
subset(mtcars, gear == 5)
#>      mpg cyl  disp  hp drat   wt  qsec vs am gear carb
#> 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

subset(mtcars, gear == 5 & cyl == 4)
#>      mpg cyl  disp  hp drat   wt  qsec vs am gear carb
#> 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
```

5.4.4 Removing columns from data frames (character subsetting)

There are two ways to remove columns from a data frame. You can set individual columns to NULL:

```
df <- data.frame(x = 1:3, y = 3:1, z = letters[1:3])
df$z <- NULL
```

Or you can subset to return only the columns you want:

```
df <- data.frame(x = 1:3, y = 3:1, z = letters[1:3])
df[c("x", "y")]
#>   x y
#> 1 1 3
#> 2 2 2
#> 3 3 1
```

If you know the columns you don't want, use set operations to work out which columns to keep:

```
df[setdiff(names(df), "z")]
#>   x y
#> 1 1 3
#> 2 2 2
#> 3 3 1
```

5.4.5 Random samples/bootstrap (integer subsetting)

You can use integer indices to perform random sampling or bootstrapping of a vector or data frame. `sample()` generates a vector of indices, then subsetting accesses the values:

```
(df <- data.frame(x = rep(1:3, each = 2), y = 6:1, z = letters[1:6]))
#>   x y z
#> 1 1 6 a
#> 2 1 5 b
#> 3 2 4 c
#> 4 2 3 d
#> 5 3 2 e
#> 6 3 1 f

# Randomly reorder
df[sample(nrow(df)), ]
#>   x y z
#> 5 3 2 e
#> 4 2 3 d
#> 1 1 6 a
#> 3 2 4 c
#> 6 3 1 f
#> 2 1 5 b

# Select 3 random rows
df[sample(nrow(df), 3), ]
#>   x y z
#> 6 3 1 f
#> 5 3 2 e
#> 3 2 4 c

# Select 6 bootstrap replicates
```

```
df[sample(nrow(df), 6, rep = TRUE), ]
#>      x y z
#> 2    1 5 b
#> 6    3 1 f
#> 3    2 4 c
#> 3.1  2 4 c
#> 6.1  3 1 f
#> 1    1 6 a
```

The arguments of `sample()` control the number of samples to extract, and whether sampling is performed with or without replacement.

5.4.6 Boolean algebra vs. sets (logical & integer subsetting)

It's useful to be aware of the natural equivalence between set operations (integer subsetting) and boolean algebra (logical subsetting). Using set operations is more effective when:

- You want to find the first (or last) `TRUE`.
- You have very few `TRUE`s and very many `FALSE`s; a set representation may be faster and require less storage.

`which()` allows you to convert a boolean representation to an integer representation.

Let's create two logical vectors and their integer equivalents and then explore the relationship between boolean and set operations.

```
(x1 <- 1:10 %% 2 == 0)
#> [1] FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE
(x2 <- which(x1))
#> [1] 2 4 6 8 10
(y1 <- 1:10 %% 5 == 0)
#> [1] FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE TRUE
(y2 <- which(y1))
#> [1] 5 10

# X & Y <-> intersect(x, y)
x1 & y1
#> [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE
intersect(x2, y2)
#> [1] 10

# X | Y <-> union(x, y)
x1 | y1
#> [1] FALSE TRUE FALSE TRUE TRUE TRUE FALSE TRUE FALSE TRUE
union(x2, y2)
#> [1] 2 4 6 8 10 5

# X & !Y <-> setdiff(x, y)
x1 & !y1
#> [1] FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE FALSE FALSE
setdiff(x2, y2)
#> [1] 2 4 6 8

# xor(X, Y) <-> setdiff(union(x, y), intersect(x, y))
```

```
xor(x1, y1)
#> [1] FALSE TRUE FALSE TRUE TRUE TRUE FALSE TRUE FALSE FALSE
setdiff(union(x2, y2), intersect(x2, y2))
#> [1] 2 4 6 8 5
```

When first learning subsetting, a common mistake is to use `x[which(y)]` instead of `x[y]`. Here the `which()` achieves nothing: it switches from logical to integer subsetting but the result will be exactly the same. In more general cases, there are two important differences. First, when the logical vector contains `NA`, logical subsetting replaces these values by `NA` while `which()` drops these values. Second, `x[-which(y)]` is **not** equivalent to `x[!y]`: if `y` is all `FALSE`, `which(y)` will be `integer(0)` and `-integer(0)` is still `integer(0)`, so you'll get no values, instead of all values. In general, avoid switching from logical to integer subsetting unless you want, for example, the first or last `TRUE` value.

5.5 Exercises

1. Install the `tidyverse` package, if you already have it installed upgrade to the latest version. This can be done by either typing `install.packages("tidyverse")` in the console or by using the “Packages” tab inside RStudio.

The `tidyverse` package is a collection of other packages and will take a while to install. Also, you may get an error that R could not move a file or package from a temporary directory to its final location. This happens because of our corporate virus scanner. The file is being virus scanned when R tries to move it. The simplest solution is to reinstall just the offending package. You can also go to the temporary directory and manually move it via windows explorer.

2. Read *A Layered Grammar of Graphics*. This is a shortish paper that introduces the concepts of the grammar of graphics and forms the basis for `ggplot`.
3. Read and do the exercises in Chapters 1-3 of *R for Data Science*.
4. Bring a couple example plots from our reports to next class. The goal is to have each of us work on a different type of plot so we can begin to build our plotting library.
5. `which()` allows you to convert a boolean representation to an integer representation. There's no reverse operation in base R. Create an `unwhich` function. `unwhich(which(x), length(x))` should return your original vector.

```
x <- sample(10) < 4
which(x)

