CHAPTER 11

Strings with stringr

Introduction

This chapter introduces you to string manipulation in R. You'll learn the basics of how strings work and how to create them by hand, but the focus of this chapter will be on regular expressions, or *regexps* for short. Regular expressions are useful because strings usually contain unstructured or semi-structured data, and regexps are a concise language for describing patterns in strings. When you first look at a regexp, you'll think a cat walked across your keyboard, but as your understanding improves they will soon start to make sense.

Prerequisites

This chapter will focus on the **stringr** package for string manipulation. **stringr** is not part of the core tidyverse because you don't always have textual data, so we need to load it explicitly.

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

String Basics

You can create strings with either single quotes or double quotes. Unlike other languages, there is no difference in behavior. I recommend always using ", unless you want to create a string that contains multiple ":

```
string1 <- "This is a string"
string2 <- 'To put a "quote" inside a string, use single quotes'
```

If you forget to close a quote, you'll see +, the continuation character:

```
> "This is a string without a closing quote
+
+
+ HELP I'M STUCK
```

If this happens to you, press Esc and try again!

To include a literal single or double quote in a string you can use \ to "escape" it:

```
double_quote <- "\"" # or '"'
single_quote <- '\'' # or "'"
```

That means if you want to include a literal backslash, you'll need to double it up: "\\".

Beware that the printed representation of a string is not the same as string itself, because the printed representation shows the escapes. To see the raw contents of the string, use `writeLines()`:

```
x <- c("\"", "\\")
x
#> [1] "\"" "\\"
writeLines(x)
#> "
#> \
```

There are a handful of other special characters. The most common are "\n", newline, and "\t", tab, but you can see the complete list by requesting help on `?'"'`, or `?"'"`. You'll also sometimes see strings like "\u00b5", which is a way of writing non-English characters that works on all platforms:

```
x <- "\u00b5"
x
#> [1] "µ"
```

Multiple strings are often stored in a character vector, which you can create with `c()`:

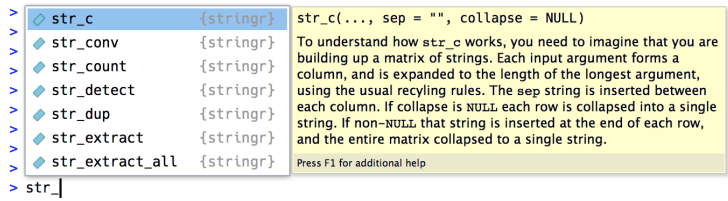
```
c("one", "two", "three")
#> [1] "one"   "two"   "three"
```

String Length

Base R contains many functions to work with strings but we'll avoid them because they can be inconsistent, which makes them hard to remember. Instead we'll use functions from **stringr**. These have more intuitive names, and all start with `str_`. For example, `str_length()` tells you the number of characters in a string:

```
str_length(c("a", "R for data science", NA))
#> [1]  1 18 NA
```

The common `str_` prefix is particularly useful if you use RStudio, because typing `str_` will trigger autocomplete, allowing you to see all **stringr** functions:



Combining Strings

To combine two or more strings, use `str_c()`:

```
str_c("x", "y")
#> [1] "xy"
str_c("x", "y", "z")
#> [1] "xyz"
```

Use the `sep` argument to control how they're separated:

```
str_c("x", "y", sep = ", ")
#> [1] "x, y"
```

Like most other functions in R, missing values are contagious. If you want them to print as `"NA"`, use `str_replace_na()`:

```
x <- c("abc", NA)
str_c("|-", x, "-|")
#> [1] "|-abc-|" NA
str_c("|-", str_replace_na(x), "-|")
#> [1] "|-abc-|" "|-NA-|"
```

As shown in the preceding code, `str_c()` is vectorized, and it automatically recycles shorter vectors to the same length as the longest:

```
str_c("prefix-", c("a", "b", "c"), "-suffix")
#> [1] "prefix-a-suffix" "prefix-b-suffix" "prefix-c-suffix"
```

Objects of length 0 are silently dropped. This is particularly useful in conjunction with `if`:

```
name <- "Hadley"
time_of_day <- "morning"
birthday <- FALSE

str_c(
  "Good ", time_of_day, " ", name,
  if (birthday) " and HAPPY BIRTHDAY",
  "."
)
#> [1] "Good morning Hadley."
```

To collapse a vector of strings into a single string, use `collapse`:

```
str_c(c("x", "y", "z"), collapse = ", ")
#> [1] "x, y, z"
```

Subsetting Strings

You can extract parts of a string using `str_sub()`. As well as the string, `str_sub()` takes `start` and `end` arguments that give the (inclusive) position of the substring:

```
x <- c("Apple", "Banana", "Pear")
str_sub(x, 1, 3)
#> [1] "App" "Ban" "Pea"

# negative numbers count backwards from end
str_sub(x, -3, -1)
#> [1] "ple" "ana" "ear"
```

Note that `str_sub()` won't fail if the string is too short; it will just return as much as possible:

```
str_sub("a", 1, 5)
#> [1] "a"
```

You can also use the assignment form of `str_sub()` to modify strings:

```
str_sub(x, 1, 1) <- str_to_lower(str_sub(x, 1, 1))
x
#> [1] "apple"  "banana" "pear"
```

Locales

Earlier I used `str_to_lower()` to change the text to lowercase. You can also use `str_to_upper()` or `str_to_title()`. However, changing case is more complicated than it might at first appear because different languages have different rules for changing case. You can pick which set of rules to use by specifying a locale:

```
# Turkish has two i's: with and without a dot, and it
# has a different rule for capitalizing them:
str_to_upper(c("i", "ı"))
#> [1] "I" "I"
str_to_upper(c("i", "ı"), locale = "tr")
#> [1] "İ" "I"
```

The locale is specified as an ISO 639 language code, which is a two- or three-letter abbreviation. If you don't already know the code for your language, Wikipedia has a good list. If you leave the locale blank, it will use the current locale, as provided by your operating system.

Another important operation that's affected by the locale is sorting. The base R `order()` and `sort()` functions sort strings using the current locale. If you want robust behavior across different computers, you may want to use `str_sort()` and `str_order()`, which take an additional `locale` argument:

```
x <- c("apple", "eggplant", "banana")

str_sort(x, locale = "en")  # English
#> [1] "apple"    "banana"   "eggplant"

str_sort(x, locale = "haw") # Hawaiian
#> [1] "apple"    "eggplant" "banana"
```

Exercises

1. In code that doesn't use **stringr**, you'll often see `paste()` and `paste0()`. What's the difference between the two functions? What **stringr** function are they equivalent to? How do the functions differ in their handling of `NA`?
2. In your own words, describe the difference between the `sep` and `collapse` arguments to `str_c()`.

3. Use `str_length()` and `str_sub()` to extract the middle character from a string. What will you do if the string has an even number of characters?
4. What does `str_wrap()` do? When might you want to use it?
5. What does `str_trim()` do? What's the opposite of `str_trim()`?
6. Write a function that turns (e.g.) a vector `c("a", "b", "c")` into the string `a, b, and c`. Think carefully about what it should do if given a vector of length 0, 1, or 2.

Matching Patterns with Regular Expressions

Regexps are a very terse language that allow you to describe patterns in strings. They take a little while to get your head around, but once you understand them, you'll find them extremely useful.

To learn regular expressions, we'll use `str_view()` and `str_view_all()`. These functions take a character vector and a regular expression, and show you how they match. We'll start with very simple regular expressions and then gradually get more and more complicated. Once you've mastered pattern matching, you'll learn how to apply those ideas with various **stringr** functions.

Basic Matches

The simplest patterns match exact strings:

```
x <- c("apple", "banana", "pear")
str_view(x, "an")
```

```
apple
banana
pear
```

The next step up in complexity is `.`, which matches any character (except a newline):

```
str_view(x, ".a.")
```

```
apple
banana
pear
```

But if "." matches any character, how do you match the character "."? You need to use an "escape" to tell the regular expression you want to match it exactly, not use its special behavior. Like strings, regexps use the backslash, \, to escape special behavior. So to match an ., you need the regexp \.. Unfortunately this creates a problem. We use strings to represent regular expressions, and \ is also used as an escape symbol in strings. So to create the regular expression \. we need the string "\\.":

```
# To create the regular expression, we need \\
dot <- "\\."

# But the expression itself only contains one:
writeLines(dot)
#> \.

# And this tells R to look for an explicit .
str_view(c("abc", "a.c", "bef"), "a\\.c")
```

```
abc
a.c
bef
```

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

```
x <- "a\\b"
writeLines(x)
#> a\b

str_view(x, "\\\\")
```

```
a\b
```

In this book, I'll write regular expressions as \. and strings that represent the regular expression as "\\.".

Exercises

1. Explain why each of these strings don't match a \: "\", "\\", "\\\".

2. How would you match the sequence `"'\`?
3. What patterns will the regular expression `\..\..\..` match? How would you represent it as a string?

Anchors

By default, regular expressions will match any part of a string. It's often useful to *anchor* the regular expression so that it matches from the start or end of the string. You can use:

- `^` to match the start of the string.
- `$` to match the end of the string.

```
x <- c("apple", "banana", "pear")
str_view(x, "^a")
```

```
apple
banana
pear
```

```
str_view(x, "a$")
```

```
apple
banana
pear
```

To remember which is which, try this mnemonic that I learned from Evan Misshula: if you begin with power (^), you end up with money ($).

To force a regular expression to only match a complete string, anchor it with both `^` and `$`:

```
x <- c("apple pie", "apple", "apple cake")
str_view(x, "apple")
```

```
apple pie
apple
apple cake
```

```
str_view(x, "^apple$")
```

```
apple pie
apple
apple cake
```

You can also match the boundary between words with `\b`. I don't often use this in R, but I will sometimes use it when I'm doing a search in RStudio when I want to find the name of a function that's a component of other functions. For example, I'll search for `\bsum\b` to avoid matching `summarize`, `summary`, `rowsum`, and so on.

Exercises

1. How would you match the literal string `"$^$"`?
2. Given the corpus of common words in `stringr::words`, create regular expressions that find all words that:
   a. Start with "y".
   b. End with "x".
   c. Are exactly three letters long. (Don't cheat by using `str_length()`!)
   d. Have seven letters or more.

   Since this list is long, you might want to use the `match` argument to `str_view()` to show only the matching or non-matching words.

Character Classes and Alternatives

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

- `\d` matches any digit.
- `\s` matches any whitespace (e.g., space, tab, newline).
- `[abc]` matches a, b, or c.
- `[^abc]` matches anything except a, b, or c.

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

You can use *alternation* to pick between one or more alternative patterns. For example, `abc|d..f` will match either `"abc"`, or `"deaf"`. Note that the precedence for `|` is low, so that `abc|xyz` matches `abc` or `xyz` not `abcyz` or `abxyz`. Like with mathematical expressions, if precedence ever gets confusing, use parentheses to make it clear what you want:

```
str_view(c("grey", "gray"), "gr(e|a)y")
```

Exercises

1. Create regular expressions to find all words that:
   a. Start with a vowel.
   b. Only contain consonants. (Hint: think about matching "not"-vowels.)
   c. End with `ed`, but not with `eed`.
   d. End with `ing` or `ize`.
2. Empirically verify the rule "i before e except after c."
3. Is "q" always followed by a "u"?
4. Write a regular expression that matches a word if it's probably written in British English, not American English.
5. Create a regular expression that will match telephone numbers as commonly written in your country.

Repetition

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

- `?`: 0 or 1
- `+`: 1 or more
- `*`: 0 or more

```
x <- "1888 is the longest year in Roman numerals: MDCCCLXXXVIII"
str_view(x, "CC?")
```

```
  1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

```
str_view(x, "CC+")
```

```
  1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

```
str_view(x, 'C[LX]+')
```

```
  1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

Note that the precedence of these operators is high, so you can write `colou?r` to match either American or British spellings. That means most uses will need parentheses, like `bana(na)+`.

You can also specify the number of matches precisely:

- `{n}`: exactly n
- `{n,}`: n or more
- `{,m}`: at most m
- `{n,m}`: between n and m

```
str_view(x, "C{2}")
```

```
  1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

```
str_view(x, "C{2,}")
```

```
  1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

```
str_view(x, "C{2,3}")
```

```
  1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

By default these matches are "greedy": they will match the longest string possible. You can make them "lazy," matching the shortest string possible, by putting a `?` after them. This is an advanced feature of regular expressions, but it's useful to know that it exists:

```
str_view(x, 'C{2,3}?')
```

```
  1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

```
str_view(x, 'C[LX]+?')
```

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

Exercises

1. Describe the equivalents of `?`, `+`, and `*` in `{m,n}` form.
2. Describe in words what these regular expressions match (read carefully to see if I'm using a regular expression or a string that defines a regular expression):
   a. `^.*$`
   b. `"\\{.+\\}"`
   c. `\d{4}-\d{2}-\d{2}`
   d. `"\\\\{4}"`
3. Create regular expressions to find all words that:
   a. Start with three consonants.
   b. Have three or more vowels in a row.
   c. Have two or more vowel-consonant pairs in a row.
4. Solve the beginner regexp crosswords at *https://regexcrossword.com/challenges/beginner*.

Grouping and Backreferences

Earlier, you learned about parentheses as a way to disambiguate complex expressions. They also define "groups" that you can refer to with *backreferences*, like `\1`, `\2`, etc. For example, the following regular expression finds all fruits that have a repeated pair of letters:

```
str_view(fruit, "(..)\\1", match = TRUE)
```

```
banana
coconut
cucumber
jujube
papaya
salal berry
```

(Shortly, you'll also see how they're useful in conjunction with `str_match()`.)

Exercises

1. Describe, in words, what these expressions will match:
   a. `(.)\1\1`
   b. `"(.)(.)\\2\\1"`
   c. `(..)\1`
   d. `"(.).\\1.\\1"`
   e. `"(.)(.)(.).*\\3\\2\\1"`
2. Construct regular expressions to match words that:
   a. Start and end with the same character.
   b. Contain a repeated pair of letters (e.g., "church" contains "ch" repeated twice).
   c. Contain one letter repeated in at least three places (e.g., "eleven" contains three "e"s).

Tools

Now that you've learned the basics of regular expressions, it's time to learn how to apply them to real problems. In this section you'll learn a wide array of **stringr** functions that let you:

- Determine which strings match a pattern.
- Find the positions of matches.
- Extract the content of matches.
- Replace matches with new values.
- Split a string based on a match.

A word of caution before we continue: because regular expressions are so powerful, it's easy to try and solve every problem with a single regular expression. In the words of Jamie Zawinski:

> Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems.

As a cautionary tale, check out this regular expression that checks if an email address is valid:

This is a somewhat pathological example (because email addresses are actually suprisingly complex), but is used in real code. See the stackoverflow discussion for more details.

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

Detect Matches

To determine if a character vector matches a pattern, use `str_detect()`. It returns a logical vector the same length as the input:

```
x <- c("apple", "banana", "pear")
str_detect(x, "e")
#> [1]  TRUE FALSE  TRUE
```

Remember that when you use a logical vector in a numeric context, `FALSE` becomes 0 and `TRUE` becomes 1. That makes `sum()` and `mean()` useful if you want to answer questions about matches across a larger vector:

```
# How many common words start with t?
sum(str_detect(words, "^t"))
#> [1] 65
# What proportion of common words end with a vowel?
mean(str_detect(words, "[aeiou]$"))
#> [1] 0.277
```

When you have complex logical conditions (e.g., match a or b but not c unless d) it's often easier to combine multiple `str_detect()` calls with logical operators, rather than trying to create a single regular expression. For example, here are two ways to find all words that don't contain any vowels:

```
# Find all words containing at least one vowel, and negate
no_vowels_1 <- !str_detect(words, "[aeiou]")
# Find all words consisting only of consonants (non-vowels)
no_vowels_2 <- str_detect(words, "^[^aeiou]+$")
identical(no_vowels_1, no_vowels_2)
#> [1] TRUE
```

The results are identical, but I think the first approach is significantly easier to understand. If your regular expression gets overly complicated, try breaking it up into smaller pieces, giving each piece a name, and then combining the pieces with logical operations.

A common use of `str_detect()` is to select the elements that match a pattern. You can do this with logical subsetting, or the convenient `str_subset()` wrapper:

```
words[str_detect(words, "x$")]
#> [1] "box" "sex" "six" "tax"
str_subset(words, "x$")
#> [1] "box" "sex" "six" "tax"
```

Typically, however, your strings will be one column of a data frame, and you'll want to use `filter` instead:

```
df <- tibble(
  word = words,
  i = seq_along(word)
)
df %>%
  filter(str_detect(words, "x$"))
#> # A tibble: 4 × 2
#>    word     i
#>   <chr> <int>
#> 1   box   108
#> 2   sex   747
#> 3   six   772
#> 4   tax   841
```

A variation on `str_detect()` is `str_count()`: rather than a simple yes or no, it tells you how many matches there are in a string:

```
x <- c("apple", "banana", "pear")
str_count(x, "a")
#> [1] 1 3 1

# On average, how many vowels per word?
mean(str_count(words, "[aeiou]"))
#> [1] 1.99
```

It's natural to use `str_count()` with `mutate()`:

```
df %>%
  mutate(
    vowels = str_count(word, "[aeiou]"),
    consonants = str_count(word, "[^aeiou]")
  )
#> # A tibble: 980 × 4
#>       word     i vowels consonants
#>      <chr> <int>  <int>      <int>
#> 1        a     1      1          0
#> 2     able     2      2          2
#> 3    about     3      3          2
#> 4 absolute     4      4          4
#> 5   accept     5      2          4
#> 6  account     6      3          4
#> # ... with 974 more rows
```

`ababaaba`

Note that matches never overlap. For example, in `"abababa"`, how many times will the pattern `"aba"` match? Regular expressions say two, not three:

```
str_count("abababa", "aba")
#> [1] 2
str_view_all("abababa", "aba")
```

Note the use of `str_view_all()`. As you'll shortly learn, many **stringr** functions come in pairs: one function works with a single match, and the other works with all matches. The second function will have the suffix `_all`.

Exercises

1. For each of the following challenges, try solving it by using both a single regular expression, and a combination of multiple `str_detect()` calls:
   a. Find all words that start or end with `x`.
   b. Find all words that start with a vowel and end with a consonant.
   c. Are there any words that contain at least one of each different vowel?
   d. What word has the highest number of vowels? What word has the highest proportion of vowels? (Hint: what is the denominator?)

Extract Matches

To extract the actual text of a match, use `str_extract()`. To show that off, we're going to need a more complicated example. I'm going to use the Harvard sentences, which were designed to test VOIP systems, but are also useful for practicing regexes. These are provided in `stringr::sentences`:

```
length(sentences)
#> [1] 720
head(sentences)
#> [1] "The birch canoe slid on the smooth planks."
#> [2] "Glue the sheet to the dark blue background."
```

```
#> [3] "It's easy to tell the depth of a well."
#> [4] "These days a chicken leg is a rare dish."
#> [5] "Rice is often served in round bowls."
#> [6] "The juice of lemons makes fine punch."
```

Imagine we want to find all sentences that contain a color. We first create a vector of color names, and then turn it into a single regular expression:

```
colors <- c(
  "red", "orange", "yellow", "green", "blue", "purple"
)
color_match <- str_c(colors, collapse = "|")
color_match
#> [1] "red|orange|yellow|green|blue|purple"
```

Now we can select the sentences that contain a color, and then extract the color to figure out which one it is:

```
has_color <- str_subset(sentences, color_match)
matches <- str_extract(has_color, color_match)
head(matches)
#> [1] "blue" "blue" "red"  "red"  "red"  "blue"
```

Note that `str_extract()` only extracts the first match. We can see that most easily by first selecting all the sentences that have more than one match:

```
more <- sentences[str_count(sentences, color_match) > 1]
str_view_all(more, color_match)
```

```
It is hard to erase blue or red ink.
The green light in the brown box flickered.
The sky in the west is tinged with orange red.
```

```
str_extract(more, color_match)
#> [1] "blue"   "green"  "orange"
```

```
It is hard to erase blue or red ink.
The green light in the brown box flickered.
The sky in the west is tinged with orange red.
```

This is a common pattern for **stringr** functions, because working with a single match allows you to use much simpler data structures. To get all matches, use `str_extract_all()`. It returns a list:

```
str_extract_all(more, color_match)
#> [[1]]
#> [1] "blue" "red"
```

```
#>
#> [[2]]
#> [1] "green" "red"
#>
#> [[3]]
#> [1] "orange" "red"
```

You'll learn more about lists in "Recursive Vectors (Lists)" on page 302 and Chapter 17.

If you use `simplify = TRUE`, `str_extract_all()` will return a matrix with short matches expanded to the same length as the longest:

```
str_extract_all(more, color_match, simplify = TRUE)
#>      [,1]     [,2]
#> [1,] "blue"   "red"
#> [2,] "green"  "red"
#> [3,] "orange" "red"

x <- c("a", "a b", "a b c")
str_extract_all(x, "[a-z]", simplify = TRUE)
#>      [,1] [,2] [,3]
#> [1,] "a"  ""   ""
#> [2,] "a"  "b"  ""
#> [3,] "a"  "b"  "c"
```

Exercises

1. In the previous example, you might have noticed that the regular expression matched "flickered," which is not a color. Modify the regex to fix the problem.
2. From the Harvard sentences data, extract:
   a. The first word from each sentence.
   b. All words ending in `ing`.
   c. All plurals.

Grouped Matches

Earlier in this chapter we talked about the use of parentheses for clarifying precedence and for backreferences when matching. You can also use parentheses to extract parts of a complex match. For example, imagine we want to extract nouns from the sentences. As a heuristic, we'll look for any word that comes after "a" or "the". Defining a "word" in a regular expression is a little tricky, so here I use a

simple approximation—a sequence of at least one character that isn't a space:

```
noun <- "(a|the) ([^ ]+)"

has_noun <- sentences %>%
  str_subset(noun) %>%
  head(10)
has_noun %>%
  str_extract(noun)
#>  [1] "the smooth" "the sheet"  "the depth"  "a chicken"
#>  [5] "the parked" "the sun"    "the huge"   "the ball"
#>  [9] "the woman"  "a helps"
```

`str_extract()` gives us the complete match; `str_match()` gives each individual component. Instead of a character vector, it returns a matrix, with one column for the complete match followed by one column for each group:

```
has_noun %>%
  str_match(noun)
#>       [,1]         [,2]  [,3]
#>  [1,] "the smooth" "the" "smooth"
#>  [2,] "the sheet"  "the" "sheet"
#>  [3,] "the depth"  "the" "depth"
#>  [4,] "a chicken"  "a"   "chicken"
#>  [5,] "the parked" "the" "parked"
#>  [6,] "the sun"    "the" "sun"
#>  [7,] "the huge"   "the" "huge"
#>  [8,] "the ball"   "the" "ball"
#>  [9,] "the woman"  "the" "woman"
#> [10,] "a helps"    "a"   "helps"
```

(Unsurprisingly, our heuristic for detecting nouns is poor, and also picks up adjectives like smooth and parked.)

If your data is in a tibble, it's often easier to use `tidyr::extract()`. It works like `str_match()` but requires you to name the matches, which are then placed in new columns:

```
tibble(sentence = sentences) %>%
  tidyr::extract(
    sentence, c("article", "noun"), "(a|the) ([^ ]+)",
    remove = FALSE
  )
#> # A tibble: 720 × 3
#>                                   sentence article    noun
#> *                                    <chr>   <chr>   <chr>
#> 1  The birch canoe slid on the smooth planks.     the  smooth
#> 2 Glue the sheet to the dark blue background.     the   sheet
#> 3      It's easy to tell the depth of a well.     the   depth
```

```
#> 4    These days a chicken leg is a rare dish.       a chicken
#> 5        Rice is often served in round bowls.    <NA>    <NA>
#> 6       The juice of lemons makes fine punch.    <NA>    <NA>
#> # ... with 714 more rows
```

Like `str_extract()`, if you want all matches for each string, you'll need `str_match_all()`.

Exercises

1. Find all words that come after a "number" like "one", "two", "three", etc. Pull out both the number and the word.
2. Find all contractions. Separate out the pieces before and after the apostrophe.

Replacing Matches

`str_replace()` and `str_replace_all()` allow you to replace matches with new strings. The simplest use is to replace a pattern with a fixed string:

```
x <- c("apple", "pear", "banana")
str_replace(x, "[aeiou]", "-")
#> [1] "-pple"  "p-ar"   "b-nana"
str_replace_all(x, "[aeiou]", "-")
#> [1] "-ppl-"  "p--r"   "b-n-n-"
```

With `str_replace_all()` you can perform multiple replacements by supplying a named vector:

```
x <- c("1 house", "2 cars", "3 people")
str_replace_all(x, c("1" = "one", "2" = "two", "3" = "three"))
#> [1] "one house"    "two cars"     "three people"
```

Instead of replacing with a fixed string you can use backreferences to insert components of the match. In the following code, I flip the order of the second and third words:

```
sentences %>%
  str_replace("([^ ]+) ([^ ]+) ([^ ]+)", "\\1 \\3 \\2") %>%
  head(5)
#> [1] "The canoe birch slid on the smooth planks."
#> [2] "Glue sheet the to the dark blue background."
#> [3] "It's to easy tell the depth of a well."
#> [4] "These a days chicken leg is a rare dish."
#> [5] "Rice often is served in round bowls."
```

Exercises

1. Replace all forward slashes in a string with backslashes.
2. Implement a simple version of `str_to_lower()` using `replace_all()`.
3. Switch the first and last letters in `words`. Which of those strings are still words?

Splitting

Use `str_split()` to split a string up into pieces. For example, we could split sentences into words:

```
sentences %>%
  head(5) %>%
  str_split(" ")
#> [[1]]
#> [1] "The"    "birch"   "canoe"   "slid"    "on"      "the"
#> [7] "smooth"  "planks."
#>
#> [[2]]
#> [1] "Glue"        "the"         "sheet"       "to"
#> [5] "the"  "dark"        "blue"        "background."
#>
#> [[3]]
#> [1] "It's"  "easy"  "to"    "tell"  "the"   "depth" "of"
#> [8] "a"   "well."
#>
#> [[4]]
#> [1] "These"   "days"    "a"       "chicken" "leg"     "is"
#> [7] "a" "rare"    "dish."
#>
#> [[5]]
#> [1] "Rice"   "is"     "often"  "served" "in"     "round"
#> [7] "bowls."
```

Because each component might contain a different number of pieces, this returns a list. If you're working with a length-1 vector, the easiest thing is to just extract the first element of the list:

```
"a|b|c|d" %>%
  str_split("\\|") %>%
  .[[1]]
#> [1] "a" "b" "c" "d"
```

Otherwise, like the other **stringr** functions that return a list, you can use `simplify = TRUE` to return a matrix:

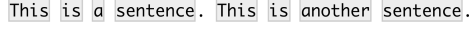
```
sentences %>%
  head(5) %>%
  str_split(" ", simplify = TRUE)
#>      [,1]    [,2]    [,3]    [,4]      [,5]  [,6]    [,7]
#> [1,] "The"   "birch" "canoe" "slid"    "on"  "the"   "smooth"
#> [2,] "Glue"  "the"   "sheet" "to"      "the" "dark"  "blue"
#> [3,] "It's"  "easy"  "to"    "tell"    "the" "depth" "of"
#> [4,] "These" "days"  "a"     "chicken" "leg" "is"    "a"
#> [5,] "Rice"  "is"    "often" "served"  "in"  "round" "bowls."
#>      [,8]          [,9]
#> [1,] "planks."     ""
#> [2,] "background." ""
#> [3,] "a"           "well."
#> [4,] "rare"        "dish."
#> [5,] ""            ""
```

You can also request a maximum number of pieces:

```
fields <- c("Name: Hadley", "Country: NZ", "Age: 35")
fields %>% str_split(": ", n = 2, simplify = TRUE)
#>      [,1]      [,2]
#> [1,] "Name"    "Hadley"
#> [2,] "Country" "NZ"
#> [3,] "Age"     "35"
```

Instead of splitting up strings by patterns, you can also split up by character, line, sentence, and word `boundary()`s:

```
x <- "This is a sentence.  This is another sentence."
str_view_all(x, boundary("word"))
```




```
str_split(x, " ")[[1]]
#> [1] "This"      "is"        "a"         "sentence." ""
#> [6]         "This"
#> [7] "is"        "another"   "sentence."
str_split(x, boundary("word"))[[1]]
#> [1] "This"      "is"        "a"         "sentence" "This"
#> [6] "is"
#> [7] "another"   "sentence"
```

Exercises

1. Split up a string like `"apples, pears, and bananas"` into individual components.
2. Why is it better to split up by `boundary("word")` than `" "`?

3. What does splitting with an empty string (`""`) do? Experiment, and then read the documentation.

Find Matches

`str_locate()` and `str_locate_all()` give you the starting and ending positions of each match. These are particularly useful when none of the other functions does exactly what you want. You can use `str_locate()` to find the matching pattern, and `str_sub()` to extract and/or modify them.

Other Types of Pattern

When you use a pattern that's a string, it's automatically wrapped into a call to `regex()`:

```
# The regular call:
str_view(fruit, "nana")
# Is shorthand for
str_view(fruit, regex("nana"))
```

You can use the other arguments of `regex()` to control details of the match:

- `ignore_case = TRUE` allows characters to match either their uppercase or lowercase forms. This always uses the current locale:

```
bananas <- c("banana", "Banana", "BANANA")
str_view(bananas, "banana")
```

```
banana
Banana
BANANA
```

```
str_view(bananas, regex("banana", ignore_case = TRUE))
```

- `multiline = TRUE` allows `^` and `$` to match the start and end of each line rather than the start and end of the complete string:

```
x <- "Line 1\nLine 2\nLine 3"
str_extract_all(x, "^Line")[[1]]
#> [1] "Line"
str_extract_all(x, regex("^Line", multiline = TRUE))[[1]]
#> [1] "Line" "Line" "Line"
```

- `comments = TRUE` allows you to use comments and white space to make complex regular expressions more understandable. Spaces are ignored, as is everything after `#`. To match a literal space, you'll need to escape it: `"\\ "`.

```
phone <- regex("
  \\(?     # optional opening parens
  (\\d{3}) # area code
  [)- ]?   # optional closing parens, dash, or space
  (\\d{3}) # another three numbers
  [ -]?    # optional space or dash
  (\\d{3}) # three more numbers
  ", comments = TRUE)

str_match("514-791-8141", phone)
#>      [,1]          [,2]  [,3]  [,4]
#> [1,] "514-791-814" "514" "791" "814"
```

- `dotall = TRUE` allows `.` to match everything, including `\n`.

There are three other functions you can use instead of `regex()`:

- `fixed()` matches exactly the specified sequence of bytes. It ignores all special regular expressions and operates at a very low level. This allows you to avoid complex escaping and can be much faster than regular expressions. The following microbenchmark shows that it's about 3x faster for a simple example:

```
microbenchmark::microbenchmark(
  fixed = str_detect(sentences, fixed("the")),
  regex = str_detect(sentences, "the"),
  times = 20
)
#> Unit: microseconds
#>   expr min  lq mean median  uq max neval cld
#>  fixed 116 117  136    120 125 389    20  a
#>  regex 333 337  346    338 342 467    20   b
```

  Beware using `fixed()` with non-English data. It is problematic because there are often multiple ways of representing the same character. For example, there are two ways to define "á": either as a single character or as an "a" plus an accent:

```
a1 <- "\u00e1"
a2 <- "a\u0301"
c(a1, a2)
#> [1] "á" "á"
```

```
a1 == a2
#> [1] FALSE
```

They render identically, but because they're defined differently, `fixed()` doesn't find a match. Instead, you can use `coll()`, defined next, to respect human character comparison rules:

```
str_detect(a1, fixed(a2))
#> [1] FALSE
str_detect(a1, coll(a2))
#> [1] TRUE
```

- `coll()` compares strings using standard *coll*ation rules. This is useful for doing case-insensitive matching. Note that `coll()` takes a `locale` parameter that controls which rules are used for comparing characters. Unfortunately different parts of the world use different rules!

```
# That means you also need to be aware of the difference
# when doing case-insensitive matches:
i <- c("I", "İ", "i", "ı")
i
#> [1] "I" "İ" "i" "ı"

str_subset(i, coll("i", ignore_case = TRUE))
#> [1] "I" "i"
str_subset(
  i,
  coll("i", ignore_case = TRUE, locale = "tr")
)
#> [1] "İ" "i"
```

Both `fixed()` and `regex()` have `ignore_case` arguments, but they do not allow you to pick the locale: they always use the default locale. You can see what that is with the following code (more on **stringi** later):

```
stringi::stri_locale_info()
#> $Language
#> [1] "en"
#>
#> $Country
#> [1] "US"
#>
#> $Variant
#> [1] ""
#>
#> $Name
#> [1] "en_US"
```

The downside of `coll()` is speed; because the rules for recognizing which characters are the same are complicated, `coll()` is relatively slow compared to `regex()` and `fixed()`.

- As you saw with `str_split()`, you can use `boundary()` to match boundaries. You can also use it with the other functions:

```
x <- "This is a sentence."
str_view_all(x, boundary("word"))

                    This is a sentence.


str_extract_all(x, boundary("word"))
#> [[1]]
#> [1] "This"     "is"       "a"        "sentence"
```

Exercises

1. How would you find all strings containing `\` with `regex()` versus with `fixed()`?
2. What are the five most common words in `sentences`?

Other Uses of Regular Expressions

There are two useful functions in base R that also use regular expressions:

- `apropos()` searches all objects available from the global environment. This is useful if you can't quite remember the name of the function:

```
apropos("replace")
#> [1] "%+replace%"  "replace"         "replace_na"
#> [4] "str_replace" "str_replace_all" "str_replace_na"
#> [7] "theme_replace"
```

- `dir()` lists all the files in a directory. The `pattern` argument takes a regular expression and only returns filenames that match the pattern. For example, you can find all the R Markdown files in the current directory with:

```
head(dir(pattern = "\\.Rmd$"))
#> [1] "communicate-plots.Rmd" "communicate.Rmd"
```

```
#> [3] "datetimes.Rmd" "EDA.Rmd"
#> [5] "explore.Rmd" "factors.Rmd"
```

(If you're more comfortable with "globs" like `*.Rmd`, you can convert them to regular expressions with `glob2rx()`).

stringi

stringr is built on top of the **stringi** package. **stringr** is useful when you're learning because it exposes a minimal set of functions, which have been carefully picked to handle the most common string manipulation functions. **stringi**, on the other hand, is designed to be comprehensive. It contains almost every function you might ever need: **stringi** has 234 functions to **stringr**'s 42.

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

Exercises

1. Find the **stringi** functions that:
   a. Count the number of words.
   b. Find duplicated strings.
   c. Generate random text.
2. How do you control the language that `stri_sort()` uses for sorting?

CHAPTER 12

Factors with forcats

Introduction

In R, factors are used to work with categorical variables, variables that have a fixed and known set of possible values. They are also useful when you want to display character vectors in a non-alphabetical order.

Historically, factors were much easier to work with than characters. As a result, many of the functions in base R automatically convert characters to factors. This means that factors often crop up in places where they're not actually helpful. Fortunately, you don't need to worry about that in the tidyverse, and can focus on situations where factors are genuinely useful.

For more historical context on factors, I recommend *stringsAsFactors: An unauthorized biography* by Roger Peng, and *stringsAsFactors = <sigh>* by Thomas Lumley.

Prerequisites

To work with factors, we'll use the **forcats** package, which provides tools for dealing with *cat*egorical variables (and it's an anagram of factors!). It provides a wide range of helpers for working with factors. **forcats** is not part of the core tidyverse, so we need to load it explicitly.

