Modern R in a Corporate Environment

R course developed for the office

Brian Davis

2018-06-02

Contents

Welcome

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

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

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

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

Part I

Preamble

Chapter 1

Introduction

1.1 Course Philosophy

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

1.1.1 Reproducible Research Approach

What is Reproducible Research About?

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

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

> Conclusions in experimental psychology often are the result of null hypothesis significance testing. Unfortunately, there is evidence ((from eight major psychology journals published between 1985 and 2013) that roughly half of all published empirical psychology articles contain at least one inconsistent p-value, and around one in eight articles contain a grossly inconsistent p-value that makes a non-significant result seem significant, or vice versa. statscheck and here
>
> "A key component of scientific communication is sufficient information for other researchers in the field to reproduce published findings. For computational and data-enabled research, this has often been interpreted to mean making available the raw data from which results were generated, the computer code that generated the findings, and any additional information needed such as workflows and input parameters. Many journals are revising author guidelines to include data and code availability. We chose a random sample of 204 scientific papers published in the journal **Science** after the implementation of their policy in February 2011. We found that were able to reproduce the findings for 26%." Proceedings of the National Academy of Sciences of the United States of America

> "Starting September 1 2016, JASA ACS will require code and data as a minimum standard for reproducibility of statistical scientific research." JASA

1.1.2 FDA Validation

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

1.1.3 The SAS Myth

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

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

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

1.2 Prerequisites

- We will assume you have minimal experience and knowledge of R
- IT should have installed:
  - R version 3.4.4
  - 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 an we will work to adjust it.

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

1. Good programming practices
2. Basics of R Programming
3. Importing / Exporting Data
4. Tidying Data
5. Visualizing Data
6. Functions

7. Strings
8. Dates and Time
9. Communicating Results

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, butitsuremakesthingseasiertoread. When I answer questions; first, I see if think I can answer the question, secondly, I check the coding style of the question and if the code is too difficult to read, I just move on. Please make your code readable by following e.g. this coding style (most examples below come from this guide).

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

2.1.1 Comments

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

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

2.1.2 Naming

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

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

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

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

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

2.1.3 Spacing

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

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

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

2.1.4 Indenting

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

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

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

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

```
message("Y is negative")

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

2.1.5 Long lines

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

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

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

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

2.1.6 Other

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

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

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

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

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

```
  x + y
}

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

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

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

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

2.2 Coding practices

2.2.1 Variables

Create variables for values that are likely to change.

2.2.2 *Rule of Three*[1]

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

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

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

2.2.3 Path names

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

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

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

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

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

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

2.3 RStudio

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

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

RStudio features:

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

The only two things that make @JennyBryan . Instead use projects + here::here() #rstats pic.twitter.com/GwxnHePL4n

— Hadley Wickham (@hadleywickham) December 11 2017

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

2.4 Getting help

2.4.1 Help yourself, learn how to debug

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

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

2.4.2 External help

Can't remember useful functions? Use cheat sheets.

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

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

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

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

2.5 Keeping up to date

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

2.6 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 constructs[1]; objects, functions, and environments.

3.1 Assignment Operators

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

3.2 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
```

[1] Technically 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 Comparision

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

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

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

```
print(y, digits = 20)
#> [1] 2.0000000000000004441
all.equal(y, 2)          ## equality with some tolerance
#> [1] TRUE
all.equal(y, 3)
#> [1] "Mean relative difference: 0.5"
isTRUE(all.equal(y, 3))  ## if you want a boolean, use isTRUE()
#> [1] FALSE
```

3.4 Logical and sets

```
x <- c(TRUE, FALSE)
df <- data.frame(expand.grid(x, x))
names(df) <- c("x", "y")
df$and  <- df$x & df$y     # logical and
df$or   <- df$x | df$y     # logical or
df$notx <- !df$x           # negation
df$xor  <- xor(df$x, df$y) # exlusive or
df
#>       x     y   and    or  notx   xor
#> 1  TRUE  TRUE  TRUE  TRUE FALSE FALSE
#> 2 FALSE  TRUE FALSE  TRUE  TRUE  TRUE
#> 3  TRUE FALSE FALSE  TRUE FALSE  TRUE
#> 4 FALSE FALSE FALSE FALSE  TRUE FALSE
```

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

```
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 `pi` is through Monte-Carlo simulation.

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

Monte-Carlo pseudo code:

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

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

  4*hits/N
}
N <- 1e6
system.time(pi_naive(N))
#>    user  system elapsed
#>   3.264   0.018   3.283
```

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
#>   0.192   0.011   0.202
```

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 an 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: 0x7fd8f21e8478>
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 be be achieved using vectorization.

Here is a formula for the Gregory-Leibniz series:

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

Here is the Gregory-Leibniz series in summation notation:

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

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

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

N <- 1e7
system.time(pi_est <- GL_naive(N))
#>    user  system elapsed
#>   0.860   0.002   0.864
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 letters[1], 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.

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

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 `NA`s 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 list[3] (with unique names). Attributes can be accessed individually with `attr()`

[3]The 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 couple[4] ways:

[4]There 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 there 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 want[5].

```
(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
```

[5] We'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:

```
dfl <- data.frame(x = 1:3, y = I(list(1:2, 1:3, 1:4)))
str(dfl)
#> '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.