unwhich <- function(x, n) {
  # your code here
}
unwhich(which(x), 10)
```

Part III

R4DS

Chapter 6

Data Visualization

6.1 Why ggplot2

The transferrable skills from ggplot2 are not the idiosyncracies of plotting syntax, but a powerful way of thinking about visualisation, as a way of **mapping between variables and the visual properties of geometric objects** that you can perceive.

— Hadley Wickham

Base plotting is **imperative**, it's about what you do. You set up your layout(), then you go to the first group (drug) You add the points for that group (drug) along with a title. Then you fit and plot a best-fit-line for the first grouping, then the second grouping, and so on. Then you go on to the next plot. After 20 of those, you end with a legend.

ggplot2 plotting is **declarative**, it's about what **your graph is**. The graph has drug group mapped to the x-axis, prevalence rate mapped to the y, and abuse type mapped to the color. The graph displays both points and best-fit lines for each drug group and it is faceted into one-plot-per-drug group, with a drug group described by its market name.

- **Functional** data visualization
 1. Wrangle your data
 2. Map data elements to visual elements
 3. Tweak scales, guides, axis, labels, theme
- Easy to **reason** about how the data drives the visualization
- Easy to **iterate**
- Easy to be **consistent**

ggplot2 is a huge package: philosophy + functions ...but it is very well organized

ggplot2 has it's one website with some **very** good examples and how to do common task.

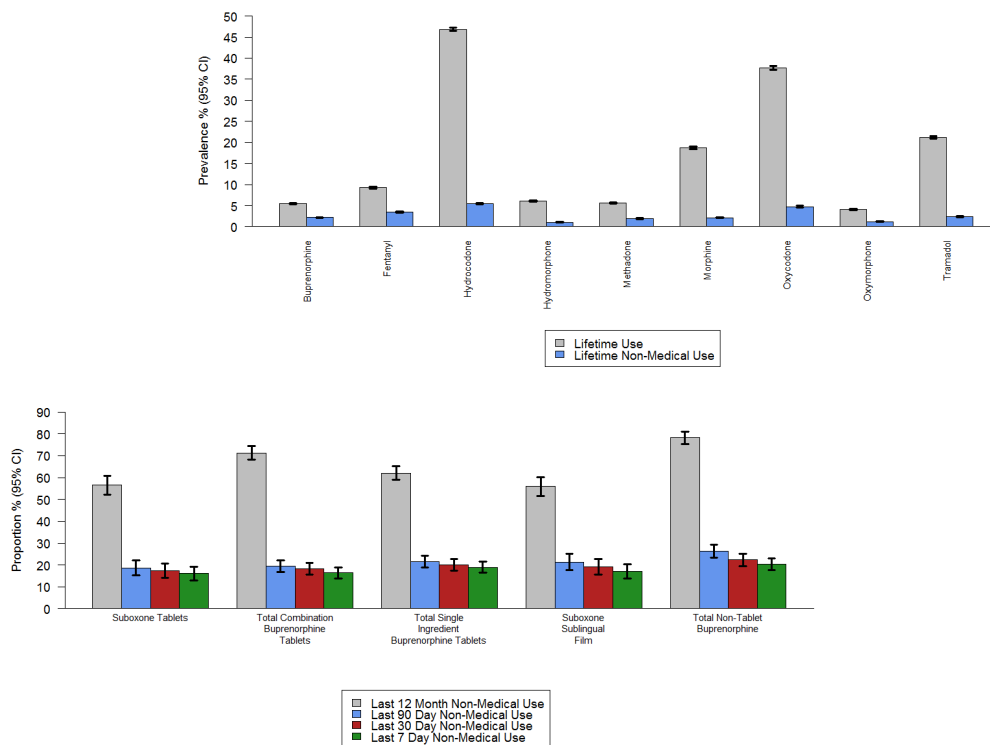
See <http://ggplot2.tidyverse.org/reference>



On 6/15 ggplot2 2.3.0 will come out and there are Breaking Changes. If you upgrade your version of ggplot there may be instances that code from the book (or websites) will no longer work.

6.2 Example

Going to throw a lot at you ...but you'll know where and what to look for. For just about every plotting task there are multiple ways to achieve the desired result.



What is similar / different between these plots? What is and what isn't driven by data?

We'll build this style of plot in stages. In chapter 9 of R for Data Science we will go into detail about how to get our data in this format.

6.2.1 Data

All plots start with data. `ggplot` expects the data to be in a “Tidy Data” format. We'll dive deeper into “tidy data” in Chapter 9 of R for Data Science, but for now the basic principle is

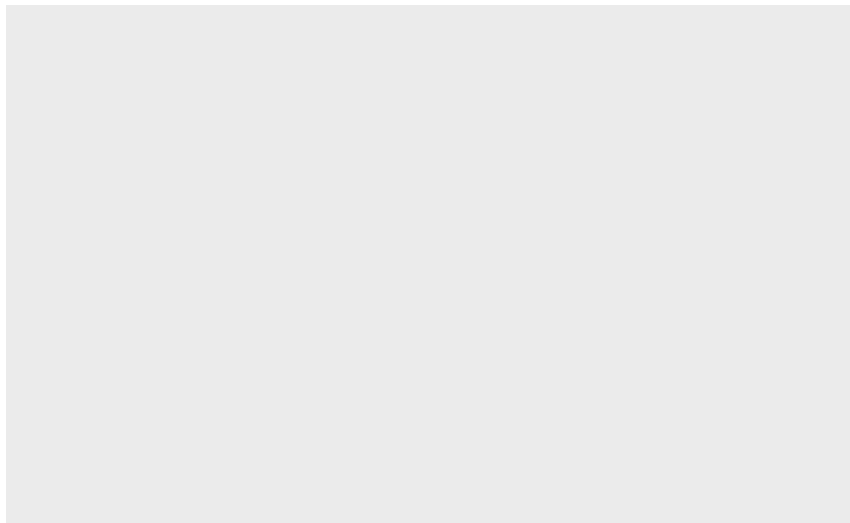
1. Each variable forms a **column**
2. Each observation forms a **row**
3. Each observational unit forms a table

```
library(tidyverse)
#> -- Attaching packages ----- tidyverse 1.2.1 --
#> v ggplot2 3.0.0    v purrr  0.2.5
#> v tibble  1.4.2    v dplyr  0.7.6
#> v tidyr   0.8.1    v stringr 1.3.1
#> v readr   1.1.1    v forcats 0.3.0
#> -- Conflicts ----- tidyverse_conflicts() --
#> x dplyr::filter() masks stats::filter()
#> x dplyr::lag()    masks stats::lag()
dat <- readRDS("./data/bargraphdat.RDS")
dat
#> # A tibble: 18 x 5
```



```
#>   drug      use_type mean lower upper  
#>   <chr>      <chr>   <dbl> <dbl> <dbl>  
#> 1 Buprenorphine use    5.45  5.26  5.64  
#> 2 Fentanyl     use    9.26  9.02  9.51  
#> 3 Hydrocodone  use   46.8  46.4  47.2  
#> 4 Hydromorphone use    6.08  5.89  6.28  
#> 5 Methadone    use    5.59  5.39  5.79  
#> 6 Morphine     use   18.7  18.4  19.0  
#> # ... with 12 more rows
```

```
p <- ggplot(data = dat)  
p
```



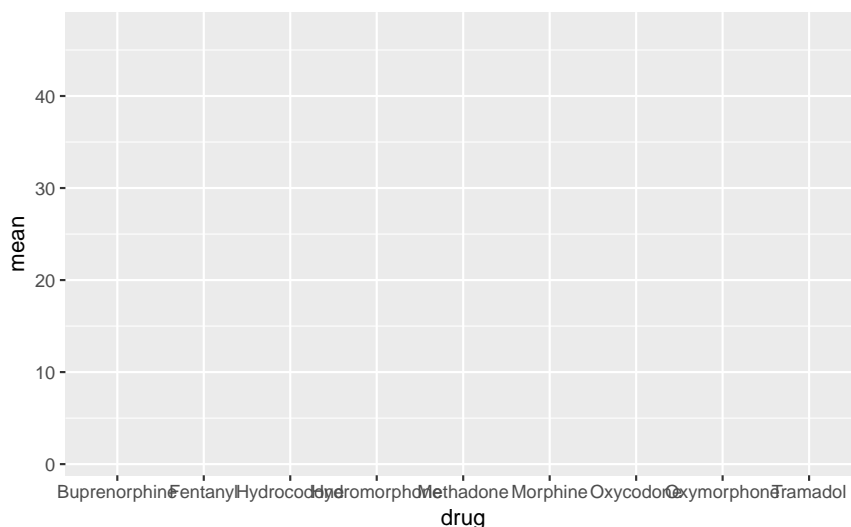
That's uninteresting. We haven't mapped the data to our plot yet. Let's work on getting the bar chart roughly right.

6.2.2 Aesthetics

Aesthetics map data to visual elements or parameters.

- drug -> x-axis
- mean -> y-axis
- use_type -> color

```
p <- ggplot(data = dat, aes(x = drug, y = mean, color = use_type))  
p
```



6.2.3 Geoms

Geoms are short for geometric objects which are displayed on the plot. Some of the more familiar ones are

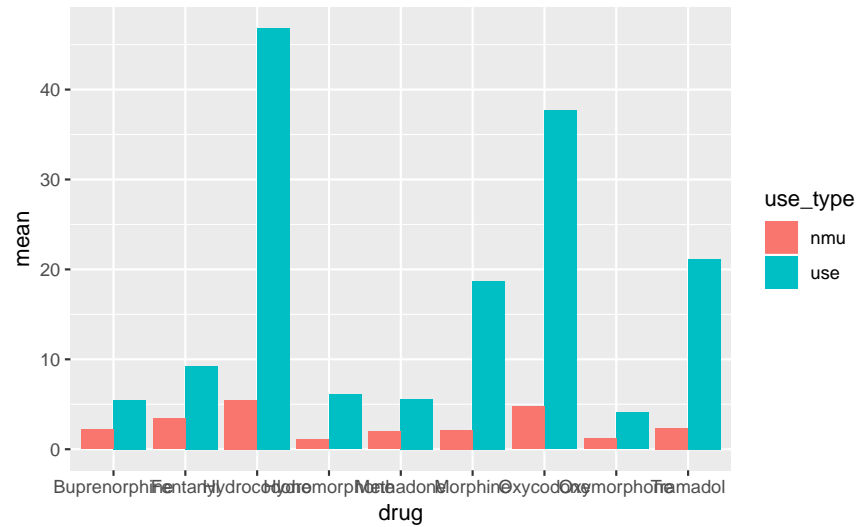
	Type	Function
Point		<code>geom_point()</code>
Line		<code>geom_line()</code>
Bar		<code>geom_bar()</code> , <code>geom_col()</code>
Histogram		<code>geom_histogram()</code>
Regression		<code>geom_smooth()</code>
Boxplot		<code>geom_boxplot()</code>
Text		<code>geom_text()</code>
Vert./Horiz. Line		<code>geom_{vh}line()</code>
Count		<code>geom_count()</code>
Density		<code>geom_density()</code>

Those are just the top 10 most popular geoms

See <http://ggplot2.tidyverse.org/reference/> for many more options

Or just start typing `geom_` in RStudio