```
library(tidyverse)
library(forcats)
```

Creating Factors

Imagine that you have a variable that records month:

```
x1 <- c("Dec", "Apr", "Jan", "Mar")
```

Using a string to record this variable has two problems:

1. There are only twelve possible months, and there's nothing saving you from typos:

   ```
   x2 <- c("Dec", "Apr", "Jam", "Mar")
   ```

2. It doesn't sort in a useful way:

   ```
   sort(x1)
   #> [1] "Apr" "Dec" "Jan" "Mar"
   ```

You can fix both of these problems with a factor. To create a factor you must start by creating a list of the valid *levels*:

```
month_levels <- c(
  "Jan", "Feb", "Mar", "Apr", "May", "Jun",
  "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
)
```

Now you can create a factor:

```
y1 <- factor(x1, levels = month_levels)
y1
#> [1] Dec Apr Jan Mar
#> Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
sort(y1)
#> [1] Jan Mar Apr Dec
#> Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

And any values not in the set will be silently converted to NA:

```
y2 <- factor(x2, levels = month_levels)
y2
#> [1] Dec  Apr  <NA> Mar
#> Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

If you want a want an error, you can use `readr::parse_factor()`:

```
y2 <- parse_factor(x2, levels = month_levels)
#> Warning: 1 parsing failure.
#> row col           expected actual
#>   3  -- value in level set    Jam
```

If you omit the levels, they'll be taken from the data in alphabetical order:

```
factor(x1)
#> [1] Dec Apr Jan Mar
#> Levels: Apr Dec Jan Mar
```

Sometimes you'd prefer that the order of the levels match the order of the first appearance in the data. You can do that when creating the factor by setting levels to `unique(x)`, or after the fact, with `fct_inorder()`:

```
f1 <- factor(x1, levels = unique(x1))
f1
#> [1] Dec Apr Jan Mar
#> Levels: Dec Apr Jan Mar

f2 <- x1 %>% factor() %>% fct_inorder()
f2
#> [1] Dec Apr Jan Mar
#> Levels: Dec Apr Jan Mar
```

If you ever need to access the set of valid levels directly, you can do so with `levels()`:

```
levels(f2)
#> [1] "Dec" "Apr" "Jan" "Mar"
```

General Social Survey

For the rest of this chapter, we're going to focus on `forcats::gss_cat`. It's a sample of data from the General Social Survey, which is a long-running US survey conducted by the independent research organization NORC at the University of Chicago. The survey has thousands of questions, so in `gss_cat` I've selected a handful that will illustrate some common challenges you'll encounter when working with factors:

```
gss_cat
#> # A tibble: 21,483 × 9
#>    year       marital   age   race        rincome
#>   <int>        <fctr> <int> <fctr>         <fctr>
#> 1  2000 Never married    26  White  $8000 to 9999
#> 2  2000      Divorced    48  White  $8000 to 9999
#> 3  2000       Widowed    67  White Not applicable
#> 4  2000 Never married    39  White Not applicable
#> 5  2000      Divorced    25  White Not applicable
#> 6  2000       Married    25  White $20000 - 24999
#> # ... with 2.148e+04 more rows, and 4 more variables:
#> #   partyid <fctr>, relig <fctr>, denom <fctr>, tvhours <int>
```

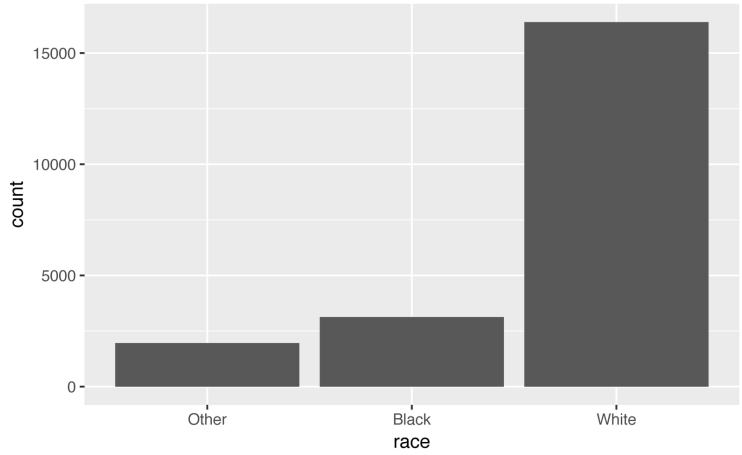
(Remember, since this dataset is provided by a package, you can get more information about the variables with `?gss_cat`.)

When factors are stored in a tibble, you can't see their levels so easily. One way to see them is with `count()`:

```
gss_cat %>%
  count(race)
#> # A tibble: 3 × 2
#>     race     n
#>   <fctr> <int>
#> 1  Other  1959
#> 2  Black  3129
#> 3  White 16395
```

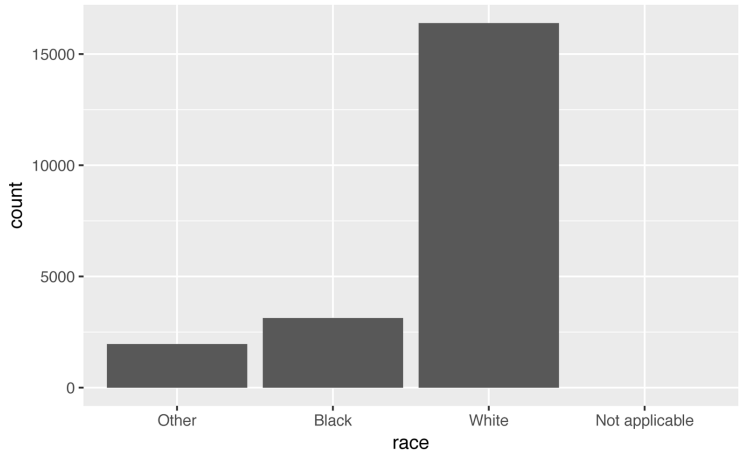
Or with a bar chart:

```
ggplot(gss_cat, aes(race)) +
  geom_bar()
```



By default, **ggplot2** will drop levels that don't have any values. You can force them to display with:

```
ggplot(gss_cat, aes(race)) +
  geom_bar() +
  scale_x_discrete(drop = FALSE)
```



These levels represent valid values that simply did not occur in this dataset. Unfortunately, **dplyr** doesn't yet have a `drop` option, but it will in the future.

When working with factors, the two most common operations are changing the order of the levels, and changing the values of the levels. Those operations are described in the following sections.

Exercises

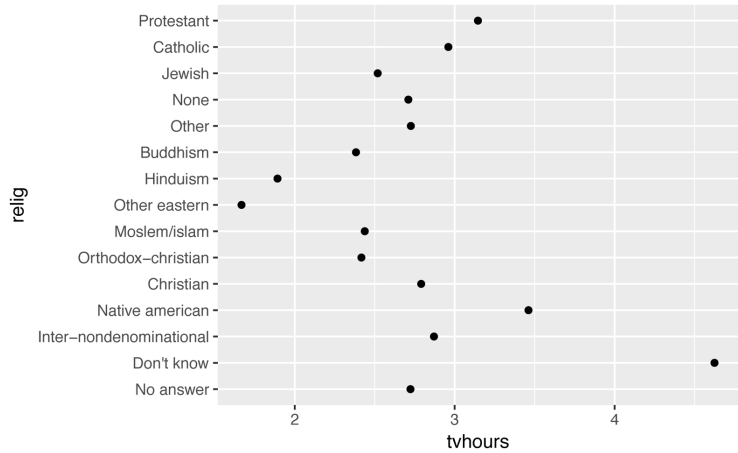
1. Explore the distribution of `rincome` (reported income). What makes the default bar chart hard to understand? How could you improve the plot?
2. What is the most common `relig` in this survey? What's the most common `partyid`?
3. Which `relig` does `denom` (denomination) apply to? How can you find out with a table? How can you find out with a visualization?

Modifying Factor Order

It's often useful to change the order of the factor levels in a visualization. For example, imagine you want to explore the average number of hours spent watching TV per day across religions:

```
relig <- gss_cat %>%
  group_by(relig) %>%
  summarize(
    age = mean(age, na.rm = TRUE),
    tvhours = mean(tvhours, na.rm = TRUE),
    n = n()
  )

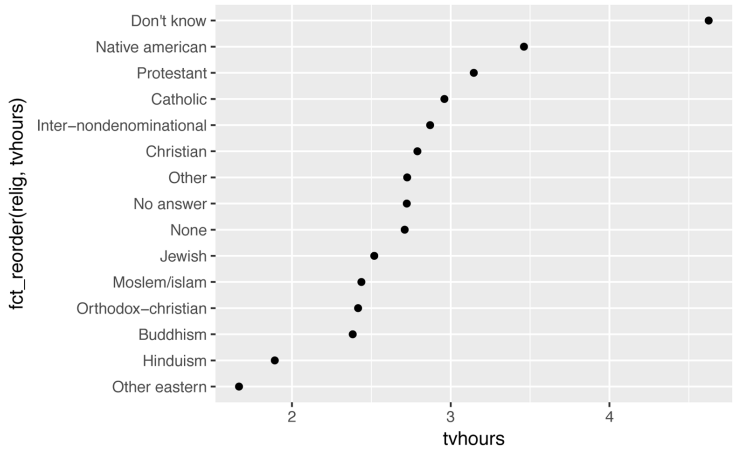
ggplot(relig, aes(tvhours, relig)) + geom_point()
```



It is difficult to interpret this plot because there's no overall pattern. We can improve it by reordering the levels of `relig` using `fct_reorder()`. `fct_reorder()` takes three arguments:

- `f`, the factor whose levels you want to modify.
- `x`, a numeric vector that you want to use to reorder the levels.
- Optionally, `fun`, a function that's used if there are multiple values of `x` for each value of `f`. The default value is `median`.

```
ggplot(relig, aes(tvhours, fct_reorder(relig, tvhours))) +
  geom_point()
```



Reordering religion makes it much easier to see that people in the "Don't know" category watch much more TV, and Hinduism and other Eastern religions watch much less.

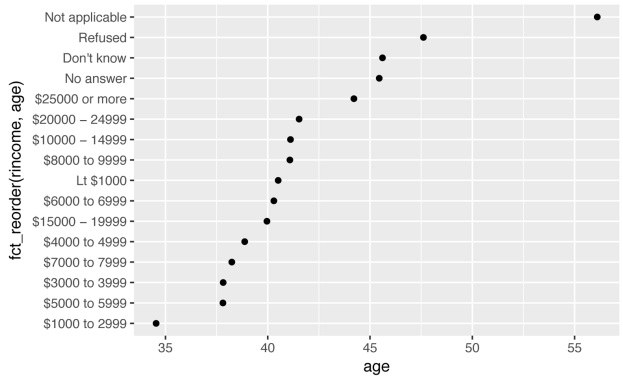
As you start making more complicated transformations, I'd recommend moving them out of `aes()` and into a separate `mutate()` step. For example, you could rewrite the preceding plot as:

```
relig %>%
  mutate(relig = fct_reorder(relig, tvhours)) %>%
  ggplot(aes(tvhours, relig)) +
    geom_point()
```

What if we create a similar plot looking at how average age varies across reported income level?

```
rincome <- gss_cat %>%
  group_by(rincome) %>%
  summarize(
    age = mean(age, na.rm = TRUE),
    tvhours = mean(tvhours, na.rm = TRUE),
    n = n()
  )

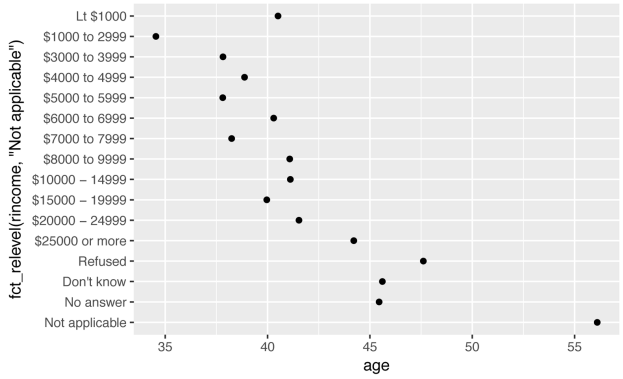
ggplot(
  rincome,
  aes(age, fct_reorder(rincome, age))
) + geom_point()
```



Here, arbitrarily reordering the levels isn't a good idea! That's because `rincome` already has a principled order that we shouldn't mess with. Reserve `fct_reorder()` for factors whose levels are arbitrarily ordered.

However, it does make sense to pull "Not applicable" to the front with the other special levels. You can use `fct_relevel()`. It takes a factor, `f`, and then any number of levels that you want to move to the front of the line:

```
ggplot(
 rincome,
 aes(age, fct_relevel(rincome, "Not applicable"))
) +
  geom_point()
```



Why do you think the average age for "Not applicable" is so high?

Another type of reordering is useful when you are coloring the lines on a plot. `fct_reorder2()` reorders the factor by the `y` values associated with the largest `x` values. This makes the plot easier to read because the line colors line up with the legend:

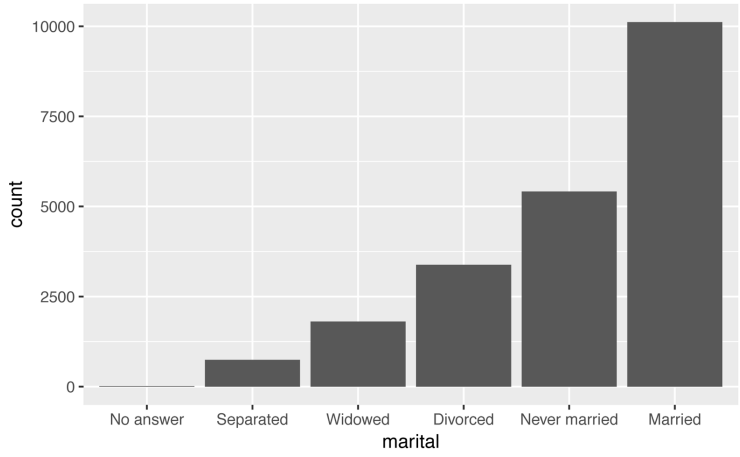
```
by_age <- gss_cat %>%
  filter(!is.na(age)) %>%
  group_by(age, marital) %>%
  count() %>%
  mutate(prop = n / sum(n))

ggplot(by_age, aes(age, prop, color = marital)) +
  geom_line(na.rm = TRUE)

ggplot(
  by_age,
  aes(age, prop, color = fct_reorder2(marital, age, prop))
) +
  geom_line() +
  labs(color = "marital")
```

Finally, for bar plots, you can use `fct_infreq()` to order levels in increasing frequency: this is the simplest type of reordering because it doesn't need any extra variables. You may want to combine with `fct_rev()`:

```
gss_cat %>%
  mutate(marital = marital %>% fct_infreq() %>% fct_rev()) %>%
  ggplot(aes(marital)) +
    geom_bar()
```



Exercises

1. There are some suspiciously high numbers in `tvhours`. Is the mean a good summary?
2. For each factor in `gss_cat` identify whether the order of the levels is arbitrary or principled.
3. Why did moving "Not applicable" to the front of the levels move it to the bottom of the plot?

Modifying Factor Levels

More powerful than changing the orders of the levels is changing their values. This allows you to clarify labels for publication, and collapse levels for high-level displays. The most general and powerful tool is `fct_recode()`. It allows you to recode, or change, the value of each level. For example, take `gss_cat$partyid`:

```
gss_cat %>% count(partyid)
#> # A tibble: 10 × 2
#>              partyid     n
#>                <fctr> <int>
#> 1          No answer   154
#> 2         Don't know     1
#> 3        Other party   393
#> 4  Strong republican  2314
#> 5 Not str republican  3032
#> 6       Ind,near rep  1791
#> # ... with 4 more rows
```

The levels are terse and inconsistent. Let's tweak them to be longer and use a parallel construction:

```
gss_cat %>%
  mutate(partyid = fct_recode(partyid,
    "Republican, strong"    = "Strong republican",
    "Republican, weak"      = "Not str republican",
    "Independent, near rep" = "Ind,near rep",
    "Independent, near dem" = "Ind,near dem",
    "Democrat, weak"        = "Not str democrat",
    "Democrat, strong"      = "Strong democrat"
  )) %>%
  count(partyid)
#> # A tibble: 10 × 2
#>                 partyid     n
#>                   <fctr> <int>
#> 1             No answer   154
```

```
#> 2            Don't know     1
#> 3          Other party    393
#> 4    Republican, strong  2314
#> 5      Republican, weak  3032
#> 6 Independent, near rep  1791
#> # ... with 4 more rows
```

`fct_recode()` will leave levels that aren't explicitly mentioned as is, and will warn you if you accidentally refer to a level that doesn't exist.

To combine groups, you can assign multiple old levels to the same new level:

```
gss_cat %>%
  mutate(partyid = fct_recode(partyid,
    "Republican, strong"    = "Strong republican",
    "Republican, weak"      = "Not str republican",
    "Independent, near rep" = "Ind,near rep",
    "Independent, near dem" = "Ind,near dem",
    "Democrat, weak"        = "Not str democrat",
    "Democrat, strong"      = "Strong democrat",
    "Other"                 = "No answer",
    "Other"                 = "Don't know",
    "Other"                 = "Other party"
  )) %>%
  count(partyid)
#> # A tibble: 8 × 2
#>                 partyid     n
#>                  <fctr> <int>
#> 1                 Other   548
#> 2    Republican, strong  2314
#> 3      Republican, weak  3032
#> 4 Independent, near rep  1791
#> 5           Independent  4119
#> 6 Independent, near dem  2499
#> # ... with 2 more rows
```

You must use this technique with care: if you group together categories that are truly different you will end up with misleading results.

If you want to collapse a lot of levels, `fct_collapse()` is a useful variant of `fct_recode()`. For each new variable, you can provide a vector of old levels:

```
gss_cat %>%
  mutate(partyid = fct_collapse(partyid,
    other = c("No answer", "Don't know", "Other party"),
    rep = c("Strong republican", "Not str republican"),
    ind = c("Ind,near rep", "Independent", "Ind,near dem"),
```

```
    dem = c("Not str democrat", "Strong democrat")
  )) %>%
  count(partyid)
#> # A tibble: 4 × 2
#>   partyid     n
#>    <fctr> <int>
#> 1   other   548
#> 2     rep  5346
#> 3     ind  8409
#> 4     dem  7180
```

Sometimes you just want to lump together all the small groups to make a plot or table simpler. That's the job of `fct_lump()`:

```
gss_cat %>%
  mutate(relig = fct_lump(relig)) %>%
  count(relig)
#> # A tibble: 2 × 2
#>        relig     n
#>       <fctr> <int>
#> 1 Protestant 10846
#> 2      Other 10637
```

The default behavior is to progressively lump together the smallest groups, ensuring that the aggregate is still the smallest group. In this case it's not very helpful: it is true that the majority of Americans in this survey are Protestant, but we've probably overcollapsed.

Instead, we can use the `n` parameter to specify how many groups (excluding other) we want to keep:

```
gss_cat %>%
  mutate(relig = fct_lump(relig, n = 10)) %>%
  count(relig, sort = TRUE) %>%
  print(n = Inf)
#> # A tibble: 10 × 2
#>                      relig     n
#>                     <fctr> <int>
#> 1               Protestant 10846
#> 2                 Catholic  5124
#> 3                     None  3523
#> 4                Christian   689
#> 5                    Other   458
#> 6                   Jewish   388
#> 7                 Buddhism   147
#> 8  Inter-nondenominational   109
#> 9             Moslem/islam   104
#> 10      Orthodox-christian    95
```

Exercises

1. How have the proportions of people identifying as Democrat, Republican, and Independent changed over time?
2. How could you collapse `rincome` into a small set of categories?

CHAPTER 13

Dates and Times with lubridate

Introduction

This chapter will show you how to work with dates and times in R. At first glance, dates and times seem simple. You use them all the time in your regular life, and they don't seem to cause much confusion. However, the more you learn about dates and times, the more complicated they seem to get. To warm up, try these three seemingly simple questions:

- Does every year have 365 days?
- Does every day have 24 hours?
- Does every minute have 60 seconds?

I'm sure you know that not every year has 365 days, but do you know the full rule for determining if a year is a leap year? (It has three parts.) You might have remembered that many parts of the world use daylight saving time (DST), so that some days have 23 hours, and others have 25. You might not have known that some minutes have 61 seconds because every now and then leap seconds are added because the Earth's rotation is gradually slowing down.

Dates and times are hard because they have to reconcile two physical phenomena (the rotation of the Earth and its orbit around the sun) with a whole raft of geopolitical phenomena including months, time zones, and DST. This chapter won't teach you every last detail about dates and times, but it will give you a solid grounding of practical skills that will help you with common data analysis challenges.

Prerequisites

This chapter will focus on the **lubridate** package, which makes it easier to work with dates and times in R. **lubridate** is not part of core tidyverse because you only need it when you're working with dates/times. We will also need **nycflights13** for practice data.

```
library(tidyverse)

library(lubridate)
library(nycflights13)
```

Creating Date/Times

There are three types of date/time data that refer to an instant in time:

- A *date*. Tibbles print this as `<date>`.
- A *time* within a day. Tibbles print this as `<time>`.
- A *date-time* is a date plus a time: it uniquely identifies an instant in time (typically to the nearest second). Tibbles print this as `<dttm>`. Elsewhere in R these are called POSIXct, but I don't think that's a very useful name.

In this chapter we are only going to focus on dates and date-times as R doesn't have a native class for storing times. If you need one, you can use the **hms** package.

You should always use the simplest possible data type that works for your needs. That means if you can use a date instead of a date-time, you should. Date-times are substantially more complicated because of the need to handle time zones, which we'll come back to at the end of the chapter.

To get the current date or date-time you can use `today()` or `now()`:

```
today()
#> [1] "2016-10-10"
now()
#> [1] "2016-10-10 15:19:39 PDT"
```

Otherwise, there are three ways you're likely to create a date/time:

- From a string.
- From individual date-time components.
- From an existing date/time object.

They work as follows.

From Strings

Date/time data often comes as strings. You've seen one approach to parsing strings into date-times in "Dates, Date-Times, and Times" on page 134. Another approach is to use the helpers provided by **lubridate**. They automatically work out the format once you specify the order of the component. To use them, identify the order in which year, month, and day appear in your dates, then arrange "y", "m", and "d" in the same order. That gives you the name of the **lubridate** function that will parse your date. For example:

```
ymd("2017-01-31")
#> [1] "2017-01-31"
mdy("January 31st, 2017")
#> [1] "2017-01-31"
dmy("31-Jan-2017")
#> [1] "2017-01-31"
```

These functions also take unquoted numbers. This is the most concise way to create a single date/time object, as you might need when filtering date/time data. `ymd()` is short and unambiguous:

```
ymd(20170131)
#> [1] "2017-01-31"
```

`ymd()` and friends create dates. To create a date-time, add an underscore and one or more of "h", "m", and "s" to the name of the parsing function:

```
ymd_hms("2017-01-31 20:11:59")
#> [1] "2017-01-31 20:11:59 UTC"
mdy_hm("01/31/2017 08:01")
#> [1] "2017-01-31 08:01:00 UTC"
```

You can also force the creation of a date-time from a date by supplying a time zone:

```
ymd(20170131, tz = "UTC")
#> [1] "2017-01-31 UTC"
```

From Individual Components

Instead of a single string, sometimes you'll have the individual components of the date-time spread across multiple columns. This is what we have in the flights data:

```
flights %>%
  select(year, month, day, hour, minute)
#> # A tibble: 336,776 × 5
#>    year month   day  hour minute
#>   <int> <int> <int> <dbl>  <dbl>
#> 1  2013     1     1     5     15
#> 2  2013     1     1     5     29
#> 3  2013     1     1     5     40
#> 4  2013     1     1     5     45
#> 5  2013     1     1     6      0
#> 6  2013     1     1     5     58
#> # ... with 3.368e+05 more rows
```

To create a date/time from this sort of input, use `make_date()` for dates, or `make_datetime()` for date-times:

```
flights %>%
  select(year, month, day, hour, minute) %>%
  mutate(
    departure = make_datetime(year, month, day, hour, minute)
  )
#> # A tibble: 336,776 × 6
#>    year month   day  hour minute           departure
#>   <int> <int> <int> <dbl>  <dbl>              <dttm>
#> 1  2013     1     1     5     15 2013-01-01 05:15:00
#> 2  2013     1     1     5     29 2013-01-01 05:29:00
#> 3  2013     1     1     5     40 2013-01-01 05:40:00
#> 4  2013     1     1     5     45 2013-01-01 05:45:00
#> 5  2013     1     1     6      0 2013-01-01 06:00:00
#> 6  2013     1     1     5     58 2013-01-01 05:58:00
#> # ... with 3.368e+05 more rows
```

Let's do the same thing for each of the four time columns in `flights`. The times are represented in a slightly odd format, so we use modulus arithmetic to pull out the hour and minute components. Once I've created the date-time variables, I focus in on the variables we'll explore in the rest of the chapter:

```
make_datetime_100 <- function(year, month, day, time) {
  make_datetime(year, month, day, time %/% 100, time %% 100)
}

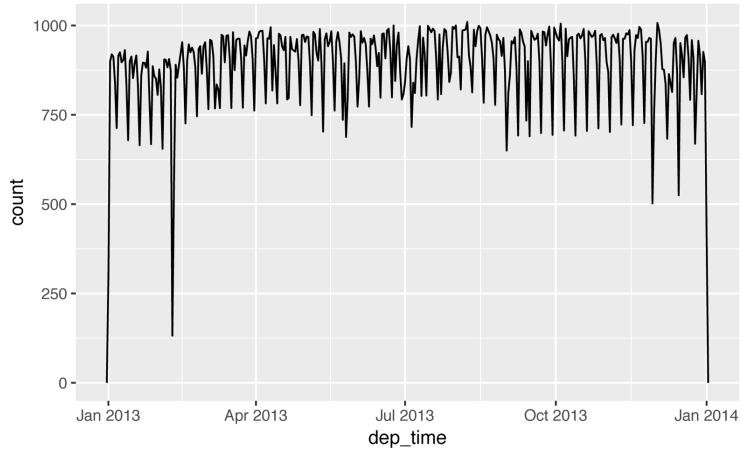
flights_dt <- flights %>%
  filter(!is.na(dep_time), !is.na(arr_time)) %>%
```

```
  mutate(
    dep_time = make_datetime_100(year, month, day, dep_time),
    arr_time = make_datetime_100(year, month, day, arr_time),
    sched_dep_time = make_datetime_100(
      year, month, day, sched_dep_time
    ),
    sched_arr_time = make_datetime_100(
      year, month, day, sched_arr_time
    )
  ) %>%
  select(origin, dest, ends_with("delay"), ends_with("time"))

flights_dt
#> # A tibble: 328,063 × 9
#>   origin  dest dep_delay arr_delay            dep_time
#>    <chr> <chr>     <dbl>     <dbl>              <dttm>
#> 1    EWR   IAH         2        11 2013-01-01 05:17:00
#> 2    LGA   IAH         4        20 2013-01-01 05:33:00
#> 3    JFK   MIA         2        33 2013-01-01 05:42:00
#> 4    JFK   BQN        -1       -18 2013-01-01 05:44:00
#> 5    LGA   ATL        -6       -25 2013-01-01 05:54:00
#> 6    EWR   ORD        -4        12 2013-01-01 05:54:00
#> # ... with 3.281e+05 more rows, and 4 more variables:
#> #   sched_dep_time <dttm>, arr_time <dttm>,
#> #   sched_arr_time <dttm>, air_time <dbl>
```

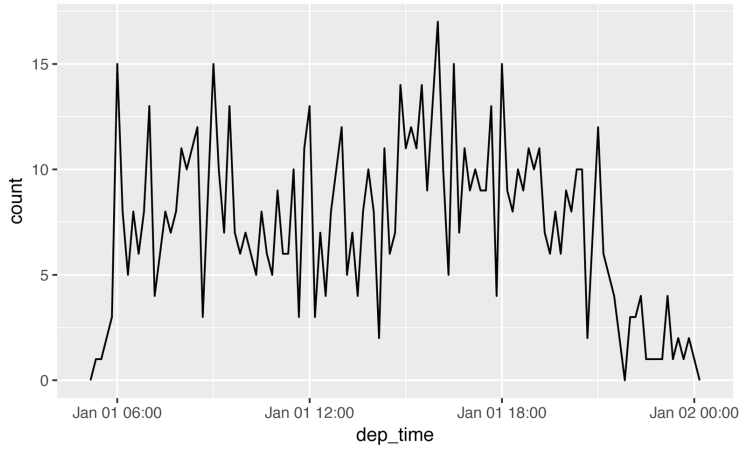
With this data, I can visualize the distribution of departure times across the year:

```
flights_dt %>%
  ggplot(aes(dep_time)) +
  geom_freqpoly(binwidth = 86400) # 86400 seconds = 1 day
```



Or within a single day:

```
flights_dt %>%
  filter(dep_time < ymd(20130102)) %>%
  ggplot(aes(dep_time)) +
  geom_freqpoly(binwidth = 600) # 600 s = 10 minutes
```



Note that when you use date-times in a numeric context (like in a histogram), 1 means 1 second, so a binwidth of 86400 means one day. For dates, 1 means 1 day.

From Other Types

You may want to switch between a date-time and a date. That's he job of `as_datetime()` and `as_date()`:

```
as_datetime(today())
#> [1] "2016-10-10 UTC"
as_date(now())
#> [1] "2016-10-10"
```

Sometimes you'll get date/times as numeric offsets from the "Unix Epoch," 1970-01-01. If the offset is in seconds, use `as_datetime()`; if it's in days, use `as_date()`:

```
as_datetime(60 * 60 * 10)
#> [1] "1970-01-01 10:00:00 UTC"
as_date(365 * 10 + 2)
#> [1] "1980-01-01"
```

Exercises

1. What happens if you parse a string that contains invalid dates?

    ```
    ymd(c("2010-10-10", "bananas"))
    ```

2. What does the `tzone` argument to `today()` do? Why is it important?
3. Use the appropriate **lubridate** function to parse each of the following dates:

    ```
    d1 <- "January 1, 2010"
    d2 <- "2015-Mar-07"
    d3 <- "06-Jun-2017"
    d4 <- c("August 19 (2015)", "July 1 (2015)")
    d5 <- "12/30/14" # Dec 30, 2014
    ```

Date-Time Components

Now that you know how to get date-time data into R's date-time data structures, let's explore what you can do with them. This section will focus on the accessor functions that let you get and set individual components. The next section will look at how arithmetic works with date-times.

Getting Components

You can pull out individual parts of the date with the accessor functions `year()`, `month()`, `mday()` (day of the month), `yday()` (day of the year), `wday()` (day of the week), `hour()`, `minute()`, and `second()`:

```
datetime <- ymd_hms("2016-07-08 12:34:56")

year(datetime)
#> [1] 2016
month(datetime)
#> [1] 7
mday(datetime)
#> [1] 8

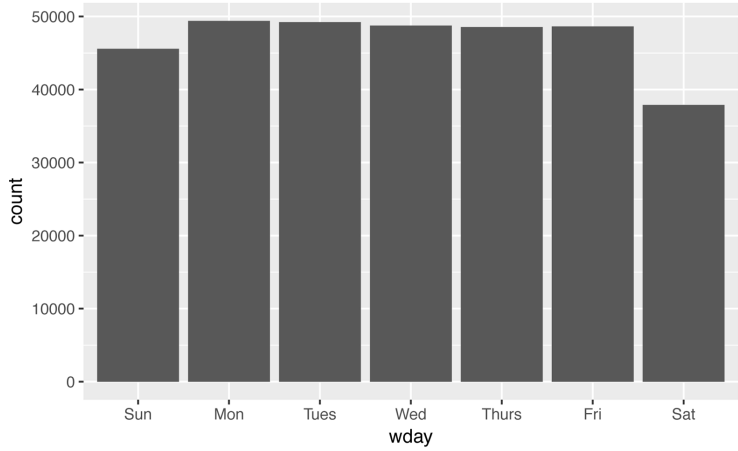
yday(datetime)
#> [1] 190
wday(datetime)
#> [1] 6
```

For `month()` and `wday()` you can set `label = TRUE` to return the abbreviated name of the month or day of the week. Set `abbr = FALSE` to return the full name:

```
month(datetime, label = TRUE)
#> [1] Jul
#> 12 Levels: Jan < Feb < Mar < Apr < May < Jun < ... < Dec
wday(datetime, label = TRUE, abbr = FALSE)
#> [1] Friday
#> 7 Levels: Sunday < Monday < Tuesday < ... < Saturday
```

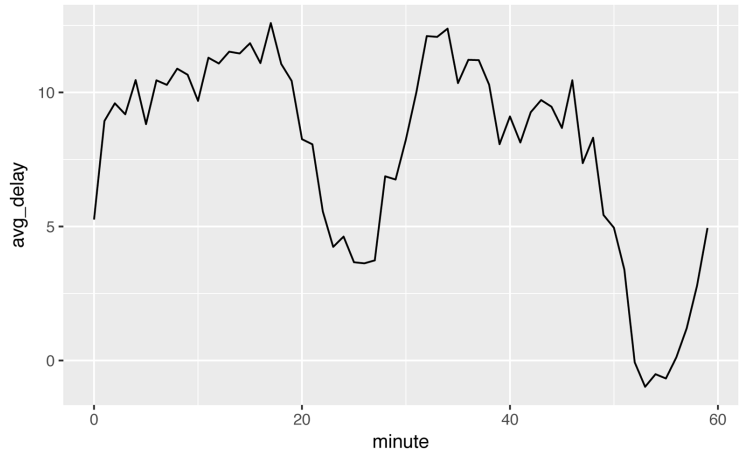
We can use `wday()` to see that more flights depart during the week than on the weekend:

```
flights_dt %>%
  mutate(wday = wday(dep_time, label = TRUE)) %>%
  ggplot(aes(x = wday)) +
    geom_bar()
```



There's an interesting pattern if we look at the average departure delay by minute within the hour. It looks like flights leaving in minutes 20–30 and 50–60 have much lower delays than the rest of the hour!

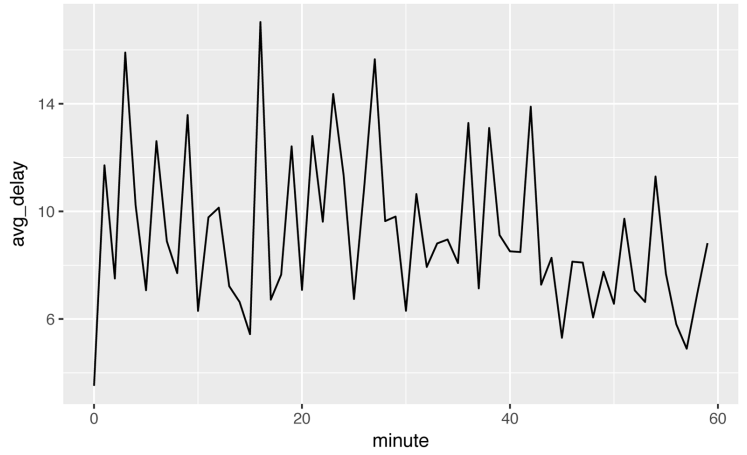
```
flights_dt %>%
  mutate(minute = minute(dep_time)) %>%
  group_by(minute) %>%
  summarize(
    avg_delay = mean(arr_delay, na.rm = TRUE),
    n = n()) %>%
  ggplot(aes(minute, avg_delay)) +
    geom_line()
```



Interestingly, if we look at the *scheduled* departure time we don't see such a strong pattern:

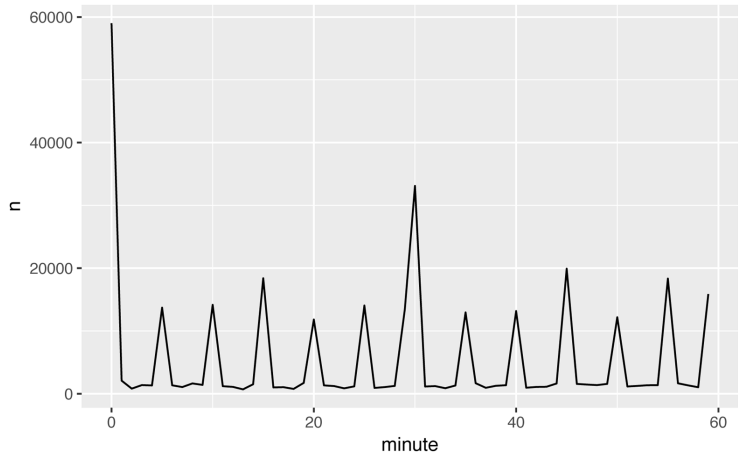
```
sched_dep <- flights_dt %>%
  mutate(minute = minute(sched_dep_time)) %>%
  group_by(minute) %>%
  summarize(
    avg_delay = mean(arr_delay, na.rm = TRUE),
    n = n())

ggplot(sched_dep, aes(minute, avg_delay)) +
  geom_line()
```



So why do we see that pattern with the actual departure times? Well, like much data collected by humans, there's a strong bias toward flights leaving at "nice" departure times. Always be alert for this sort of pattern whenever you work with data that involves human judgment!

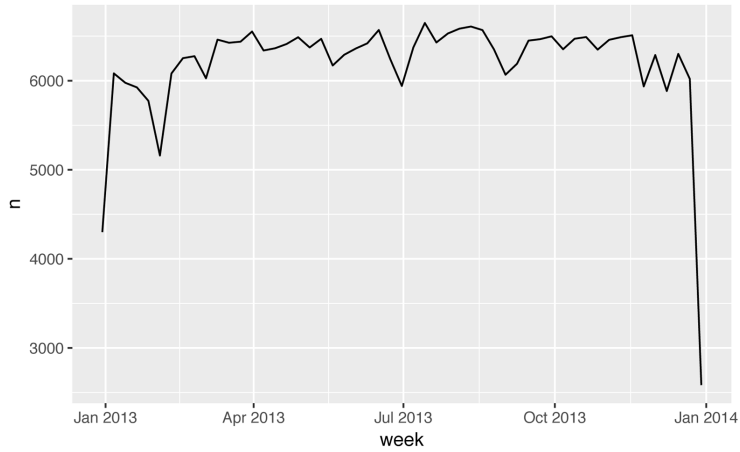
```
ggplot(sched_dep, aes(minute, n)) +
  geom_line()
```



Rounding

An alternative approach to plotting individual components is to round the date to a nearby unit of time, with `floor_date()`, `round_date()`, and `ceiling_date()`. Each ceiling_date()function takes a vector of dates to adjust and then the name of the unit to round down (floor), round up (ceiling), or round to. This, for example, allows us to plot the number of flights per week:

```
flights_dt %>%
  count(week = floor_date(dep_time, "week")) %>%
  ggplot(aes(week, n)) +
    geom_line()
```



Computing the difference between a rounded and unrounded date can be particularly useful.

Setting Components

You can also use each accessor function to set the components of a date/time:

```
(datetime <- ymd_hms("2016-07-08 12:34:56"))
#> [1] "2016-07-08 12:34:56 UTC"

year(datetime) <- 2020
datetime
#> [1] "2020-07-08 12:34:56 UTC"
month(datetime) <- 01
datetime
#> [1] "2020-01-08 12:34:56 UTC"
hour(datetime) <- hour(datetime) + 1
```

Alternatively, rather than modifying in place, you can create a new date-time with `update()`. This also allows you to set multiple values at once:

```
update(datetime, year = 2020, month = 2, mday = 2, hour = 2)
#> [1] "2020-02-02 02:34:56 UTC"
```

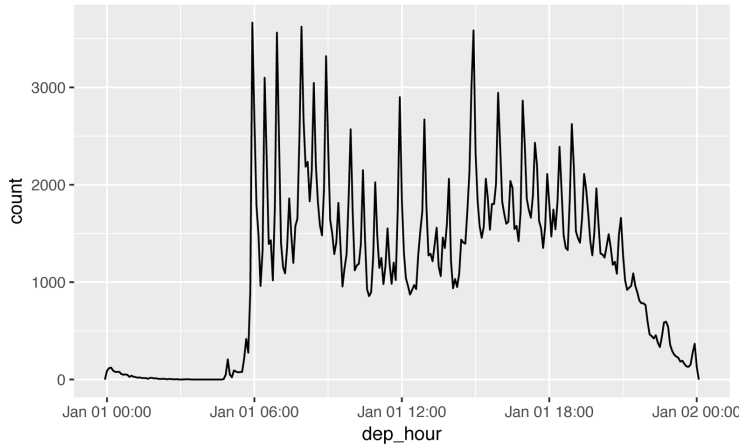
If values are too big, they will roll over:

```
ymd("2015-02-01") %>%
  update(mday = 30)
#> [1] "2015-03-02"
ymd("2015-02-01") %>%
```

```
  update(hour = 400)
#> [1] "2015-02-17 16:00:00 UTC"
```

You can use `update()` to show the distribution of flights across the course of the day for every day of the year:

```
flights_dt %>%
  mutate(dep_hour = update(dep_time, yday = 1)) %>%
  ggplot(aes(dep_hour)) +
    geom_freqpoly(binwidth = 300)
```



Setting larger components of a date to a constant is a powerful technique that allows you to explore patterns in the smaller components.

Exercises

1. How does the distribution of flight times within a day change over the course of the year?
2. Compare `dep_time`, `sched_dep_time`, and `dep_delay`. Are they consistent? Explain your findings.
3. Compare `air_time` with the duration between the departure and arrival. Explain your findings. (Hint: consider the location of the airport.)
4. How does the average delay time change over the course of a day? Should you use `dep_time` or `sched_dep_time`? Why?
5. On what day of the week should you leave if you want to minimize the chance of a delay?

6. What makes the distribution of `diamonds$carat` and `flights$sched_dep_time` similar?
7. Confirm my hypothesis that the early departures of flights in minutes 20–30 and 50–60 are caused by scheduled flights that leave early. Hint: create a binary variable that tells you whether or not a flight was delayed.

Time Spans

Next you'll learn about how arithmetic with dates works, including subtraction, addition, and division. Along the way, you'll learn about three important classes that represent time spans:

- *Durations*, which represent an exact number of seconds.
- *Periods*, which represent human units like weeks and months.
- *Intervals*, which represent a starting and ending point.

Durations

In R, when you subtract two dates, you get a difftime object:

```
# How old is Hadley?
h_age <- today() - ymd(19791014)
h_age
#> Time difference of 13511 days
```

A difftime class object records a time span of seconds, minutes, hours, days, or weeks. This ambiguity can make difftimes a little painful to work with, so **lubridate** provides an alternative that always uses seconds—the *duration*:

```
as.duration(h_age)
#> [1] "1167350400s (~36.99 years)"
```

Durations come with a bunch of convenient constructors:

```
dseconds(15)
#> [1] "15s"
dminutes(10)
#> [1] "600s (~10 minutes)"
dhours(c(12, 24))
#> [1] "43200s (~12 hours)" "86400s (~1 days)"
ddays(0:5)
#> [1] "0s"                "86400s (~1 days)"
#> [3] "172800s (~2 days)" "259200s (~3 days)"
```

```
#> [5] "345600s (~4 days)" "432000s (~5 days)"
dweeks(3)
#> [1] "1814400s (~3 weeks)"
dyears(1)
#> [1] "31536000s (~52.14 weeks)"
```

Durations always record the time span in seconds. Larger units are created by converting minutes, hours, days, weeks, and years to seconds at the standard rate (60 seconds in a minute, 60 minutes in an hour, 24 hours in day, 7 days in a week, 365 days in a year).

You can add and multiply durations:

```
2 * dyears(1)
#> [1] "63072000s (~2 years)"
dyears(1) + dweeks(12) + dhours(15)
#> [1] "38847600s (~1.23 years)"
```

You can add and subtract durations to and from days:

```
tomorrow <- today() + ddays(1)
last_year <- today() - dyears(1)
```

However, because durations represent an exact number of seconds, sometimes you might get an unexpected result:

```
one_pm <- ymd_hms(
  "2016-03-12 13:00:00",
  tz = "America/New_York"
)

one_pm
#> [1] "2016-03-12 13:00:00 EST"
one_pm + ddays(1)
#> [1] "2016-03-13 14:00:00 EDT"
```

Why is one day after 1 p.m. on March 12, 2 p.m. on March 13?! If you look carefully at the date you might also notice that the time zones have changed. Because of DST, March 12 only has 23 hours, so if we add a full day's worth of seconds we end up with a different time.

Periods

To solve this problem, **lubridate** provides *periods*. Periods are time spans but don't have a fixed length in seconds; instead they work with "human" times, like days and months. That allows them to work in a more intuitive way:

```
one_pm
#> [1] "2016-03-12 13:00:00 EST"
one_pm + days(1)
#> [1] "2016-03-13 13:00:00 EDT"
```

Like durations, periods can be created with a number of friendly constructor functions:

```
seconds(15)
#> [1] "15S"
minutes(10)
#> [1] "10M 0S"
hours(c(12, 24))
#> [1] "12H 0M 0S" "24H 0M 0S"
days(7)
#> [1] "7d 0H 0M 0S"
months(1:6)
#> [1] "1m 0d 0H 0M 0S" "2m 0d 0H 0M 0S" "3m 0d 0H 0M 0S"
#> [4] "4m 0d 0H 0M 0S" "5m 0d 0H 0M 0S" "6m 0d 0H 0M 0S"
weeks(3)
#> [1] "21d 0H 0M 0S"
years(1)
#> [1] "1y 0m 0d 0H 0M 0S"
```

You can add and multiply periods:

```
10 * (months(6) + days(1))
#> [1] "60m 10d 0H 0M 0S"
days(50) + hours(25) + minutes(2)
#> [1] "50d 25H 2M 0S"
```

And of course, add them to dates. Compared to durations, periods are more likely to do what you expect:

```
# A leap year
ymd("2016-01-01") + dyears(1)
#> [1] "2016-12-31"
ymd("2016-01-01") + years(1)
#> [1] "2017-01-01"

# Daylight Savings Time
one_pm + ddays(1)
#> [1] "2016-03-13 14:00:00 EDT"
one_pm + days(1)
#> [1] "2016-03-13 13:00:00 EDT"
```

Let's use periods to fix an oddity related to our flight dates. Some planes appear to have arrived at their destination *before* they departed from New York City:

```
flights_dt %>%
  filter(arr_time < dep_time)
```

```
#> # A tibble: 10,633 × 9
#>   origin  dest dep_delay arr_delay            dep_time
#>    <chr> <chr>     <dbl>     <dbl>              <dttm>
#> 1    EWR   BQN         9        -4 2013-01-01 19:29:00
#> 2    JFK   DFW        59        NA 2013-01-01 19:39:00
#> 3    EWR   TPA        -2         9 2013-01-01 20:58:00
#> 4    EWR   SJU        -6       -12 2013-01-01 21:02:00
#> 5    EWR   SFO        11       -14 2013-01-01 21:08:00
#> 6    LGA   FLL       -10        -2 2013-01-01 21:20:00
#> # ... with 1.063e+04 more rows, and 4 more variables:
#> #   sched_dep_time <dttm>, arr_time <dttm>,
#> #   sched_arr_time <dttm>, air_time <dbl>
```

These are overnight flights. We used the same date information for both the departure and the arrival times, but these flights arrived on the following day. We can fix this by adding `days(1)` to the arrival time of each overnight flight:

```
flights_dt <- flights_dt %>%
  mutate(
    overnight = arr_time < dep_time,
    arr_time = arr_time + days(overnight * 1),
    sched_arr_time = sched_arr_time + days(overnight * 1)
  )
```

Now all of our flights obey the laws of physics:

```
flights_dt %>%
  filter(overnight, arr_time < dep_time)
#> # A tibble: 0 × 10
#> # ... with 10 variables: origin <chr>, dest <chr>,
#> #   dep_delay <dbl>, arr_delay <dbl>, dep_time <dttm>,
#> #   sched_dep_time <dttm>, arr_time <dttm>,
#> #   sched_arr_time <dttm>, air_time <dbl>, overnight <lgl>
```

Intervals

It's obvious what `dyears(1) / ddays(365)` should return: one, because durations are always represented by a number of seconds, and a duration of a year is defined as 365 days' worth of seconds.

What should `years(1) / days(1)` return? Well, if the year was 2015 it should return 365, but if it was 2016, it should return 366! There's not quite enough information for **lubridate** to give a single clear answer. What it does instead is give an estimate, with a warning:

```
years(1) / days(1)
#> estimate only: convert to intervals for accuracy
#> [1] 365
```

If you want a more accurate measurement, you'll have to use an *interval*. An interval is a duration with a starting point; that makes it precise so you can determine exactly how long it is:

```
next_year <- today() + years(1)
(today() %--% next_year) / ddays(1)
#> [1] 365
```

To find out how many periods fall into an interval, you need to use integer division:

```
(today() %--% next_year) %/% days(1)
#> [1] 365
```

Summary

How do you pick between duration, periods, and intervals? As always, pick the simplest data structure that solves your problem. If you only care about physical time, use a duration; if you need to add human times, use a period; if you need to figure out how long a span is in human units, use an interval.

Figure 13-1 summarizes permitted arithmetic operations between the different data types.

	date				date time				duration				period				interval				number			
date	-								-	+			-	+							-	+		
date time					-				-	+			-	+							-	+		
duration	-	+			-	+			-	+		/									-	+	×	/
period	-	+			-	+							-	+							-	+	×	/
interval												/				/								
number	-	+			-	+			-	+	×		-	+	×		-	+	×		-	+	×	/

Figure 13-1. The allowed arithmetic operations between pairs of date/time classes

Exercises

1. Why is there `months()` but no `dmonths()`?
2. Explain `days(overnight * 1)` to someone who has just started learning R. How does it work?

3. Create a vector of dates giving the first day of every month in 2015. Create a vector of dates giving the first day of every month in the *current* year.
4. Write a function that, given your birthday (as a date), returns how old you are in years.
5. Why can't `(today() %--% (today() + years(1)) / months(1)` work?

Time Zones

Time zones are an enormously complicated topic because of their interaction with geopolitical entities. Fortunately we don't need to dig into all the details as they're not all that important for data analysis, but there are a few challenges we'll need to tackle head on.

The first challenge is that everyday names of time zones tend to be ambiguous. For example, if you're American you're probably familiar with EST, or Eastern Standard Time. However, both Australia and Canada also have EST! To avoid confusion, R uses the international standard IANA time zones. These use a consistent naming scheme with "/", typically in the form "<continent>/<city>" (there are a few exceptions because not every country lies on a continent). Examples include "America/New_York," "Europe/Paris," and "Pacific/Auckland."

You might wonder why the time zone uses a city, when typically you think of time zones as associated with a country or region within a country. This is because the IANA database has to record decades' worth of time zone rules. In the course of decades, countries change names (or break apart) fairly frequently, but city names tend to stay the same. Another problem is that name needs to reflect not only to the current behavior, but also the complete history. For example, there are time zones for both "America/New_York" and "America/Detroit." These cities both currently use Eastern Standard Time but in 1969–1972, Michigan (the state in which Detroit is located) did not follow DST, so it needs a different name. It's worth reading the raw time zone database (available at *http://www.iana.org/time-zones*) just to read some of these stories!

You can find out what R thinks your current time zone is with `Sys.timezone()`:

```
Sys.timezone()
#> [1] "America/Los_Angeles"
```

(If R doesn't know, you'll get an `NA`.)

And see the complete list of all time zone names with `OlsonNames()`:

```
length(OlsonNames())
#> [1] 589
head(OlsonNames())
#> [1] "Africa/Abidjan"     "Africa/Accra"
#> [3] "Africa/Addis_Ababa" "Africa/Algiers"
#> [5] "Africa/Asmara"      "Africa/Asmera"
```

In R, the time zone is an attribute of the date-time that only controls printing. For example, these three objects represent the same instant in time:

```
(x1 <- ymd_hms("2015-06-01 12:00:00", tz = "America/New_York"))
#> [1] "2015-06-01 12:00:00 EDT"
(x2 <- ymd_hms("2015-06-01 18:00:00", tz = "Europe/Copenhagen"))
#> [1] "2015-06-01 18:00:00 CEST"
(x3 <- ymd_hms("2015-06-02 04:00:00", tz = "Pacific/Auckland"))
#> [1] "2015-06-02 04:00:00 NZST"
```

You can verify that they're the same time using subtraction:

```
x1 - x2
#> Time difference of 0 secs
x1 - x3
#> Time difference of 0 secs
```

Unless otherwise specified, **lubridate** always uses UTC. UTC (Coordinated Universal Time) is the standard time zone used by the scientific community and is roughly equivalent to its predecessor GMT (Greenwich Mean Time). It does not have DST, which makes it a convenient representation for computation. Operations that combine date-times, like `c()`, will often drop the time zone. In that case, the date-times will display in your local time zone:

```
x4 <- c(x1, x2, x3)
x4
#> [1] "2015-06-01 09:00:00 PDT" "2015-06-01 09:00:00 PDT"
#> [3] "2015-06-01 09:00:00 PDT"
```

You can change the time zone in two ways:

- Keep the instant in time the same, and change how it's displayed. Use this when the instant is correct, but you want a more natural display:

```
x4a <- with_tz(x4, tzone = "Australia/Lord_Howe")
x4a
#> [1] "2015-06-02 02:30:00 LHST"
#> [2] "2015-06-02 02:30:00 LHST"
#> [3] "2015-06-02 02:30:00 LHST"
x4a - x4
#> Time differences in secs
#> [1] 0 0 0
```

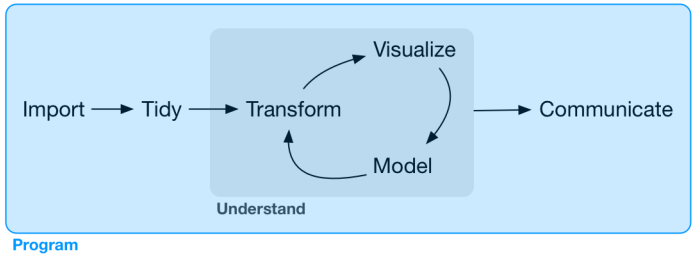
(This also illustrates another challenge of times zones: they're not all integer hour offsets!)

- Change the underlying instant in time. Use this when you have an instant that has been labeled with the incorrect time zone, and you need to fix it:

```
x4b <- force_tz(x4, tzone = "Australia/Lord_Howe")
x4b
#> [1] "2015-06-01 09:00:00 LHST"
#> [2] "2015-06-01 09:00:00 LHST"
#> [3] "2015-06-01 09:00:00 LHST"
x4b - x4
#> Time differences in hours
#> [1] -17.5 -17.5 -17.5
```

PART III
Program

In this part of the book, you'll improve your programming skills. Programming is a cross-cutting skill needed for all data science work: you must use a computer to do data science; you cannot do it in your head, or with pencil and paper.



Programming produces code, and code is a tool of communication. Obviously code tells the computer what you want it to do. But it also communicates meaning to other humans. Thinking about code as a vehicle for communication is important because every project you do is fundamentally collaborative. Even if you're not working with other people, you'll definitely be working with future-you! Writing clear code is important so that others (like future-you) can understand why you tackled an analysis in the way you did. That means getting better at programming also involves getting better at com-

municating. Over time, you want your code to become not just easier to write, but easier for others to read.

Writing code is similar in many ways to writing prose. One parallel that I find particularly useful is that in both cases rewriting is the key to clarity. The first expression of your ideas is unlikely to be particularly clear, and you may need to rewrite multiple times. After solving a data analysis challenge, it's often worth looking at your code and thinking about whether or not it's obvious what you've done. If you spend a little time rewriting your code while the ideas are fresh, you can save a lot of time later trying to re-create what your code did. But this doesn't mean you should rewrite every function: you need to balance what you need to achieve now with saving time in the long run. (But the more you rewrite your functions the more likely your first attempt will be clear.)

In the following four chapters, you'll learn skills that will allow you to both tackle new programs and solve existing problems with greater clarity and ease:

- In Chapter 14, you will dive deep into the *pipe*, `%>%`, and learn more about how it works, what the alternatives are, and when not to use it.
- Copy-and-paste is a powerful tool, but you should avoid doing it more than twice. Repeating yourself in code is dangerous because it can easily lead to errors and inconsistencies. Instead, in Chapter 15, you'll learn how to write *functions*, which let you extract out repeated code so that it can be easily reused.
- As you start to write more powerful functions, you'll need a solid grounding in R's *data structures*, provided by Chapter 16. You must master the four common atomic vectors and the three important S3 classes built on top of them, and understand the mysteries of the list and data frame.
- Functions extract out repeated code, but you often need to repeat the same actions on different inputs. You need tools for *iteration* that let you do similar things again and again. These tools include for loops and functional programming, which you'll learn about in Chapter 17.

Learning More

The goal of these chapters is to teach you the minimum about programming that you need to practice data science, which turns out to be a reasonable amount. Once you have mastered the material in this book, I strongly believe you should invest further in your programming skills. Learning more about programming is a long-term investment: it won't pay off immediately, but in the long term it will allow you to solve new problems more quickly, and let you reuse your insights from previous problems in new scenarios.

To learn more you need to study R as a programming language, not just an interactive environment for data science. We have written two books that will help you do so:

- *Hands-On Programming with R*, by Garrett Grolemund. This is an introduction to R as a programming language and is a great place to start if R is your first programming language. It covers similar material to these chapters, but with a different style and different motivation examples (based in the casino). It's a useful complement if you find that these four chapters go by too quickly.
- *Advanced R* by Hadley Wickham. This dives into the details of R the programming language. This is a great place to start if you have existing programming experience. It's also a great next step once you've internalized the ideas in these chapters. You can read it online at *http://adv-r.had.co.nz*.

CHAPTER 14

Pipes with magrittr

Introduction

Pipes are a powerful tool for clearly expressing a sequence of multiple operations. So far, you've been using them without knowing how they work, or what the alternatives are. Now, in this chapter, it's time to explore the pipe in more detail. You'll learn the alternatives to the pipe, when you shouldn't use the pipe, and some useful related tools.

Prerequisites

The pipe, `%>%`, comes from the **magrittr** package by Stefan Milton Bache. Packages in the tidyverse load `%>%` for you automatically, so you don't usually load **magrittr** explicitly. Here, however, we're focusing on piping, and we aren't loading any other packages, so we will load it explicitly.

```
library(magrittr)
```

Piping Alternatives

The point of the pipe is to help you write code in a way that is easier to read and understand. To see why the pipe is so useful, we're going to explore a number of ways of writing the same code. Let's use code to tell a story about a little bunny named Foo Foo:

Little bunny Foo Foo
Went hopping through the forest
Scooping up the field mice
And bopping them on the head

This is a popular children's poem that is accompanied by hand actions.

We'll start by defining an object to represent little bunny Foo Foo:

```
foo_foo <- little_bunny()
```

And we'll use a function for each key verb: `hop()`, `scoop()`, and `bop()`. Using this object and these verbs, there are (at least) four ways we could retell the story in code:

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

We'll work through each approach, showing you the code and talking about the advantages and disadvantages.

Intermediate Steps

The simplest approach is to save each step as a new object:

```
foo_foo_1 <- hop(foo_foo, through = forest)
foo_foo_2 <- scoop(foo_foo_1, up = field_mice)
foo_foo_3 <- bop(foo_foo_2, on = head)
```

The main downside of this form is that it forces you to name each intermediate element. If there are natural names, this is a good idea, and you should do it. But many times, like in this example, there aren't natural names, and you add numeric suffixes to make the names unique. That leads to two problems:

- The code is cluttered with unimportant names.
- You have to carefully increment the suffix on each line.

Whenever I write code like this, I invariably use the wrong number on one line and then spend 10 minutes scratching my head and trying to figure out what went wrong with my code.

You may also worry that this form creates many copies of your data and takes up a lot of memory. Surprisingly, that's not the case. First, note that proactively worrying about memory is not a useful way to spend your time: worry about it when it becomes a problem (i.e., you run out of memory), not before. Second, R isn't stupid, and it will share columns across data frames, where possible. Let's take a look at an actual data manipulation pipeline where we add a new column to `ggplot2::diamonds`:

```
diamonds <- ggplot2::diamonds
diamonds2 <- diamonds %>%
  dplyr::mutate(price_per_carat = price / carat)

pryr::object_size(diamonds)
#> 3.46 MB
pryr::object_size(diamonds2)
#> 3.89 MB
pryr::object_size(diamonds, diamonds2)
#> 3.89 MB
```

`pryr::object_size()` gives the memory occupied by all of its arguments. The results seem counterintuitive at first:

- `diamonds` takes up 3.46 MB.
- `diamonds2` takes up 3.89 MB.
- `diamonds` and `diamonds2` together take up 3.89 MB!

How can that work? Well, `diamonds2` has 10 columns in common with `diamonds`: there's no need to duplicate all that data, so the two data frames have variables in common. These variables will only get copied if you modify one of them. In the following example, we modify a single value in `diamonds$carat`. That means the `carat` variable can no longer be shared between the two data frames, and a copy must be made. The size of each data frame is unchanged, but the collective size increases:

```
diamonds$carat[1] <- NA
pryr::object_size(diamonds)
#> 3.46 MB
pryr::object_size(diamonds2)
#> 3.89 MB
pryr::object_size(diamonds, diamonds2)
#> 4.32 MB
```

(Note that we use `pryr::object_size()` here, not the built-in `object.size()`. `object.size()` only takes a single object so it can't compute how data is shared across multiple objects.)

Overwrite the Original

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

```
foo_foo <- hop(foo_foo, through = forest)
foo_foo <- scoop(foo_foo, up = field_mice)
foo_foo <- bop(foo_foo, on = head)
```

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

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

Function Composition

Another approach is to abandon assignment and just string the function calls together:

```
bop(
  scoop(
    hop(foo_foo, through = forest),
    up = field_mice
  ),
  on = head
)
```

Here the disadvantage is that you have to read from inside-out, from right-to-left, and that the arguments end up spread far apart (evocatively called the Dagwood sandwich problem). In short, this code is hard for a human to consume.

Use the Pipe

Finally, we can use the pipe:

```
foo_foo %>%
  hop(through = forest) %>%
  scoop(up = field_mouse) %>%
  bop(on = head)
```

This is my favorite form, because it focuses on verbs, not nouns. You can read this series of function compositions like it's a set of imperative actions. Foo Foo hops, then scoops, then bops. The downside, of course, is that you need to be familiar with the pipe. If you've never seen `%>%` before, you'll have no idea what this code does. Fortunately, most people pick up the idea very quickly, so when you share your code with others who aren't familiar with the pipe, you can easily teach them.

The pipe works by performing a "lexical transformation": behind the scenes, **magrittr** reassembles the code in the pipe to a form that works by overwriting an intermediate object. When you run a pipe like the preceding one, **magrittr** does something like this:

```
my_pipe <- function(.) {
  . <- hop(., through = forest)
  . <- scoop(., up = field_mice)
  bop(., on = head)
}
my_pipe(foo_foo)
```

This means that the pipe won't work for two classes of functions:

- Functions that use the current environment. For example, `assign()` will create a new variable with the given name in the current environment:

  ```
  assign("x", 10)
  x
  #> [1] 10

  "x" %>% assign(100)
  x
  #> [1] 10
  ```

  The use of `assign` with the pipe does not work because it assigns it to a temporary environment used by `%>%`. If you do want to use `assign` with the pipe, you must be explicit about the environment:

  ```
  env <- environment()
  "x" %>% assign(100, envir = env)
  x
  #> [1] 100
  ```

  Other functions with this problem include `get()` and `load()`.

- Functions that use lazy evaluation. In R, function arguments are only computed when the function uses them, not prior to call-

ing the function. The pipe computes each element in turn, so you can't rely on this behavior.

One place that this is a problem is `tryCatch()`, which lets you capture and handle errors:

```
tryCatch(stop("!"), error = function(e) "An error")
#> [1] "An error"

stop("!") %>%
  tryCatch(error = function(e) "An error")
#> Error in eval(expr, envir, enclos): !
```

There are a relatively wide class of functions with this behavior, including `try()`, `suppressMessages()`, and `suppressWarnings()` in base R.

When Not to Use the Pipe

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

- Your pipes are longer than (say) 10 steps. In that case, create intermediate objects with meaningful names. That will make debugging easier, because you can more easily check the intermediate results, and it makes it easier to understand your code, because the variable names can help communicate intent.
- You have multiple inputs or outputs. If there isn't one primary object being transformed, but two or more objects being combined together, don't use the pipe.
- You are starting to think about a directed graph with a complex dependency structure. Pipes are fundamentally linear and expressing complex relationships with them will typically yield confusing code.

Other Tools from magrittr

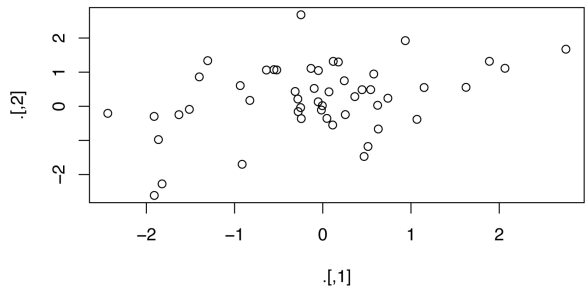
All packages in the tidyverse automatically make `%>%` available for you, so you don't normally load **magrittr** explicitly. However, there

are some other useful tools inside **magrittr** that you might want to try out:

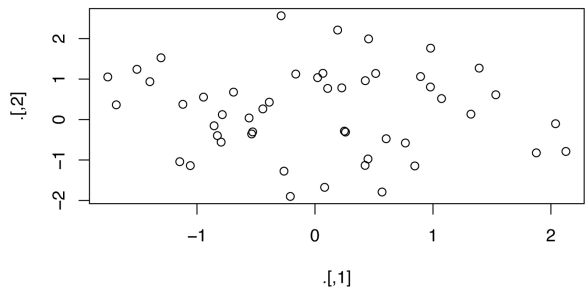
- When working with more complex pipes, it's sometimes useful to call a function for its side effects. Maybe you want to print out the current object, or plot it, or save it to disk. Many times, such functions don't return anything, effectively terminating the pipe.

  To work around this problem, you can use the "tee" pipe. `%T>%` works like `%>%` except that it returns the lefthand side instead of the righthand side. It's called "tee" because it's like a literal T-shaped pipe:

```
rnorm(100) %>%
  matrix(ncol = 2) %>%
  plot() %>%
  str()
#>  NULL
```




```
rnorm(100) %>%
  matrix(ncol = 2) %T>%
  plot() %>%
  str()
#>  num [1:50, 1:2] -0.387 -0.785 -1.057 -0.796 -1.756 ...
```



- If you're working with functions that don't have a data frame–based API (i.e., you pass them individual vectors, not a data frame and expressions to be evaluated in the context of that data frame), you might find `%$%` useful. It "explodes" out the variables in a data frame so that you can refer to them explicitly. This is useful when working with many functions in base R:

  ```
  mtcars %$%
    cor(disp, mpg)
  #> [1] -0.848
  ```

- For assignment **magrittr** provides the `%<>%` operator, which allows you to replace code like:

  ```
  mtcars <- mtcars %>%
    transform(cyl = cyl * 2)
  ```

  with:

  ```
  mtcars %<>% transform(cyl = cyl * 2)
  ```

  I'm not a fan of this operator because I think assignment is such a special operation that it should always be clear when it's occurring. In my opinion, a little bit of duplication (i.e., repeating the name of the object twice) is fine in return for making assignment more explicit.

CHAPTER 15

Functions

Introduction

One of the best ways to improve your reach as a data scientist is to write functions. Functions allow you to automate common tasks in a more powerful and general way than copying and pasting. Writing a function has three big advantages over using copy-and-paste:

- You can give a function an evocative name that makes your code easier to understand.
- As requirements change, you only need to update code in one place, instead of many.
- You eliminate the chance of making incidental mistakes when you copy and paste (i.e., updating a variable name in one place, but not in another).

Writing good functions is a lifetime journey. Even after using R for many years I still learn new techniques and better ways of approaching old problems. The goal of this chapter is not to teach you every esoteric detail of functions but to get you started with some pragmatic advice that you can apply immediately.

As well as practical advice for writing functions, this chapter also gives you some suggestions for how to style your code. Good code style is like correct punctuation. Youcanmanagewithoutit, but it sure makes things easier to read! As with styles of punctuation, there are many possible variations. Here we present the style we use in our code, but the most important thing is to be consistent.

Prerequisites

The focus of this chapter is on writing functions in base R, so you won't need any extra packages.

When Should You Write a Function?

You should consider writing a function whenever you've copied and pasted a block of code more than twice (i.e., you now have three copies of the same code). For example, take a look at this code. What does it do?

```
df <- tibble::tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)

df$a <- (df$a - min(df$a, na.rm = TRUE)) /
  (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))
df$b <- (df$b - min(df$b, na.rm = TRUE)) /
  (max(df$b, na.rm = TRUE) - min(df$a, na.rm = TRUE))
df$c <- (df$c - min(df$c, na.rm = TRUE)) /
  (max(df$c, na.rm = TRUE) - min(df$c, na.rm = TRUE))
df$d <- (df$d - min(df$d, na.rm = TRUE)) /
  (max(df$d, na.rm = TRUE) - min(df$d, na.rm = TRUE))
```

You might be able to puzzle out that this rescales each column to have a range from 0 to 1. But did you spot the mistake? I made an error when copying and pasting the code for `df$b`: I forgot to change an `a` to a `b`. Extracting repeated code out into a function is a good idea because it prevents you from making this type of mistake.

To write a function you need to first analyze the code. How many inputs does it have?

```
(df$a - min(df$a, na.rm = TRUE)) /
  (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))
```

This code only has one input: `df$a`. (If you're surprised that `TRUE` is not an input, you can explore why in the following exercise.) To make the inputs more clear, it's a good idea to rewrite the code using temporary variables with general names. Here this code only requires a single numeric vector, so I'll call it `x`:

```
x <- df$a
(x - min(x, na.rm = TRUE)) /
```

```
(max(x, na.rm = TRUE) - min(x, na.rm = TRUE))
#>  [1] 0.289 0.751 0.000 0.678 0.853 1.000 0.172 0.611 0.612
#>  [10] 0.601
```

There is some duplication in this code. We're computing the range of the data three times, but it makes sense to do it in one step:

```
rng <- range(x, na.rm = TRUE)
(x - rng[1]) / (rng[2] - rng[1])
#>  [1] 0.289 0.751 0.000 0.678 0.853 1.000 0.172 0.611 0.612
#>  [10] 0.601
```

Pulling out intermediate calculations into named variables is a good practice because it makes it more clear what the code is doing. Now that I've simplified the code, and checked that it still works, I can turn it into a function:

```
rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}
rescale01(c(0, 5, 10))
#> [1] 0.0 0.5 1.0
```

There are three key steps to creating a new function:

1. You need to pick a *name* for the function. Here I've used `rescale01` because this function rescales a vector to lie between 0 and 1.
2. You list the inputs, or *arguments*, to the function inside `function`. Here we have just one argument. If we had more the call would look like `function(x, y, z)`.
3. You place the code you have developed in the *body* of the function, a `{` block that immediately follows `function(...)`.

Note the overall process: I only made the function after I'd figured out how to make it work with a simple input. It's easier to start with working code and turn it into a function; it's harder to create a function and then try to make it work.

At this point it's a good idea to check your function with a few different inputs:

```
rescale01(c(-10, 0, 10))
#> [1] 0.0 0.5 1.0
rescale01(c(1, 2, 3, NA, 5))
#> [1] 0.00 0.25 0.50   NA 1.00
```

As you write more and more functions you'll eventually want to convert these informal, interactive tests into formal, automated tests. That process is called unit testing. Unfortunately, it's beyond the scope of this book, but you can learn about it at *http://r-pkgs.had.co.nz/tests.html*.

We can simplify the original example now that we have a function:

```
df$a <- rescale01(df$a)
df$b <- rescale01(df$b)
df$c <- rescale01(df$c)
df$d <- rescale01(df$d)
```

Compared to the original, this code is easier to understand and we've eliminated one class of copy-and-paste errors. There is still quite a bit of duplication since we're doing the same thing to multiple columns. We'll learn how to eliminate that duplication in Chapter 17, once you've learned more about R's data structures in Chapter 16.

Another advantage of functions is that if our requirements change, we only need to make the change in one place. For example, we might discover that some of our variables include infinite values, and `rescale01()` fails:

```
x <- c(1:10, Inf)
rescale01(x)
#>  [1]   0   0   0   0   0   0   0   0   0   0 NaN
```

Because we've extracted the code into a function, we only need to make the fix in one place:

```
rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE, finite = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}
rescale01(x)
#>  [1] 0.000 0.111 0.222 0.333 0.444 0.556 0.667 0.778 0.889
#>  [10] 1.000   Inf
```

This is an important part of the "do not repeat yourself" (or DRY) principle. The more repetition you have in your code, the more places you need to remember to update when things change (and they always do!), and the more likely you are to create bugs over time.

Exercises

1. Why is `TRUE` not a parameter to `rescale01()`? What would happen if `x` contained a single missing value, and `na.rm` was `FALSE`?
2. In the second variant of `rescale01()`, infinite values are left unchanged. Rewrite `rescale01()` so that `-Inf` is mapped to 0, and `Inf` is mapped to 1.
3. Practice turning the following code snippets into functions. Think about what each function does. What would you call it? How many arguments does it need? Can you rewrite it to be more expressive or less duplicative?

```
mean(is.na(x))

x / sum(x, na.rm = TRUE)

sd(x, na.rm = TRUE) / mean(x, na.rm = TRUE)
```

4. Follow *http://nicercode.github.io/intro/writing-functions.html* to write your own functions to compute the variance and skew of a numeric vector.
5. Write `both_na()`, a function that takes two vectors of the same length and returns the number of positions that have an `NA` in both vectors.
6. What do the following functions do? Why are they useful even though they are so short?

```
is_directory <- function(x) file.info(x)$isdir
is_readable <- function(x) file.access(x, 4) == 0
```

7. Read the complete lyrics to "Little Bunny Foo Foo." There's a lot of duplication in this song. Extend the initial piping example to re-create the complete song, and use functions to reduce the duplication.

Functions Are for Humans and Computers

It's important to remember that functions are not just for the computer, but are also for humans. R doesn't care what your function is called, or what comments it contains, but these are important for human readers. This section discusses some things that you should bear in mind when writing functions that humans can understand.

The name of a function is important. Ideally, the name of your function will be short, but clearly evoke what the function does. That's hard! But it's better to be clear than short, as RStudio's autocomplete makes it easy to type long names.

Generally, function names should be verbs, and arguments should be nouns. There are some exceptions: nouns are OK if the function computes a very well known noun (i.e., `mean()` is better than `compute_mean()`), or is accessing some property of an object (i.e., `coef()` is better than `get_coefficients()`). A good sign that a noun might be a better choice is if you're using a very broad verb like "get," "compute," "calculate," or "determine." Use your best judgment and don't be afraid to rename a function if you figure out a better name later:

```
# Too short
f()

# Not a verb, or descriptive
my_awesome_function()

# Long, but clear
impute_missing()
collapse_years()
```

If your function name is composed of multiple words, I recommend using "snake_case," where each lowercase word is separated by an underscore. camelCase is a popular alternative. It doesn't really matter which one you pick; the important thing is to be consistent: pick one or the other and stick with it. R itself is not very consistent, but there's nothing you can do about that. Make sure you don't fall into the same trap by making your code as consistent as possible:

```
# Never do this!
col_mins <- function(x, y) {}
rowMaxes <- function(y, x) {}
```

If you have a family of functions that do similar things, make sure they have consistent names and arguments. Use a common prefix to indicate that they are connected. That's better than a common suffix because autocomplete allows you to type the prefix and see all the members of the family:

```
# Good
input_select()
input_checkbox()
input_text()
```

```
# Not so good
select_input()
checkbox_input()
text_input()
```

A good example of this design is the **stringr** package: if you don't remember exactly which function you need, you can type `str_` and jog your memory.

Where possible, avoid overriding existing functions and variables. It's impossible to do in general because so many good names are already taken by other packages, but avoiding the most common names from base R will avoid confusion:

```
# Don't do this!
T <- FALSE
c <- 10
mean <- function(x) sum(x)
```

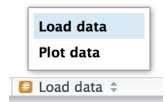
Use comments, lines starting with #, to explain the "why" of your code. You generally should avoid comments that explain the "what" or the "how." If you can't understand what the code does from reading it, you should think about how to rewrite it to be more clear. Do you need to add some intermediate variables with useful names? Do you need to break out a subcomponent of a large function so you can name it? However, your code can never capture the reasoning behind your decisions: why did you choose this approach instead of an alternative? What else did you try that didn't work? It's a great idea to capture that sort of thinking in a comment.

Another important use of comments is to break up your file into easily readable chunks. Use long lines of - or = to make it easy to spot the breaks:

```
# Load data --------------------------------------

# Plot data --------------------------------------
```

RStudio provides a keyboard shortcut to create these headers (Cmd/Ctrl-Shift-R), and will display them in the code navigation drop-down at the bottom-left of the editor:



Exercises

1. Read the source code for each of the following three functions, puzzle out what they do, and then brainstorm better names:

```
f1 <- function(string, prefix) {
  substr(string, 1, nchar(prefix)) == prefix
}
f2 <- function(x) {
  if (length(x) <= 1) return(NULL)
  x[-length(x)]
}
f3 <- function(x, y) {
  rep(y, length.out = length(x))
}
```

2. Take a function that you've written recently and spend five minutes brainstorming a better name for it and its arguments.
3. Compare and contrast `rnorm()` and `MASS::mvrnorm()`. How could you make them more consistent?
4. Make a case for why `norm_r()`, `norm_d()`, etc., would be better than `rnorm()`, `dnorm()`. Make a case for the opposite.

Conditional Execution

An `if` statement allows you to conditionally execute code. It looks like this:

```
if (condition) {
  # code executed when condition is TRUE
} else {
  # code executed when condition is FALSE
}
```

To get help on `if` you need to surround it in backticks: `` ?`if` ``. The help isn't particularly helpful if you're not already an experienced programmer, but at least you know how to get to it!

Here's a simple function that uses an `if` statement. The goal of this function is to return a logical vector describing whether or not each element of a vector is named:

```
has_name <- function(x) {
  nms <- names(x)
  if (is.null(nms)) {
    rep(FALSE, length(x))
```

```
  } else {
    !is.na(nms) & nms != ""
  }
}
```

This function takes advantage of the standard return rule: a function returns the last value that it computed. Here that is either one of the two branches of the `if` statement.

Conditions

The `condition` must evaluate to either `TRUE` or `FALSE`. If it's a vector, you'll get a warning message; if it's an `NA`, you'll get an error. Watch out for these messages in your own code:

```
if (c(TRUE, FALSE)) {}
#> Warning in if (c(TRUE, FALSE)) {:
#> the condition has length > 1 and only the
#> first element will be used
#> NULL

if (NA) {}
#> Error in if (NA) {: missing value where TRUE/FALSE needed
```

You can use `||` (or) and `&&` (and) to combine multiple logical expressions. These operators are "short-circuiting": as soon as `||` sees the first `TRUE` it returns `TRUE` without computing anything else. As soon as `&&` sees the first `FALSE` it returns `FALSE`. You should never use `|` or `&` in an `if` statement: these are vectorized operations that apply to multiple values (that's why you use them in `filter()`). If you do have a logical vector, you can use `any()` or `all()` to collapse it to a single value.

Be careful when testing for equality. `==` is vectorized, which means that it's easy to get more than one output. Either check the length is already 1, collapse with `all()` or `any()`, or use the nonvectorized `identical()`. `identical()` is very strict: it always returns either a single `TRUE` or a single `FALSE`, and doesn't coerce types. This means that you need to be careful when comparing integers and doubles:

```
identical(0L, 0)
#> [1] FALSE
```

You also need to be wary of floating-point numbers:

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

```
x == 2
#> [1] FALSE
x - 2
#> [1] 4.44e-16
```

Instead use `dplyr::near()` for comparisons, as described in "Comparisons" on page 46.

And remember, `x == NA` doesn't do anything useful!

Multiple Conditions

You can chain multiple `if` statements together:

```
if (this) {
  # do that
} else if (that) {
  # do something else
} else {
  #
}
```

But if you end up with a very long series of chained `if` statements, you should consider rewriting. One useful technique is the `switch()` function. It allows you to evaluate selected code based on position or name:

```
#> function(x, y, op) {
#>   switch(op,
#>     plus = x + y,
#>     minus = x - y,
#>     times = x * y,
#>     divide = x / y,
#>     stop("Unknown op!")
#>   )
#> }
```

Another useful function that can often eliminate long chains of `if` statements is `cut()`. It's used to discretize continuous variables.

Code Style

Both `if` and `function` should (almost) always be followed by squiggly brackets (`{}`), and the contents should be indented by two spaces. This makes it easier to see the hierarchy in your code by skimming the lefthand margin.

An opening curly brace should never go on its own line and should always be followed by a new line. A closing curly brace should

always go on its own line, unless it's followed by `else`. Always indent the code inside curly braces:

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

if (y == 0) {
  log(x)
} else {
  y ^ x
}

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

if (y == 0) {
  log(x)
}
else {
  y ^ x
}
```

It's OK to drop the curly braces if you have a very short `if` statement that can fit on one line:

```
y <- 10
x <- if (y < 20) "Too low" else "Too high"
```

I recommend this only for very brief `if` statements. Otherwise, the full form is easier to read:

```
if (y < 20) {
  x <- "Too low"
} else {
  x <- "Too high"
}
```

Exercises

1. What's the difference between `if` and `ifelse()`? Carefully read the help and construct three examples that illustrate the key differences.
2. Write a greeting function that says "good morning," "good afternoon," or "good evening," depending on the time of day. (Hint: use a time argument that defaults to `lubridate::now()`. That will make it easier to test your function.)

3. Implement a `fizzbuzz` function. It takes a single number as input. If the number is divisible by three, it returns "fizz". If it's divisible by five it returns "buzz". If it's divisible by three and five, it returns "fizzbuzz". Otherwise, it returns the number. Make sure you first write working code before you create the function.
4. How could you use `cut()` to simplify this set of nested if-else statements?

```
if (temp <= 0) {
  "freezing"
} else if (temp <= 10) {
  "cold"
} else if (temp <= 20) {
  "cool"
} else if (temp <= 30) {
  "warm"
} else {
  "hot"
}
```

   How would you change the call to `cut()` if I'd used `<` instead of `<=`? What is the other chief advantage of `cut()` for this problem? (Hint: what happens if you have many values in `temp`?)
5. What happens if you use `switch()` with numeric values?
6. What does this `switch()` call do? What happens if `x` is "e"?

```
switch(x,
  a = ,
  b = "ab",
  c = ,
  d = "cd"
)
```

   Experiment, then carefully read the documentation.

Function Arguments

The arguments to a function typically fall into two broad sets: one set supplies the *data* to compute on, and the other supplies arguments that control the *details* of the computation. For example:

- In `log()`, the data is `x`, and the detail is the `base` of the logarithm.

- In `mean()`, the data is `x`, and the details are how much data to trim from the ends (`trim`) and how to handle missing values (`na.rm`).
- In `t.test()`, the data are `x` and `y`, and the details of the test are `alternative`, `mu`, `paired`, `var.equal`, and `conf.level`.
- In `str_c()` you can supply any number of strings to `...`, and the details of the concatenation are controlled by `sep` and `collapse`.

Generally, data arguments should come first. Detail arguments should go on the end, and usually should have default values. You specify a default value in the same way you call a function with a named argument:

```
# Compute confidence interval around
# mean using normal approximation
mean_ci <- function(x, conf = 0.95) {
  se <- sd(x) / sqrt(length(x))
  alpha <- 1 - conf
  mean(x) + se * qnorm(c(alpha / 2, 1 - alpha / 2))
}

x <- runif(100)
mean_ci(x)
#> [1] 0.498 0.610
mean_ci(x, conf = 0.99)
#> [1] 0.480 0.628
```

The default value should almost always be the most common value. The few exceptions to this rule have to do with safety. For example, it makes sense for `na.rm` to default to `FALSE` because missing values are important. Even though `na.rm = TRUE` is what you usually put in your code, it's a bad idea to silently ignore missing values by default.

When you call a function, you typically omit the names of the data arguments, because they are used so commonly. If you override the default value of a detail argument, you should use the full name:

```
# Good
mean(1:10, na.rm = TRUE)

# Bad
mean(x = 1:10, , FALSE)
mean(, TRUE, x = c(1:10, NA))
```

You can refer to an argument by its unique prefix (e.g., `mean(x, n = TRUE)`), but this is generally best avoided given the possibilities for confusion.

Notice that when you call a function, you should place a space around `=` in function calls, and always put a space after a comma, not before (just like in regular English). Using whitespace makes it easier to skim the function for the important components:

```
# Good
average <- mean(feet / 12 + inches, na.rm = TRUE)

# Bad
average<-mean(feet/12+inches,na.rm=TRUE)
```

Choosing Names

The names of the arguments are also important. R doesn't care, but the readers of your code (including future-you!) will. Generally you should prefer longer, more descriptive names, but there are a handful of very common, very short names. It's worth memorizing these:

- `x`, `y`, `z`: vectors.
- `w`: a vector of weights.
- `df`: a data frame.
- `i`, `j`: numeric indices (typically rows and columns).
- `n`: length, or number of rows.
- `p`: number of columns.