```
dfl[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 it 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
```

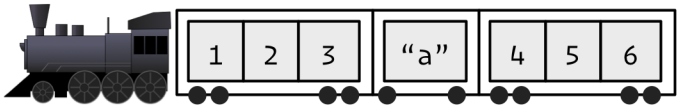
5.2 Selecting a single elements

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:

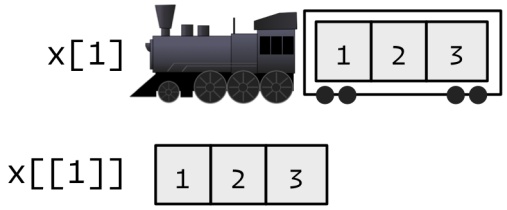
> "If list `x` is a train carrying objects, then `x[[5]]` is the object in car 5; `x[4:6]` is a train of cars 4-6."
>
> — @RLangTip, https://twitter.com/RLangTip/status/268375867468681216

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

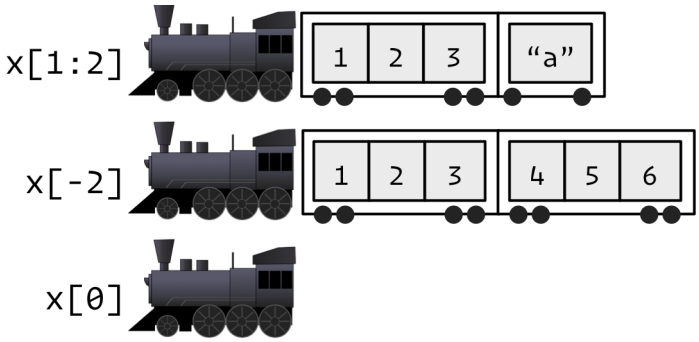
```
x <- list(1:3, "a", 4:6)
```



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 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 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 happen 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	`logical(0)`	`NA`	`NA`
List	`list()`	`list(NULL)`	`list(NULL)`
`NULL`	`NULL`	`NULL`	`NULL`

```
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	`NULL`	`NULL`
`NULL`	`NULL`	`NULL`	`NULL`	`NULL`

```
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 tideverse 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 it's 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 it 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 out reports to next class. The goal is to have each of us work an 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, prevalance 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's faceted into one-plot-per-drug group, with a drug group described by its market name.

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

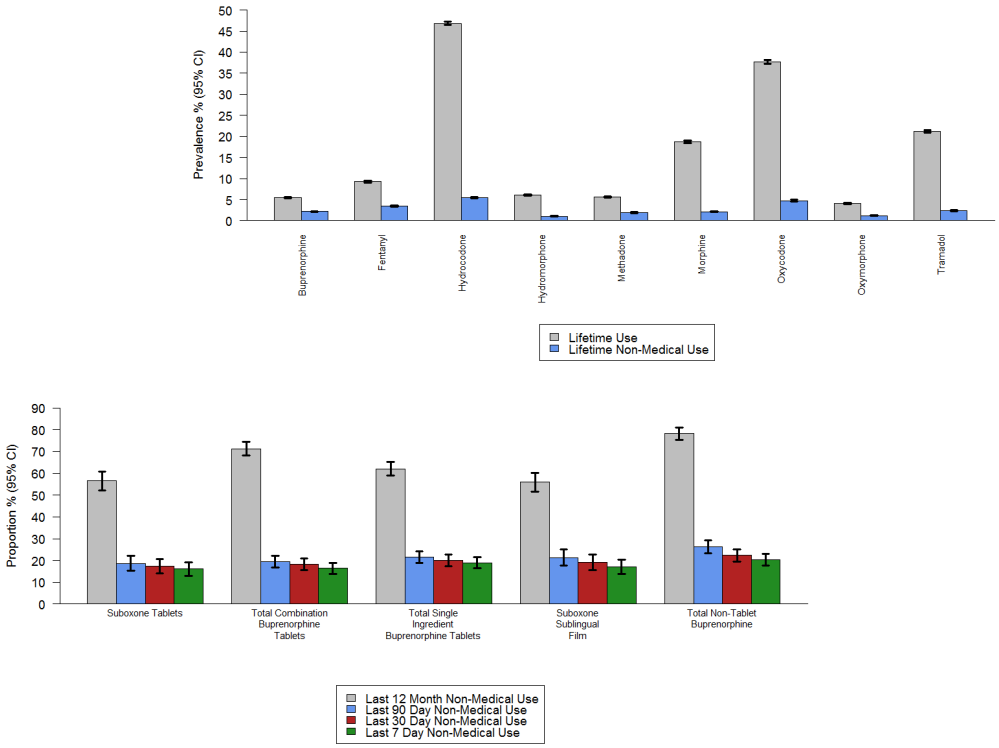
`ggplot2` is a huge package: philosophy + functions …but it's very well organized

Going to throw a lot at you …but you'll know where and what to look for

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

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

6.2 Example



What is similar / difrerent 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 varible forms a **column**
2. Each observation forms a **row**
3. Each observational unit forms a table

```
library(tidyverse)
#> -- Attaching packages --------------------- tidyverse 1.2.1 --
#> ✓ ggplot2 2.2.1     ✓ purrr   0.2.4
#> ✓ tibble  1.4.2     ✓ dplyr   0.7.4
#> ✓ tidyr   0.8.0     ✓ stringr 1.3.0
#> ✓ readr   1.1.1     ✓ 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>
```