```
#> [1] "geom_abline"      "geom_area"        "geom_bar"         "geom_bin2d"
#> [5] "geom_blank"       "geom_boxplot"     "geom_col"         "geom_contour"
#> [9] "geom_count"       "geom_crossbar"    "geom_curve"       "geom_density"
#> [13] "geom_density_2d"  "geom_density2d"   "geom_dotplot"     "geom_errorbar"
#> [17] "geom_errorbarh"   "geom_freqpoly"    "geom_hex"         "geom_histogram"
#> [21] "geom_hline"       "geom_jitter"      "geom_label"       "geom_line"
#> [25] "geom_linerange"   "geom_map"         "geom_path"        "geom_point"
#> [29] "geom_pointrange"  "geom_polygon"     "geom_qq"          "geom_qq_line"
#> [33] "geom_quantile"    "geom_raster"      "geom_rect"        "geom_ribbon"
#> [37] "geom_rug"         "geom_segment"     "geom_sf"          "geom_smooth"
#> [41] "geom_spoke"       "geom_step"        "geom_text"        "geom_tile"
#> [45] "geom_violin"      "geom_vline"
```

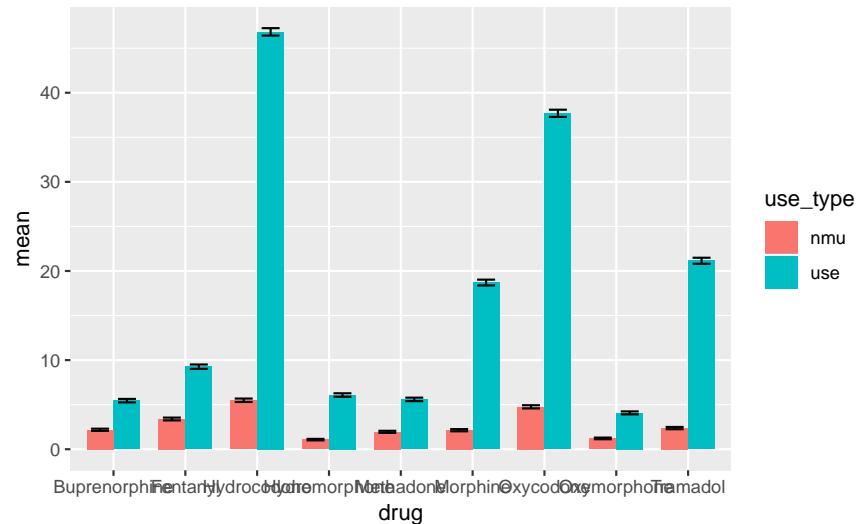



```
geom_*(mapping, data, stat, position)
```

- **data** Geoms can have their own data
 - Has to map onto global coordinates
- **map** Geoms can have their own aesthetics
 - Inherits global aesthetics
 - Have geom-specific aesthetics
 - * `geom_point` needs `x` and `y`, optional `shape`, `color`, `size`, etc.
 - * `geom_ribbon` requires `x`, `ymin` and `ymax`, optional `fill`
 - `?geom_ribbon`
- **stat** Some geoms apply further transformations to the data
 - All respect `stat = 'identity'`
 - Ex: `geom_histogram` uses `stat_bin()` to group observations
- **position** Some adjust location of objects
 - `'dodge'`, `'stack'`, `'jitter'`

Now let's add the error bars to our plot. We will have to add the upper and lower bounds to our aesthetics, and align them with our bars.

```
p <- ggplot(data = dat, aes(x = drug, y = mean, fill = use_type, ymin = lower, ymax = upper)) +
  geom_col(position = "dodge", width = 0.75) +
  geom_errorbar(position = position_dodge(width = 0.75), width = 0.5)
p
```



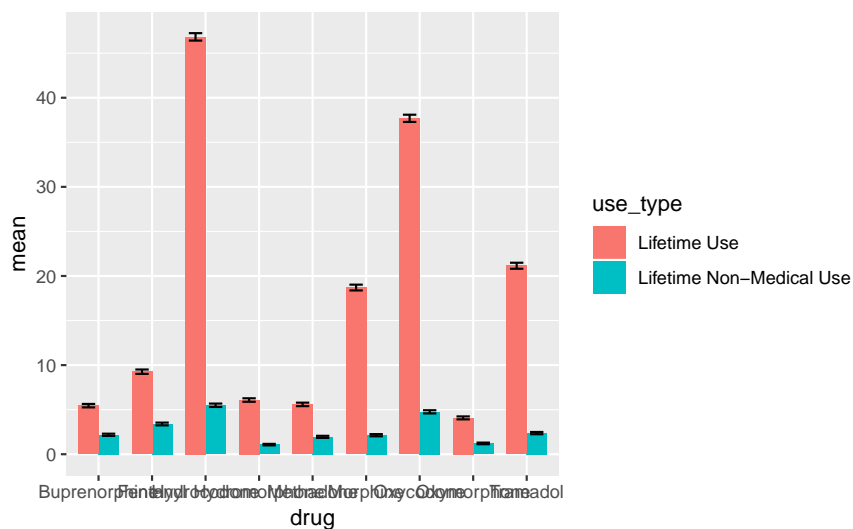
We've come pretty close to recreating the original plot. We still have some tweaking to do.

1. Reorder the grouping so that “Use” comes before “Non-Medical Use” and use the full description.
2. Change the fill colors
3. Change the y-axis label to “Prevalence % (95% CI)”
4. Remove the x-axis label “drug”.
5. Change the y-axis scales to go in increments of 5
6. Rotate the x-axis labels
7. Remove the variable name over the legend.
8. Move the legend to the bottom

The first one is handled with our data. Factors to the rescue, while the second can be done with a named vector.

```
# convert the use_type to a factor with the correct label
dat$use_type <- factor(dat$use_type,
  levels = c("use", "nmu"),
  labels = c("Lifetime Use", "Lifetime Non-Medical Use"))

p <- ggplot(data = dat, aes(x = drug, y = mean, fill = use_type, ymin = lower, ymax = upper)) +
  geom_col(position = "dodge", width = 0.75) +
  geom_errorbar(position = position_dodge(width = 0.75), width = 0.5)
p
```



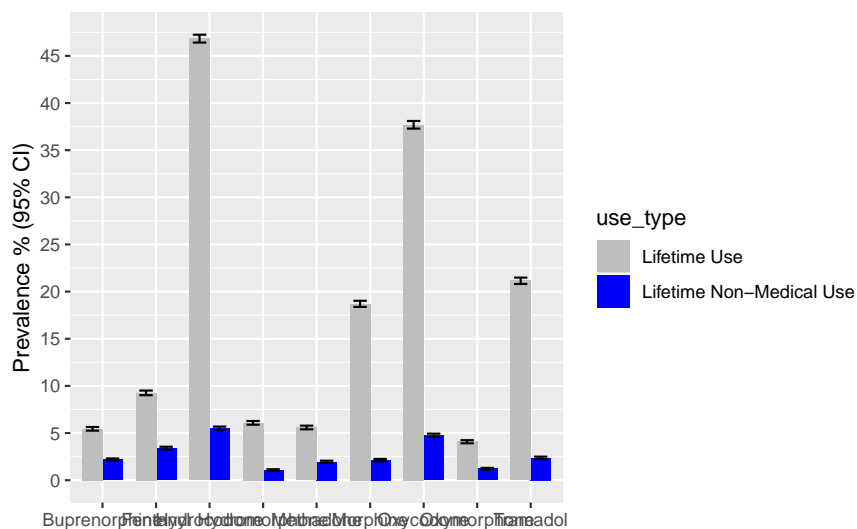
6.2.4 Scales

Scales control the details of how data values are translated to visual properties. Override the default scales to tweak details like the axis labels or legend keys, or to use a completely different translation from data to aesthetic.

`labs()` `xlab()` `ylab()` and `ggtitle()` modify the axis, legend, and plot labels.