Otherwise, consider matching names of arguments in existing R functions. For example, use `na.rm` to determine if missing values should be removed.

Checking Values

As you start to write more functions, you'll eventually get to the point where you don't remember exactly how your function works. At this point it's easy to call your function with invalid inputs. To avoid this problem, it's often useful to make constraints explicit. For example, imagine you've written some functions for computing weighted summary statistics:

```
wt_mean <- function(x, w) {
  sum(x * w) / sum(x)
}
wt_var <- function(x, w) {
  mu <- wt_mean(x, w)
  sum(w * (x - mu) ^ 2) / sum(w)
}
wt_sd <- function(x, w) {
  sqrt(wt_var(x, w))
}
```

What happens if `x` and `w` are not the same length?

```
wt_mean(1:6, 1:3)
#> [1] 2.19
```

In this case, because of R's vector recycling rules, we don't get an error.

It's good practice to check important preconditions, and throw an error (with `stop()`) if they are not true:

```
wt_mean <- function(x, w) {
  if (length(x) != length(w)) {
    stop("`x` and `w` must be the same length", call. = FALSE)
  }
  sum(w * x) / sum(x)
}
```

Be careful not to take this too far. There's a trade-off between how much time you spend making your function robust, versus how long you spend writing it. For example, if you also added a `na.rm` argument, I probably wouldn't check it carefully:

```
wt_mean <- function(x, w, na.rm = FALSE) {
  if (!is.logical(na.rm)) {
    stop("`na.rm` must be logical")
  }
  if (length(na.rm) != 1) {
    stop("`na.rm` must be length 1")
  }
  if (length(x) != length(w)) {
    stop("`x` and `w` must be the same length", call. = FALSE)
  }

  if (na.rm) {
    miss <- is.na(x) | is.na(w)
    x <- x[!miss]
    w <- w[!miss]
  }
  sum(w * x) / sum(x)
}
```

This is a lot of extra work for little additional gain. A useful compromise is the built-in `stopifnot()`; it checks that each argument is `TRUE`, and produces a generic error message if not:

```
wt_mean <- function(x, w, na.rm = FALSE) {
  stopifnot(is.logical(na.rm), length(na.rm) == 1)
  stopifnot(length(x) == length(w))

  if (na.rm) {
    miss <- is.na(x) | is.na(w)
    x <- x[!miss]
    w <- w[!miss]
  }
  sum(w * x) / sum(x)
}
wt_mean(1:6, 6:1, na.rm = "foo")
#> Error: is.logical(na.rm) is not TRUE
```

Note that when using `stopifnot()` you assert what should be true rather than checking for what might be wrong.

Dot-Dot-Dot (...)

Many functions in R take an arbitrary number of inputs:

```
sum(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
#> [1] 55
stringr::str_c("a", "b", "c", "d", "e", "f")
#> [1] "abcdef"
```

How do these functions work? They rely on a special argument: `...` (pronounced dot-dot-dot). This special argument captures any number of arguments that aren't otherwise matched.

It's useful because you can then send those `...` on to another function. This is a useful catch-all if your function primarily wraps another function. For example, I commonly create these helper functions that wrap around `str_c()`:

```
commas <- function(...) stringr::str_c(..., collapse = ", ")
commas(letters[1:10])
#> [1] "a, b, c, d, e, f, g, h, i, j"

rule <- function(..., pad = "-") {
  title <- paste0(...)
  width <- getOption("width") - nchar(title) - 5
  cat(title, " ", stringr::str_dup(pad, width), "\n", sep = "")
}
rule("Important output")
#> Important output ----------------------------------------
```

Here `...` lets me forward on any arguments that I don't want to deal with to `str_c()`. It's a very convenient technique. But it does come at a price: any misspelled arguments will not raise an error. This makes it easy for typos to go unnoticed:

```
x <- c(1, 2)
sum(x, na.mr = TRUE)
#> [1] 4
```

If you just want to capture the values of the `...`, use `list(...)`.

Lazy Evaluation

Arguments in R are lazily evaluated: they're not computed until they're needed. That means if they're never used, they're never called. This is an important property of R as a programming language, but is generally not important when you're writing your own functions for data analysis. You can read more about lazy evaluation at *http://adv-r.had.co.nz/Functions.html#lazy-evaluation*.

Exercises

1. What does `commas(letters, collapse = "-")` do? Why?
2. It'd be nice if you could supply multiple characters to the `pad` argument, e.g., `rule("Title", pad = "-+")`. Why doesn't this currently work? How could you fix it?
3. What does the `trim` argument to `mean()` do? When might you use it?
4. The default value for the `method` argument to `cor()` is `c("pearson", "kendall", "spearman")`. What does that mean? What value is used by default?

Return Values

Figuring out what your function should return is usually straightforward: it's why you created the function in the first place! There are two things you should consider when returning a value:

- Does returning early make your function easier to read?
- Can you make your function pipeable?

Explicit Return Statements

The value returned by the function is usually the last statement it evaluates, but you can choose to return early by using `return()`. I think it's best to save the use of `return()` to signal that you can return early with a simpler solution. A common reason to do this is because the inputs are empty:

```
complicated_function <- function(x, y, z) {
  if (length(x) == 0 || length(y) == 0) {
    return(0)
  }

  # Complicated code here
}
```

Another reason is because you have a `if` statement with one complex block and one simple block. For example, you might write an `if` statement like this:

```
f <- function() {
  if (x) {
    # Do
    # something
    # that
    # takes
    # many
    # lines
    # to
    # express
  } else {
    # return something short
  }
}
```

But if the first block is very long, by the time you get to the `else`, you've forgotten the `condition`. One way to rewrite it is to use an early return for the simple case:

```
f <- function() {
  if (!x) {
    return(something_short)
  }

  # Do
  # something
  # that
  # takes
  # many
  # lines
```

```
  # to
  # express
}
```

This tends to make the code easier to understand, because you don't need quite so much context to understand it.

Writing Pipeable Functions

If you want to write your own pipeable functions, thinking about the return value is important. There are two main types of pipeable functions: transformation and side-effect.

In *transformation* functions, there's a clear "primary" object that is passed in as the first argument, and a modified version is returned by the function. For example, the key objects for **dplyr** and **tidyr** are data frames. If you can identify what the object type is for your domain, you'll find that your functions just work with the pipe.

Side-effect functions are primarily called to perform an action, like drawing a plot or saving a file, not transforming an object. These functions should "invisibly" return the first argument, so they're not printed by default, but can still be used in a pipeline. For example, this simple function prints out the number of missing values in a data frame:

```
show_missings <- function(df) {
  n <- sum(is.na(df))
  cat("Missing values: ", n, "\n", sep = "")

  invisible(df)
}
```

If we call it interactively, the `invisible()` means that the input `df` doesn't get printed out:

```
show_missings(mtcars)
#> Missing values: 0
```

But it's still there, it's just not printed by default:

```
x <- show_missings(mtcars)
#> Missing values: 0
class(x)
#> [1] "data.frame"
dim(x)
#> [1] 32 11
```

And we can still use it in a pipe:

```
mtcars %>%
  show_missings() %>%
  mutate(mpg = ifelse(mpg < 20, NA, mpg)) %>%
  show_missings()
#> Missing values: 0
#> Missing values: 18
```

Environment

The last component of a function is its environment. This is not something you need to understand deeply when you first start writing functions. However, it's important to know a little bit about environments because they are crucial to how functions work. The environment of a function controls how R finds the value associated with a name. For example, take this function:

```
f <- function(x) {
  x + y
}
```

In many programming languages, this would be an error, because `y` is not defined inside the function. In R, this is valid code because R uses rules called *lexical scoping* to find the value associated with a name. Since `y` is not defined inside the function, R will look in the *environment* where the function was defined:

```
y <- 100
f(10)
#> [1] 110

y <- 1000
f(10)
#> [1] 1010
```

This behavior seems like a recipe for bugs, and indeed you should avoid creating functions like this deliberately, but by and large it doesn't cause too many problems (especially if you regularly restart R to get to a clean slate).

The advantage of this behavior is that from a language standpoint it allows R to be very consistent. Every name is looked up using the same set of rules. For `f()` that includes the behavior of two things that you might not expect: `{` and `+`. This allows you to do devious things like:

```
`+` <- function(x, y) {
  if (runif(1) < 0.1) {
    sum(x, y)
  } else {
    sum(x, y) * 1.1
  }
}
table(replicate(1000, 1 + 2))
#>
#>   3 3.3
#> 100 900
rm(`+`)
```

This is a common phenomenon in R. R places few limits on your power. You can do many things that you can't do in other programming languages. You can do many things that 99% of the time are extremely ill-advised (like overriding how addition works!). But this power and flexibility is what makes tools like **ggplot2** and **dplyr** possible. Learning how to make best use of this flexibility is beyond the scope of this book, but you can read about in *Advanced R*.

CHAPTER 16

Vectors

Introduction

So far this book has focused on tibbles and packages that work with them. But as you start to write your own functions, and dig deeper into R, you need to learn about vectors, the objects that underlie tibbles. If you've learned R in a more traditional way, you're probably already familiar with vectors, as most R resources start with vectors and work their way up to tibbles. I think it's better to start with tibbles because they're immediately useful, and then work your way down to the underlying components.

Vectors are particularly important as most of the functions you will write will work with vectors. It is possible to write functions that work with tibbles (like in **ggplot2**, **dplyr**, and **tidyr**), but the tools you need to write such functions are currently idiosyncratic and immature. I am working on a better approach, *https://github.com/hadley/lazyeval*, but it will not be ready in time for the publication of the book. Even when complete, you'll still need to understand vectors; it'll just make it easier to write a user-friendly layer on top.

Prerequisites

The focus of this chapter is on base R data structures, so it isn't essential to load any packages. We will, however, use a handful of functions from the **purrr** package to avoid some inconsistences in base R.

```
library(tidyverse)
#> Loading tidyverse: ggplot2
#> Loading tidyverse: tibble
#> Loading tidyverse: tidyr
#> Loading tidyverse: readr
#> Loading tidyverse: purrr
#> Loading tidyverse: dplyr
#> Conflicts with tidy packages ---------------------------------
#> filter(): dplyr, stats
#> lag():    dplyr, stats
```

Vector Basics

There are two types of vectors:

- *Atomic* vectors, of which there are six types: *logical*, *integer*, *double*, *character*, *complex*, and *raw*. Integer and double vectors are collectively known as *numeric* vectors.
- *Lists*, which are sometimes called recursive vectors because lists can contain other lists.

The chief difference between atomic vectors and lists is that atomic vectors are *homogeneous*, while lists can be *heterogeneous*. There's one other related object: `NULL`. `NULL` is often used to represent the absence of a vector (as opposed to `NA`, which is used to represent the absence of a value in a vector). `NULL` typically behaves like a vector of length 0. Figure 16-1 summarizes the interrelationships.

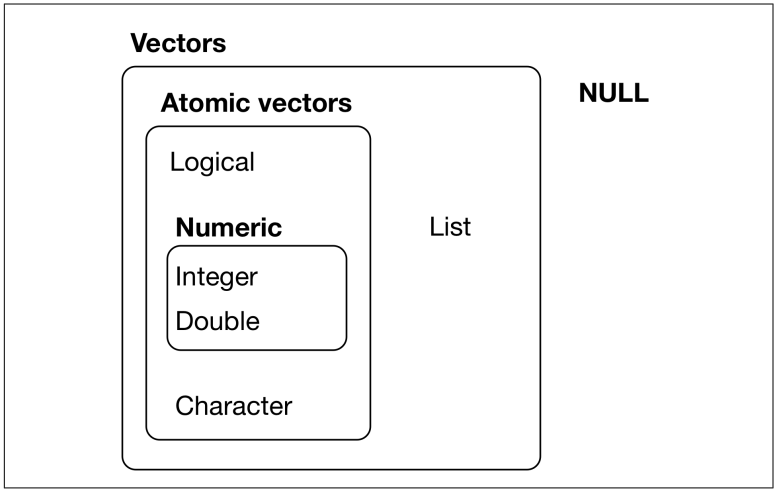


Figure 16-1. The hierarchy of R's vector types

Every vector has two key properties:

- Its *type*, which you can determine with `typeof()`:

```
typeof(letters)
#> [1] "character"
typeof(1:10)
#> [1] "integer"
```

- Its *length*, which you can determine with `length()`:

```
x <- list("a", "b", 1:10)
length(x)
#> [1] 3
```

Vectors can also contain arbitrary additional metadata in the form of attributes. These attributes are used to create *augmented vectors*, which build on additional behavior. There are four important types of augmented vector:

- Factors are built on top of integer vectors.
- Dates and date-times are built on top of numeric vectors.
- Data frames and tibbles are built on top of lists.

This chapter will introduce you to these important vectors from simplest to most complicated. You'll start with atomic vectors, then build up to lists, and finish off with augmented vectors.

Important Types of Atomic Vector

The four most important types of atomic vector are logical, integer, double, and character. Raw and complex are rarely used during a data analysis, so I won't discuss them here.

Logical

Logical vectors are the simplest type of atomic vector because they can take only three possible values: `FALSE`, `TRUE`, and `NA`. Logical vectors are usually constructed with comparison operators, as described in "Comparisons" on page 46. You can also create them by hand with `c()`:

```
1:10 %% 3 == 0
#>  [1] FALSE FALSE  TRUE FALSE FALSE
#>  [2] TRUE FALSE FALSE  TRUE FALSE
```

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

Numeric

Integer and double vectors are known collectively as numeric vectors. In R, numbers are doubles by default. To make an integer, place a `L` after the number:

```
typeof(1)
#> [1] "double"
typeof(1L)
#> [1] "integer"
1.5L
#> [1] 1.5
```

The distinction between integers and doubles is not usually important, but there are two important differences that you should be aware of:

- Doubles are approximations. Doubles represent floating-point numbers that cannot always be precisely represented with a fixed amount of memory. This means that you should consider all doubles to be approximations. For example, what is square of the square root of two?

  ```
  x <- sqrt(2) ^ 2
  x
  #> [1] 2
  x - 2
  #> [1] 4.44e-16
  ```

  This behavior is common when working with floating-point numbers: most calculations include some approximation error. Instead of comparing floating-point numbers using `==`, you should use `dplyr::near()`, which allows for some numerical tolerance.

- Integers have one special value, `NA`, while doubles have four, `NA`, `NaN`, `Inf`, and `-Inf`. All three special values can arise during division:

  ```
  c(-1, 0, 1) / 0
  #> [1] -Inf  NaN  Inf
  ```

  Avoid using `==` to check for these other special values. Instead use the helper functions `is.finite()`, `is.infinite()`, and `is.nan()`:

	0	Inf	NA	NaN
is.finite()	x			
is.infinite()		x		
is.na()			x	x
is.nan()				x

Character

Character vectors are the most complex type of atomic vector, because each element of a character vector is a string, and a string can contain an arbitrary amount of data.

You've already learned a lot about working with strings in Chapter 11. Here I want to mention one important feature of the underlying string implementation: R uses a global string pool. This means that each unique string is only stored in memory once, and every use of the string points to that representation. This reduces the amount of memory needed by duplicated strings. You can see this behavior in practice with `pryr::object_size()`:

```
x <- "This is a reasonably long string."
pryr::object_size(x)
#> 136 B

y <- rep(x, 1000)
pryr::object_size(y)
#> 8.13 kB
```

`y` doesn't take up 1000x as much memory as `x`, because each element of `y` is just a pointer to that same string. A pointer is 8 bytes, so 1000 pointers to a 136 B string is 8 * 1000 + 136 = 8.13 kB.

Missing Values

Note that each type of atomic vector has its own missing value:

```
NA            # logical
#> [1] NA
NA_integer_   # integer
#> [1] NA
NA_real_      # double
#> [1] NA
NA_character_ # character
#> [1] NA
```

Normally you don't need to know about these different types because you can always use `NA` and it will be converted to the correct type using the implicit coercion rules described next. However, there are some functions that are strict about their inputs, so it's useful to have this knowledge sitting in your back pocket so you can be specific when needed.

Exercises

1. Describe the difference between `is.finite(x)` and `!is.infinite(x)`.
2. Read the source code for `dplyr::near()` (Hint: to see the source code, drop the `()`). How does it work?
3. A logical vector can take three possible values. How many possible values can an integer vector take? How many possible values can a double take? Use Google to do some research.
4. Brainstorm at least four functions that allow you to convert a double to an integer. How do they differ? Be precise.
5. What functions from the **readr** package allow you to turn a string into a logical, integer, and double vector?

Using Atomic Vectors

Now that you understand the different types of atomic vector, it's useful to review some of the important tools for working with them. These include:

- How to convert from one type to another, and when that happens automatically.
- How to tell if an object is a specific type of vector.
- What happens when you work with vectors of different lengths.
- How to name the elements of a vector.
- How to pull out elements of interest.

Coercion

There are two ways to convert, or coerce, one type of vector to another:

- Explicit coercion happens when you call a function like `as.logical()`, `as.integer()`, `as.double()`, or `as.character()`. Whenever you find yourself using explicit coercion, you should always check whether you can make the fix upstream, so that the vector never had the wrong type in the first place. For example, you may need to tweak your **readr** `col_types` specification.
- Implicit coercion happens when you use a vector in a specific context that expects a certain type of vector. For example, when you use a logical vector with a numeric summary function, or when you use a double vector where an integer vector is expected.

Because explicit coercion is used relatively rarely, and is largely easy to understand, I'll focus on implicit coercion here.

You've already seen the most important type of implicit coercion: using a logical vector in a numeric context. In this case `TRUE` is converted to `1` and `FALSE` is converted to 0. That means the sum of a logical vector is the number of trues, and the mean of a logical vector is the proportion of trues:

```
x <- sample(20, 100, replace = TRUE)
y <- x > 10
sum(y)  # how many are greater than 10?
#> [1] 44
mean(y) # what proportion are greater than 10?
#> [1] 0.44
```

You may see some code (typically older) that relies on implicit coercion in the opposite direction, from integer to logical:

```
if (length(x)) {
  # do something
}
```

In this case, 0 is converted to `FALSE` and everything else is converted to `TRUE`. I think this makes it harder to understand your code, and I don't recommend it. Instead be explicit: `length(x) > 0`.

It's also important to understand what happens when you try and create a vector containing multiple types with `c()`—the most complex type always wins:

```
typeof(c(TRUE, 1L))
#> [1] "integer"
typeof(c(1L, 1.5))
```

```
#> [1] "double"
typeof(c(1.5, "a"))
#> [1] "character"
```

An atomic vector cannot have a mix of different types because the type is a property of the complete vector, not the individual elements. If you need to mix multiple types in the same vector, you should use a list, which you'll learn about shortly.

Test Functions

Sometimes you want to do different things based on the type of vector. One option is to use `typeof()`. Another is to use a test function that returns a `TRUE` or `FALSE`. Base R provides many functions like `is.vector()` and `is.atomic()`, but they often return surprising results. Instead, it's safer to use the `is_*` functions provided by **purrr**, which are summarized in the following table.

	lgl	int	dbl	chr	list
`is_logical()`	x				
`is_integer()`		x			
`is_double()`			x		
`is_numeric()`		x	x		
`is_character()`				x	
`is_atomic()`	x	x	x	x	
`is_list()`					x
`is_vector()`	x	x	x	x	x

Each predicate also comes with a "scalar" version, like `is_scalar_atomic()`, which checks that the length is 1. This is useful, for example, if you want to check that an argument to your function is a single logical value.

Scalars and Recycling Rules

As well as implicitly coercing the types of vectors to be compatible, R will also implicitly coerce the length of vectors. This is called vector *recycling*, because the shorter vector is repeated, or recycled, to the same length as the longer vector.

This is generally most useful when you are mixing vectors and "scalars." I put scalars in quotes because R doesn't actually have

scalars: instead, a single number is a vector of length 1. Because there are no scalars, most built-in functions are *vectorized*, meaning that they will operate on a vector of numbers. That's why, for example, this code works:

```
sample(10) + 100
#>  [1] 109 108 104 102 103 110 106 107 105 101
runif(10) > 0.5
#>  [1]  TRUE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE
#>  [10]  TRUE
```

In R, basic mathematical operations work with vectors. That means that you should never need to perform explicit iteration when performing simple mathematical computations.

It's intuitive what should happen if you add two vectors of the same length, or a vector and a "scalar," but what happens if you add two vectors of different lengths?

```
1:10 + 1:2
#>  [1]  2  4  4  6  6  8  8 10 10 12
```

Here, R will expand the shortest vector to the same length as the longest, so-called recycling. This is silent except when the length of the longer is not an integer multiple of the length of the shorter:

```
1:10 + 1:3
#> Warning in 1:10 + 1:3:
#> longer object length is not a multiple of shorter
#> object length
#>  [1]  2  4  6  5  7  9  8 10 12 11
```

While vector recycling can be used to create very succinct, clever code, it can also silently conceal problems. For this reason, the vectorized functions in tidyverse will throw errors when you recycle anything other than a scalar. If you do want to recycle, you'll need to do it yourself with `rep()`:

```
tibble(x = 1:4, y = 1:2)
#> Error: Variables must be length 1 or 4.
#> Problem variables: 'y'

tibble(x = 1:4, y = rep(1:2, 2))
#> # A tibble: 4 × 2
#>       x     y
#>   <int> <int>
#> 1     1     1
#> 2     2     2
#> 3     3     1
#> 4     4     2
```

```
tibble(x = 1:4, y = rep(1:2, each = 2))
#> # A tibble: 4 × 2
#>       x     y
#>   <int> <int>
#> 1     1     1
#> 2     2     1
#> 3     3     2
#> 4     4     2
```

Naming Vectors

All types of vectors can be named. You can name them during creation with `c()`:

```
c(x = 1, y = 2, z = 4)
#> x y z
#> 1 2 4
```

Or after the fact with `purrr::set_names()`:

```
set_names(1:3, c("a", "b", "c"))
#> a b c
#> 1 2 3
```

Named vectors are most useful for subsetting, described next.

Subsetting

So far we've used `dplyr::filter()` to filter the rows in a tibble. `filter()` only works with tibble, so we'll need a new tool for vectors: `[`. `[` is the subsetting function, and is called like `x[a]`. There are four types of things that you can subset a vector with:

- A numeric vector containing only integers. The integers must either be all positive, all negative, or zero.

  Subsetting with positive integers keeps the elements at those positions:

  ```
  x <- c("one", "two", "three", "four", "five")
  x[c(3, 2, 5)]
  #> [1] "three" "two"   "five"
  ```

  By repeating a position, you can actually make a longer output than input:

  ```
  x[c(1, 1, 5, 5, 5, 2)]
  #> [1] "one"  "one"  "five" "five" "five" "two"
  ```

Negative values drop the elements at the specified positions:

```
x[c(-1, -3, -5)]
#> [1] "two"  "four"
```

It's an error to mix positive and negative values:

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

The error message mentions subsetting with zero, which returns no values:

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

This is not useful very often, but it can be helpful if you want to create unusual data structures to test your functions with.

- Subsetting with a logical vector keeps all values corresponding to a `TRUE` value. This is most often useful in conjunction with the comparison functions:

```
x <- c(10, 3, NA, 5, 8, 1, NA)

# All non-missing values of x
x[!is.na(x)]
#> [1] 10  3  5  8  1

# All even (or missing!) values of x
x[x %% 2 == 0]
#> [1] 10 NA  8 NA
```

- If you have a named vector, you can subset it with a character vector:

```
x <- c(abc = 1, def = 2, xyz = 5)
x[c("xyz", "def")]
#> xyz def
#>   5   2
```

Like with positive integers, you can also use a character vector to duplicate individual entries.

- The simplest type of subsetting is nothing, `x[]`, which returns the complete `x`. This is not useful for subsetting vectors, but it is useful when subsetting matrices (and other high-dimensional structures) because it lets you select all the rows or all the columns, by leaving that index blank. For example, if `x` is 2D,

`x[1, ]` selects the first row and all the columns, and `x[, -1]` selects all rows and all columns except the first.

To learn more about the applications of subsetting, read the "Subsetting" chapter of *Advanced R*.

There is an important variation of `[` called `[[`. `[[` only ever extracts a single element, and always drops names. It's a good idea to use it whenever you want to make it clear that you're extracting a single item, as in a for loop. The distinction between `[` and `[[` is most important for lists, as we'll see shortly.

Exercises

1. What does `mean(is.na(x))` tell you about a vector `x`? What about `sum(!is.finite(x))`?
2. Carefully read the documentation of `is.vector()`. What does it actually test for? Why does `is.atomic()` not agree with the definition of atomic vectors above?
3. Compare and contrast `setNames()` with `purrr::set_names()`.
4. Create functions that take a vector as input and return:
   a. The last value. Should you use `[` or `[[`?
   b. The elements at even numbered positions.
   c. Every element except the last value.
   d. Only even numbers (and no missing values).
5. Why is `x[-which(x > 0)]` not the same as `x[x <= 0]`?
6. What happens when you subset with a positive integer that's bigger than the length of the vector? What happens when you subset with a name that doesn't exist?

Recursive Vectors (Lists)

Lists are a step up in complexity from atomic vectors, because lists can contain other lists. This makes them suitable for representing hierarchical or tree-like structures. You create a list with `list()`:

```
x <- list(1, 2, 3)
x
#> [[1]]
```

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

A very useful tool for working with lists is `str()` because it focuses on the *str*ucture, not the contents:

```
str(x)
#> List of 3
#>  $ : num 1
#>  $ : num 2
#>  $ : num 3

x_named <- list(a = 1, b = 2, c = 3)
str(x_named)
#> List of 3
#>  $ a: num 1
#>  $ b: num 2
#>  $ c: num 3
```

Unlike atomic vectors, `lists()` can contain a mix of objects:

```
y <- list("a", 1L, 1.5, TRUE)
str(y)
#> List of 4
#>  $ : chr "a"
#>  $ : int 1
#>  $ : num 1.5
#>  $ : logi TRUE
```

Lists can even contain other lists!

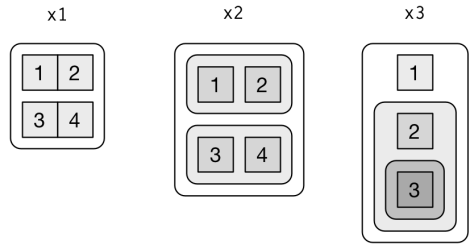
```
z <- list(list(1, 2), list(3, 4))
str(z)
#> List of 2
#>  $ :List of 2
#>   ..$ : num 1
#>   ..$ : num 2
#>  $ :List of 2
#>   ..$ : num 3
#>   ..$ : num 4
```

Visualizing Lists

To explain more complicated list manipulation functions, it's helpful to have a visual representation of lists. For example, take these three lists:

```
x1 <- list(c(1, 2), c(3, 4))
x2 <- list(list(1, 2), list(3, 4))
x3 <- list(1, list(2, list(3)))
```

I'll draw them as follows:



There are three principles:

- Lists have rounded corners. Atomic vectors have square corners.
- Children are drawn inside their parent, and have a slightly darker background to make it easier to see the hierarchy.
- The orientation of the children (i.e., rows or columns) isn't important, so I'll pick a row or column orientation to either save space or illustrate an important property in the example.

Subsetting

There are three ways to subset a list, which I'll illustrate with `a`:

```
a <- list(a = 1:3, b = "a string", c = pi, d = list(-1, -5))
```

- `[` extracts a sublist. The result will always be a list:

  ```
  str(a[1:2])
  #> List of 2
  #>  $ a: int [1:3] 1 2 3
  #>  $ b: chr "a string"
  str(a[4])
  #> List of 1
  #>  $ d:List of 2
  #>   ..$ : num -1
  #>   ..$ : num -5
  ```

  Like with vectors, you can subset with a logical, integer, or character vector.

- `[[` extracts a single component from a list. It removes a level of hierarchy from the list:

```
str(y[[1]])
#>  chr "a"
str(y[[4]])
#>  logi TRUE
```

- `$` is a shorthand for extracting named elements of a list. It works similarly to `[[` except that you don't need to use quotes:

```
a$a
#> [1] 1 2 3
a[["a"]]
#> [1] 1 2 3
```

The distinction between `[` and `[[` is really important for lists, because `[[` drills down into the list while `[` returns a new, smaller list. Compare the preceding code and output with the visual representation in Figure 16-2.

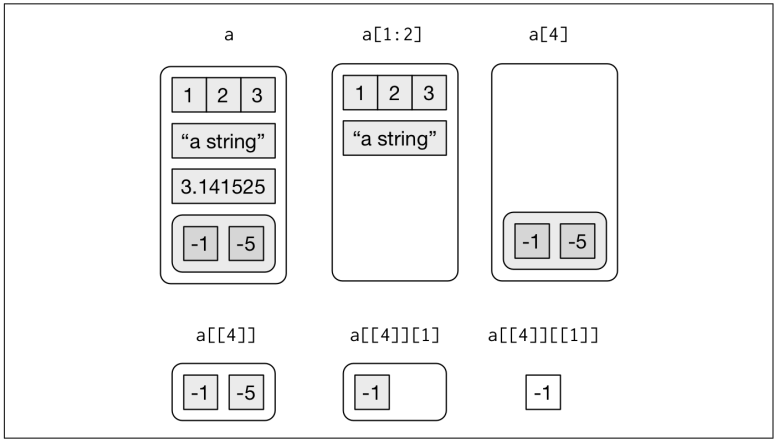


Figure 16-2. Subsetting a list, visually

Lists of Condiments

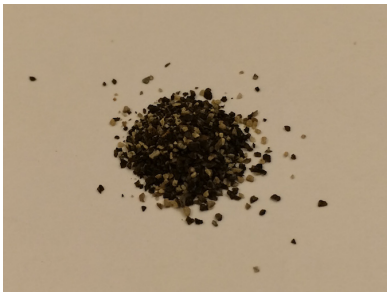
The difference between `[` and `[[` is very important, but it's easy to get confused. To help you remember, let me show you an unusual pepper shaker:

If this pepper shaker is your list `x`, then, `x[1]` is a pepper shaker containing a single pepper packet:

`x[2]` would look the same, but would contain the second packet. `x[1:2]` would be a pepper shaker containing two pepper packets.

`x[[1]]` is:

If you wanted to get the content of the pepper package, you'd need `x[[1]][[1]]`:



Exercises

1. Draw the following lists as nested sets:
   a. `list(a, b, list(c, d), list(e, f))`
   b. `list(list(list(list(list(list(a))))))`
2. What happens if you subset a tibble as if you're subsetting a list? What are the key differences between a list and a tibble?

Attributes

Any vector can contain arbitrary additional metadata through its *attributes*. You can think of attributes as a named list of vectors that can be attached to any object. You can get and set individual

attribute values with `attr()` or see them all at once with `attributes()`:

```
x <- 1:10
attr(x, "greeting")
#> NULL
attr(x, "greeting") <- "Hi!"
attr(x, "farewell") <- "Bye!"
attributes(x)
#> $greeting
#> [1] "Hi!"
#>
#> $farewell
#> [1] "Bye!"
```

There are three very important attributes that are used to implement fundamental parts of R:

- *Names* are used to name the elements of a vector.
- *Dimensions* (dims, for short) make a vector behave like a matrix or array.
- *Class* is used to implement the S3 object-oriented system.

You've seen names earlier, and we won't cover dimensions because we don't use matrices in this book. It remains to describe the class, which controls how *generic functions* work. Generic functions are key to object-oriented programming in R, because they make functions behave differently for different classes of input. A detailed discussion of object-oriented programming is beyond the scope of this book, but you can read more about it in *Advanced R*.

Here's what a typical generic function looks like:

```
as.Date
#> function (x, ...)
#> UseMethod("as.Date")
#> <bytecode: 0x7fa61e0590d8>
#> <environment: namespace:base>
```

The call to "UseMethod" means that this is a generic function, and it will call a specific *method*, a function, based on the class of the first argument. (All methods are functions; not all functions are methods.) You can list all the methods for a generic with `methods()`:

```
methods("as.Date")
#> [1] as.Date.character as.Date.date      as.Date.dates
#> [4] as.Date.default as.Date.factor    as.Date.numeric
#> [7] as.Date.POSIXct   as.Date.POSIXlt
#> see '?methods' for accessing help and source code
```

For example, if `x` is a character vector, `as.Date()` will call `as.Date.character()`; if it's a factor, it'll call `as.Date.factor()`.

You can see the specific implementation of a method with `getS3method()`:

```
getS3method("as.Date", "default")
#> function (x, ...)
#> {
#>     if (inherits(x, "Date"))
#>         return(x)
#>     if (is.logical(x) && all(is.na(x)))
#>         return(structure(as.numeric(x), class = "Date"))
#>     stop(
#>       gettextf("do not know how to convert '%s' to class %s",
#>       deparse(substitute(x)), dQuote("Date")), domain = NA)
#> }
#> <bytecode: 0x7fa61dd47e78>
#> <environment: namespace:base>
getS3method("as.Date", "numeric")
#> function (x, origin, ...)
#> {
#>     if (missing(origin))
#>         stop("'origin' must be supplied")
#>     as.Date(origin, ...) + x
#> }
#> <bytecode: 0x7fa61dd463b8>
#> <environment: namespace:base>
```

The most important S3 generic is `print()`: it controls how the object is printed when you type its name at the console. Other important generics are the subsetting functions `[`, `[[`, and `$`.

Augmented Vectors

Atomic vectors and lists are the building blocks for other important vector types like factors and dates. I call these *augmented vectors*, because they are vectors with additional *attributes*, including class. Because augmented vectors have a class, they behave differently to the atomic vector on which they are built. In this book, we make use of four important augmented vectors:

- Factors
- Date-times and times
- Tibbles

These are described next.

Factors

Factors are designed to represent categorical data that can take a fixed set of possible values. Factors are built on top of integers, and have a levels attribute:

```
x <- factor(c("ab", "cd", "ab"), levels = c("ab", "cd", "ef"))
typeof(x)
#> [1] "integer"
attributes(x)
#> $levels
#> [1] "ab" "cd" "ef"
#>
#> $class
#> [1] "factor"
```

Dates and Date-Times

Dates in R are numeric vectors that represent the number of days since 1 January 1970:

```
x <- as.Date("1971-01-01")
unclass(x)
#> [1] 365

typeof(x)
#> [1] "double"
attributes(x)
#> $class
#> [1] "Date"
```

Date-times are numeric vectors with class `POSIXct` that represent the number of seconds since 1 January 1970. (In case you were wondering, "POSIXct" stands for "Portable Operating System Interface," calendar time.)

```
x <- lubridate::ymd_hm("1970-01-01 01:00")
unclass(x)
#> [1] 3600
#> attr(,"tzone")
#> [1] "UTC"
```

```
typeof(x)
#> [1] "double"
attributes(x)
#> $tzone
#> [1] "UTC"
#>
#> $class
#> [1] "POSIXct" "POSIXt"
```

The `tzone` attribute is optional. It controls how the time is printed, not what absolute time it refers to:

```
attr(x, "tzone") <- "US/Pacific"
x
#> [1] "1969-12-31 17:00:00 PST"

attr(x, "tzone") <- "US/Eastern"
x
#> [1] "1969-12-31 20:00:00 EST"
```

There is another type of date-times called `POSIXlt`. These are built on top of named lists:

```
y <- as.POSIXlt(x)
typeof(y)
#> [1] "list"
attributes(y)
#> $names
#>  [1] "sec"    "min"    "hour"   "mday"   "mon"    "year"
#>  [7] "wday"   "yday"   "isdst"  "zone"   "gmtoff"
#>
#> $class
#> [1] "POSIXlt" "POSIXt"
#>
#> $tzone
#> [1] "US/Eastern" "EST"        "EDT"
```

`POSIXlt`s are rare inside the tidyverse. They do crop up in base R, because they are needed to extract specific components of a date, like the year or month. Since **lubridate** provides helpers for you to do this instead, you don't need them. `POSIXct`'s are always easier to work with, so if you find you have a `POSIXlt`, you should always convert it to a regular date-time with `lubridate::as_date_time()`.

Tibbles

Tibbles are augmented lists. They have three classes: `tbl_df`, `tbl`, and `data.frame`. They have two attributes: (column) `names` and `row.names`.

```
tb <- tibble::tibble(x = 1:5, y = 5:1)
typeof(tb)
#> [1] "list"
attributes(tb)
#> $names
#> [1] "x" "y"
#>
#> $class
#> [1] "tbl_df"     "tbl"        "data.frame"
#>
#> $row.names
#> [1] 1 2 3 4 5
```

Traditional `data.frames` have a very similar structure:

```
df <- data.frame(x = 1:5, y = 5:1)
typeof(df)
#> [1] "list"
attributes(df)
#> $names
#> [1] "x" "y"
#>
#> $row.names
#> [1] 1 2 3 4 5
#>
#> $class
#> [1] "data.frame"
```

The main difference is the class. The class of tibble includes "data.frame," which means tibbles inherit the regular data frame behavior by default.

The difference between a tibble or a data frame and a list is that all of the elements of a tibble or data frame must be vectors with the same length. All functions that work with tibbles enforce this constraint.

Exercises

1. What does `hms::hms(3600)` return? How does it print? What primitive type is the augmented vector built on top of? What attributes does it use?
2. Try and make a tibble that has columns with different lengths. What happens?
3. Based of the previous definition, is it OK to have a list as a column of a tibble?

CHAPTER 17

Iteration with purrr

Introduction

In Chapter 15, we talked about how important it is to reduce duplication in your code by creating functions instead of copying and pasting. Reducing code duplication has three main benefits:

- It's easier to see the intent of your code, because your eyes are drawn to what's different, not what stays the same.
- It's easier to respond to changes in requirements. As your needs change, you only need to make changes in one place, rather than remembering to change every place that you copied and pasted the code.
- You're likely to have fewer bugs because each line of code is used in more places.

One tool for reducing duplication is functions, which reduce duplication by identifying repeated patterns of code and extracting them out into independent pieces that can be easily reused and updated. Another tool for reducing duplication is *iteration*, which helps you when you need to do the same thing to multiple inputs: repeating the same operation on different columns, or on different datasets. In this chapter you'll learn about two important iteration paradigms: imperative programming and functional programming. On the imperative side you have tools like for loops and while loops, which are a great place to start because they make iteration very explicit, so it's obvious what's happening. However, for loops are quite verbose,

and require quite a bit of bookkeeping code that is duplicated for every for loop. Functional programming (FP) offers tools to extract out this duplicated code, so each common for loop pattern gets its own function. Once you master the vocabulary of FP, you can solve many common iteration problems with less code, more ease, and fewer errors.

Prerequisites

Once you've mastered the for loops provided by base R, you'll learn some of the powerful programming tools provided by **purrr**, one of the tidyverse core packages.

```
library(tidyverse)
```

For Loops

Imagine we have this simple tibble:

```
df <- tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)
```

We want to compute the median of each column. You *could* do it with copy-and-paste:

```
median(df$a)
#> [1] -0.246
median(df$b)
#> [1] -0.287
median(df$c)
#> [1] -0.0567
median(df$d)
#> [1] 0.144
```

But that breaks our rule of thumb: never copy and paste more than twice. Instead, we could use a for loop:

```
output <- vector("double", ncol(df))  # 1. output
for (i in seq_along(df)) {            # 2. sequence
  output[[i]] <- median(df[[i]])      # 3. body
}
output
#> [1] -0.2458 -0.2873 -0.0567  0.1443
```

Every for loop has three components:

output `output <- vector("double", length(x))`
: Before you start the loop, you must always allocate sufficient space for the output. This is very important for efficiency: if you grow the for loop at each iteration using `c()` (for example), your for loop will be very slow.

  A general way of creating an empty vector of given length is the `vector()` function. It has two arguments: the type of the vector ("logical," "integer," "double," "character," etc.) and the length of the vector.

sequence `i in seq_along(df)`
: This determines what to loop over: each run of the for loop will assign `i` to a different value from `seq_along(df)`. It's useful to think of `i` as a pronoun, like "it."