```
bar_colors <- c("Lifetime Use" = "grey", "Lifetime Non-Medical Use" = "blue")

p <- ggplot(data = dat, aes(x = drug, y = mean, fill = use_type, ymin = lower, ymax = upper)) +
  geom_col(position = "dodge", width = 0.75) +
  geom_errorbar(position = position_dodge(width = 0.75), width = 0.5) +
  scale_fill_manual(values=bar_colors) + # change the bar colors
  scale_y_continuous(breaks = seq(0, ceiling(max(dat$upper)), 5) ) + # change the y-axis scale
  labs(x = NULL, # Remove the x-axis label "drug"
       y = "Prevalence % (95% CI)" # Change the y-axis label
  )
p
```





library `scales` provides many useful functions for automatically determining breaks and labels for axes and legends. Also has many useful formatting functions such as commas and percentages

6.2.5 Themes

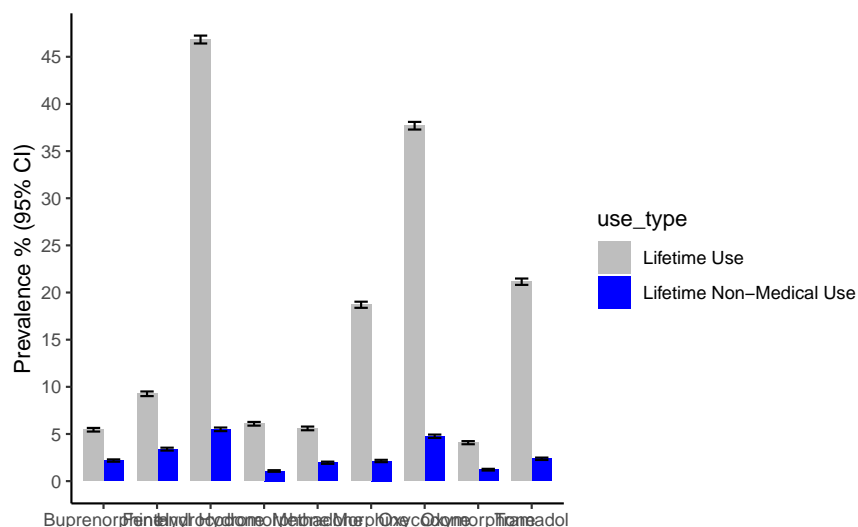
Themes control the display of all non-data elements of the plot. You can change just about everything, fonts, font sizes, background colors, etc. You can override all settings with a complete theme like `theme_bw()`, or choose to tweak individual settings by using `theme()` and the `element_` functions.

There are a handful of built in themes and tons of packages that have additional themes. `ggthemes` has a collection of themes used by various organization (Ex. The Economist, Fivethirtyeight.com, The Wall St. Journal, etc)

Themes contain a huge number of parameters, grouped by plot area:

- **Global options:** `line`, `rect`, `text`, `title`
- **axis:** x-, y- or other axis title, ticks, lines
- **legend:** Plot legends
- **panel:** Actual plot area
- **plot:** Whole image
- **strip:** Facet labels

```
p + theme_classic()
```



This is almost what we want. Our final code would look like:

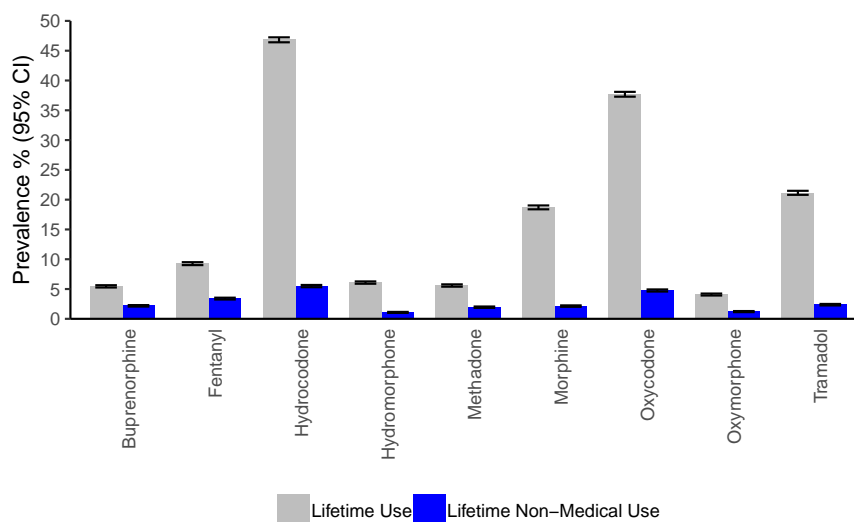
```
library(tidyverse)
dat <- readRDS("./data/bargraphdat.RDS")
# convert the use_type to a factor with the correct label
dat$use_type <- factor(dat$use_type, levels = c("use", "nmu"), labels = c("Lifetime Use", "Lifetime Non-
bar_colors <- c("Lifetime Use" = "grey", "Lifetime Non-Medical Use" = "blue")

p <- ggplot(data = dat, aes(x = drug, y = mean, fill = use_type, ymin = lower, ymax = upper)) +
  geom_col(position = "dodge", width = 0.75) +
  geom_errorbar(position = position_dodge(width = 0.75), width = 0.5) +
  scale_fill_manual(values=bar_colors) +      # change the bar colors
```

```

coord_cartesian(ylim=c(0, 50)) +
scale_y_continuous(breaks = seq(0, ceiling(max(dat$upper)+5), 5), # change the y-axis scale
                  expand = c(0,0)) + # remove the spacing between the x axis and the bars
labs(x = NULL, # Remove the x-axis label "drug"
     y = "Prevalence % (95% CI)" + # Change the y-axis label
theme_classic() +
theme(legend.position = "bottom", # move the legend to the bottom
      legend.title = element_blank(), # remove the legend variable
      axis.text.x = element_text(angle = 90, hjust = 1), # rotate the x-axis text
      axis.ticks.x = element_blank()) # remove the x axis tick marks
p

```



6.3 Facets

Facets are subplots of the data with each subplot displaying one subset of the data. there are two ways to create facets: `facet_grid` and `facet_wrap`.

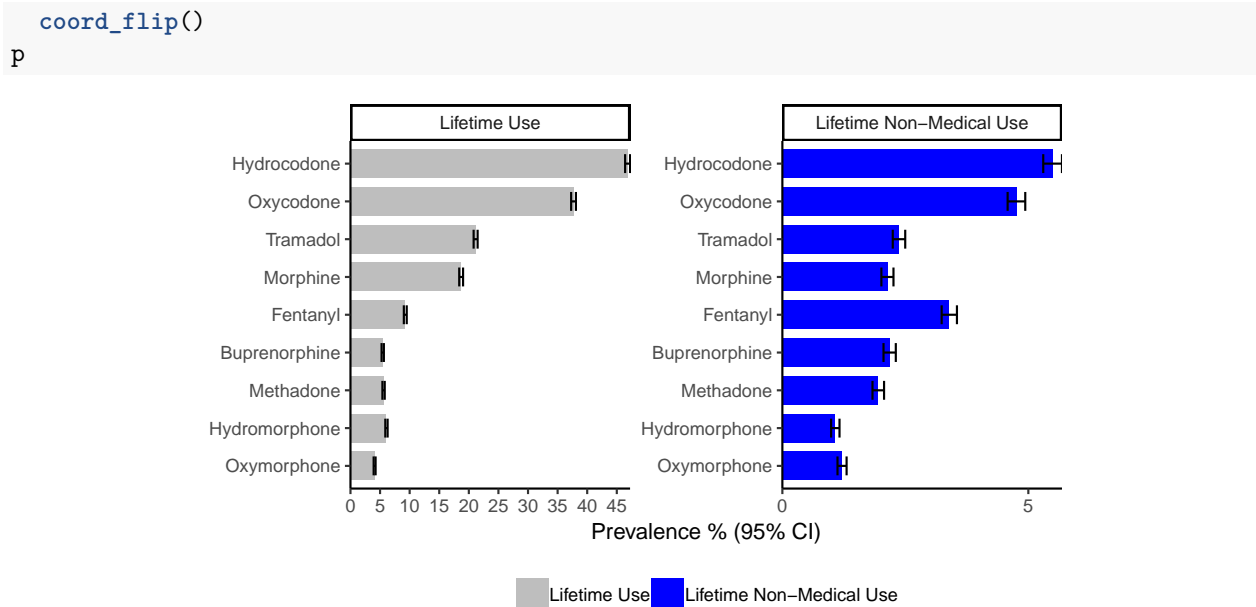
`facet_grid` forms a matrix of panels defined by row and column faceting variables. It is most useful when you have two discrete variables, and all combinations of the variables exist in the data.

`facet_wrap` wraps a 1d sequence of panels into 2d. This is generally a better use of screen space than `facet_grid` because most displays are roughly rectangular.

```

p <- ggplot(data = dat, aes(x = fct_reorder(drug, mean), y = mean, fill = use_type, ymin = lower, ymax = upper)) +
  geom_col(width = 0.75) +
  geom_errorbar(position = position_dodge(width = 0.75), width = 0.5) +
  facet_wrap(~ use_type, scales = "free") +
  scale_fill_manual(values=bar_colors) + # change the bar colors
  scale_y_continuous(breaks = seq(0, ceiling(max(dat$upper)), 5), # change the y-axis scale
                    expand = c(0,0)) + # remove the spacing between the x axis and the bars
  labs(x = NULL, # Remove the x-axis label "drug"
       y = "Prevalence % (95% CI)" + # Change the y-axis label
  theme_classic() +
  theme(legend.position = "bottom", # move the legend to the bottom
        legend.title = element_blank()) + # remove the legend variable

```

6.4 Stats

While we didn't use them for this particular plot `stat_*()` function can be a huge time saver. `stat_*` functions display statistical summaries of the data. For a bar plot there is no reason the count then number of items in a group (or percentage) on the data. Instead we can use the appropriate function have it calculated automatically for us.

```
#> [1] "stat_bin"           "stat_bin_2d"        "stat_bin_hex"       "stat_bin2d"
#> [5] "stat_binhex"        "stat_boxplot"       "stat_contour"       "stat_count"
#> [9] "stat_density"       "stat_density_2d"    "stat_density2d"     "stat_ecdf"
#> [13] "stat_ellipse"       "stat_function"      "stat_identity"      "stat_qq"
#> [17] "stat_qq_line"       "stat_quantile"      "stat_sf"            "stat_smooth"
#> [21] "stat_spoke"         "stat_sum"           "stat_summary"       "stat_summary_2d"
#> [25] "stat_summary_bin"   "stat_summary_hex"   "stat_summary2d"     "stat_unique"
#> [29] "stat_ydensity"
```

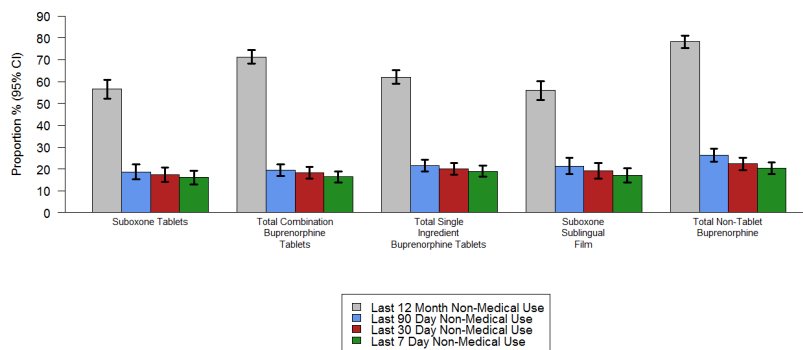
There are many more useful `stat_*()` functions in various packages.

6.5 Saving

Save your plot with `ggsave`. Use the correct extension for the plot type you wish to save. E.g .pdf for pdf, .png for png, etc. See `?ggsave` for details and other parameters.

6.6 Exercises

1. Modify the above code to produce the plot below. You can read in the data with: `dat <- readRDS("../data/bargraphdat2.RDS")`



2. If you wanted to make this style of plot a function, what would you need to pass to the function? What customization would you allow a user to make and what would you not?
3. For the plot you brought, create a data set and create the the plot using ggplot.
4. For the above plot (exercise 3). Re-imagine a different visualization for the data and create the plot using ggplot.
5. Begin making a RADARS theme. What is our font, font size for various elements, background, etc. We will end up making a custom theme based on this for everyone to use. This will allow us to get presentation quality graphics quickly.
6. Read Chapter 2 (Workflow: Basics)

6.7 Resources and Links

Learn more

- **ggplot2 docs:** <http://ggplot2.tidyverse.org/>
- **Hadley Wickham's ggplot2 book:** <https://www.amazon.com/dp/0387981403/>

Noteworthy RStudio Add-Ins

- ggplotThemeAssist: Customize your ggplot theme interactively
- ggedit: Layer, scale, and theme editing

General Help and How-To's

- <http://ggplot2.tidyverse.org/reference/>
- ggplot wiki
- R Cookbook
- ggplot2-toolbox
- ggplot tutorial
- Examples and Themes
- hmrthemes
- Visualizing Data

Tips and Tricks

- Beautiful Plots Cheatsheet
- Pretty Scatter Plots
- Corporate Palettes
- Maps
- Writing Functions with ggplot
- ggplot2 function writing tips

- Cowplot Vignette

Math and symbols

- labeller
- bquote method

Base Plot

- Base plot
- Base plot limits
- base R Graphics

Chapter 7

Data Transformation

```
library(tidyverse)
```

7.1 Tibbles

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



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

Key advantages of tibbles:

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

When creating a tibble:

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

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

versus

```
mydf <- data.frame(  
  x = 1:5,  
  y = 1,  
  z = x ^ 2 + y  
)
```

```

  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.frame()`.



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

7.2 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.

7.3 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.

7.4 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.

7.5 Add new variables with `mutate()`

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

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



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

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

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

7.6 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 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.

7.6.1 Intermediate steps

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

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

7.6.2 Overwrite the original

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

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

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



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

7.6.3 Function composition

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



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

7.6.4 Use the pipe operator `%>%`

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

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

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

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

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

7.7 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()`

7.8 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.

Chapter 8

Data Import

```
library(tidyverse)
```

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

8.1 Text Files

8.1.1 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

8.1.2 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

8.2 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 `sas7bcat` 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.

8.3 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

8.4 Databases

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

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

```
con <- dbConnect(odbc::odbc(),                               # for a Microsoft 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.

Chapter 9

Strings

This section covers basic string manipulation in R. See Chapter 11 in R for Data Science for a more complete coverage. The first part of this chapter will probably look familiar, but the rest will focus on **regular expressions**. Regular expressions (sometimes referred to as **regex** or **regexp**) are a concise language for describing patterns in strings.

While base R has some string functions they tend to be inconsistent in the order of the parameters, which makes them hard to remember and you end up needing the help files. As with most of this course we will focus on the Tidyverse, and in particular the **stringr** package, which is not part of the core tidyverse package so you will have to load it explicitly with `library(stringr)`

```
library(tidyverse)
library(stringr)
```

9.1 Basics

As with most languages you create a string by putting the text in single or double quotes `x <- "This is my string!"`, and you can create a vector of strings with `c()`. There are a handful of special characters but the most common are the newline character `"\n"`, and the tab character `"\t"`. You may also see strings like `\u00AE` or `\u00b5` which is a way of including non-English characters or special symbols that work on all platforms. These are typically known as Unicode characters.

```
x <- "This my registered\u00AE drug"
x
#> [1] "This my registered® drug"
```

One nice feature of the stringr package is all string functions begin with `str_`. This makes it easy in RStudio to find useful string processing functions.



See the tidyverse package **glue** which can glue strings to data in R. Small, fast, dependency free interpreted string literals.

9.1.1 Helpful Basic String Functions

- `str_c()` joins multiple strings together into a single string (similar to `paste` or `paste0` from base R)
- `str_to_upper()` converts the string to all uppercase characters

- `str_to_lower()` converts the string to all lowercase characters
- `str_to_title()` capitalizes the first character of each word in the string
- `str_length()` the length of a string
- `str_wrap()` wrap strings into nicely formatted paragraphs
- `str_trim()` trim white space from a string
- `str_pad()` pad a string
- `str_order()` order a character vector
- `str_sort()` sort a character vector
- `str_sub()` extract and replace substrings from a character vector



While `str_c` looks and acts very similar to the base R functions `paste` or `paste0`, they act very differently with regards to missing data. `paste` and `paste0` replace missing values with the text “NA”, while `str_c` propagates the missing value.

```
str_c("This", "is", "a", "string", NA, "with", "a", "missing", "value")
#> [1] NA
paste("This", "is", "a", "string", NA, "with", "a", "missing", "value")
#> [1] "This is a string NA with a missing value"
```

Notice `str_c` and `paste` have both a `sep` and `collapse` argument. While these appear to be the same they are not. The `sep` argument is the string inserted between arguments to `str_c`, while `collapse` is the string used to separate any elements of the character vector into a character vector of length one.