  You might not have seen `seq_along()` before. It's a safe version of the familiar `1:length(l)`, with an important difference; if you have a zero-length vector, `seq_along()` does the right thing:

```
y <- vector("double", 0)
seq_along(y)
#> integer(0)
1:length(y)
#> [1] 1 0
```

  You probably won't create a zero-length vector deliberately, but it's easy to create them accidentally. If you use `1:length(x)` instead of `seq_along(x)`, you're likely to get a confusing error message.

body `output[[i]] <- median(df[[i]])`
: This is the code that does the work. It's run repeatedly, each time with a different value for `i`. The first iteration will run `output[[1]] <- median(df[[1]])`, the second will run `output[[2]] <- median(df[[2]])`, and so on.

That's all there is to the for loop! Now is a good time to practice creating some basic (and not so basic) for loops using the following exercises. Then we'll move on to some variations of the for loop that help you solve other problems that will crop up in practice.

Exercises

1. Write for loops to:
   a. Compute the mean of every column in `mtcars`.
   b. Determine the type of each column in `nyc flights13::flights`.
   c. Compute the number of unique values in each column of `iris`.
   d. Generate 10 random normals for each of $\mu = -10$, 0, 10, and 100.

   Think about the output, sequence, and body *before* you start writing the loop.
2. Eliminate the for loop in each of the following examples by taking advantage of an existing function that works with vectors:

```
out <- ""
for (x in letters) {
  out <- stringr::str_c(out, x)
}

x <- sample(100)
sd <- 0
for (i in seq_along(x)) {
  sd <- sd + (x[i] - mean(x)) ^ 2
}
sd <- sqrt(sd / (length(x) - 1))

x <- runif(100)
out <- vector("numeric", length(x))
out[1] <- x[1]
for (i in 2:length(x)) {
  out[i] <- out[i - 1] + x[i]
}
```

3. Combine your function writing and for loop skills:
   a. Write a for loop that `prints()` the lyrics to the children's song "Alice the Camel."
   b. Convert the nursery rhyme "Ten in the Bed" to a function. Generalize it to any number of people in any sleeping structure.

c. Convert the song "99 Bottles of Beer on the Wall" to a function. Generalize to any number of any vessel containing any liquid on any surface.

4. It's common to see for loops that don't preallocate the output and instead increase the length of a vector at each step:

```
output <- vector("integer", 0)
for (i in seq_along(x)) {
  output <- c(output, lengths(x[[i]]))
}
output
```

How does this affect performance? Design and execute an experiment.

For Loop Variations

Once you have the basic for loop under your belt, there are some variations that you should be aware of. These variations are important regardless of how you do iteration, so don't forget about them once you've mastered the FP techniques you'll learn about in the next section.

There are four variations on the basic theme of the for loop:

- Modifying an existing object, instead of creating a new object.
- Looping over names or values, instead of indices.
- Handling outputs of unknown length.
- Handling sequences of unknown length.

Modifying an Existing Object

Sometimes you want to use a for loop to modify an existing object. For example, remember our challenge from Chapter 15. We wanted to rescale every column in a data frame:

```
df <- tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)
rescale01 <- function(x) {
```

```
  rng <- range(x, na.rm = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}

df$a <- rescale01(df$a)
df$b <- rescale01(df$b)
df$c <- rescale01(df$c)
df$d <- rescale01(df$d)
```

To solve this with a for loop we again think about the three components:

Output
: We already have the output—it's the same as the input!

Sequence
: We can think about a data frame as a list of columns, so we can iterate over each column with `seq_along(df)`.

Body
: Apply `rescale01()`.

This gives us:

```
for (i in seq_along(df)) {
  df[[i]] <- rescale01(df[[i]])
}
```

Typically you'll be modifying a list or data frame with this sort of loop, so remember to use `[[`, not `[`. You might have spotted that I used `[[` in all my for loops: I think it's better to use `[[` even for atomic vectors because it makes it clear that I want to work with a single element.

Looping Patterns

There are three basic ways to loop over a vector. So far I've shown you the most general: looping over the numeric indices with `for (i in seq_along(xs))`, and extracting the value with `x[[i]]`. There are two other forms:

- Loop over the elements: `for (x in xs)`. This is most useful if you only care about side effects, like plotting or saving a file, because it's difficult to save the output efficiently.
- Loop over the names: `for (nm in names(xs))`. This gives you a name, which you can use to access the value with `x[[nm]]`. This is useful if you want to use the name in a plot title or a filename.

If you're creating named output, make sure to name the results vector like so:

```
results <- vector("list", length(x))
names(results) <- names(x)
```

Iteration over the numeric indices is the most general form, because given the position you can extract both the name and the value:

```
for (i in seq_along(x)) {
  name <- names(x)[[i]]
  value <- x[[i]]
}
```

Unknown Output Length

Sometimes you might not know how long the output will be. For example, imagine you want to simulate some random vectors of random lengths. You might be tempted to solve this problem by progressively growing the vector:

```
means <- c(0, 1, 2)

output <- double()
for (i in seq_along(means)) {
  n <- sample(100, 1)
  output <- c(output, rnorm(n, means[[i]]))
}
str(output)
#>  num [1:202] 0.912 0.205 2.584 -0.789 0.588 ...
```

But this is not very efficient because in each iteration, R has to copy all the data from the previous iterations. In technical terms you get "quadratic" ($O(n^2)$) behavior, which means that a loop with three times as many elements would take nine (3^2) times as long to run.

A better solution is to save the results in a list, and then combine into a single vector after the loop is done:

```
out <- vector("list", length(means))
for (i in seq_along(means)) {
  n <- sample(100, 1)
  out[[i]] <- rnorm(n, means[[i]])
}
str(out)
#> List of 3
#>  $ : num [1:83] 0.367 1.13 -0.941 0.218 1.415 ...
#>  $ : num [1:21] -0.485 -0.425 2.937 1.688 1.324 ...
#>  $ : num [1:40] 2.34 1.59 2.93 3.84 1.3 ...
```

```
str(unlist(out))
#>  num [1:144] 0.367 1.13 -0.941 0.218 1.415 ...
```

Here I've used `unlist()` to flatten a list of vectors into a single vector. A stricter option is to use `purrr::flatten_dbl()`—it will throw an error if the input isn't a list of doubles.

This pattern occurs in other places too:

- You might be generating a long string. Instead of `paste()`ing together each iteration with the previous, save the output in a character vector and then combine that vector into a single string with `paste(output, collapse = "")`.
- You might be generating a big data frame. Instead of sequentially `rbind()`ing in each iteration, save the output in a list, then use `dplyr::bind_rows(output)` to combine the output into a single data frame.

Watch out for this pattern. Whenever you see it, switch to a more complex result object, and then combine in one step at the end.

Unknown Sequence Length

Sometimes you don't even know how long the input sequence should be. This is common when doing simulations. For example, you might want to loop until you get three heads in a row. You can't do that sort of iteration with the for loop. Instead, you can use a while loop. A while loop is simpler than a for loop because it only has two components, a condition and a body:

```
while (condition) {
  # body
}
```

A while loop is also more general than a for loop, because you can rewrite any for loop as a while loop, but you can't rewrite every while loop as a for loop:

```
for (i in seq_along(x)) {
  # body
}

# Equivalent to
i <- 1
while (i <= length(x)) {
  # body
```

```
  i <- i + 1
}
```

Here's how we could use a while loop to find how many tries it takes to get three heads in a row:

```
flip <- function() sample(c("T", "H"), 1)

flips <- 0
nheads <- 0

while (nheads < 3) {
  if (flip() == "H") {
    nheads <- nheads + 1
  } else {
    nheads <- 0
  }
  flips <- flips + 1
}
flips
#> [1] 3
```

I mention while loops only briefly, because I hardly ever use them. They're most often used for simulation, which is outside the scope of this book. However, it is good to know they exist so that you're prepared for problems where the number of iterations is not known in advance.

Exercises

1. Imagine you have a directory full of CSV files that you want to read in. You have their paths in a vector, `files <- dir("data/", pattern = "\\.csv$", full.names = TRUE)`, and now want to read each one with `read_csv()`. Write the for loop that will load them into a single data frame.
2. What happens if you use `for (nm in names(x))` and `x` has no names? What if only some of the elements are named? What if the names are not unique?
3. Write a function that prints the mean of each numeric column in a data frame, along with its name. For example, `show_mean(iris)` would print:

```
show_mean(iris)
#> Sepal.Length: 5.84
#> Sepal.Width:  3.06
#> Petal.Length: 3.76
#> Petal.Width:  1.20
```

(Extra challenge: what function did I use to make sure that the numbers lined up nicely, even though the variable names had different lengths?)

4. What does this code do? How does it work?

```
trans <- list(
  disp = function(x) x * 0.0163871,
  am = function(x) {
    factor(x, labels = c("auto", "manual"))
  }
)
for (var in names(trans)) {
  mtcars[[var]] <- trans[[var]](mtcars[[var]])
}
```

For Loops Versus Functionals

For loops are not as important in R as they are in other languages because R is a functional programming language. This means that it's possible to wrap up for loops in a function, and call that function instead of using the for loop directly.

To see why this is important, consider (again) this simple data frame:

```
df <- tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)
```

Imagine you want to compute the mean of every column. You could do that with a for loop:

```
output <- vector("double", length(df))
for (i in seq_along(df)) {
  output[[i]] <- mean(df[[i]])
}
output
#> [1]  0.2026 -0.2068  0.1275 -0.0917
```

You realize that you're going to want to compute the means of every column pretty frequently, so you extract it out into a function:

```
col_mean <- function(df) {
  output <- vector("double", length(df))
  for (i in seq_along(df)) {
```

```
    output[i] <- mean(df[[i]])
  }
  output
}
```

But then you think it'd also be helpful to be able to compute the median, and the standard deviation, so you copy and paste your `col_mean()` function and replace the `mean()` with `median()` and `sd()`:

```
col_median <- function(df) {
  output <- vector("double", length(df))
  for (i in seq_along(df)) {
    output[i] <- median(df[[i]])
  }
  output
}
col_sd <- function(df) {
  output <- vector("double", length(df))
  for (i in seq_along(df)) {
    output[i] <- sd(df[[i]])
  }
  output
}
```

Uh oh! You've copied and pasted this code twice, so it's time to think about how to generalize it. Notice that most of this code is for-loop boilerplate and it's hard to see the one thing (`mean()`, `median()`, `sd()`) that is different between the functions.

What would you do if you saw a set of functions like this?

```
f1 <- function(x) abs(x - mean(x)) ^ 1
f2 <- function(x) abs(x - mean(x)) ^ 2
f3 <- function(x) abs(x - mean(x)) ^ 3
```

Hopefully, you'd notice that there's a lot of duplication, and extract it out into an additional argument:

```
f <- function(x, i) abs(x - mean(x)) ^ i
```

You've reduced the chance of bugs (because you now have 1/3 less code), and made it easy to generalize to new situations.

We can do exactly the same thing with `col_mean()`, `col_median()`, and `col_sd()` by adding an argument that supplies the function to apply to each column:

```
col_summary <- function(df, fun) {
  out <- vector("double", length(df))
  for (i in seq_along(df)) {
```

```
    out[i] <- fun(df[[i]])
  }
  out
}
col_summary(df, median)
#> [1]  0.237 -0.218  0.254 -0.133
col_summary(df, mean)
#> [1]  0.2026 -0.2068  0.1275 -0.0917
```

The idea of passing a function to another function is an extremely powerful idea, and it's one of the behaviors that makes R a functional programming language. It might take you a while to wrap your head around the idea, but it's worth the investment. In the rest of the chapter, you'll learn about and use the **purrr** package, which provides functions that eliminate the need for many common for loops. The apply family of functions in base R (`apply()`, `lapply()`, `tapply()`, etc.) solve a similar problem, but **purrr** is more consistent and thus is easier to learn.

The goal of using **purrr** functions instead of for loops is to allow you to break common list manipulation challenges into independent pieces:

- How can you solve the problem for a single element of the list? Once you've solved that problem, **purrr** takes care of generalizing your solution to every element in the list.
- If you're solving a complex problem, how can you break it down into bite-sized pieces that allow you to advance one small step toward a solution? With **purrr**, you get lots of small pieces that you can compose together with the pipe.

This structure makes it easier to solve new problems. It also makes it easier to understand your solutions to old problems when you reread your old code.

Exercises

1. Read the documentation for `apply()`. In the second case, what two for loops does it generalize?
2. Adapt `col_summary()` so that it only applies to numeric columns. You might want to start with an `is_numeric()` function that returns a logical vector that has a `TRUE` corresponding to each numeric column.

The Map Functions

The pattern of looping over a vector, doing something to each element, and saving the results is so common that the **purrr** package provides a family of functions to do it for you. There is one function for each type of output:

- `map()` makes a list.
- `map_lgl()` makes a logical vector.
- `map_int()` makes an integer vector.
- `map_dbl()` makes a double vector.
- `map_chr()` makes a character vector.

Each function takes a vector as input, applies a function to each piece, and then returns a new vector that's the same length (and has the same names) as the input. The type of the vector is determined by the suffix to the map function.

Once you master these functions, you'll find it takes much less time to solve iteration problems. But you should never feel bad about using a for loop instead of a map function. The map functions are a step up a tower of abstraction, and it can take a long time to get your head around how they work. The important thing is that you solve the problem that you're working on, not write the most concise and elegant code (although that's definitely something you want to strive toward!).

Some people will tell you to avoid for loops because they are slow. They're wrong! (Well at least they're rather out of date, as for loops haven't been slow for many years). The chief benefit of using functions like `map()` is not speed, but clarity: they make your code easier to write and to read.

We can use these functions to perform the same computations as the last for loop. Those summary functions returned doubles, so we need to use `map_dbl()`:

```
map_dbl(df, mean)
#>       a       b       c       d
#>  0.2026 -0.2068  0.1275 -0.0917
map_dbl(df, median)
#>      a      b      c      d
#>  0.237 -0.218  0.254 -0.133
```

```
map_dbl(df, sd)
#>     a     b     c     d
#> 0.796 0.759 1.164 1.062
```

Compared to using a for loop, focus is on the operation being performed (i.e., `mean()`, `median()`, `sd()`), not the bookkeeping required to loop over every element and store the output. This is even more apparent if we use the pipe:

```
df %>% map_dbl(mean)
#>       a       b       c       d
#>  0.2026 -0.2068  0.1275 -0.0917
df %>% map_dbl(median)
#>      a      b      c      d
#>  0.237 -0.218  0.254 -0.133
df %>% map_dbl(sd)
#>     a     b     c     d
#> 0.796 0.759 1.164 1.062
```

There are a few differences between `map_*()` and `col_summary()`:

- All **purrr** functions are implemented in C. This makes them a little faster at the expense of readability.
- The second argument, `.f`, the function to apply, can be a formula, a character vector, or an integer vector. You'll learn about those handy shortcuts in the next section.
- `map_*()` uses ... ("Dot-Dot-Dot (…)" on page 284) to pass along additional arguments to `.f` each time it's called:

  ```
  map_dbl(df, mean, trim = 0.5)
  #>      a      b      c      d
  #>  0.237 -0.218  0.254 -0.133
  ```

- The map functions also preserve names:

  ```
  z <- list(x = 1:3, y = 4:5)
  map_int(z, length)
  #> x y
  #> 3 2
  ```

Shortcuts

There are a few shortcuts that you can use with `.f` in order to save a little typing. Imagine you want to fit a linear model to each group in a dataset. The following toy example splits up the `mtcars` dataset into three pieces (one for each value of cylinder) and fits the same linear model to each piece:

```
models <- mtcars %>%
  split(.$cyl) %>%
  map(function(df) lm(mpg ~ wt, data = df))
```

The syntax for creating an anonymous function in R is quite verbose so **purrr** provides a convenient shortcut—a one-sided formula:

```
models <- mtcars %>%
  split(.$cyl) %>%
  map(~lm(mpg ~ wt, data = .))
```

Here I've used `.` as a pronoun: it refers to the current list element (in the same way that `i` referred to the current index in the for loop).

When you're looking at many models, you might want to extract a summary statistic like the R^2. To do that we need to first run `summary()` and then extract the component called `r.squared`. We could do that using the shorthand for anonymous functions:

```
models %>%
  map(summary) %>%
  map_dbl(~.$r.squared)
#>     4     6     8
#> 0.509 0.465 0.423
```

But extracting named components is a common operation, so **purrr** provides an even shorter shortcut: you can use a string.

```
models %>%
  map(summary) %>%
  map_dbl("r.squared")
#>     4     6     8
#> 0.509 0.465 0.423
```

You can also use an integer to select elements by position:

```
x <- list(list(1, 2, 3), list(4, 5, 6), list(7, 8, 9))
x %>% map_dbl(2)
#> [1] 2 5 8
```

Base R

If you're familiar with the apply family of functions in base R, you might have noticed some similarities with the **purrr** functions:

- `lapply()` is basically identical to `map()`, except that `map()` is consistent with all the other functions in **purrr**, and you can use the shortcuts for `.f`.

- Base `sapply()` is a wrapper around `lapply()` that automatically simplifies the output. This is useful for interactive work but is problematic in a function because you never know what sort of output you'll get:

```
x1 <- list(
  c(0.27, 0.37, 0.57, 0.91, 0.20),
  c(0.90, 0.94, 0.66, 0.63, 0.06),
  c(0.21, 0.18, 0.69, 0.38, 0.77)
)
x2 <- list(
  c(0.50, 0.72, 0.99, 0.38, 0.78),
  c(0.93, 0.21, 0.65, 0.13, 0.27),
  c(0.39, 0.01, 0.38, 0.87, 0.34)
)

threshold <- function(x, cutoff = 0.8) x[x > cutoff]
x1 %>% sapply(threshold) %>% str()
#> List of 3
#>  $ : num 0.91
#>  $ : num [1:2] 0.9 0.94
#>  $ : num(0)
x2 %>% sapply(threshold) %>% str()
#>  num [1:3] 0.99 0.93 0.87
```

- `vapply()` is a safe alternative to `sapply()` because you supply an additional argument that defines the type. The only problem with `vapply()` is that it's a lot of typing: `vapply(df, is.numeric, logical(1))` is equivalent to `map_lgl(df, is.numeric)`. One advantage of `vapply()` over **purrr**'s map functions is that it can also produce matrices—the map functions only ever produce vectors.

I focus on **purrr** functions here because they have more consistent names and arguments, helpful shortcuts, and in the future will provide easy parallelism and progress bars.

Exercises

1. Write code that uses one of the map functions to:
   a. Compute the mean of every column in `mtcars`.
   b. Determine the type of each column in `nyc flights13::flights`.

c. Compute the number of unique values in each column of `iris`.

d. Generate 10 random normals for each of $\mu = -10$, 0, 10, and 100.

2. How can you create a single vector that for each column in a data frame indicates whether or not it's a factor?
3. What happens when you use the map functions on vectors that aren't lists? What does `map(1:5, runif)` do? Why?
4. What does `map(-2:2, rnorm, n = 5)` do? Why? What does `map_dbl(-2:2, rnorm, n = 5)` do? Why?
5. Rewrite `map(x, function(df) lm(mpg ~ wt, data = df))` to eliminate the anonymous function.

Dealing with Failure

When you use the map functions to repeat many operations, the chances are much higher that one of those operations will fail. When this happens, you'll get an error message, and no output. This is annoying: why does one failure prevent you from accessing all the other successes? How do you ensure that one bad apple doesn't ruin the whole barrel?

In this section you'll learn how to deal with this situation with a new function: `safely()`. `safely()` is an adverb: it takes a function (a verb) and returns a modified version. In this case, the modified function will never throw an error. Instead, it always returns a list with two elements:

`result`
: The original result. If there was an error, this will be `NULL`.

`error`
: An error object. If the operation was successful, this will be `NULL`.

(You might be familiar with the `try()` function in base R. It's similar, but because it sometimes returns the original result and it sometimes returns an error object it's more difficult to work with.)

Let's illustrate this with a simple example, `log()`:

```
safe_log <- safely(log)
str(safe_log(10))
#> List of 2
#>  $ result: num 2.3
#>  $ error : NULL
str(safe_log("a"))
#> List of 2
#>  $ result: NULL
#>  $ error :List of 2
#>   ..$ message: chr "non-numeric argument to mathematical ..."
#>   ..$ call   : language .f(...)
#>   ..- attr(*, "class")= chr [1:3] "simpleError" "error" ...
```

When the function succeeds the `result` element contains the result and the `error` element is `NULL`. When the function fails, the `result` element is `NULL` and the `error` element contains an error object.

`safely()` is designed to work with `map`:

```
x <- list(1, 10, "a")
y <- x %>% map(safely(log))
str(y)
#> List of 3
#>  $ :List of 2
#>   ..$ result: num 0
#>   ..$ error : NULL
#>  $ :List of 2
#>   ..$ result: num 2.3
#>   ..$ error : NULL
#>  $ :List of 2
#>   ..$ result: NULL
#>   ..$ error :List of 2
#>   .. ..$ message: chr "non-numeric argument to ..."
#>   .. ..$ call   : language .f(...)
#>   .. ..- attr(*, "class")=chr [1:3] "simpleError" "error" ...
```

This would be easier to work with if we had two lists: one of all the errors and one of all the output. That's easy to get with `purrr::transpose()`:

```
y <- y %>% transpose()
str(y)
#> List of 2
#>  $ result:List of 3
#>   ..$ : num 0
#>   ..$ : num 2.3
#>   ..$ : NULL
#>  $ error :List of 3
#>   ..$ : NULL
```

```
#>   ..$ : NULL
#>   ..$ :List of 2
#>   .. ..$ message: chr "non-numeric argument to ..."
#>   .. ..$ call   : language .f(...)
#>   .. ..- attr(*, "class")=chr [1:3] "simpleError" "error" ...
```

It's up to you how to deal with the errors, but typically you'll either look at the values of `x` where `y` is an error, or work with the values of `y` that are OK:

```
is_ok <- y$error %>% map_lgl(is_null)
x[!is_ok]
#> [[1]]
#> [1] "a"
y$result[is_ok] %>% flatten_dbl()
#> [1] 0.0 2.3
```

purrr provides two other useful adverbs:

- Like `safely()`, `possibly()` always succeeds. It's simpler than `safely()`, because you give it a default value to return when there is an error:

  ```
  x <- list(1, 10, "a")
  x %>% map_dbl(possibly(log, NA_real_))
  #> [1] 0.0 2.3  NA
  ```

- `quietly()` performs a similar role to `safely()`, but instead of capturing errors, it captures printed output, messages, and warnings:

  ```
  x <- list(1, -1)
  x %>% map(quietly(log)) %>% str()
  #> List of 2
  #>  $ :List of 4
  #>   ..$ result  : num 0
  #>   ..$ output  : chr ""
  #>   ..$ warnings: chr(0)
  #>   ..$ messages: chr(0)
  #>  $ :List of 4
  #>   ..$ result  : num NaN
  #>   ..$ output  : chr ""
  #>   ..$ warnings: chr "NaNs produced"
  #>   ..$ messages: chr(0)
  ```

Mapping over Multiple Arguments

So far we've mapped along a single input. But often you have multiple related inputs that you need to iterate along in parallel. That's the job of the `map2()` and `pmap()` functions. For example, imagine you want to simulate some random normals with different means. You know how to do that with `map()`:

```
mu <- list(5, 10, -3)
mu %>%
  map(rnorm, n = 5) %>%
  str()
#> List of 3
#>  $ : num [1:5] 5.45 5.5 5.78 6.51 3.18
#>  $ : num [1:5] 10.79 9.03 10.89 10.76 10.65
#>  $ : num [1:5] -3.54 -3.08 -5.01 -3.51 -2.9
```

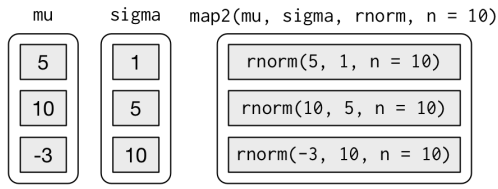
What if you also want to vary the standard deviation? One way to do that would be to iterate over the indices and index into vectors of means and sds:

```
sigma <- list(1, 5, 10)
seq_along(mu) %>%
  map(~rnorm(5, mu[[.]], sigma[[.]])) %>%
  str()
#> List of 3
#>  $ : num [1:5] 4.94 2.57 4.37 4.12 5.29
#>  $ : num [1:5] 11.72 5.32 11.46 10.24 12.22
#>  $ : num [1:5] 3.68 -6.12 22.24 -7.2 10.37
```

But that obfuscates the intent of the code. Instead we could use `map2()`, which iterates over two vectors in parallel:

```
map2(mu, sigma, rnorm, n = 5) %>% str()
#> List of 3
#>  $ : num [1:5] 4.78 5.59 4.93 4.3 4.47
#>  $ : num [1:5] 10.85 10.57 6.02 8.82 15.93
#>  $ : num [1:5] -1.12 7.39 -7.5 -10.09 -2.7
```

`map2()` generates this series of function calls:



Note that the arguments that vary for each call come *before* the function; arguments that are the same for every call come *after*.

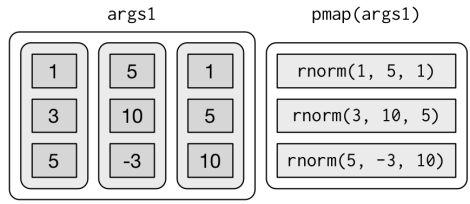
Like `map()`, `map2()` is just a wrapper around a for loop:

```
map2 <- function(x, y, f, ...) {
  out <- vector("list", length(x))
  for (i in seq_along(x)) {
    out[[i]] <- f(x[[i]], y[[i]], ...)
  }
  out
}
```

You could also imagine `map3()`, `map4()`, `map5()`, `map6()`, etc., but that would get tedious quickly. Instead, **purrr** provides `pmap()`, which takes a list of arguments. You might use that if you wanted to vary the mean, standard deviation, and number of samples:

```
n <- list(1, 3, 5)
args1 <- list(n, mu, sigma)
args1 %>%
  pmap(rnorm) %>%
  str()
#> List of 3
#>  $ : num 4.55
#>  $ : num [1:3] 13.4 18.8 13.2
#>  $ : num [1:5] 0.685 10.801 -11.671 21.363 -2.562
```

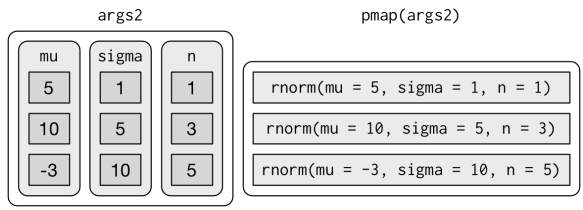
That looks like:



If you don't name the elements of list, `pmap()` will use positional matching when calling the function. That's a little fragile, and makes the code harder to read, so it's better to name the arguments:

```
args2 <- list(mean = mu, sd = sigma, n = n)
args2 %>%
  pmap(rnorm) %>%
  str()
```

That generates longer, but safer, calls:



Since the arguments are all the same length, it makes sense to store them in a data frame:

```
params <- tribble(
  ~mean, ~sd, ~n,
    5,     1,  1,
   10,     5,  3,
   -3,    10,  5
)
params %>%
  pmap(rnorm)
#> [[1]]
#> [1] 4.68
#>
#> [[2]]
#> [1] 23.44 12.85  7.28
#>
#> [[3]]
#> [1]  -5.34 -17.66   0.92   6.06   9.02
```

As soon as your code gets complicated, I think a data frame is a good approach because it ensures that each column has a name and is the same length as all the other columns.

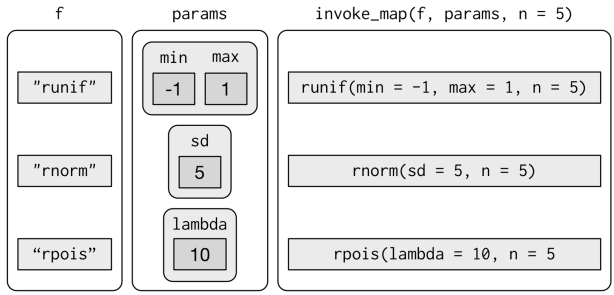
Invoking Different Functions

There's one more step up in complexity—as well as varying the arguments to the function you might also vary the function itself:

```
f <- c("runif", "rnorm", "rpois")
param <- list(
  list(min = -1, max = 1),
  list(sd = 5),
  list(lambda = 10)
)
```

To handle this case, you can use `invoke_map()`:

```
invoke_map(f, param, n = 5) %>% str()
#> List of 3
#>  $ : num [1:5] 0.762 0.36 -0.714 0.531 0.254
#>  $ : num [1:5] 3.07 -3.09 1.1 5.64 9.07
#>  $ : int [1:5] 9 14 8 9 7
```



The first argument is a list of functions or a character vector of function names. The second argument is a list of lists giving the arguments that vary for each function. The subsequent arguments are passed on to every function.

And again, you can use `tribble()` to make creating these matching pairs a little easier:

```
sim <- tribble(
  ~f,      ~params,
  "runif", list(min = -1, max = 1),
  "rnorm", list(sd = 5),
  "rpois", list(lambda = 10)
)
sim %>%
  mutate(sim = invoke_map(f, params, n = 10))
```

Walk

Walk is an alternative to map that you use when you want to call a function for its side effects, rather than for its return value. You typically do this because you want to render output to the screen or save files to disk—the important thing is the action, not the return value. Here's a very simple example:

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

x %>%
```

```
  walk(print)
#> [1] 1
#> [1] "a"
#> [1] 3
```

`walk()` is generally not that useful compared to `walk2()` or `pwalk()`. For example, if you had a list of plots and a vector of filenames, you could use `pwalk()` to save each file to the corresponding location on disk:

```
library(ggplot2)
plots <- mtcars %>%
  split(.$cyl) %>%
  map(~ggplot(., aes(mpg, wt)) + geom_point())
paths <- stringr::str_c(names(plots), ".pdf")

pwalk(list(paths, plots), ggsave, path = tempdir())
```

`walk()`, `walk2()`, and `pwalk()` all invisibly return `.x`, the first argument. This makes them suitable for use in the middle of pipelines.

Other Patterns of For Loops

purrr provides a number of other functions that abstract over other types of for loops. You'll use them less frequently than the map functions, but they're useful to know about. The goal here is to briefly illustrate each function, so hopefully it will come to mind if you see a similar problem in the future. Then you can go look up the documentation for more details.

Predicate Functions

A number of functions work with *predicate* functions that return either a single `TRUE` or `FALSE`.

`keep()` and `discard()` keep elements of the input where the predicate is `TRUE` or `FALSE`, respectively:

```
iris %>%
  keep(is.factor) %>%
  str()
#> 'data.frame':    150 obs. of  1 variable:
#>  $ Species: Factor w/ 3 levels "setosa","versicolor",..: ...

iris %>%
  discard(is.factor) %>%
  str()
#> 'data.frame':    150 obs. of  4 variables:
```

```
#>  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
#>  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3 ...
#>  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 ...
#>  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 ...
```

`some()` and `every()` determine if the predicate is true for any or for all of the elements:

```
x <- list(1:5, letters, list(10))

x %>%
  some(is_character)
#> [1] TRUE

x %>%
  every(is_vector)
#> [1] TRUE
```

`detect()` finds the first element where the predicate is true; `detect_index()` returns its position:

```
x <- sample(10)
x
#>  [1]  8  7  5  6  9  2 10  1  3  4

x %>%
  detect(~ . > 5)
#> [1] 8

x %>%
  detect_index(~ . > 5)
#> [1] 1
```

`head_while()` and `tail_while()` take elements from the start or end of a vector while a predicate is true:

```
x %>%
  head_while(~ . > 5)
#> [1] 8 7

x %>%
  tail_while(~ . > 5)
#> integer(0)
```

Reduce and Accumulate

Sometimes you have a complex list that you want to reduce to a simple list by repeatedly applying a function that reduces a pair to a singleton. This is useful if you want to apply a two-table **dplyr** verb to multiple tables. For example, you might have a list of data frames,

and you want to reduce to a single data frame by joining the elements together:

```
dfs <- list(
  age = tibble(name = "John", age = 30),
  sex = tibble(name = c("John", "Mary"), sex = c("M", "F")),
  trt = tibble(name = "Mary", treatment = "A")
)

dfs %>% reduce(full_join)
#> Joining, by = "name"
#> Joining, by = "name"
#> # A tibble: 2 × 4
#>    name   age   sex treatment
#>   <chr> <dbl> <chr>     <chr>
#> 1  John    30     M      <NA>
#> 2  Mary    NA     F         A
```

Or maybe you have a list of vectors, and want to find the intersection:

```
vs <- list(
  c(1, 3, 5, 6, 10),
  c(1, 2, 3, 7, 8, 10),
  c(1, 2, 3, 4, 8, 9, 10)
)

vs %>% reduce(intersect)
#> [1]  1  3 10
```

The `reduce` function takes a "binary" function (i.e., a function with two primary inputs), and applies it repeatedly to a list until there is only a single element left.

Accumulate is similar but it keeps all the interim results. You could use it to implement a cumulative sum:

```
x <- sample(10)
x
#>  [1]  6  9  8  5  2  4  7  1 10  3
x %>% accumulate(`+`)
#>  [1]  6 15 23 28 30 34 41 42 52 55
```

Exercises

1. Implement your own version of `every()` using a for loop. Compare it with `purrr::every()`. What does **purrr**'s version do that your version doesn't?

2. Create an enhanced `col_sum()` that applies a summary function to every numeric column in a data frame.
3. A possible base R equivalent of `col_sum()` is:

```
col_sum3 <- function(df, f) {
  is_num <- sapply(df, is.numeric)
  df_num <- df[, is_num]

  sapply(df_num, f)
}
```

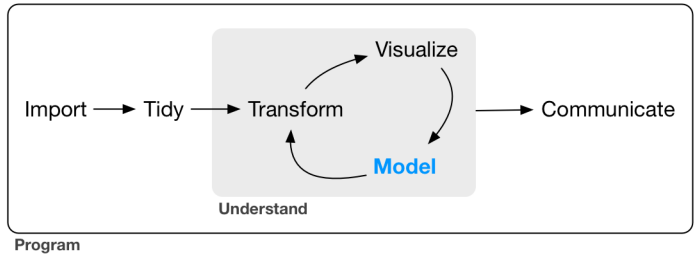
But it has a number of bugs as illustrated with the following inputs:

```
df <- tibble(
  x = 1:3,
  y = 3:1,
  z = c("a", "b", "c")
)
# OK
col_sum3(df, mean)
# Has problems: don't always return numeric vector
col_sum3(df[1:2], mean)
col_sum3(df[1], mean)
col_sum3(df[0], mean)
```

What causes the bugs?

PART IV
Model

Now that you are equipped with powerful programming tools we can finally return to modeling. You'll use your new tools of data wrangling and programming to fit many models and understand how they work. The focus of this book is on exploration, not confirmation or formal inference. But you'll learn a few basic tools that help you understand the variation within your models.



The goal of a model is to provide a simple low-dimensional summary of a dataset. Ideally, the model will capture true "signals" (i.e., patterns generated by the phenomenon of interest), and ignore "noise" (i.e., random variation that you're not interested in). Here we only cover "predictive" models, which, as the name suggests, generate predictions. There is another type of model that we're not going to discuss: "data discovery" models. These models don't make pre-

dictions, but instead help you discover interesting relationships within your data. (These two categories of models are sometimes called supervised and unsupervised, but I don't think that terminology is particularly illuminating.)

This book is not going to give you a deep understanding of the mathematical theory that underlies models. It will, however, build your intuition about how statistical models work, and give you a family of useful tools that allow you to use models to better understand your data:

- In Chapter 18, you'll learn how models work mechanistically, focusing on the important family of linear models. You'll learn general tools for gaining insight into what a predictive model tells you about your data, focusing on simple simulated datasets.
- In Chapter 19, you'll learn how to use models to pull out known patterns in real data. Once you have recognized an important pattern it's useful to make it explicit in a model, because then you can more easily see the subtler signals that remain.
- In Chapter 20, you'll learn how to use many simple models to help understand complex datasets. This is a powerful technique, but to access it you'll need to combine modeling and programming tools.

These topics are notable because of what they don't include: any tools for quantitatively assessing models. That is deliberate: precisely quantifying a model requires a couple of big ideas that we just don't have the space to cover here. For now, you'll rely on qualitative assessment and your natural skepticism. In "Learning More About Models" on page 396, we'll point you to other resources where you can learn more.

Hypothesis Generation Versus Hypothesis Confirmation

In this book, we are going to use models as a tool for exploration, completing the trifecta of the tools for EDA that were introduced in Part I. This is not how models are usually taught, but as you will see, models are an important tool for exploration. Traditionally, the focus of modeling is on inference, or for confirming that a hypothesis is true. Doing this correctly is not complicated, but it is hard.

There is a pair of ideas that you must understand in order to do inference correctly:

- Each observation can either be used for exploration or confirmation, not both.
- You can use an observation as many times as you like for exploration, but you can only use it once for confirmation. As soon as you use an observation twice, you've switched from confirmation to exploration.

This is necessary because to confirm a hypothesis you must use data independent of the data that you used to generate the hypothesis. Otherwise you will be overoptimistic. There is absolutely nothing wrong with exploration, but you should never sell an exploratory analysis as a confirmatory analysis because it is fundamentally misleading.

If you are serious about doing a confirmatory analysis, one approach is to split your data into three pieces before you begin the analysis:

- 60% of your data goes into a *training* (or exploration) set. You're allowed to do anything you like with this data: visualize it and fit tons of models to it.
- 20% goes into a *query* set. You can use this data to compare models or visualizations by hand, but you're not allowed to use it as part of an automated process.
- 20% is held back for a *test* set. You can only use this data ONCE, to test your final model.

This partitioning allows you to explore the training data, occasionally generating candidate hypotheses that you check with the query set. When you are confident you have the right model, you can check it once with the test data.

(Note that even when doing confirmatory modeling, you will still need to do EDA. If you don't do any EDA you will remain blind to the quality problems with your data.)

CHAPTER 18

Model Basics with modelr

Introduction

The goal of a model is to provide a simple low-dimensional summary of a dataset. In the context of this book we're going to use models to partition data into patterns and residuals. Strong patterns will hide subtler trends, so we'll use models to help peel back layers of structure as we explore a dataset.

However, before we can start using models on interesting, real datasets, you need to understand the basics of how models work. For that reason, this chapter of the book is unique because it uses only simulated datasets. These datasets are very simple, and not at all interesting, but they will help you understand the essence of modeling before you apply the same techniques to real data in the next chapter.

There are two parts to a model:

1. First, you define a *family of models* that express a precise, but generic, pattern that you want to capture. For example, the pattern might be a straight line, or a quadatric curve. You will express the model family as an equation like `y = a_1 * x + a_2` or `y = a_1 * x ^ a_2`. Here, `x` and `y` are known variables from your data, and `a_1` and `a_2` are parameters that can vary to capture different patterns.
2. Next, you generate a *fitted model* by finding the model from the family that is the closest to your data. This takes the generic

model family and makes it specific, like `y = 3 * x + 7` or `y = 9 * x ^ 2`.

It's important to understand that a fitted model is just the closest model from a family of models. That implies that you have the "best" model (according to some criteria); it doesn't imply that you have a good model and it certainly doesn't imply that the model is "true." George Box puts this well in his famous aphorism:

> All models are wrong, but some are useful.

It's worth reading the fuller context of the quote:

> Now it would be very remarkable if any system existing in the real world could be exactly represented by any simple model. However, cunningly chosen parsimonious models often do provide remarkably useful approximations. For example, the law PV = RT relating pressure P, volume V and temperature T of an "ideal" gas via a constant R is not exactly true for any real gas, but it frequently provides a useful approximation and furthermore its structure is informative since it springs from a physical view of the behavior of gas molecules.
>
> For such a model there is no need to ask the question "Is the model true?" If "truth" is to be the "whole truth" the answer must be "No." The only question of interest is "Is the model illuminating and useful?"

The goal of a model is not to uncover truth, but to discover a simple approximation that is still useful.

Prerequisites

In this chapter we'll use the **modelr** package, which wraps around base R's modeling functions to make them work naturally in a pipe.

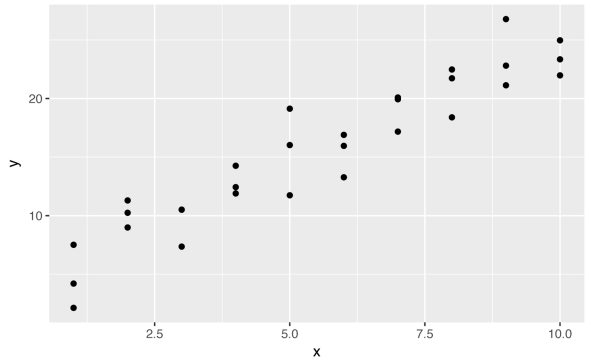
```
library(tidyverse)

library(modelr)
options(na.action = na.warn)
```

A Simple Model

Let's take a look at the simulated dataset `sim1`. It contains two continuous variables, `x` and `y`. Let's plot them to see how they're related:

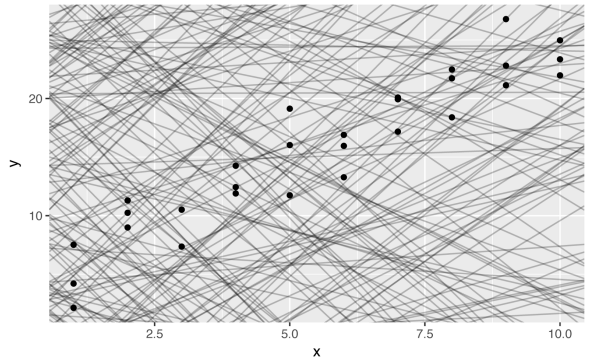
```
ggplot(sim1, aes(x, y)) +
  geom_point()
```



You can see a strong pattern in the data. Let's use a model to capture that pattern and make it explicit. It's our job to supply the basic form of the model. In this case, the relationship looks linear, i.e., `y = a_0 + a_1 * x`. Let's start by getting a feel for what models from that family look like by randomly generating a few and overlaying them on the data. For this simple case, we can use `geom_abline()`, which takes a slope and intercept as parameters. Later on we'll learn more general techniques that work with any model:

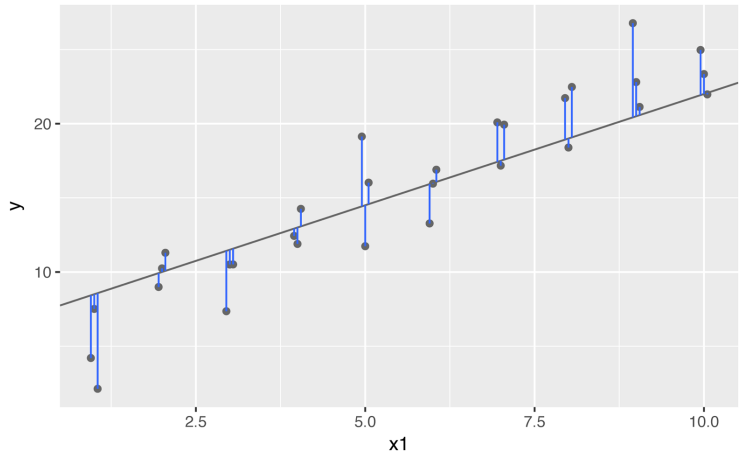
```
models <- tibble(
  a1 = runif(250, -20, 40),
  a2 = runif(250, -5, 5)
)

ggplot(sim1, aes(x, y)) +
  geom_abline(
    aes(intercept = a1, slope = a2),
    data = models, alpha = 1/4
  ) +
  geom_point()
```



There are 250 models on this plot, but a lot are really bad! We need to find the good models by making precise our intuition that a good model is "close" to the data. We need a way to quantify the distance between the data and a model. Then we can fit the model by finding the values of `a_0` and `a_1` that generate the model with the smallest distance from this data.

One easy place to start is to find the vertical distance between each point and the model, as in the following diagram. (Note that I've shifted the x values slightly so you can see the individual distances.)



This distance is just the difference between the `y` value given by the model (the *prediction*), and the actual `y` value in the data (the *response*).

To compute this distance, we first turn our model family into an R function. This takes the model parameters and the data as inputs, and gives values predicted by the model as output:

```
model1 <- function(a, data) {
  a[1] + data$x * a[2]
}
model1(c(7, 1.5), sim1)
#>  [1]  8.5  8.5  8.5 10.0 10.0 10.0 11.5 11.5 11.5 13.0 13.0
#> [12] 13.0 14.5 14.5 14.5 16.0 16.0 16.0 17.5 17.5 17.5 19.0
#> [23] 19.0 19.0 20.5 20.5 20.5 22.0 22.0 22.0
```

Next, we need some way to compute an overall distance between the predicted and actual values. In other words, the plot shows 30 distances: how do we collapse that into a single number?

One common way to do this in statistics is to use the "root-mean-squared deviation." We compute the difference between actual and predicted, square them, average them, and then take the square root. This distance has lots of appealing mathematical properties, which we're not going to talk about here. You'll just have to take my word for it!