```
str_c("This", "is", "not", "a", "character", "vector", sep = "_")
#> [1] "This_is_not_a_character_vector"
x <- c("This", "is", "a", "character", "vector")
x
#> [1] "This"      "is"        "a"         "character" "vector"
str_c(x, sep = "_")
#> [1] "This"      "is"        "a"         "character" "vector"
str_c(x, collapse = "_")
#> [1] "This_is_a_character_vector"
```

9.2 Regular Expressions

A regular expression (regex or regexp for short) is a special text string for describing a search pattern. Usually this pattern is then used by string searching algorithms for “find” or “find and replace” operations on strings. Patterns can be a bit tricky to wrap your head around at first since the most common case is simply looking for a sequence of letters. Patterns are much more general. Think about how would you tell a computer to look for any phone number or any email address.



Just about every modern programming language has the ability to use regular expressions. In SAS regular expressions are implemented in the PRX series of functions such as: `PRXMATCH`, `PRXSEARCH`, `PRXPARSE`, `PRXCHANGE`.

Regexps are a very terse language that allow you to describe patterns in strings. They take a little while to get your head around, but once you understand them, you’ll find them extremely useful. Any non-trivial regular expression looks like a cat walked across your keyboard.



To learn regular expressions, use `str_view()` and `str_view_all()`. These functions take a character vector and a regular expression, and show you how they match.

9.2.1 Basic Matches

We'll start with very simple regular expressions and then gradually get more and more complicated. Once you've mastered pattern matching, you'll learn how to apply those ideas with various stringr functions. We'll use the built in fruit data set.

```
fruit
#> [1] "apple"           "apricot"           "avocado"
#> [4] "banana"          "bell pepper"       "bilberry"
#> [7] "blackberry"      "blackcurrant"      "blood orange"
#> [10] "blueberry"       "boysenberry"       "breadfruit"
#> [13] "canary melon"    "cantaloupe"        "cherimoya"
#> [16] "cherry"          "chili pepper"      "clementine"
#> [19] "cloudberry"     "coconut"           "cranberry"
#> [22] "cucumber"       "currant"           "damson"
#> [25] "date"           "dragonfruit"       "durian"
#> [28] "eggplant"       "elderberry"        "feijoa"
#> [31] "fig"            "goji berry"        "gooseberry"
#> [34] "grape"          "grapefruit"        "guava"
#> [37] "honeydew"       "huckleberry"       "jackfruit"
#> [40] "jambul"         "jujube"            "kiwi fruit"
#> [43] "kumquat"        "lemon"             "lime"
#> [46] "loquat"         "lychee"            "mandarine"
#> [49] "mango"          "mulberry"          "nectarine"
#> [52] "nut"            "olive"             "orange"
#> [55] "pamelo"         "papaya"            "passionfruit"
#> [58] "peach"          "pear"              "persimmon"
#> [61] "physalis"       "pineapple"         "plum"
#> [64] "pomegranate"    "pomelo"            "purple mangosteen"
#> [67] "quince"         "raisin"            "rambutan"
#> [70] "raspberry"      "redcurrant"        "rock melon"
#> [73] "salal berry"    "satsuma"           "star fruit"
#> [76] "strawberry"     "tamarillo"         "tangerine"
#> [79] "ugli fruit"     "watermelon"
```

One of the most common uses of regular expressions is to find strings that contain an exact sequence of letters.

```
str_subset(fruit, "berry")
#> [1] "bilberry" "blackberry" "blueberry" "boysenberry" "cloudberry"
#> [6] "cranberry" "elderberry" "goji berry" "gooseberry" "huckleberry"
#> [11] "mulberry" "raspberry" "salal berry" "strawberry"
```



`str_subset()` keeps strings matching a pattern. We will use these as examples of how to use patterns, but work with any function which takes a pattern.

This works great for quite a few cases, but what if you wanted to find just “grape” and not “grapefruit”?

```
str_subset(fruit, "grape")
#> [1] "grape"      "grapefruit"
```

9.2.2 Special Characters

Once you get beyond basic substring matching you need some helpers. Below is a list of common helpers.

- `.` match any character (except a newline)
- `^` match the start of the string
- `$` match the end of the string

So to match “grape” but not “grapefruit” you could do this is a couple ways.

```
str_subset(fruit, "grape$")  # match all words that end in "grape"
#> [1] "grape"
str_subset(fruit, "^grape$") # match all words the start AND end in "grape"
#> [1] "grape"
```

Find the fruits that have an “a” in them but not ones that have the only “a” as the starting character.

```
str_subset(fruit, ".a")
#> [1] "avocado"      "banana"      "blackberry"
#> [4] "blackcurrant" "blood orange" "breadfruit"
#> [7] "canary melon" "cantaloupe"  "cherimoya"
#> [10] "cranberry"    "currant"     "damson"
#> [13] "date"         "dragonfruit" "durian"
#> [16] "eggplant"     "feijoa"      "grape"
#> [19] "grapefruit"   "guava"       "jackfruit"
#> [22] "jambul"       "kumquat"     "loquat"
#> [25] "mandarine"    "mango"       "nectarine"
#> [28] "orange"       "pamelo"      "papaya"
#> [31] "passionfruit" "peach"       "pear"
#> [34] "physalis"     "pineapple"   "pomegranate"
#> [37] "purple mangosteen" "raisin"     "rambutan"
#> [40] "raspberry"    "redcurrant"  "salal berry"
#> [43] "satsuma"      "star fruit"  "strawberry"
#> [46] "tamarillo"    "tangerine"   "watermelon"
```

If “.” matches any character, how do you match the character “.”?

You need to use an “escape” to tell the regular expression you want to match it exactly, not use its special behavior. Like strings, regexps use the backslash, `\`, to escape special behavior. So to match an `.`, you need the regexp `\.`. Unfortunately this creates a problem. We use strings to represent regular expressions, and `\` is also used as an escape symbol in strings. So to create the regular expression `\.` we need the string `"\\."`.

If `\` is used as an escape character in regular expressions, how do you match a literal `\`? Well you need to escape it, creating the regular expression `\\`. To create that regular expression, you need to use a string, which also needs to escape `\`. That means to match a literal `\` you need to write `"\\\\"` — you need four backslashes to match one!

9.2.3 Character Classes and Alternatives

There are a number of special patterns that match more than one character. You’ve already seen `.`, which matches any character apart from a newline. There are many other useful tools:

- `\d` match any digit
- `[:digits]` match any digit
- `\D` match any non digit
- `[:alpha:]` match any letter
- `[:lower:]` match any lowercase letter
- `[:upper:]` match any uppercase letter
- `[:alnum:]` match any alpha numeric character.
- `[:punct:]` match any punctuation (see cheat sheet)
- `|` or (ex `ad|e` matches “ad” or “e”)
- `[]` matches on of (ex `[ade]` matches “a”, “d” or “e”, equivalent to `a|d|e`)
- `[^]` match anything but (ex `[^ade]` matches everything but “a”, “d” or “e”)
- `[-]` match a range (ex `[0-5]` matches numbers between 0 and 5 inclusive)



Remember, to create a regular expression containing `\d`, you need to escape the `\` for the string, so you type `"\\d"`.

A character class containing a single character is a nice alternative to backslash escapes when you want to include a single meta character in a regex. Many people find this more readable.

9.2.4 Repetition

The next step up in power involves controlling how many times a pattern matches:

- `?` match zero or 1
- `*` match zero or more
- `+` match one or more
- `{n}`: exactly n
- `{n,}`: n or more
- `{,m}`: at most m
- `{n,m}`: between n and m

Find all the fruits with three or more “e”’s.

```
str_subset(fruit, ".*e.*e.*e")
#> [1] "bell pepper"      "clementine"      "elderberry"
#> [4] "purple mangosteen"

# or more succinctly
str_subset(fruit, "(.*e){3,}")
#> [1] "bell pepper"      "clementine"      "elderberry"
#> [4] "purple mangosteen"
```

9.2.5 Grouping

Parentheses are a way to disambiguate complex expressions. Parentheses also create a **numbered capturing group** (number 1, 2 etc.). A capturing group stores **the part of the string** matched by the part of the regular expression inside the parentheses. You can refer to the same text as previously matched by a capturing group with **backreferences**, like `\1`, `\2` etc.

For example, find all the fruits with a double letter

```
str_subset(fruit, "([:alpha:])\\1")
#> [1] "apple"           "bell pepper"     "bilberry"
#> [4] "blackberry"      "blackcurrant"    "blood orange"
```

```
#> [7] "blueberry"      "boysenberry"    "cherry"
#> [10] "chili pepper"   "cloudberry"     "cranberry"
#> [13] "currant"        "eggplant"       "elderberry"
#> [16] "goji berry"     "gooseberry"     "huckleberry"
#> [19] "lychee"         "mulberry"       "passionfruit"
#> [22] "persimmon"      "pineapple"      "purple mangosteen"
#> [25] "raspberry"      "redcurrant"     "salal berry"
#> [28] "strawberry"     "tamarillo"
```

Find all fruits that have a repeated pair of letters.

```
str_subset(fruit, "(.)\\1")
#> [1] "banana"      "coconut"      "cucumber"     "jujube"      "papaya"
#> [6] "salal berry"
```

9.3 Helpful String Functions

Don't forget that you're in a programming language and you have other tools at your disposal. Instead of creating one complex regular expression, it's often easier to write a series of simpler regexps. If you get stuck trying to create a single regexp that solves your problem, take a step back and think if you could break the problem down into smaller pieces, solving each challenge before moving onto the next one.

9.3.1 Detect Matches

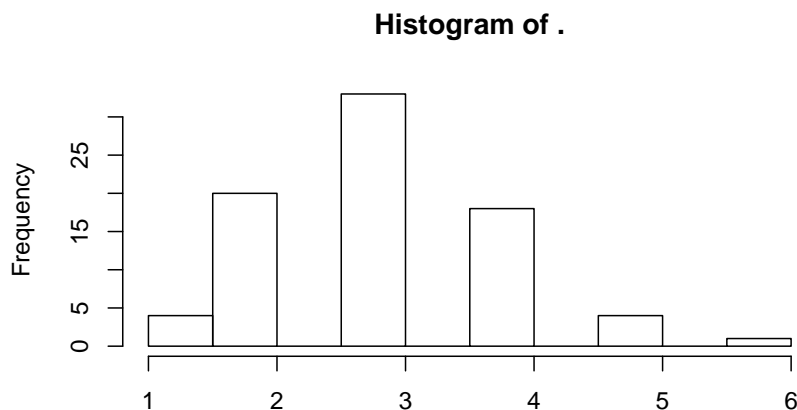
- `str_detect()` Detect the presence of strings matching a pattern (similar to base R `grep1()`)
- `str_count()` Count the number of matches in a string.

What proportion of the fruits have double letters?

```
str_detect(fruit, "([:alpha:])\\1") %>% mean()
#> [1] 0.362
```

What is the distribution of vowels in fruits?

```
str_count(fruit, "[aeiou]") %>% hist()
```





`str_detect()` has a natural fit with `filter` to subset data sets.

9.3.2 Extract Matches

- `str_subset()` keep strings matching a pattern
- `str_extract()` extract first matching patterns from a string.
- `str_extract_all()` extract all matching patterns from a string.
- `str_which()` find the positions of strings matching a pattern (similar to base R `grep()`)

Which fruits have a color in their name?

```
# create our regular expression from a character vector
colors <- c("red", "orange", "yellow", "green", "blue", "purple", "black")
color_match <- str_c(colors, collapse = "|")
color_match
#> [1] "red/orange/yellow/green/blue/purple/black"

str_subset(fruit, color_match)
#> [1] "blackberry"      "blackcurrant"    "blood orange"
#> [4] "blueberry"       "orange"          "purple mangosteen"
#> [7] "redcurrant"
```

Pull out the vowels in all the fruits

```
str_extract(fruit, "[aeiou]")
#> [1] "a" "a" "a" "a" "e" "i" "a" "a" "o" "u" "o" "e" "a" "a" "e" "e" "i"
#> [18] "e" "o" "o" "a" "u" "u" "a" "a" "a" "u" "e" "e" "e" "i" "o" "o" "a"
#> [35] "a" "u" "o" "u" "a" "a" "u" "i" "u" "e" "i" "o" "e" "a" "a" "u" "e"
#> [52] "u" "o" "o" "a" "a" "a" "e" "e" "e" "a" "i" "u" "o" "o" "u" "u" "a"
#> [69] "a" "a" "e" "o" "a" "a" "a" "a" "a" "a" "u" "a"

str_extract_all(fruit, "[aeiou]") %>% head()
#> [[1]]
#> [1] "a" "e"
#>
#> [[2]]
#> [1] "a" "i" "o"
#>
#> [[3]]
#> [1] "a" "o" "a" "o"
#>
#> [[4]]
#> [1] "a" "a" "a"
#>
#> [[5]]
#> [1] "e" "e" "e"
#>
#> [[6]]
#> [1] "i" "e"
```

9.3.3 Replacing Matches

- `str_replace()` replace first matched pattern in a string.

- `str_replace_all()` replace all matched pattern in a string.



These functions are similar to base R functions `sub()` and `gsub()`

Replace vowels with a hyphen

```
str_replace(fruit, "[aeiou]", "-") %>% head()
#> [1] "-pple"      "-pricot"     "-vocado"     "b-nana"      "b-ll pepper"
#> [6] "b-lberry"
str_replace_all(fruit, "[aeiou]", "-") %>% head()
#> [1] "-ppl-"      "-pr-c-t"     "-v-c-d-"     "b-n-n-"      "b-ll p-pp-r"
#> [6] "b-lb-rry"
```

Also, `str_replace_all()` can perform multiple replacements by supplying a named vector.

Instead of replacing with a fixed string you can use backreferences to insert components of the match.

If the fruit name is comprised of 2 or more words swap the first and second word.

```
str_replace(fruit, "([^\s]+) ([^\s]+)", "\\2 \\1")
#> [1] "apple"      "apricot"     "avocado"
#> [4] "banana"     "pepper bell" "bilberry"
#> [7] "blackberry" "blackcurrant" "orange blood"
#> [10] "blueberry"  "boysenberry"  "breadfruit"
#> [13] "melon canary" "cantaloupe"   "cherimoya"
#> [16] "cherry"     "pepper chili" "clementine"
#> [19] "cloudberry" "coconut"      "cranberry"
#> [22] "cucumber"   "currant"      "damson"
#> [25] "date"       "dragonfruit"  "durian"
#> [28] "eggplant"   "elderberry"   "feijoa"
#> [31] "fig"        "berry goji"   "gooseberry"
#> [34] "grape"      "grapefruit"   "guava"
#> [37] "honeydew"   "huckleberry"  "jackfruit"
#> [40] "jambul"     "jujube"       "fruit kiwi"
#> [43] "kumquat"    "lemon"        "lime"
#> [46] "loquat"     "lychee"       "mandarine"
#> [49] "mango"      "mulberry"     "nectarine"
#> [52] "nut"        "olive"        "orange"
#> [55] "pamelo"     "papaya"       "passionfruit"
#> [58] "peach"      "pear"         "persimmon"
#> [61] "physalis"   "pineapple"    "plum"
#> [64] "pomegranate" "pomelo"      "mangosteen purple"
#> [67] "quince"     "raisin"       "rambutan"
#> [70] "raspberry"  "redcurrant"   "melon rock"
#> [73] "berry salal" "satsuma"      "fruit star"
#> [76] "strawberry" "tamarillo"    "tangerine"
#> [79] "fruit ugli" "watermelon"
```

9.3.4 Other String Functions

- `str_split()` split up a string into pieces
- `str_locate()` returns the starting and ending positions of the first match
- `str_locate_all()` returns the starting and ending positions of all matches

Split the fruits up by word

```
str_split(fruit, " ") %>% head(., 10) # split by a space
#> [[1]]
#> [1] "apple"
#>
#> [[2]]
#> [1] "apricot"
#>
#> [[3]]
#> [1] "avocado"
#>
#> [[4]]
#> [1] "banana"
#>
#> [[5]]
#> [1] "bell" "pepper"
#>
#> [[6]]
#> [1] "bilberry"
#>
#> [[7]]
#> [1] "blackberry"
#>
#> [[8]]
#> [1] "blackcurrant"
#>
#> [[9]]
#> [1] "blood" "orange"
#>
#> [[10]]
#> [1] "blueberry"
str_split(fruit, boundary("word")) %>% head(., 10) # split by word boundary
#> [[1]]
#> [1] "apple"
#>
#> [[2]]
#> [1] "apricot"
#>
#> [[3]]
#> [1] "avocado"
#>
#> [[4]]
#> [1] "banana"
#>
#> [[5]]
#> [1] "bell" "pepper"
#>
#> [[6]]
#> [1] "bilberry"
#>
#> [[7]]
#> [1] "blackberry"
#>
#> [[8]]
```

```

#> [1] "blackcurrant"
#>
#> [[9]]
#> [1] "blood" "orange"
#>
#> [[10]]
#> [1] "blueberry"
str_split(fruit, " ", n = 2, simplify = TRUE) # only split into 2 peices
#>      [,1]      [,2]
#> [1,] "apple"    ""
#> [2,] "apricot"   ""
#> [3,] "avocado"   ""
#> [4,] "banana"    ""
#> [5,] "bell"      "pepper"
#> [6,] "bilberry"  ""
#> [7,] "blackberry" ""
#> [8,] "blackcurrant" ""
#> [9,] "blood"     "orange"
#> [10,] "blueberry" ""
#> [11,] "boysenberry" ""
#> [12,] "breadfruit" ""
#> [13,] "canary"    "melon"
#> [14,] "cantaloupe" ""
#> [15,] "cherimoya" ""
#> [16,] "cherry"    ""
#> [17,] "chili"     "pepper"
#> [18,] "clementine" ""
#> [19,] "cloudberry" ""
#> [20,] "coconut"   ""
#> [21,] "cranberry" ""
#> [22,] "cucumber"  ""
#> [23,] "currant"    ""
#> [24,] "damson"     ""
#> [25,] "date"       ""
#> [26,] "dragonfruit" ""
#> [27,] "durian"     ""
#> [28,] "eggplant"   ""
#> [29,] "elderberry" ""
#> [30,] "feijoa"     ""
#> [31,] "fig"        ""
#> [32,] "goji"       "berry"
#> [33,] "gooseberry" ""
#> [34,] "grape"      ""
#> [35,] "grapefruit" ""
#> [36,] "guava"      ""
#> [37,] "honeydew"   ""
#> [38,] "huckleberry" ""
#> [39,] "jackfruit"  ""
#> [40,] "jambul"     ""
#> [41,] "jujube"     ""
#> [42,] "kiwi"       "fruit"
#> [43,] "kumquat"    ""
#> [44,] "lemon"      ""

```



```

#> [45,] "lime"      ""
#> [46,] "loquat"   ""
#> [47,] "lychee"   ""
#> [48,] "mandarine" ""
#> [49,] "mango"    ""
#> [50,] "mulberry" ""
#> [51,] "nectarine" ""
#> [52,] "nut"      ""
#> [53,] "olive"    ""
#> [54,] "orange"   ""
#> [55,] "pamelo"   ""
#> [56,] "papaya"   ""
#> [57,] "passionfruit" ""
#> [58,] "peach"    ""
#> [59,] "pear"     ""
#> [60,] "persimmon" ""
#> [61,] "physalis" ""
#> [62,] "pineapple" ""
#> [63,] "plum"     ""
#> [64,] "pomegranate" ""
#> [65,] "pomelo"   ""
#> [66,] "purple"   "mangosteen"
#> [67,] "quince"   ""
#> [68,] "raisin"   ""
#> [69,] "rambutan" ""
#> [70,] "raspberry" ""
#> [71,] "redcurrant" ""
#> [72,] "rock"     "melon"
#> [73,] "salal"    "berry"
#> [74,] "satsuma"  ""
#> [75,] "star"     "fruit"
#> [76,] "strawberry" ""
#> [77,] "tamarillo" ""
#> [78,] "tangerine" ""
#> [79,] "ugli"     "fruit"
#> [80,] "watermelon" ""

```

stringr is built on top of the **stringi** package. **stringr** is useful when you're learning because it exposes a minimal set of functions, which have been carefully picked to handle the most common string manipulation functions. **stringi**, on the other hand, is designed to be comprehensive. It contains almost every function you might ever need: **stringi** has 234 functions to **stringr**'s 46.

If you find yourself struggling to do something in **stringr**, it's worth taking a look at **stringi**. The packages work very similarly, so you should be able to translate your **stringr** knowledge in a natural way. The main difference is the prefix: **str_** vs. **stri_**.

Chapter 10

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>.

10.1 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 ?? 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.

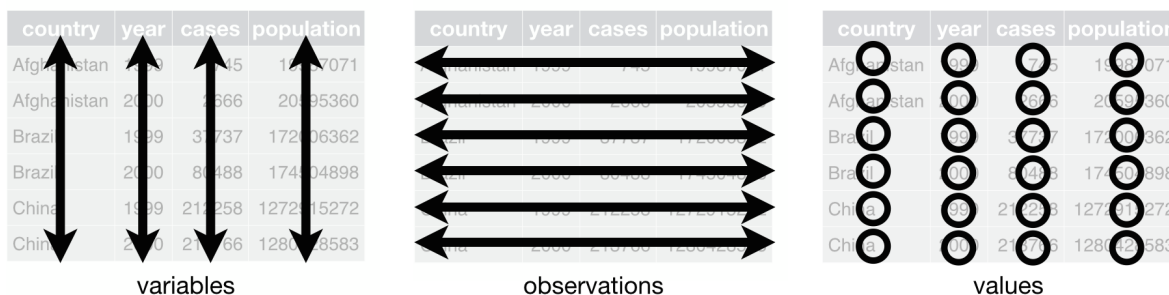


Figure 10.1: Following three rules makes a dataset tidy: variables are in columns, observations are in rows, and values are in cells.

10.2 Example

You can represent the same underlying data in multiple ways. The example below shows the same data organized 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?

10.3 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 tidy: `gather()` and `spread()`.

10.3.1 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

country	year	cases
Afghanistan	1999	745
Afghanistan	2000	2666
Brazil	1999	37737
Brazil	2000	80488
China	1999	212258
China	2000	213766

country	1999	2000
Afghanistan	745	2666
Brazil	37737	80488
China	212258	213766

table4

Figure 10.2: 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 ???. 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
```

10.3.2 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 ??.

```
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
```

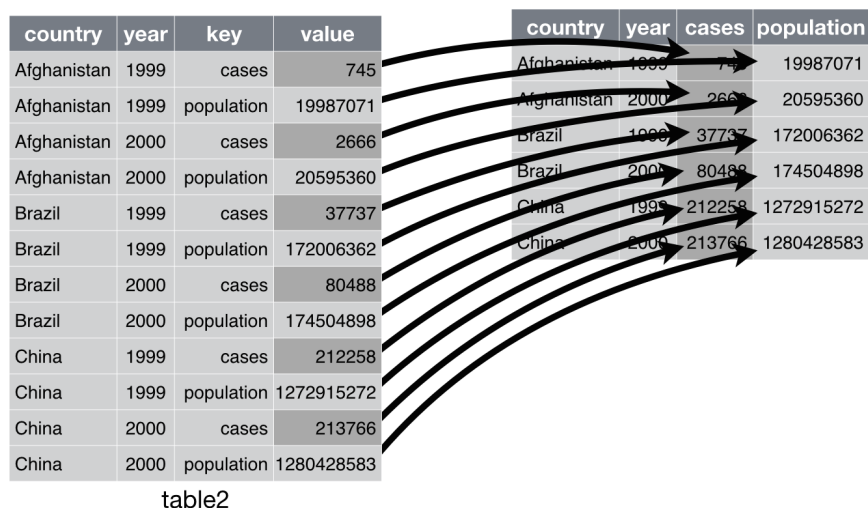


Figure 10.3: Spreading ‘table2’ makes it tidy

```
#> 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.


10.4 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.

10.4.1 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
Afghanistan	1999	745 / 19987071
Afghanistan	2000	2666 / 20595360
Brazil	1999	37737 / 172006362
Brazil	2000	80488 / 174504898
China	1999	212258 / 1272915272
China	2000	213766 / 1280428583

table3

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

Figure 10.4: 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 ?? 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>
```

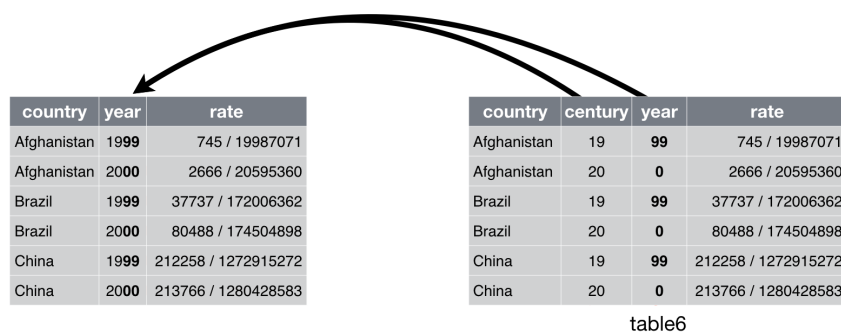


Figure 10.5: Uniting ‘table5’ makes it tidy

```
#> 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
```

10.4.2 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
```

10.5 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 NAs 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 was 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 complete 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
```

10.6 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/>