```
measure_distance <- function(mod, data) {
  diff <- data$y - model1(mod, data)
  sqrt(mean(diff ^ 2))
}
measure_distance(c(7, 1.5), sim1)
#> [1] 2.67
```

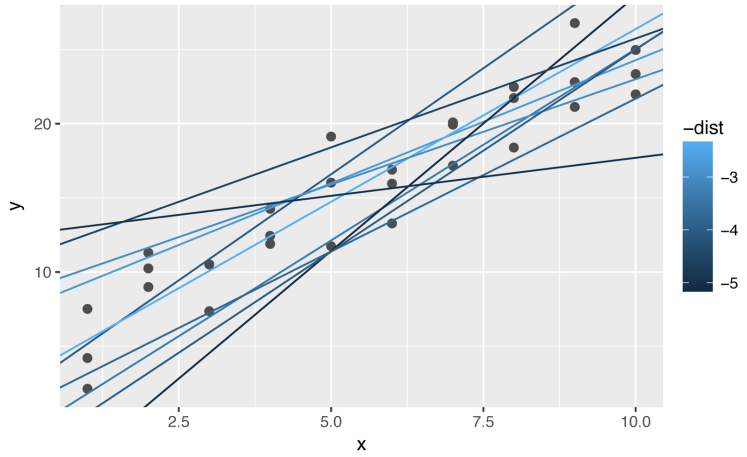
Now we can use **purrr** to compute the distance for all the models defined previously. We need a helper function because our distance function expects the model as a numeric vector of length 2:

```
sim1_dist <- function(a1, a2) {
  measure_distance(c(a1, a2), sim1)
}

models <- models %>%
  mutate(dist = purrr::map2_dbl(a1, a2, sim1_dist))
models
#> # A tibble: 250 × 3
#>       a1      a2  dist
#>    <dbl>   <dbl> <dbl>
#> 1 -15.15  0.0889  30.8
#> 2  30.06 -0.8274  13.2
#> 3  16.05  2.2695  13.2
#> 4 -10.57  1.3769  18.7
#> 5 -19.56 -1.0359  41.8
#> 6   7.98  4.5948  19.3
#> # ... with 244 more rows
```

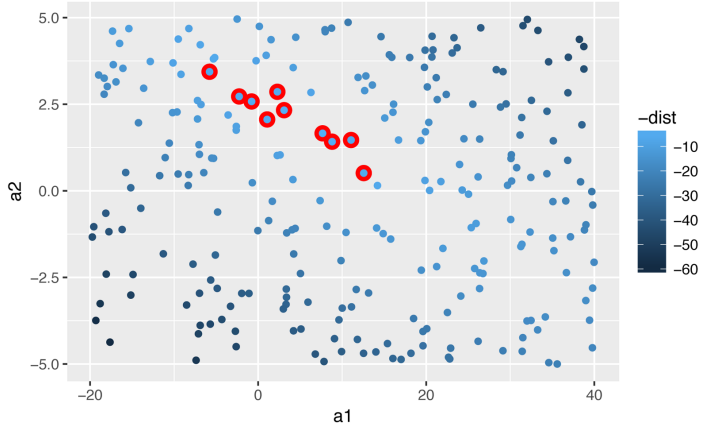
Next, let's overlay the 10 best models on to the data. I've colored the models by `-dist`: this is an easy way to make sure that the best models (i.e., the ones with the smallest distance) get the brighest colors:

```
ggplot(sim1, aes(x, y)) +
  geom_point(size = 2, color = "grey30") +
  geom_abline(
    aes(intercept = a1, slope = a2, color = -dist),
    data = filter(models, rank(dist) <= 10)
  )
```



We can also think about these models as observations, and visualize them with a scatterplot of `a1` versus `a2`, again colored by `-dist`. We can no longer directly see how the model compares to the data, but we can see many models at once. Again, I've highlighted the 10 best models, this time by drawing red circles underneath them:

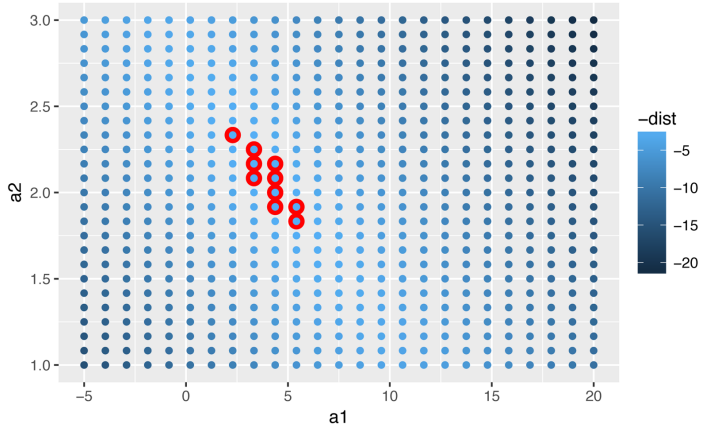
```
ggplot(models, aes(a1, a2)) +
  geom_point(
    data = filter(models, rank(dist) <= 10),
    size = 4, color = "red"
  ) +
  geom_point(aes(colour = -dist))
```



Instead of trying lots of random models, we could be more systematic and generate an evenly spaced grid of points (this is called a grid search). I picked the parameters of the grid roughly by looking at where the best models were in the preceding plot:

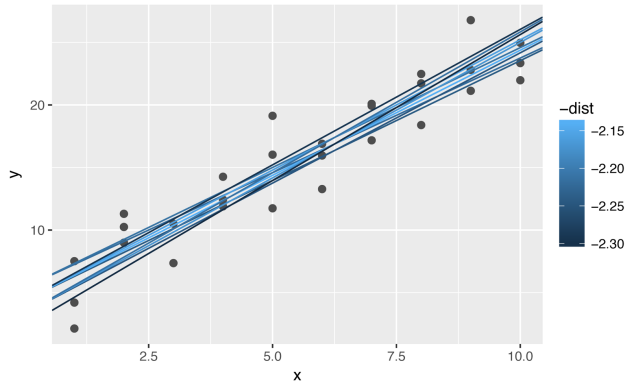
```
grid <- expand.grid(
  a1 = seq(-5, 20, length = 25),
  a2 = seq(1, 3, length = 25)
  ) %>%
  mutate(dist = purrr::map2_dbl(a1, a2, sim1_dist))

grid %>%
  ggplot(aes(a1, a2)) +
  geom_point(
    data = filter(grid, rank(dist) <= 10),
    size = 4, colour = "red"
  ) +
  geom_point(aes(color = -dist))
```



When you overlay the best 10 models back on the original data, they all look pretty good:

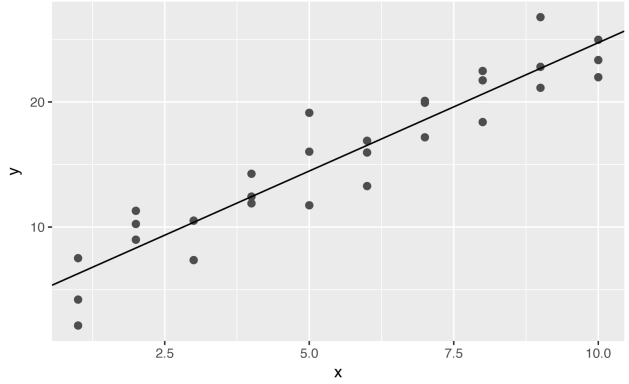
```
ggplot(sim1, aes(x, y)) +
  geom_point(size = 2, color = "grey30") +
  geom_abline(
    aes(intercept = a1, slope = a2, color = -dist),
    data = filter(grid, rank(dist) <= 10)
  )
```



You could imagine iteratively making the grid finer and finer until you narrowed in on the best model. But there's a better way to tackle that problem: a numerical minimization tool called Newton–Raphson search. The intuition of Newton–Raphson is pretty simple: you pick a starting point and look around for the steepest slope. You then ski down that slope a little way, and then repeat again and again, until you can't go any lower. In R, we can do that with `optim()`:

```
best <- optim(c(0, 0), measure_distance, data = sim1)
best$par
#> [1] 4.22 2.05

ggplot(sim1, aes(x, y)) +
  geom_point(size = 2, color = "grey30") +
  geom_abline(intercept = best$par[1], slope = best$par[2])
```



Don't worry too much about the details of how `optim()` works. It's the intuition that's important here. If you have a function that defines the distance between a model and a dataset, and an algorithm that can minimize that distance by modifying the parameters of the model, you can find the best model. The neat thing about this approach is that it will work for any family of models that you can write an equation for.

There's one more approach that we can use for this model, because it is a special case of a broader family: linear models. A linear model has the general form `y = a_1 + a_2 * x_1 + a_3 * x_2 + ... + a_n * x_(n - 1)`. So this simple model is equivalent to a general linear model where `n` is `2` and `x_1` is `x`. R has a tool specifically designed for fitting linear models called `lm()`. `lm()` has a special way to specify the model family: formulas. Formulas look like `y ~ x`, which `lm()` will translate to a function like `y = a_1 + a_2 * x`. We can fit the model and look at the output:

```
sim1_mod <- lm(y ~ x, data = sim1)
coef(sim1_mod)
#> (Intercept)           x
#>        4.22        2.05
```

These are exactly the same values we got with `optim()`! Behind the scenes `lm()` doesn't use `optim()` but instead takes advantage of the mathematical structure of linear models. Using some connections between geometry, calculus, and linear algebra, `lm()` actually finds the closest model in a single step, using a sophisticated algorithm. This approach is faster and guarantees that there is a global minimum.

Exercises

1. One downside of the linear model is that it is sensitive to unusual values because the distance incorporates a squared term. Fit a linear model to the following simulated data, and visualize the results. Rerun a few times to generate different simulated datasets. What do you notice about the model?

```
sim1a <- tibble(
  x = rep(1:10, each = 3),
  y = x * 1.5 + 6 + rt(length(x), df = 2)
)
```

2. One way to make linear models more robust is to use a different distance measure. For example, instead of root-mean-squared distance, you could use mean-absolute distance:

```
measure_distance <- function(mod, data) {
  diff <- data$y - make_prediction(mod, data)
  mean(abs(diff))
}
```

   Use `optim()` to fit this model to the previous simulated data and compare it to the linear model.

3. One challenge with performing numerical optimization is that it's only guaranteed to find one local optima. What's the problem with optimizing a three-parameter model like this?

```
model1 <- function(a, data) {
  a[1] + data$x * a[2] + a[3]
}
```

Visualizing Models

For simple models, like the one in the previous section, you can figure out what pattern the model captures by carefully studying the model family and the fitted coefficients. And if you ever take a statistics course on modeling, you're likely to spend a lot of time doing just that. Here, however, we're going to take a different tack. We're going to focus on understanding a model by looking at its predictions. This has a big advantage: every type of predictive model makes predictions (otherwise what use would it be?) so we can use the same set of techniques to understand any type of predictive model.

It's also useful to see what the model doesn't capture, the so-called residuals that are left after subtracting the predictions from the data. Residuals are powerful because they allow us to use models to remove striking patterns so we can study the subtler trends that remain.

Predictions

To visualize the predictions from a model, we start by generating an evenly spaced grid of values that covers the region where our data lies. The easiest way to do that is to use `modelr::data_grid()`. Its

first argument is a data frame, and for each subsequent argument it finds the unique variables and then generates all combinations:

```
grid <- sim1 %>%
  data_grid(x)
grid
#> # A tibble: 10 × 1
#>       x
#>   <int>
#> 1     1
#> 2     2
#> 3     3
#> 4     4
#> 5     5
#> 6     6
#> # ... with 4 more rows
```

(This will get more interesting when we start to add more variables to our model.)

Next we add predictions. We'll use `modelr::add_predictions()`, which takes a data frame and a model. It adds the predictions from the model to a new column in the data frame:

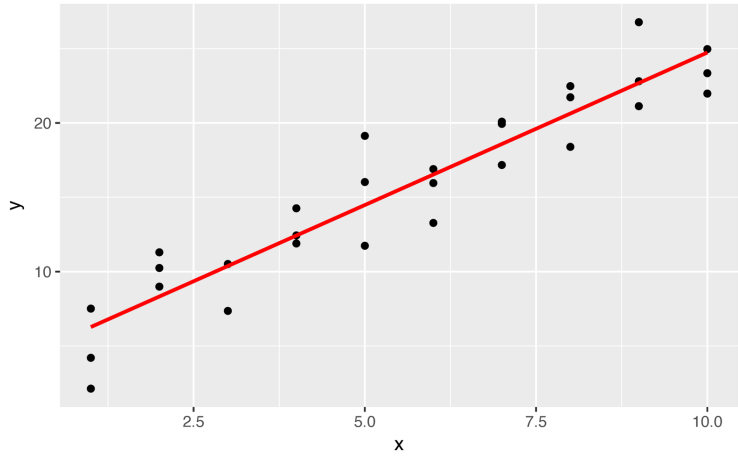
```
grid <- grid %>%
  add_predictions(sim1_mod)
grid
#> # A tibble: 10 × 2
#>       x  pred
#>   <int> <dbl>
#> 1     1  6.27
#> 2     2  8.32
#> 3     3 10.38
#> 4     4 12.43
#> 5     5 14.48
#> 6     6 16.53
#> # ... with 4 more rows
```

(You can also use this function to add predictions to your original dataset.)

Next, we plot the predictions. You might wonder about all this extra work compared to just using `geom_abline()`. But the advantage of this approach is that it will work with *any* model in R, from the simplest to the most complex. You're only limited by your visualization skills. For more ideas about how to visualize more complex model types, you might try *http://vita.had.co.nz/papers/model-vis.html*.

```
ggplot(sim1, aes(x)) +
  geom_point(aes(y = y)) +
```

```
  geom_line(
    aes(y = pred),
    data = grid,
    colour = "red",
    size = 1
  )
```



Residuals

The flip side of predictions are *residuals*. The predictions tell you the pattern that the model has captured, and the residuals tell you what the model has missed. The residuals are just the distances between the observed and predicted values that we computed earlier.

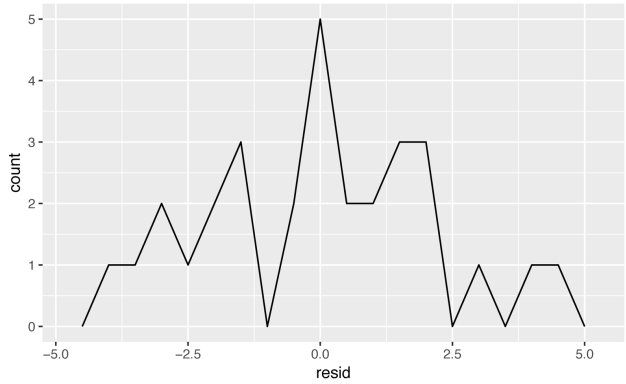
We add residuals to the data with `add_residuals()`, which works much like `add_predictions()`. Note, however, that we use the original dataset, not a manufactured grid. This is because to compute residuals we need actual `y` values:

```
sim1 <- sim1 %>%
  add_residuals(sim1_mod)
sim1
#> # A tibble: 30 × 3
#>       x     y  resid
#>   <int> <dbl>  <dbl>
#> 1     1  4.20 -2.072
#> 2     1  7.51  1.238
#> 3     1  2.13 -4.147
#> 4     2  8.99  0.665
#> 5     2 10.24  1.919
```

```
#> 6     2 11.30  2.973
#> # ... with 24 more rows
```

There are a few different ways to understand what the residuals tell us about the model. One way is to simply draw a frequency polygon to help us understand the spread of the residuals:

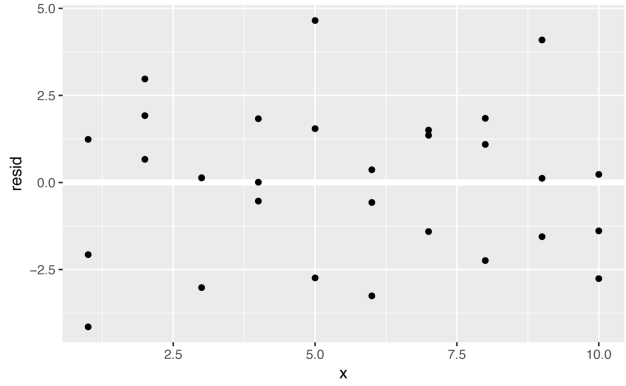
```
ggplot(sim1, aes(resid)) +
  geom_freqpoly(binwidth = 0.5)
```



This helps you calibrate the quality of the model: how far away are the predictions from the observed values? Note that the average of the residual will always be 0.

You'll often want to re-create plots using the residuals instead of the original predictor. You'll see a lot of that in the next chapter:

```
ggplot(sim1, aes(x, resid)) +
  geom_ref_line(h = 0) +
  geom_point()
```



This looks like random noise, suggesting that our model has done a good job of capturing the patterns in the dataset.

Exercises

1. Instead of using `lm()` to fit a straight line, you can use `loess()` to fit a smooth curve. Repeat the process of model fitting, grid generation, predictions, and visualization on `sim1` using `loess()` instead of `lm()`. How does the result compare to `geom_smooth()`?
2. `add_predictions()` is paired with `gather_predictions()` and `spread_predictions()`. How do these three functions differ?
3. What does `geom_ref_line()` do? What package does it come from? Why is displaying a reference line in plots showing residuals useful and important?
4. Why might you want to look at a frequency polygon of absolute residuals? What are the pros and cons compared to looking at the raw residuals?

Formulas and Model Families

You've seen formulas before when using `facet_wrap()` and `facet_grid()`. In R, formulas provide a general way of getting "special behavior." Rather than evaluating the values of the variables right away, they capture them so they can be interpreted by the function.

The majority of modeling functions in R use a standard conversion from formulas to functions. You've seen one simple conversion already: `y ~ x` is translated to `y = a_1 + a_2 * x`. If you want to see what R actually does, you can use the `model_matrix()` function. It takes a data frame and a formula and returns a tibble that defines the model equation: each column in the output is associated with one coefficient in the model, and the function is always `y = a_1 * out1 + a_2 * out_2`. For the simplest case of `y ~ x1` this shows us something interesting:

```
df <- tribble(
  ~y, ~x1, ~x2,
  4, 2, 5,
  5, 1, 6
```

```
)
model_matrix(df, y ~ x1)
#> # A tibble: 2 × 2
#>   `(Intercept)`    x1
#>           <dbl> <dbl>
#> 1             1     2
#> 2             1     1
```

The way that R adds the intercept to the model is just by having a column that is full of ones. By default, R will always add this column. If you don't want that, you need to explicitly drop it with `-1`:

```
model_matrix(df, y ~ x1 - 1)
#> # A tibble: 2 × 1
#>      x1
#>   <dbl>
#> 1     2
#> 2     1
```

The model matrix grows in an unsurprising way when you add more variables to the model:

```
model_matrix(df, y ~ x1 + x2)
#> # A tibble: 2 × 3
#>   `(Intercept)`    x1    x2
#>           <dbl> <dbl> <dbl>
#> 1             1     2     5
#> 2             1     1     6
```

This formula notation is sometimes called "Wilkinson-Rogers notation," and was initially described in *Symbolic Description of Factorial Models for Analysis of Variance*, by G. N. Wilkinson and C. E. Rogers. It's worth digging up and reading the original paper if you'd like to understand the full details of the modeling algebra.

The following sections expand on how this formula notation works for categorcal variables, interactions, and transformation.

Categorical Variables

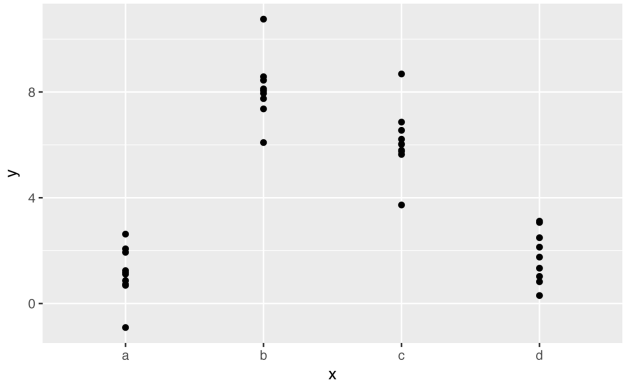
Generating a function from a formula is straightforward when the predictor is continuous, but things get a bit more complicated when the predictor is categorical. Imagine you have a formula like `y ~ sex`, where `sex` could either be male or female. It doesn't make sense to convert that to a formula like `y = x_0 + x_1 * sex` because `sex` isn't a number—you can't multiply it! Instead what R does is convert it to `y = x_0 + x_1 * sex_male` where `sex_male` is one if `sex` is male and zero otherwise:

```
df <- tribble(
  ~ sex, ~ response,
  "male", 1,
  "female", 2,
  "male", 1
)
model_matrix(df, response ~ sex)
#> # A tibble: 3 × 2
#>   `(Intercept)` sexmale
#>           <dbl>   <dbl>
#> 1             1       1
#> 2             1       0
#> 3             1       1
```

You might wonder why R also doesn't create a `sexfemale` column. The problem is that would create a column that is perfectly predictable based on the other columns (i.e., `sexfemale = 1 - sexmale`). Unfortunately the exact details of why this is a problem is beyond the scope of this book, but basically it creates a model family that is too flexible, and will have infinitely many models that are equally close to the data.

Fortunately, however, if you focus on visualizing predictions you don't need to worry about the exact parameterization. Let's look at some data and models to make that concrete. Here's the `sim2` dataset from **modelr**:

```
ggplot(sim2) +
  geom_point(aes(x, y))
```



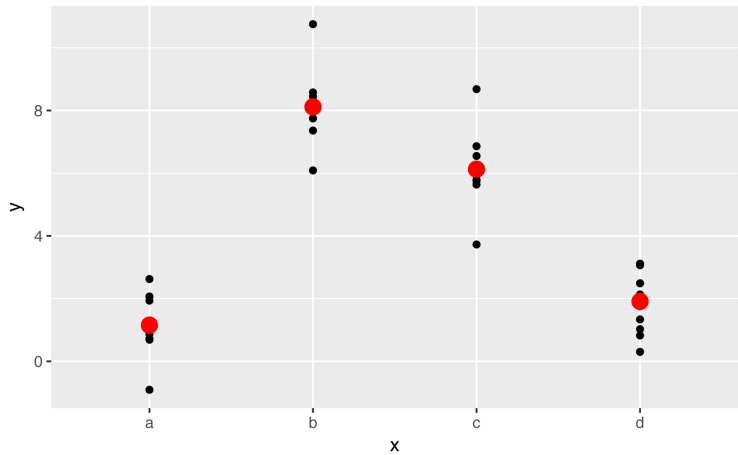
We can fit a model to it, and generate predictions:

```
mod2 <- lm(y ~ x, data = sim2)
```

```
grid <- sim2 %>%
  data_grid(x) %>%
  add_predictions(mod2)
grid
#> # A tibble: 4 × 2
#>       x  pred
#>   <chr> <dbl>
#> 1     a  1.15
#> 2     b  8.12
#> 3     c  6.13
#> 4     d  1.91
```

Effectively, a model with a categorical x will predict the mean value for each category. (Why? Because the mean minimizes the root-mean-squared distance.) That's easy to see if we overlay the predictions on top of the original data:

```
ggplot(sim2, aes(x)) +
  geom_point(aes(y = y)) +
  geom_point(
    data = grid,
    aes(y = pred),
    color = "red",
    size = 4
  )
```



You can't make predictions about levels that you didn't observe. Sometimes you'll do this by accident so it's good to recognize this error message:

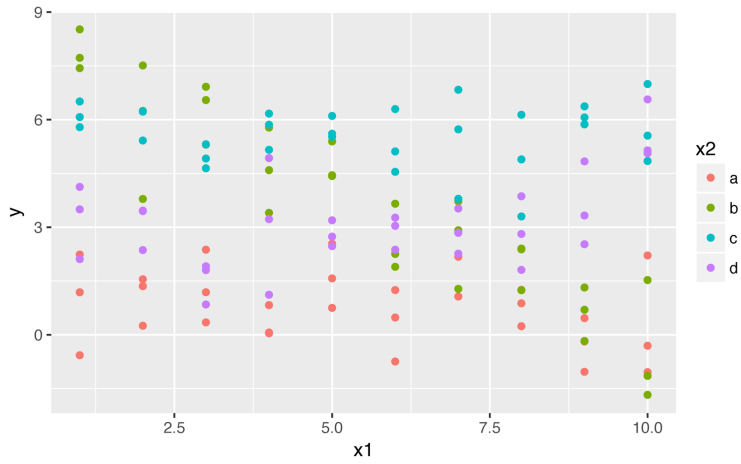
```
tibble(x = "e") %>%
  add_predictions(mod2)
```

```
#> Error in model.frame.default(Terms, newdata, na.action =
#> na.action, xlev = object$xlevels): factor x has new level e
```

Interactions (Continuous and Categorical)

What happens when you combine a continuous and a categorical variable? `sim3` contains a categorical predictor and a continuous predictor. We can visualize it with a simple plot:

```
ggplot(sim3, aes(x1, y)) +
  geom_point(aes(color = x2))
```



There are two possible models you could fit to this data:

```
mod1 <- lm(y ~ x1 + x2, data = sim3)
mod2 <- lm(y ~ x1 * x2, data = sim3)
```

When you add variables with `+`, the model will estimate each effect independent of all the others. It's possible to fit the so-called interaction by using `*`. For example, `y ~ x1 * x2` is translated to `y = a_0 + a_1 * a1 + a_2 * a2 + a_12 * a1 * a2`. Note that whenever you use `*`, both the interaction and the individual components are included in the model.

To visualize these models we need two new tricks:

- We have two predictors, so we need to give `data_grid()` both variables. It finds all the unique values of `x1` and `x2` and then generates all combinations.

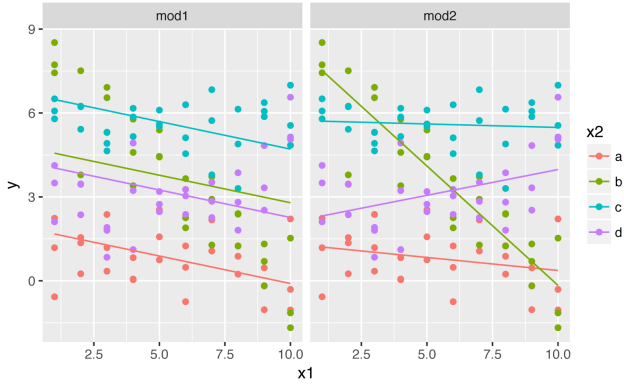
- To generate predictions from both models simultaneously, we can use `gather_predictions()`, which adds each prediction as a row. The complement of `gather_predictions()` is `spread_predictions()`, which adds each prediction to a new column.

Together this gives us:

```
grid <- sim3 %>%
  data_grid(x1, x2) %>%
  gather_predictions(mod1, mod2)
grid
#> # A tibble: 80 × 4
#>   model    x1     x2  pred
#>   <chr> <int> <fctr> <dbl>
#> 1  mod1     1      a  1.67
#> 2  mod1     1      b  4.56
#> 3  mod1     1      c  6.48
#> 4  mod1     1      d  4.03
#> 5  mod1     2      a  1.48
#> 6  mod1     2      b  4.37
#> # ... with 74 more rows
```

We can visualize the results for both models on one plot using faceting:

```
ggplot(sim3, aes(x1, y, color = x2)) +
  geom_point() +
  geom_line(data = grid, aes(y = pred)) +
  facet_wrap(~ model)
```

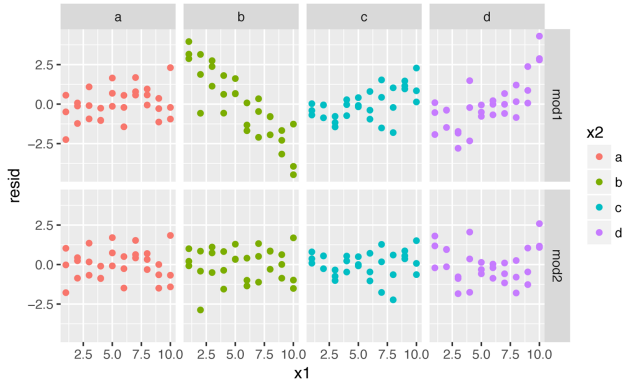


Note that the model that uses + has the same slope for each line, but different intercepts. The model that uses * has a different slope and intercept for each line.

Which model is better for this data? We can take look at the residuals. Here I've faceted by both model and `x2` because it makes it easier to see the pattern within each group:

```
sim3 <- sim3 %>%
  gather_residuals(mod1, mod2)

ggplot(sim3, aes(x1, resid, color = x2)) +
  geom_point() +
  facet_grid(model ~ x2)
```



There is little obvious pattern in the residuals for `mod2`. The residuals for `mod1` show that the model has clearly missed some pattern in `b`, and less so, but still present, is pattern in `c`, and `d`. You might wonder if there's a precise way to tell which of `mod1` or `mod2` is better. There is, but it requires a lot of mathematical background, and we don't really care. Here, we're interested in a qualitative assessment of whether or not the model has captured the pattern that we're interested in.

Interactions (Two Continuous)

Let's take a look at the equivalent model for two continuous variables. Initially things proceed almost identically to the previous example:

```
mod1 <- lm(y ~ x1 + x2, data = sim4)
mod2 <- lm(y ~ x1 * x2, data = sim4)

grid <- sim4 %>%
  data_grid(
    x1 = seq_range(x1, 5),
```

```
    x2 = seq_range(x2, 5)
  ) %>%
  gather_predictions(mod1, mod2)
grid
#> # A tibble: 50 × 4
#>   model    x1    x2   pred
#>   <chr> <dbl> <dbl>  <dbl>
#> 1  mod1  -1.0  -1.0  0.996
#> 2  mod1  -1.0  -0.5 -0.395
#> 3  mod1  -1.0   0.0 -1.786
#> 4  mod1  -1.0   0.5 -3.177
#> 5  mod1  -1.0   1.0 -4.569
#> 6  mod1  -0.5  -1.0  1.907
#> # ... with 44 more rows
```

Note my use of `seq_range()` inside `data_grid()`. Instead of using every unique value of `x`, I'm going to use a regularly spaced grid of five values between the minimum and maximum numbers. It's probably not super important here, but it's a useful technique in general. There are three other useful arguments to `seq_range()`:

- `pretty = TRUE` will generate a "pretty" sequence, i.e., something that looks nice to the human eye. This is useful if you want to produce tables of output:

  ```
  seq_range(c(0.0123, 0.923423), n = 5)
  #> [1] 0.0123 0.2401 0.4679 0.6956 0.9234
  seq_range(c(0.0123, 0.923423), n = 5, pretty = TRUE)
  #> [1] 0.0 0.2 0.4 0.6 0.8 1.0
  ```

- `trim = 0.1` will trim off 10% of the tail values. This is useful if the variable has a long-tailed distribution and you want to focus on generating values near the center:

  ```
  x1 <- rcauchy(100)
  seq_range(x1, n = 5)
  #> [1] -115.9  -83.5  -51.2  -18.8   13.5
  seq_range(x1, n = 5, trim = 0.10)
  #> [1] -13.84  -8.71  -3.58   1.55   6.68
  seq_range(x1, n = 5, trim = 0.25)
  #> [1] -2.1735 -1.0594  0.0547  1.1687  2.2828
  seq_range(x1, n = 5, trim = 0.50)
  #> [1] -0.725 -0.268  0.189  0.647  1.104
  ```

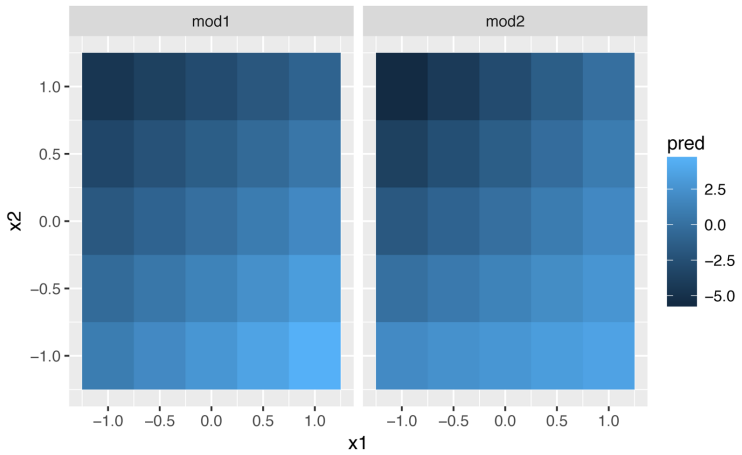
- `expand = 0.1` is in some sense the opposite of `trim()`; it expands the range by 10%:

  ```
  x2 <- c(0, 1)
  seq_range(x2, n = 5)
  #> [1] 0.00 0.25 0.50 0.75 1.00
  ```

```
seq_range(x2, n = 5, expand = 0.10)
#> [1] -0.050  0.225  0.500  0.775  1.050
seq_range(x2, n = 5, expand = 0.25)
#> [1] -0.125  0.188  0.500  0.812  1.125
seq_range(x2, n = 5, expand = 0.50)
#> [1] -0.250  0.125  0.500  0.875  1.250
```

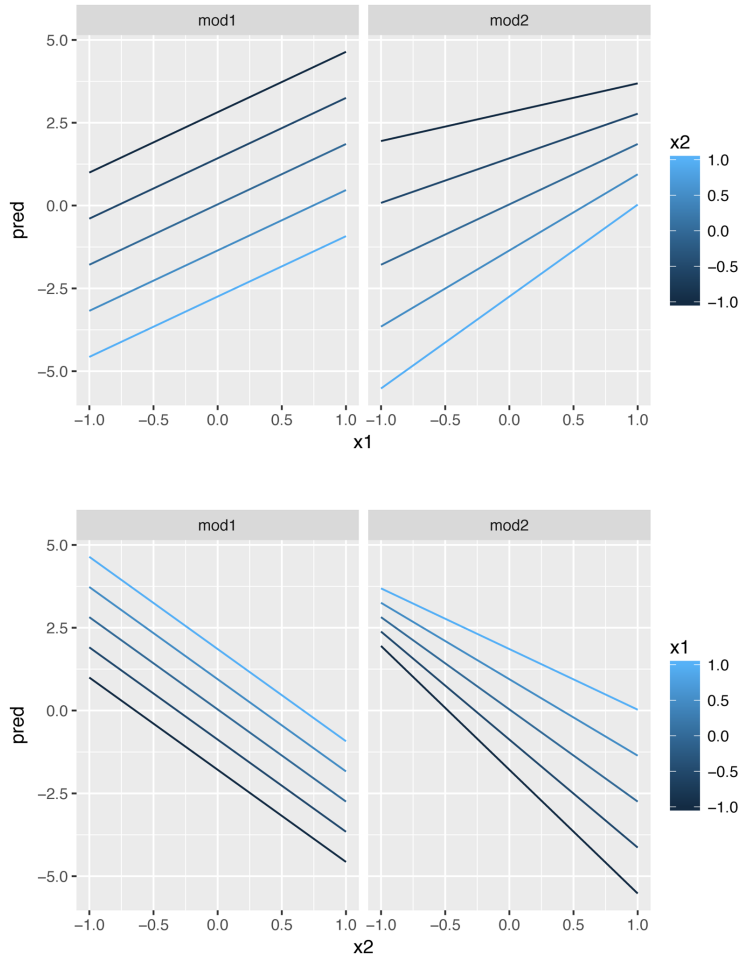
Next let's try and visualize that model. We have two continuous predictors, so you can imagine the model like a 3D surface. We could display that using `geom_tile()`:

```
ggplot(grid, aes(x1, x2)) +
  geom_tile(aes(fill = pred)) +
  facet_wrap(~ model)
```



That doesn't suggest that the models are very different! But that's partly an illusion: our eyes and brains are not very good at accurately comparing shades of color. Instead of looking at the surface from the top, we could look at it from either side, showing multiple slices:

```
ggplot(grid, aes(x1, pred, color = x2, group = x2)) +
  geom_line() +
  facet_wrap(~ model)
ggplot(grid, aes(x2, pred, color = x1, group = x1)) +
  geom_line() +
  facet_wrap(~ model)
```



This shows you that interaction between two continuous variables works basically the same way as for a categorical and continuous variable. An interaction says that there's not a fixed offset: you need to consider both values of `x1` and `x2` simultaneously in order to predict `y`.

You can see that even with just two continuous variables, coming up with good visualizations are hard. But that's reasonable: you shouldn't expect it will be easy to understand how three or more variables simultaneously interact! But again, we're saved a little because we're using models for exploration, and you can gradually

build up your model over time. The model doesn't have to be perfect, it just has to help you reveal a little more about your data.

I spent some time looking at the residuals to see if I could figure if `mod2` did better than `mod1`. I think it does, but it's pretty subtle. You'll have a chance to work on it in the exercises.

Transformations

You can also perform transformations inside the model formula. For example, `log(y) ~ sqrt(x1) + x2` is transformed to `y = a_1 + a_2 * x1 * sqrt(x) + a_3 * x2`. If your transformation involves `+`, `*`, `^`, or `-`, you'll need to wrap it in `I()` so R doesn't treat it like part of the model specification. For example, `y ~ x + I(x ^ 2)` is translated to `y = a_1 + a_2 * x + a_3 * x^2`. If you forget the `I()` and specify `y ~ x ^ 2 + x`, R will compute `y ~ x * x + x`. `x * x` means the interaction of `x` with itself, which is the same as `x`. R automatically drops redundant variables so `x + x` becomes `x`, meaning that `y ~ x ^ 2 + x` specifies the function `y = a_1 + a_2 * x`. That's probably not what you intended!

Again, if you get confused about what your model is doing, you can always use `model_matrix()` to see exactly what equation `lm()` is fitting:

```
df <- tribble(
  ~y, ~x,
   1,  1,
   2,  2,
   3,  3
)
model_matrix(df, y ~ x^2 + x)
#> # A tibble: 3 × 2
#>   `(Intercept)`     x
#>           <dbl> <dbl>
#> 1             1     1
#> 2             1     2
#> 3             1     3
model_matrix(df, y ~ I(x^2) + x)
#> # A tibble: 3 × 3
#>   `(Intercept)` `I(x^2)`     x
#>           <dbl>    <dbl> <dbl>
#> 1             1        1     1
#> 2             1        4     2
#> 3             1        9     3
```

Transformations are useful because you can use them to approximate nonlinear functions. If you've taken a calculus class, you may have heard of Taylor's theorem, which says you can approximate any smooth function with an infinite sum of polynomials. That means you can use a linear function to get arbitrarily close to a smooth function by fitting an equation like `y = a_1 + a_2 * x + a_3 * x^2 + a_4 * x ^ 3`. Typing that sequence by hand is tedious, so R provides a helper function, `poly()`:

```
model_matrix(df, y ~ poly(x, 2))
#> # A tibble: 3 × 3
#>   `(Intercept)` `poly(x, 2)1` `poly(x, 2)2`
#>           <dbl>         <dbl>         <dbl>
#> 1             1     -7.07e-01         0.408
#> 2             1     -7.85e-17        -0.816
#> 3             1      7.07e-01         0.408
```

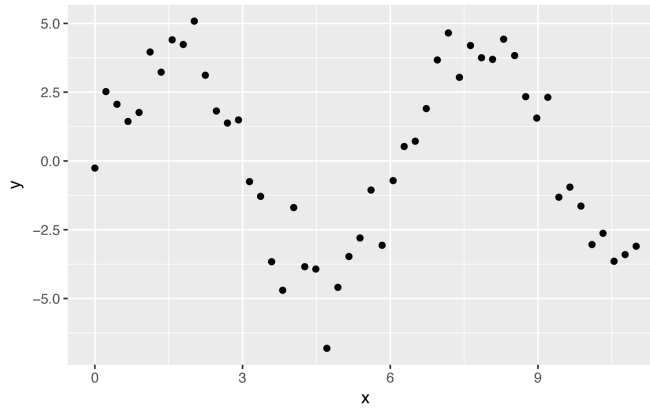
However there's one major problem with using `poly()`: outside the range of the data, polynomials rapidly shoot off to positive or negative infinity. One safer alternative is to use the natural spline, `splines::ns()`:

```
library(splines)
model_matrix(df, y ~ ns(x, 2))
#> # A tibble: 3 × 3
#>   `(Intercept)` `ns(x, 2)1` `ns(x, 2)2`
#>           <dbl>       <dbl>       <dbl>
#> 1             1       0.000       0.000
#> 2             1       0.566      -0.211
#> 3             1       0.344       0.771
```

Let's see what that looks like when we try and approximate a nonlinear function:

```
sim5 <- tibble(
  x = seq(0, 3.5 * pi, length = 50),
  y = 4 * sin(x) + rnorm(length(x))
)

ggplot(sim5, aes(x, y)) +
  geom_point()
```

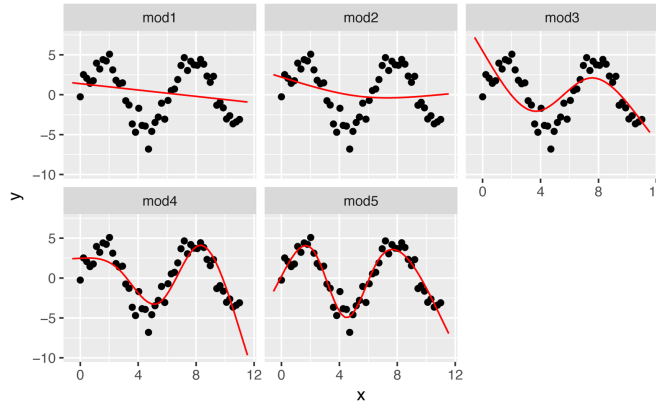


I'm going to fit five models to this data:

```
mod1 <- lm(y ~ ns(x, 1), data = sim5)
mod2 <- lm(y ~ ns(x, 2), data = sim5)
mod3 <- lm(y ~ ns(x, 3), data = sim5)
mod4 <- lm(y ~ ns(x, 4), data = sim5)
mod5 <- lm(y ~ ns(x, 5), data = sim5)

grid <- sim5 %>%
  data_grid(x = seq_range(x, n = 50, expand = 0.1)) %>%
  gather_predictions(mod1, mod2, mod3, mod4, mod5, .pred = "y")

ggplot(sim5, aes(x, y)) +
  geom_point() +
  geom_line(data = grid, color = "red") +
  facet_wrap(~ model)
```



Notice that the extrapolation outside the range of the data is clearly bad. This is the downside to approximating a function with a polynomial. But this is a very real problem with every model: the model can never tell you if the behavior is true when you start extrapolating outside the range of the data that you have seen. You must rely on theory and science.

Exercises

1. What happens if you repeat the analysis of `sim2` using a model without an intercept? What happens to the model equation? What happens to the predictions?
2. Use `model_matrix()` to explore the equations generated for the models I fit to `sim3` and `sim4`. Why is `*` a good shorthand for interaction?
3. Using the basic principles, convert the formulas in the following two models into functions. (Hint: start by converting the categorical variable into 0-1 variables.)

```
mod1 <- lm(y ~ x1 + x2, data = sim3)
mod2 <- lm(y ~ x1 * x2, data = sim3)
```

4. For `sim4`, which of `mod1` and `mod2` is better? I think `mod2` does a slightly better job at removing patterns, but it's pretty subtle. Can you come up with a plot to support my claim?

Missing Values

Missing values obviously cannot convey any information about the relationship between the variables, so modeling functions will drop any rows that contain missing values. R's default behavior is to silently drop them, but `options(na.action = na.warn)` (run in the prerequisites), makes sure you get a warning:

```
df <- tribble(
  ~x, ~y,
  1, 2.2,
  2, NA,
  3, 3.5,
  4, 8.3,
  NA, 10
)
```

```
mod <- lm(y ~ x, data = df)
#> Warning: Dropping 2 rows with missing values
```

To suppress the warning, set `na.action = na.exclude`:

```
mod <- lm(y ~ x, data = df, na.action = na.exclude)
```

You can always see exactly how many observations were used with `nobs()`:

```
nobs(mod)
#> [1] 3
```

Other Model Families

This chapter has focused exclusively on the class of linear models, which assume a relationship of the form `y = a_1 * x1 + a_2 * x2 + ... + a_n * xn`. Linear models additionally assume that the residuals have a normal distribution, which we haven't talked about. There is a large set of model classes that extend the linear model in various interesting ways. Some of them are:

- *Generalized linear models*, e.g., `stats::glm()`. Linear models assume that the response is continuous and the error has a normal distribution. Generalized linear models extend linear models to include noncontinuous responses (e.g., binary data or counts). They work by defining a distance metric based on the statistical idea of likelihood.
- *Generalized additive models*, e.g., `mgcv::gam()`, extend generalized linear models to incorporate arbitrary smooth functions. That means you can write a formula like `y ~ s(x)`, which becomes an equation like `y = f(x)`, and let `gam()` estimate what that function is (subject to some smoothness constraints to make the problem tractable).
- *Penalized linear models*, e.g., `glmnet::glmnet()`, add a penalty term to the distance that penalizes complex models (as defined by the distance between the parameter vector and the origin). This tends to make models that generalize better to new datasets from the same population.
- *Robust linear models*, e.g., `MASS:rlm()`, tweak the distance to downweight points that are very far away. This makes them less sensitive to the presence of outliers, at the cost of being not quite as good when there are no outliers.

- *Trees*, e.g., `rpart::rpart()`, attack the problem in a completely different way than linear models. They fit a piece-wise constant model, splitting the data into progressively smaller and smaller pieces. Trees aren't terribly effective by themselves, but they are very powerful when used in aggregate by models like *random forests* (e.g., `randomForest::randomForest()`) or *gradient boosting machines* (e.g., `xgboost::xgboost`.)

These models all work similarly from a programming perspective. Once you've mastered linear models, you should find it easy to master the mechanics of these other model classes. Being a skilled modeler is a mixture of some good general principles and having a big toolbox of techniques. Now that you've learned some general tools and one useful class of models, you can go on and learn more classes from other sources.

CHAPTER 19

Model Building

Introduction

In the previous chapter you learned how linear models worked, and learned some basic tools for understanding what a model is telling you about your data. The previous chapter focused on simulated datasets to help you learn about how models work. This chapter will focus on real data, showing you how you can progressively build up a model to aid your understanding of the data.

We will take advantage of the fact that you can think about a model partitioning your data into patterns and residuals. We'll find patterns with visualization, then make them concrete and precise with a model. We'll then repeat the process, but replace the old response variable with the residuals from the model. The goal is to transition from implicit knowledge in the data and your head to explicit knowledge in a quantitative model. This makes it easier to apply to new domains, and easier for others to use.

For very large and complex datasets this will be a lot of work. There are certainly alternative approaches—a more machine learning approach is simply to focus on the predictive ability of the model. These approaches tend to produce black boxes: the model does a really good job at generating predictions, but you don't know why. This is a totally reasonable approach, but it does make it hard to apply your real-world knowledge to the model. That, in turn, makes it difficult to assess whether or not the model will continue to work in the long term, as fundamentals change. For most real models, I'd

expect you to use some combination of this approach and a more classic automated approach.

It's a challenge to know when to stop. You need to figure out when your model is good enough, and when additional investment is unlikely to pay off. I particularly like this quote from reddit user Broseidon241:

> A long time ago in art class, my teacher told me "An artist needs to know when a piece is done. You can't tweak something into perfection—wrap it up. If you don't like it, do it over again. Otherwise begin something new." Later in life, I heard "A poor seamstress makes many mistakes. A good seamstress works hard to correct those mistakes. A great seamstress isn't afraid to throw out the garment and start over."
>
> —Broseidon241

Prerequisites

We'll use the same tools as in the previous chapter, but add in some real datasets: `diamonds` from **ggplot2**, and `flights` from **nycflights13**. We'll also need **lubridate** in order to work with the date/times in `flights`.

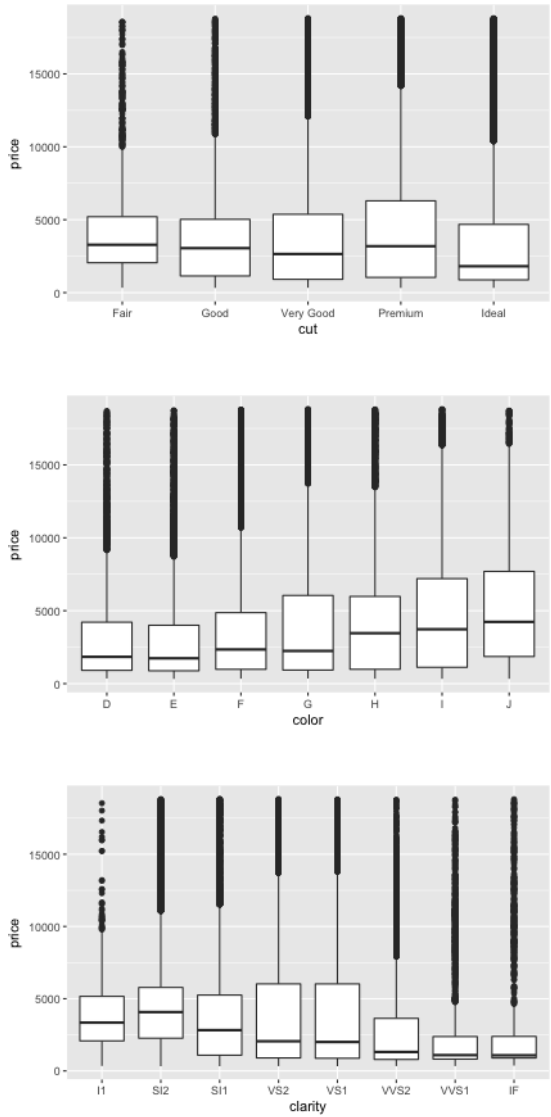
```
library(tidyverse)
library(modelr)
options(na.action = na.warn)

library(nycflights13)
library(lubridate)
```

Why Are Low-Quality Diamonds More Expensive?

In previous chapters we've seen a surprising relationship between the quality of diamonds and their price: low-quality diamonds (poor cuts, bad colors, and inferior clarity) have higher prices:

```
ggplot(diamonds, aes(cut, price)) + geom_boxplot()
ggplot(diamonds, aes(color, price)) + geom_boxplot()
ggplot(diamonds, aes(clarity, price)) + geom_boxplot()
```



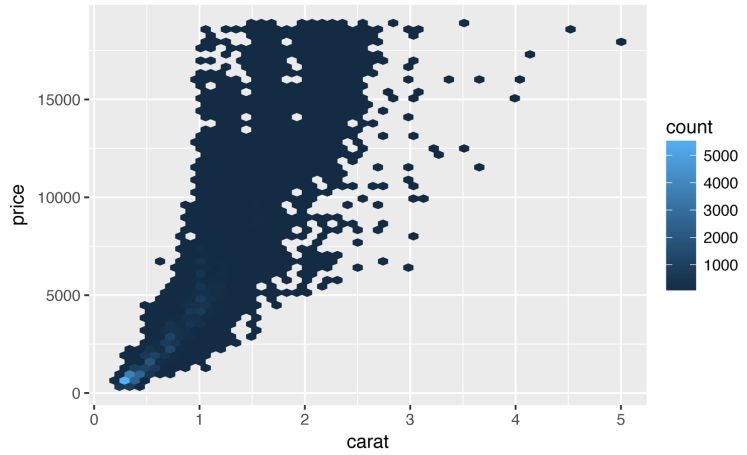
Note that the worst diamond color is J (slightly yellow), and the worst clarity is I1 (inclusions visible to the naked eye).

Price and Carat

It looks like lower-quality diamonds have higher prices because there is an important confounding variable: the weight (`carat`) of

the diamond. The weight of the diamond is the single most important factor for determining the price of the diamond, and lower-quality diamonds tend to be larger:

```
ggplot(diamonds, aes(carat, price)) +
  geom_hex(bins = 50)
```



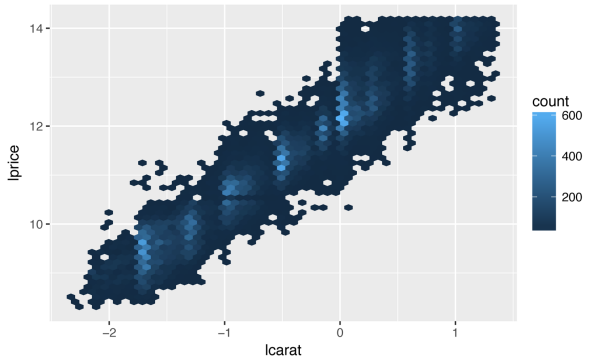
We can make it easier to see how the other attributes of a diamond affect its relative `price` by fitting a model to separate out the effect of `carat`. But first, let's make a couple of tweaks to the diamonds dataset to make it easier to work with:

1. Focus on diamonds smaller than 2.5 carats (99.7% of the data).
2. Log-transform the carat and price variables:

```
diamonds2 <- diamonds %>%
  filter(carat <= 2.5) %>%
  mutate(lprice = log2(price), lcarat = log2(carat))
```

Together, these changes make it easier to see the relationship between `carat` and `price`:

```
ggplot(diamonds2, aes(lcarat, lprice)) +
  geom_hex(bins = 50)
```



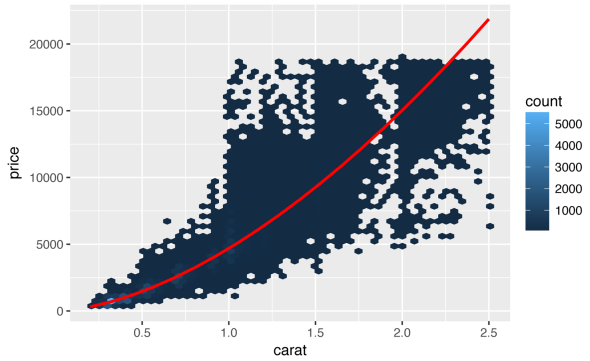
The log transformation is particularly useful here because it makes the pattern linear, and linear patterns are the easiest to work with. Let's take the next step and remove that strong linear pattern. We first make the pattern explicit by fitting a model:

```
mod_diamond <- lm(lprice ~ lcarat, data = diamonds2)
```

Then we look at what the model tells us about the data. Note that I back-transform the predictions, undoing the log transformation, so I can overlay the predictions on the raw data:

```
grid <- diamonds2 %>%
  data_grid(carat = seq_range(carat, 20)) %>%
  mutate(lcarat = log2(carat)) %>%
  add_predictions(mod_diamond, "lprice") %>%
  mutate(price = 2 ^ lprice)

ggplot(diamonds2, aes(carat, price)) +
  geom_hex(bins = 50) +
  geom_line(data = grid, color = "red", size = 1)
```

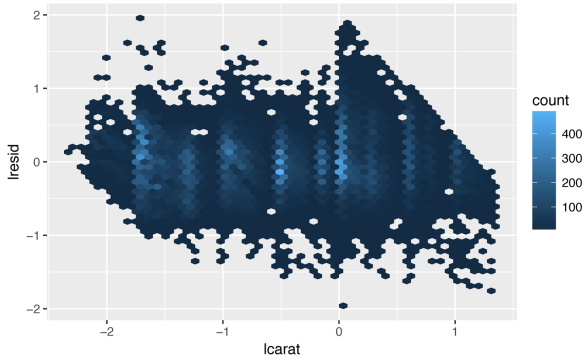


That tells us something interesting about our data. If we believe our model, then the large diamonds are much cheaper than expected. This is probably because no diamond in this dataset costs more than $19,000.

Now we can look at the residuals, which verifies that we've successfully removed the strong linear pattern:

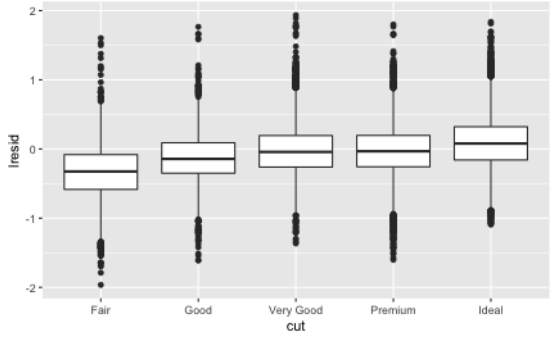
```
diamonds2 <- diamonds2 %>%
  add_residuals(mod_diamond, "lresid")

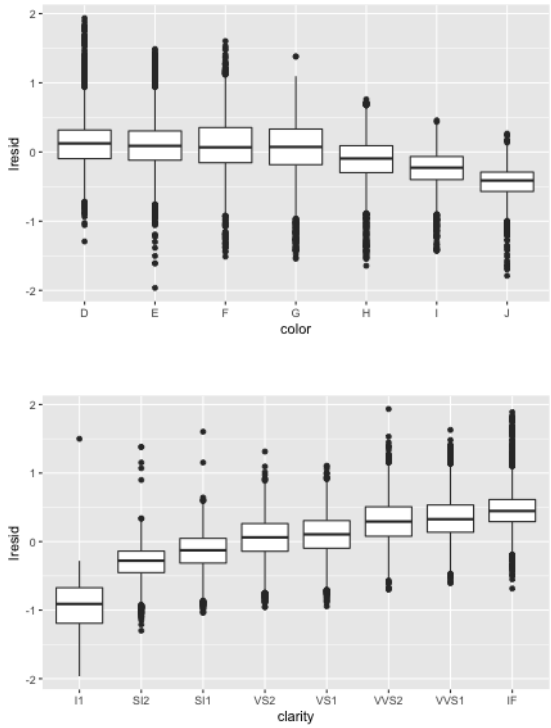
ggplot(diamonds2, aes(lcarat, lresid)) +
  geom_hex(bins = 50)
```



Importantly, we can now redo our motivating plots using those residuals instead of `price`:

```
ggplot(diamonds2, aes(cut, lresid)) + geom_boxplot()
ggplot(diamonds2, aes(color, lresid)) + geom_boxplot()
ggplot(diamonds2, aes(clarity, lresid)) + geom_boxplot()
```





Now we see the relationship we expect: as the quality of the diamond increases, so to does its relative price. To interpret the y-axis, we need to think about what the residuals are telling us, and what scale they are on. A residual of –1 indicates that `lprice` was 1 unit lower than a prediction based solely on its weight. 2^{-1} is 1/2, so points with a value of –1 are half the expected price, and residuals with value 1 are twice the predicted price.

A More Complicated Model

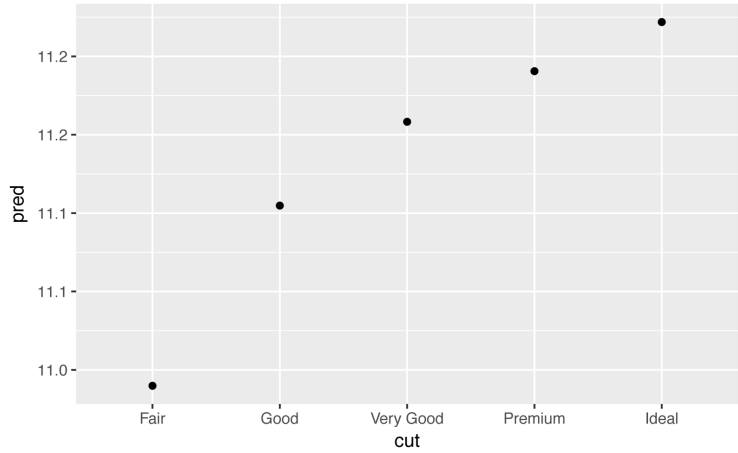
If we wanted to, we could continue to build up our model, moving the effects we've observed into the model to make them explicit. For example, we could include `color`, `cut`, and `clarity` into the model so that we also make explicit the effect of these three categorical variables:

```
mod_diamond2 <- lm(
  lprice ~ lcarat + color + cut + clarity,
  data = diamonds2
)
```

This model now includes four predictors, so it's getting harder to visualize. Fortunately, they're currently all independent, which means that we can plot them individually in four plots. To make the process a little easier, we're going to use the `.model` argument to `data_grid`:

```
grid <- diamonds2 %>%
  data_grid(cut, .model = mod_diamond2) %>%
  add_predictions(mod_diamond2)
grid
#> # A tibble: 5 × 5
#>         cut lcarat color clarity  pred
#>       <ord>  <dbl> <chr>   <chr> <dbl>
#> 1      Fair -0.515     G     SI1  11.0
#> 2      Good -0.515     G     SI1  11.1
#> 3 Very Good -0.515     G     SI1  11.2
#> 4   Premium -0.515     G     SI1  11.2
#> 5     Ideal -0.515     G     SI1  11.2

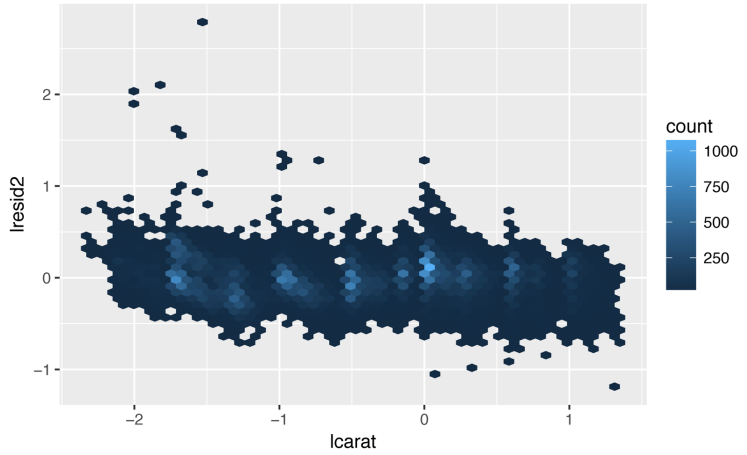
ggplot(grid, aes(cut, pred)) +
  geom_point()
```



If the model needs variables that you haven't explicitly supplied, `data_grid()` will automatically fill them in with the "typical" value. For continuous variables, it uses the median, and for categorical variables, it uses the most common value (or values, if there's a tie):

```
diamonds2 <- diamonds2 %>%
  add_residuals(mod_diamond2, "lresid2")
```

```
ggplot(diamonds2, aes(lcarat, lresid2)) +
  geom_hex(bins = 50)
```



This plot indicates that there are some diamonds with quite large residuals—remember a residual of 2 indicates that the diamond is 4x the price that we expected. It's often useful to look at unusual values individually:

```
diamonds2 %>%
  filter(abs(lresid2) > 1) %>%
  add_predictions(mod_diamond2) %>%
  mutate(pred = round(2 ^ pred)) %>%
  select(price, pred, carat:table, x:z) %>%
  arrange(price)
#> # A tibble: 16 × 11
#>   price  pred carat     cut color clarity depth table     x
#>   <int> <dbl> <dbl>   <ord> <ord>   <ord> <dbl> <dbl> <dbl>
#> 1  1013   264  0.25    Fair     F     SI2  54.4    64  4.30
#> 2  1186   284  0.25 Premium     G     SI2  59.0    60  5.33
#> 3  1186   284  0.25 Premium     G     SI2  58.8    60  5.33
#> 4  1262  2644  1.03    Fair     E      I1  78.2    54  5.72
#> 5  1415   639  0.35    Fair     G     VS2  65.9    54  5.57
#> 6  1415   639  0.35    Fair     G     VS2  65.9    54  5.57
#> # ... with 10 more rows, and 2 more variables: y <dbl>,
#> #   z <dbl>
```

Nothing really jumps out at me here, but it's probably worth spending time considering if this indicates a problem with our model, or if there are errors in the data. If there are mistakes in the data, this could be an opportunity to buy diamonds that have been priced low incorrectly.

Exercises

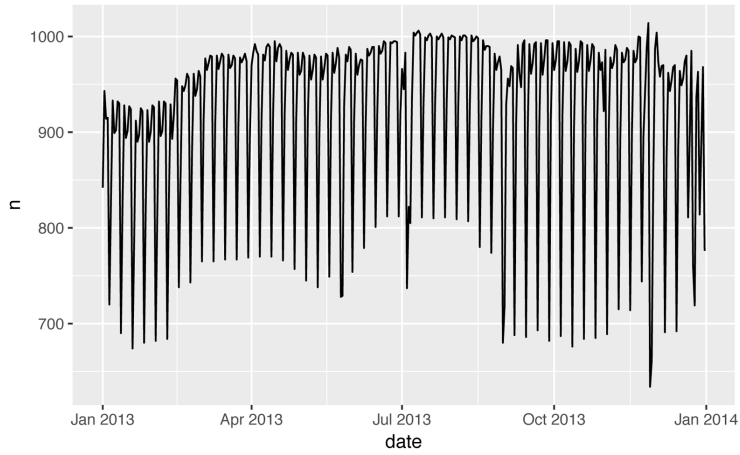
1. In the plot of `lcarat` versus `lprice`, there are some bright vertical strips. What do they represent?
2. If `log(price) = a_0 + a_1 * log(carat)`, what does that say about the relationship between `price` and `carat`?
3. Extract the diamonds that have very high and very low residuals. Is there anything unusual about these diamonds? Are they particularly bad or good, or do you think these are pricing errors?
4. Does the final model, `mod_diamonds2`, do a good job of predicting diamond prices? Would you trust it to tell you how much to spend if you were buying a diamond?

What Affects the Number of Daily Flights?

Let's work through a similar process for a dataset that seems even simpler at first glance: the number of flights that leave NYC per day. This is a really small dataset—only 365 rows and 2 columns—and we're not going to end up with a fully realized model, but as you'll see, the steps along the way will help us better understand the data. Let's get started by counting the number of flights per day and visualizing it with **ggplot2**:

```
daily <- flights %>%
  mutate(date = make_date(year, month, day)) %>%
  group_by(date) %>%
  summarize(n = n())
daily
#> # A tibble: 365 × 2
#>         date     n
#>       <date> <int>
#> 1 2013-01-01   842
#> 2 2013-01-02   943
#> 3 2013-01-03   914
#> 4 2013-01-04   915
#> 5 2013-01-05   720
#> 6 2013-01-06   832
#> # ... with 359 more rows

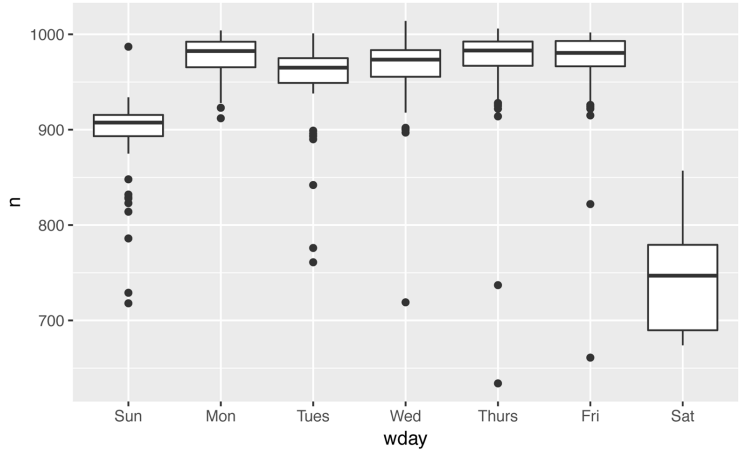
ggplot(daily, aes(date, n)) +
  geom_line()
```



Day of Week

Understanding the long-term trend is challenging because there's a very strong day-of-week effect that dominates the subtler patterns. Let's start by looking at the distribution of flight numbers by day of week:

```
daily <- daily %>%
  mutate(wday = wday(date, label = TRUE))
ggplot(daily, aes(wday, n)) +
  geom_boxplot()
```



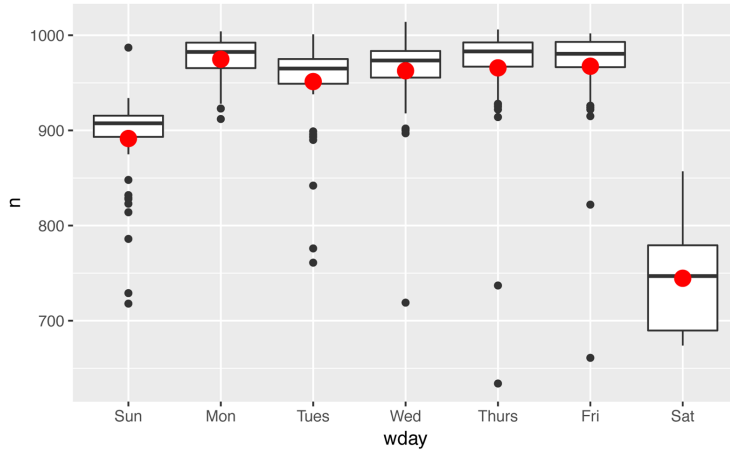
There are fewer flights on weekends because most travel is for business. The effect is particularly pronounced on Saturday: you might sometimes leave on Sunday for a Monday morning meeting, but it's very rare that you'd leave on Saturday as you'd much rather be at home with your family.

One way to remove this strong pattern is to use a model. First, we fit the model, and display its predictions overlaid on the original data:

```
mod <- lm(n ~ wday, data = daily)

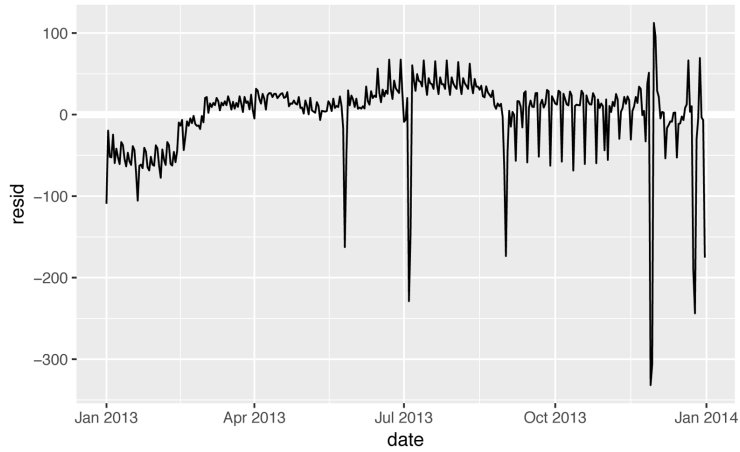
grid <- daily %>%
  data_grid(wday) %>%
  add_predictions(mod, "n")

ggplot(daily, aes(wday, n)) +
  geom_boxplot() +
  geom_point(data = grid, color = "red", size = 4)
```



Next we compute and visualize the residuals:

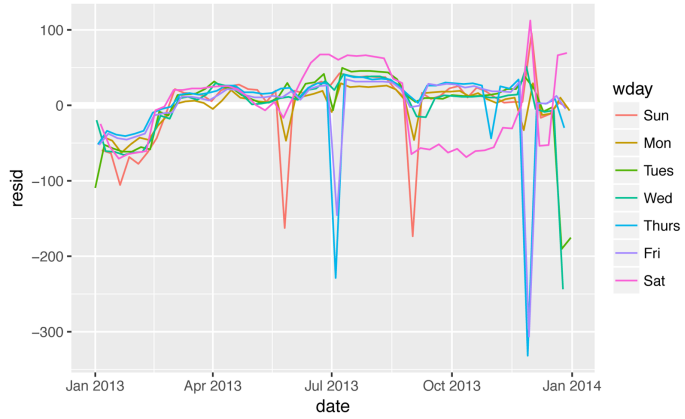
```
daily <- daily %>%
  add_residuals(mod)
daily %>%
  ggplot(aes(date, resid)) +
  geom_ref_line(h = 0) +
  geom_line()
```



Note the change in the y-axis: now we are seeing the deviation from the expected number of flights, given the day of week. This plot is useful because now that we've removed much of the large day-of-week effect, we can see some of the subtler patterns that remain:

- Our model seems to fail starting in June: you can still see a strong regular pattern that our model hasn't captured. Drawing a plot with one line for each day of the week makes the cause easier to see:

```
ggplot(daily, aes(date, resid, color = wday)) +
  geom_ref_line(h = 0) +
  geom_line()
```



Our model fails to accurately predict the number of flights on Saturday: during summer there are more flights than we expect, and during fall there are fewer. We'll see how we can do better to capture this pattern in the next section.

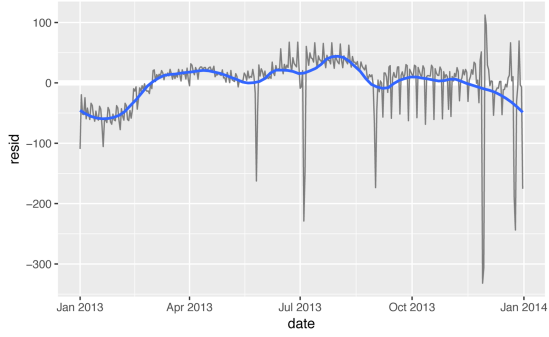
- There are some days with far fewer flights than expected:

```
daily %>%
  filter(resid < -100)
#> # A tibble: 11 × 4
#>         date     n  wday resid
#>       <date> <int> <ord> <dbl>
#> 1 2013-01-01   842  Tues  -109
#> 2 2013-01-20   786   Sun  -105
#> 3 2013-05-26   729   Sun  -162
#> 4 2013-07-04   737 Thurs  -229
#> 5 2013-07-05   822   Fri  -145
#> 6 2013-09-01   718   Sun  -173
#> # ... with 5 more rows
```

  If you're familiar with American public holidays, you might spot New Year's Day, July 4th, Thanksgiving, and Christmas. There are some others that don't seem to correspond to public holidays. You'll work on those in one of the exercises.

- There seems to be some smoother long-term trend over the course of a year. We can highlight that trend with `geom_smooth()`:

```
daily %>%
  ggplot(aes(date, resid)) +
  geom_ref_line(h = 0) +
  geom_line(color = "grey50") +
  geom_smooth(se = FALSE, span = 0.20)
#> `geom_smooth()` using method = 'loess'
```

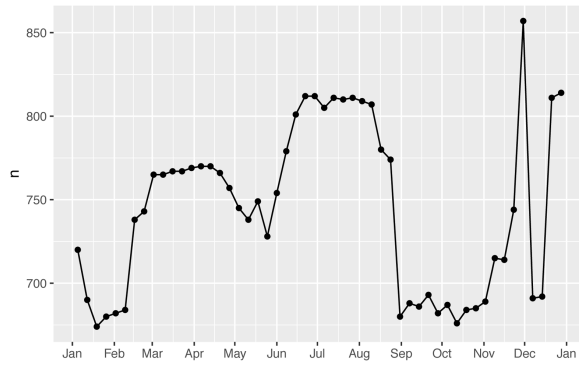


There are fewer flights in January (and December), and more in summer (May–Sep). We can't do much with this pattern quantitatively, because we only have a single year of data. But we can use our domain knowledge to brainstorm potential explanations.

Seasonal Saturday Effect

Let's first tackle our failure to accurately predict the number of flights on Saturday. A good place to start is to go back to the raw numbers, focusing on Saturdays:

```
daily %>%
  filter(wday == "Sat") %>%
  ggplot(aes(date, n)) +
    geom_point() +
    geom_line() +
    scale_x_date(
      NULL,
      date_breaks = "1 month",
      date_labels = "%b"
    )
```



(I've used both points and lines to make it more clear what is data and what is interpolation.)

I suspect this pattern is caused by summer holidays: many people go on holiday in the summer, and people don't mind travelling on Saturdays for vacation. Looking at this plot, we might guess that summer holidays are from early June to late August. That seems to line up fairly well with the state's school terms: summer break in 2013 was June 26–September 9.

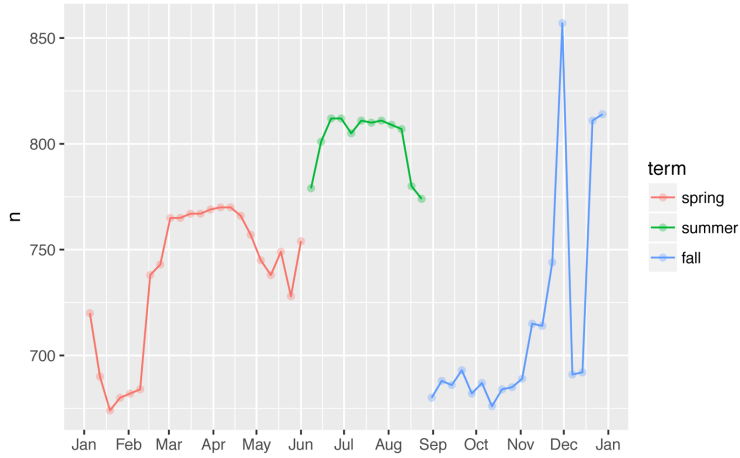
Why are there more Saturday flights in the spring than the fall? I asked some American friends and they suggested that it's less common to plan family vacations during the fall because of the big Thanksgiving and Christmas holidays. We don't have the data to know for sure, but it seems like a plausible working hypothesis.

Let's create a "term" variable that roughly captures the three school terms, and check our work with a plot:

```
term <- function(date) {
  cut(date,
    breaks = ymd(20130101, 20130605, 20130825, 20140101),
    labels = c("spring", "summer", "fall")
  )
}

daily <- daily %>%
  mutate(term = term(date))

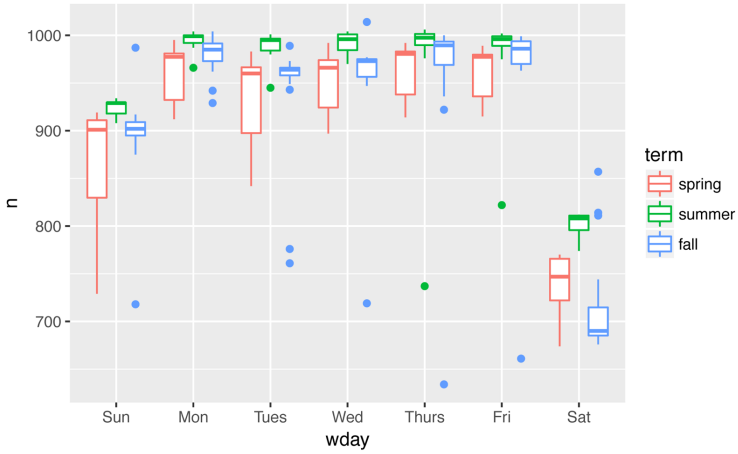
daily %>%
  filter(wday == "Sat") %>%
  ggplot(aes(date, n, color = term)) +
  geom_point(alpha = 1/3) +
  geom_line() +
  scale_x_date(
    NULL,
    date_breaks = "1 month",
    date_labels = "%b"
  )
```



(I manually tweaked the dates to get nice breaks in the plot. Using a visualization to help you understand what your function is doing is a really powerful and general technique.)

It's useful to see how this new variable affects the other days of the week:

```
daily %>%
  ggplot(aes(wday, n, color = term)) +
    geom_boxplot()
```



It looks like there is significant variation across the terms, so fitting a separate day-of-week effect for each term is reasonable. This improves our model, but not as much as we might hope:

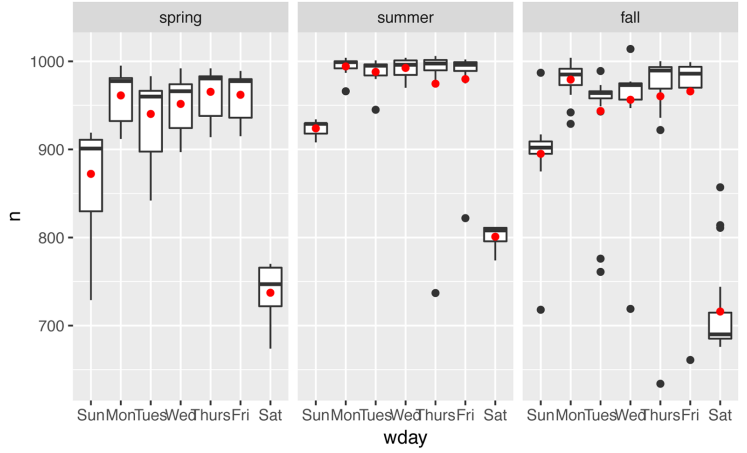
```
mod1 <- lm(n ~ wday, data = daily)
mod2 <- lm(n ~ wday * term, data = daily)

daily %>%
  gather_residuals(without_term = mod1, with_term = mod2) %>%
  ggplot(aes(date, resid, color = model)) +
    geom_line(alpha = 0.75)
```

We can see the problem by overlaying the predictions from the model onto the raw data:

```
grid <- daily %>%
  data_grid(wday, term) %>%
  add_predictions(mod2, "n")

ggplot(daily, aes(wday, n)) +
  geom_boxplot() +
  geom_point(data = grid, color = "red") +
  facet_wrap(~ term)
```

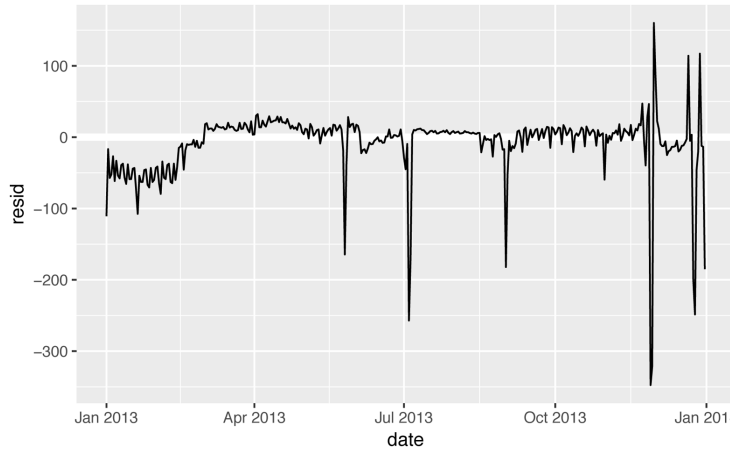


Our model is finding the *mean* effect, but we have a lot of big outliers, so the mean tends to be far away from the typical value. We can

alleviate this problem by using a model that is robust to the effect of outliers: `MASS::rlm()`. This greatly reduces the impact of the outliers on our estimates, and gives a model that does a good job of removing the day-of-week pattern:

```
mod3 <- MASS::rlm(n ~ wday * term, data = daily)

daily %>%
  add_residuals(mod3, "resid") %>%
  ggplot(aes(date, resid)) +
  geom_hline(yintercept = 0, size = 2, color = "white") +
  geom_line()
```



It's now much easier to see the long-term trend, and the positive and negative outliers.

Computed Variables

If you're experimenting with many models and many visualizations, it's a good idea to bundle the creation of variables up into a function so there's no chance of accidentally applying a different transformation in different places. For example, we could write:

```
compute_vars <- function(data) {
  data %>%
    mutate(
      term = term(date),
      wday = wday(date, label = TRUE)
    )
}
```

Another option is to put the transformations directly in the model formula:

```
wday2 <- function(x) wday(x, label = TRUE)
mod3 <- lm(n ~ wday2(date) * term(date), data = daily)
```

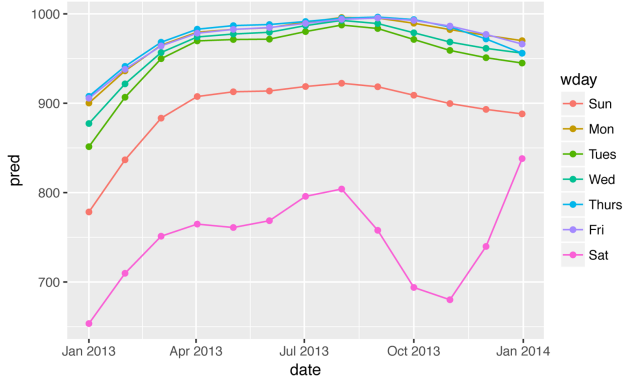
Either approach is reasonable. Making the transformed variable explicit is useful if you want to check your work, or use them in a visualization. But you can't easily use transformations (like splines) that return multiple columns. Including the transformations in the model function makes life a little easier when you're working with many different datasets because the model is self-contained.

Time of Year: An Alternative Approach

In the previous section we used our domain knowledge (how the US school term affects travel) to improve the model. An alternative to making our knowledge explicit in the model is to give the data more room to speak. We could use a more flexible model and allow that to capture the pattern we're interested in. A simple linear trend isn't adequate, so we could try using a natural spline to fit a smooth curve across the year:

```
library(splines)
mod <- MASS::rlm(n ~ wday * ns(date, 5), data = daily)

daily %>%
  data_grid(wday, date = seq_range(date, n = 13)) %>%
  add_predictions(mod) %>%
  ggplot(aes(date, pred, color = wday)) +
    geom_line() +
    geom_point()
```



We see a strong pattern in the numbers of Saturday flights. This is reassuring, because we also saw that pattern in the raw data. It's a good sign when you get the same signal from different approaches.

Exercises

1. Use your Google sleuthing skills to brainstorm why there were fewer than expected flights on January 20, May 26, and September 1. (Hint: they all have the same explanation.) How would these days generalize to another year?

2. What do the three days with high positive residuals represent? How would these days generalize to another year?

```
daily %>%
  top_n(3, resid)
#> # A tibble: 3 × 5
#>         date     n  wday resid   term
#>       <date> <int> <ord> <dbl> <fctr>
#> 1 2013-11-30   857   Sat 112.4   fall
#> 2 2013-12-01   987   Sun  95.5   fall
#> 3 2013-12-28   814   Sat  69.4   fall
```

3. Create a new variable that splits the `wday` variable into terms, but only for Saturdays, i.e., it should have `Thurs`, `Fri`, but `Sat-summer`, `Sat-spring`, `Sat-fall`. How does this model compare with the model with every combination of `wday` and `term`?

4. Create a new `wday` variable that combines the day of week, term (for Saturdays), and public holidays. What do the residuals of that model look like?

5. What happens if you fit a day-of-week effect that varies by month (i.e., `n ~ wday * month`)? Why is this not very helpful?

6. What would you expect the model `n ~ wday + ns(date, 5)` to look like? Knowing what you know about the data, why would you expect it to be not particularly effective?

7. We hypothesized that people leaving on Sundays are more likely to be business travelers who need to be somewhere on Monday. Explore that hypothesis by seeing how it breaks down based on distance and time: if it's true, you'd expect to see more Sunday evening flights to places that are far away.

8. It's a little frustrating that Sunday and Saturday are on separate ends of the plot. Write a small function to set the levels of the factor so that the week starts on Monday.

Learning More About Models

We have only scratched the absolute surface of modeling, but you have hopefully gained some simple, but general-purpose tools that you can use to improve your own data analyses. It's OK to start simple! As you've seen, even very simple models can make a dramatic difference in your ability to tease out interactions between variables.

These modeling chapters are even more opinionated than the rest of the book. I approach modeling from a somewhat different perspective to most others, and there is relatively little space devoted to it. Modeling really deserves a book on its own, so I'd highly recommend that you read at least one of these three books:

- *Statistical Modeling: A Fresh Approach* by Danny Kaplan. This book provides a gentle introduction to modeling, where you build your intuition, mathematical tools, and R skills in parallel. The book replaces a traditional "introduction to statistics" course, providing a curriculum that is up-to-date and relevant to data science.
- *An Introduction to Statistical Learning* by Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani (available online for free). This book presents a family of modern modeling techniques collectively known as statistical learning. For an even deeper understanding of the math behind the models, read the classic *Elements of Statistical Learning* by Trevor Hastie, Robert Tibshirani, and Jerome Friedman (also available online for free).
- *Applied Predictive Modeling* by Max Kuhn and Kjell Johnson. This book is a companion to the **caret** package and provides practical tools for dealing with real-life predictive modeling challenges.

CHAPTER 20

Many Models with purrr and broom

Introduction

In this chapter you're going to learn three powerful ideas that help you to work with large numbers of models with ease:

- Using many simple models to better understand complex datasets.
- Using list-columns to store arbitrary data structures in a data frame. For example, this will allow you to have a column that contains linear models.
- Using the **broom** package, by David Robinson, to turn models into tidy data. This is a powerful technique for working with large numbers of models because once you have tidy data, you can apply all of the techniques that you've learned about earlier in the book.

We'll start by diving into a motivating example using data about life expectancy around the world. It's a small dataset but it illustrates how important modeling can be for improving your visualizations. We'll use a large number of simple models to partition out some of the strongest signals so we can see the subtler signals that remain. We'll also see how model summaries can help us pick out outliers and unusual trends.

The following sections will dive into more detail about the individual techniques:

- In "gapminder" on page 398, you'll see a motivating example that puts list-columns to use to fit per-county models to world economic data.
- In "List-Columns" on page 402, you'll learn more about the list-column data structure, and why it's valid to put lists in data frames.
- In "Creating List-Columns" on page 411, you'll learn the three main ways in which you'll create list-columns.
- In "Simplifying List-Columns" on page 416 you'll learn how to convert list-columns back to regular atomic vectors (or sets of atomic vectors) so you can work with them more easily.
- In "Making Tidy Data with broom" on page 419, you'll learn about the full set of tools provided by broom, and see how they can be applied to other types of data structure.

This chapter is somewhat aspirational: if this book is your first introduction to R, this chapter is likely to be a struggle. It requires you to have deeply internalized ideas about modeling, data structures, and iteration. So don't worry if you don't get it—just put this chapter aside for a few months, and come back when you want to stretch your brain.

Prerequisites

Working with many models requires many of the packages of the tidyverse (for data exploration, wrangling, and programming) and **modelr** to facilitate modeling.

```
library(modelr)
library(tidyverse)
```

gapminder

To motivate the power of many simple models, we're going to look into the "gapminder" data. This data was popularized by Hans Rosling, a Swedish doctor and statistician. If you've never heard of him, stop reading this chapter right now and go watch one of his videos! He is a fantastic data presenter and illustrates how you can use data

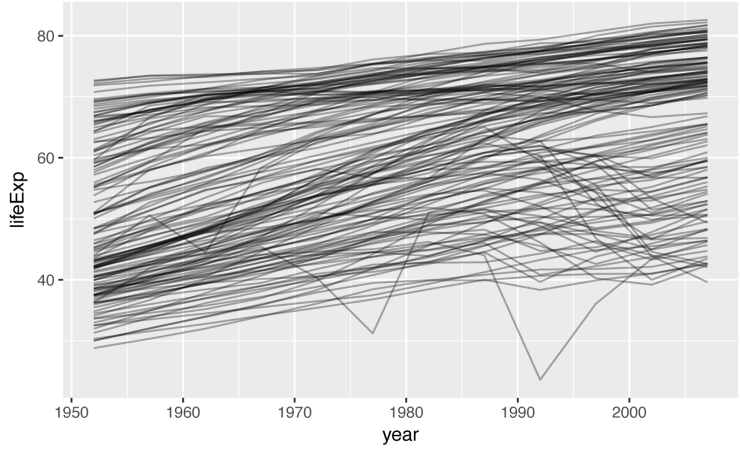
to present a compelling story. A good place to start is this short video filmed in conjunction with the BBC.

The gapminder data summarizes the progression of countries over time, looking at statistics like life expectancy and GDP. The data is easy to access in R, thanks to Jenny Bryan, who created the **gapminder** package:

```
library(gapminder)
gapminder
#> # A tibble: 1,704 × 6
#>       country continent  year lifeExp      pop gdpPercap
#>        <fctr>    <fctr> <int>   <dbl>    <int>     <dbl>
#> 1 Afghanistan      Asia  1952    28.8  8425333       779
#> 2 Afghanistan      Asia  1957    30.3  9240934       821
#> 3 Afghanistan      Asia  1962    32.0 10267083       853
#> 4 Afghanistan      Asia  1967    34.0 11537966       836
#> 5 Afghanistan      Asia  1972    36.1 13079460       740
#> 6 Afghanistan      Asia  1977    38.4 14880372       786
#> # ... with 1,698 more rows
```

In this case study, we're going to focus on just three variables to answer the question "How does life expectancy (`lifeExp`) change over time (`year`) for each country (`country`)?" A good place to start is with a plot:

```
gapminder %>%
  ggplot(aes(year, lifeExp, group = country)) +
    geom_line(alpha = 1/3)
```



This is a small dataset: it only has ~1,700 observations and 3 variables. But it's still hard to see what's going on! Overall, it looks like

life expectancy has been steadily improving. However, if you look closely, you might notice some countries that don't follow this pattern. How can we make those countries easier to see?

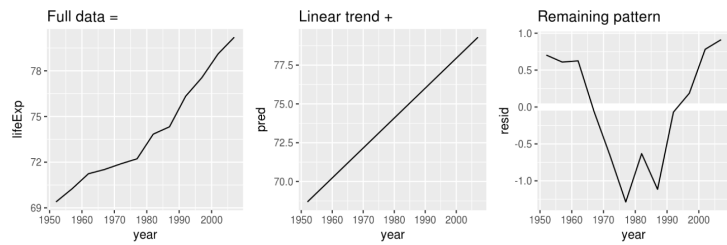
One way is to use the same approach as in the last chapter: there's a strong signal (overall linear growth) that makes it hard to see subtler trends. We'll tease these factors apart by fitting a model with a linear trend. The model captures steady growth over time, and the residuals will show what's left.

You already know how to do that if we had a single country:

```
nz <- filter(gapminder, country == "New Zealand")
nz %>%
  ggplot(aes(year, lifeExp)) +
  geom_line() +
  ggtitle("Full data = ")

nz_mod <- lm(lifeExp ~ year, data = nz)
nz %>%
  add_predictions(nz_mod) %>%
  ggplot(aes(year, pred)) +
  geom_line() +
  ggtitle("Linear trend + ")

nz %>%
  add_residuals(nz_mod) %>%
  ggplot(aes(year, resid)) +
  geom_hline(yintercept = 0, color = "white", size = 3) +
  geom_line() +
  ggtitle("Remaining pattern")
```



How can we easily fit that model to every country?

Nested Data

You could imagine copying and pasting that code multiple times; but you've already learned a better way! Extract out the common

code with a function and repeat using a map function from **purrr**. This problem is structured a little differently to what you've seen before. Instead of repeating an action for each variable, we want to repeat an action for each country, a subset of rows. To do that, we need a new data structure: the *nested data frame*. To create a nested data frame we start with a grouped data frame, and "nest" it:

```
by_country <- gapminder %>%
  group_by(country, continent) %>%
  nest()

by_country
#> # A tibble: 142 × 3
#>       country continent              data
#>        <fctr>    <fctr>            <list>
#> 1 Afghanistan      Asia <tibble [12 × 4]>
#> 2     Albania    Europe <tibble [12 × 4]>
#> 3     Algeria    Africa <tibble [12 × 4]>
#> 4      Angola    Africa <tibble [12 × 4]>
#> 5   Argentina  Americas <tibble [12 × 4]>
#> 6   Australia   Oceania <tibble [12 × 4]>
#> # ... with 136 more rows
```

(I'm cheating a little by grouping on both `continent` and `country`. Given `country`, `continent` is fixed, so this doesn't add any more groups, but it's an easy way to carry an extra variable along for the ride.)

This creates a data frame that has one row per group (per country), and a rather unusual column: `data`. `data` is a list of data frames (or tibbles, to be precise). This seems like a crazy idea: we have a data frame with a column that is a list of other data frames! I'll explain shortly why I think this is a good idea.

The `data` column is a little tricky to look at because it's a moderately complicated list, and we're still working on good tools to explore these objects. Unfortunately using `str()` is not recommended as it will often produce very long output. But if you pluck out a single element from the `data` column you'll see that it contains all the data for that country (in this case, Afghanistan):

```
by_country$data[[1]]
#> # A tibble: 12 × 4
#>    year lifeExp      pop gdpPercap
#>   <int>   <dbl>    <int>     <dbl>
#> 1  1952    28.8  8425333       779
#> 2  1957    30.3  9240934       821
#> 3  1962    32.0 10267083       853
```

```
#> 4  1967    34.0 11537966       836
#> 5  1972    36.1 13079460       740
#> 6  1977    38.4 14880372       786
#> # ... with 6 more rows
```

Note the difference between a standard grouped data frame and a nested data frame: in a grouped data frame, each row is an observation; in a nested data frame, each row is a group. Another way to think about a nested dataset is we now have a meta-observation: a row that represents the complete time course for a country, rather than a single point in time.

List-Columns

Now that we have our nested data frame, we're in a good position to fit some models. We have a model-fitting function:

```
country_model <- function(df) {
  lm(lifeExp ~ year, data = df)
}
```

And we want to apply it to every data frame. The data frames are in a list, so we can use `purrr::map()` to apply `country_model` to each element:

```
models <- map(by_country$data, country_model)
```

However, rather than leaving the list of models as a free-floating object, I think it's better to store it as a column in the `by_country` data frame. Storing related objects in columns is a key part of the value of data frames, and why I think list-columns are such a good idea. In the course of working with these countries, we are going to have lots of lists where we have one element per country. So why not store them all together in one data frame?

In other words, instead of creating a new object in the global environment, we're going to create a new variable in the `by_country` data frame. That's a job for `dplyr::mutate()`:

```
by_country <- by_country %>%
  mutate(model = map(data, country_model))
by_country
#> # A tibble: 142 × 4
#>       country continent              data    model
#>        <fctr>    <fctr>            <list>   <list>
#> 1 Afghanistan      Asia <tibble [12 × 4]> <S3: lm>
#> 2     Albania    Europe <tibble [12 × 4]> <S3: lm>
#> 3     Algeria    Africa <tibble [12 × 4]> <S3: lm>
```

```
#> 4      Angola    Africa <tibble [12 × 4]> <S3: lm>
#> 5   Argentina  Americas <tibble [12 × 4]> <S3: lm>
#> 6   Australia   Oceania <tibble [12 × 4]> <S3: lm>
#> # ... with 136 more rows
```

This has a big advantage: because all the related objects are stored together, you don't need to manually keep them in sync when you filter or arrange. The semantics of the data frame takes care of that for you:

```
by_country %>%
  filter(continent == "Europe")
#> # A tibble: 30 × 4
#>                  country continent              data    model
#>                   <fctr>    <fctr>            <list>   <list>
#> 1                Albania    Europe <tibble [12 × 4]> <S3: lm>
#> 2                Austria    Europe <tibble [12 × 4]> <S3: lm>
#> 3                Belgium    Europe <tibble [12 × 4]> <S3: lm>
#> 4 Bosnia and Herzegovina    Europe <tibble [12 × 4]> <S3: lm>
#> 5               Bulgaria    Europe <tibble [12 × 4]> <S3: lm>
#> 6                Croatia    Europe <tibble [12 × 4]> <S3: lm>
#> # ... with 24 more rows
by_country %>%
  arrange(continent, country)
#> # A tibble: 142 × 4
#>        country continent              data    model
#>         <fctr>    <fctr>            <list>   <list>
#> 1      Algeria    Africa <tibble [12 × 4]> <S3: lm>
#> 2       Angola    Africa <tibble [12 × 4]> <S3: lm>
#> 3        Benin    Africa <tibble [12 × 4]> <S3: lm>
#> 4     Botswana    Africa <tibble [12 × 4]> <S3: lm>
#> 5 Burkina Faso    Africa <tibble [12 × 4]> <S3: lm>
#> 6      Burundi    Africa <tibble [12 × 4]> <S3: lm>
#> # ... with 136 more rows
```

If your list of data frames and list of models were separate objects, you have to remember that whenever you reorder or subset one vector, you need to reorder or subset all the others in order to keep them in sync. If you forget, your code will continue to work, but it will give the wrong answer!

Unnesting

Previously we computed the residuals of a single model with a single dataset. Now we have 142 data frames and 142 models. To compute the residuals, we need to call `add_residuals()` with each model–data pair:

```
by_country <- by_country %>%
  mutate(
    resids = map2(data, model, add_residuals)
  )
by_country
#> # A tibble: 142 × 5
#>       country continent              data    model
#>        <fctr>    <fctr>            <list>   <list>
#> 1 Afghanistan      Asia <tibble [12 × 4]> <S3: lm>
#> 2     Albania    Europe <tibble [12 × 4]> <S3: lm>
#> 3     Algeria    Africa <tibble [12 × 4]> <S3: lm>
#> 4      Angola    Africa <tibble [12 × 4]> <S3: lm>
#> 5   Argentina  Americas <tibble [12 × 4]> <S3: lm>
#> 6   Australia   Oceania <tibble [12 × 4]> <S3: lm>
#> # ... with 136 more rows, and 1 more variable:
#> #   resids <list>
```

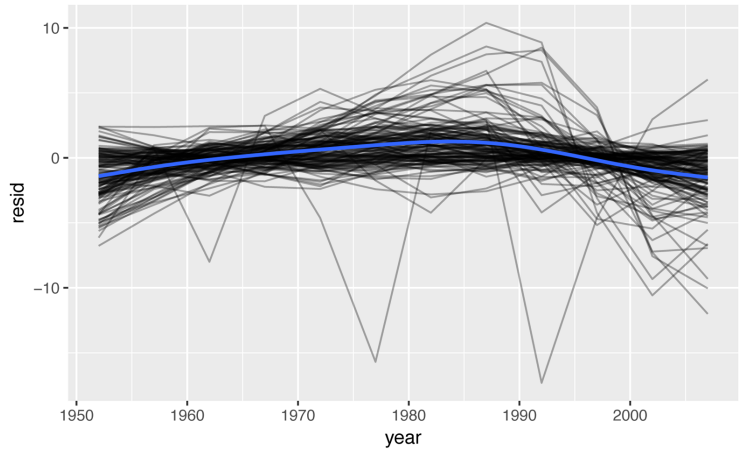
But how can you plot a list of data frames? Instead of struggling to answer that question, let's turn the list of data frames back into a regular data frame. Previously we used `nest()` to turn a regular data frame into a nested data frame, and now we do the opposite with `unnest()`:

```
resids <- unnest(by_country, resids)
resids
#> # A tibble: 1,704 × 7
#>       country continent  year lifeExp      pop gdpPercap
#>        <fctr>    <fctr> <int>   <dbl>    <int>     <dbl>
#> 1 Afghanistan      Asia  1952    28.8  8425333       779
#> 2 Afghanistan      Asia  1957    30.3  9240934       821
#> 3 Afghanistan      Asia  1962    32.0 10267083       853
#> 4 Afghanistan      Asia  1967    34.0 11537966       836
#> 5 Afghanistan      Asia  1972    36.1 13079460       740
#> 6 Afghanistan      Asia  1977    38.4 14880372       786
#> # ... with 1,698 more rows, and 1 more variable: resid <dbl>
```

Note that each regular column is repeated once for each row in the nested column.

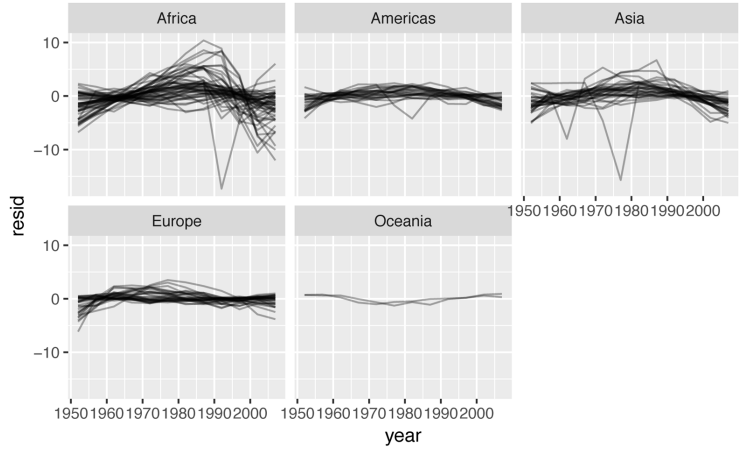
Now that we have regular data frame, we can plot the residuals:

```
resids %>%
  ggplot(aes(year, resid)) +
    geom_line(aes(group = country), alpha = 1 / 3) +
    geom_smooth(se = FALSE)
#> `geom_smooth()` using method = 'gam'
```



Faceting by continent is particularly revealing:

```
resids %>%
  ggplot(aes(year, resid, group = country)) +
    geom_line(alpha = 1 / 3) +
    facet_wrap(~continent)
```



It looks like we've missed some mild pattern. There's also something interesting going on in Africa: we see some very large residuals, which suggests our model isn't fitting so well there. We'll explore that more in the next section, attacking it from a slightly different angle.

Model Quality

Instead of looking at the residuals from the model, we could look at some general measurements of model quality. You learned how to compute some specific measures in the previous chapter. Here we'll show a different approach using the **broom** package. The **broom** package provides a general set of functions to turn models into tidy data. Here we'll use `broom::glance()` to extract some model quality metrics. If we apply it to a model, we get a data frame with a single row:

```
broom::glance(nz_mod)
#>   r.squared adj.r.squared sigma statistic  p.value df logLik
#>   AIC  BIC
#> 1     0.954         0.949 0.804       205 5.41e-08  2  -13.3
#>   32.6 34.1
#>   deviance df.residual
#> 1     6.47          10
```

We can use `mutate()` and `unnest()` to create a data frame with a row for each country:

```
by_country %>%
  mutate(glance = map(model, broom::glance)) %>%
  unnest(glance)
#> # A tibble: 142 × 16
#>       country continent              data    model
#>        <fctr>    <fctr>            <list>   <list>
#> 1 Afghanistan      Asia <tibble [12 × 4]> <S3: lm>
#> 2     Albania    Europe <tibble [12 × 4]> <S3: lm>
#> 3     Algeria    Africa <tibble [12 × 4]> <S3: lm>
#> 4      Angola    Africa <tibble [12 × 4]> <S3: lm>
#> 5   Argentina  Americas <tibble [12 × 4]> <S3: lm>
#> 6   Australia   Oceania <tibble [12 × 4]> <S3: lm>
#> # ... with 136 more rows, and 12 more variables:
#> #   resids <list>, r.squared <dbl>, adj.r.squared <dbl>,
#> #   sigma <dbl>, statistic <dbl>, p.value <dbl>, df <int>,
#> #   logLik <dbl>, AIC <dbl>, BIC <dbl>, deviance <dbl>,
#> #   df.residual <int>
```

This isn't quite the output we want, because it still includes all the list-columns. This is default behavior when `unnest()` works on single-row data frames. To suppress these columns we use `.drop = TRUE`:

```
glance <- by_country %>%
  mutate(glance = map(model, broom::glance)) %>%
  unnest(glance, .drop = TRUE)
glance
```

```
#> # A tibble: 142 × 13
#>       country continent r.squared adj.r.squared sigma
#>        <fctr>    <fctr>     <dbl>         <dbl> <dbl>
#> 1 Afghanistan      Asia     0.948         0.942 1.223
#> 2     Albania    Europe     0.911         0.902 1.983
#> 3     Algeria    Africa     0.985         0.984 1.323
#> 4      Angola    Africa     0.888         0.877 1.407
#> 5   Argentina  Americas     0.996         0.995 0.292
#> 6   Australia   Oceania     0.980         0.978 0.621
#> # ... with 136 more rows, and 8 more variables:
#> #   statistic <dbl>, p.value <dbl>, df <int>, logLik <dbl>,
#> #   AIC <dbl>, BIC <dbl>, deviance <dbl>, df.residual <int>
```

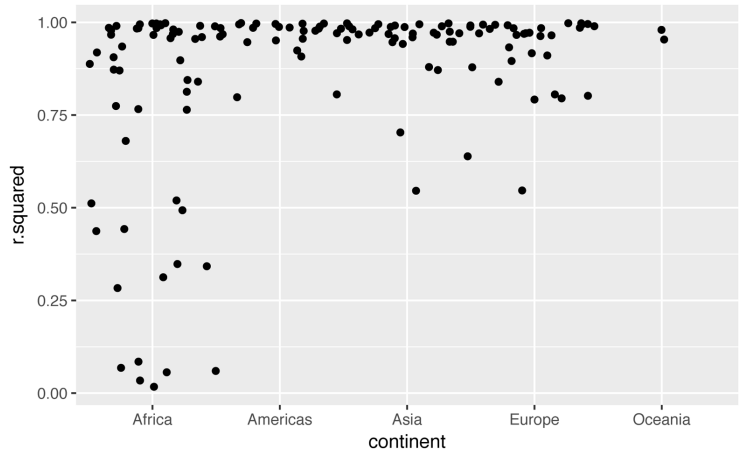
(Pay attention to the variables that aren't printed: there's a lot of useful stuff there.)

With this data frame in hand, we can start to look for models that don't fit well:

```
glance %>%
  arrange(r.squared)
#> # A tibble: 142 × 13
#>     country continent r.squared adj.r.squared sigma
#>      <fctr>    <fctr>     <dbl>         <dbl> <dbl>
#> 1    Rwanda    Africa    0.0172      -0.08112  6.56
#> 2  Botswana    Africa    0.0340      -0.06257  6.11
#> 3  Zimbabwe    Africa    0.0562      -0.03814  7.21
#> 4    Zambia    Africa    0.0598      -0.03418  4.53
#> 5 Swaziland    Africa    0.0682      -0.02497  6.64
#> 6   Lesotho    Africa    0.0849      -0.00666  5.93
#> # ... with 136 more rows, and 8 more variables:
#> #   statistic <dbl>, p.value <dbl>, df <int>, logLik <dbl>,
#> #   AIC <dbl>, BIC <dbl>, deviance <dbl>, df.residual <int>
```

The worst models all appear to be in Africa. Let's double-check that with a plot. Here we have a relatively small number of observations and a discrete variable, so `geom_jitter()` is effective:

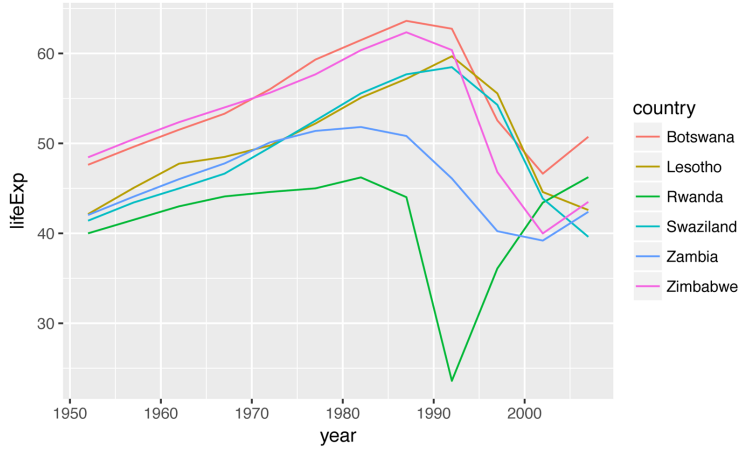
```
glance %>%
  ggplot(aes(continent, r.squared)) +
    geom_jitter(width = 0.5)
```



We could pull out the countries with particularly bad R^2 and plot the data:

```
bad_fit <- filter(glance, r.squared < 0.25)

gapminder %>%
  semi_join(bad_fit, by = "country") %>%
  ggplot(aes(year, lifeExp, color = country)) +
    geom_line()
```



We see two main effects here: the tragedies of the HIV/AIDS epidemic and the Rwandan genocide.

Exercises

1. A linear trend seems to be slightly too simple for the overall trend. Can you do better with a quadratic polynomial? How can you interpret the coefficients of the quadratic? (Hint: you might want to transform `year` so that it has mean zero.)
2. Explore other methods for visualizing the distribution of R^2 per continent. You might want to try the **ggbeeswarm** package, which provides similar methods for avoiding overlaps as jitter, but uses deterministic methods.
3. To create the last plot (showing the data for the countries with the worst model fits), we needed two steps: we created a data frame with one row per country and then semi-joined it to the original dataset. It's possible avoid this join if we use `unnest()` instead of `unnest(.drop = TRUE)`. How?

List-Columns

Now that you've seen a basic workflow for managing many models, let's dive back into some of the details. In this section, we'll explore the list-column data structure in a little more detail. It's only recently that I've really appreciated the idea of the list-column. List-columns are implicit in the definition of the data frame: a data frame is a named list of equal length vectors. A list is a vector, so it's always been legitimate to use a list as a column of a data frame. However, base R doesn't make it easy to create list-columns, and `data.frame()` treats a list as a list of columns:

```
data.frame(x = list(1:3, 3:5))
#>   x.1.3 x.3.5
#> 1     1     3
#> 2     2     4
#> 3     3     5
```

You can prevent `data.frame()` from doing this with `I()`, but the result doesn't print particularly well:

```
data.frame(
  x = I(list(1:3, 3:5)),
  y = c("1, 2", "3, 4, 5")
)
#>         x       y
```

```
#> 1 1, 2, 3    1, 2
#> 2 3, 4, 5 3, 4, 5
```

Tibble alleviates this problem by being lazier (`tibble()` doesn't modify its inputs) and by providing a better print method:

```
tibble(
  x = list(1:3, 3:5),
  y = c("1, 2", "3, 4, 5")
)
#> # A tibble: 2 × 2
#>           x       y
#>      <list>   <chr>
#> 1 <int [3]>    1, 2
#> 2 <int [3]> 3, 4, 5
```

It's even easier with `tribble()` as it can automatically work out that you need a list:

```
tribble(
   ~x, ~y,
  1:3, "1, 2",
  3:5, "3, 4, 5"
)
#> # A tibble: 2 × 2
#>           x       y
#>      <list>   <chr>
#> 1 <int [3]>    1, 2
#> 2 <int [3]> 3, 4, 5
```

List-columns are often most useful as an intermediate data structure. They're hard to work with directly, because most R functions work with atomic vectors or data frames, but the advantage of keeping related items together in a data frame is worth a little hassle.

Generally there are three parts of an effective list-column pipeline:

1. You create the list-column using one of `nest()`, `summarize()` + `list()`, or `mutate()` + a map function, as described in "Creating List-Columns" on page 411.
2. You create other intermediate list-columns by transforming existing list columns with `map()`, `map2()`, or `pmap()`. For example, in the previous case study, we created a list-column of models by transforming a list-column of data frames.
3. You simplify the list-column back down to a data frame or atomic vector, as described in "Simplifying List-Columns" on page 416.

Creating List-Columns

Typically, you won't create list-columns with `tibble()`. Instead, you'll create them from regular columns, using one of three methods:

1. With `tidyr::nest()` to convert a grouped data frame into a nested data frame where you have list-column of data frames.
2. With `mutate()` and vectorized functions that return a list.
3. With `summarize()` and summary functions that return multiple results.

Alternatively, you might create them from a named list, using `tibble::enframe()`.

Generally, when creating list-columns, you should make sure they're homogeneous: each element should contain the same type of thing. There are no checks to make sure this is true, but if you use **purrr** and remember what you've learned about type-stable functions, you should find it happens naturally.

With Nesting

`nest()` creates a nested data frame, which is a data frame with a list-column of data frames. In a nested data frame each row is a meta-observation: the other columns give variables that define the observation (like country and continent earlier), and the list-column of data frames gives the individual observations that make up the meta-observation.

There are two ways to use `nest()`. So far you've seen how to use it with a grouped data frame. When applied to a grouped data frame, `nest()` keeps the grouping columns as is, and bundles everything else into the list-column:

```
gapminder %>%
  group_by(country, continent) %>%
  nest()
#> # A tibble: 142 × 3
#>       country continent              data
#>        <fctr>    <fctr>            <list>
#> 1 Afghanistan      Asia <tibble [12 × 4]>
#> 2     Albania    Europe <tibble [12 × 4]>
#> 3     Algeria    Africa <tibble [12 × 4]>
```

```
#> 4      Angola    Africa <tibble [12 × 4]>
#> 5   Argentina  Americas <tibble [12 × 4]>
#> 6   Australia   Oceania <tibble [12 × 4]>
#> # ... with 136 more rows
```

You can also use it on an ungrouped data frame, specifying which columns you want to nest:

```
gapminder %>%
  nest(year:gdpPercap)
#> # A tibble: 142 × 3
#>       country continent              data
#>        <fctr>    <fctr>            <list>
#> 1 Afghanistan      Asia <tibble [12 × 4]>
#> 2     Albania    Europe <tibble [12 × 4]>
#> 3     Algeria    Africa <tibble [12 × 4]>
#> 4      Angola    Africa <tibble [12 × 4]>
#> 5   Argentina  Americas <tibble [12 × 4]>
#> 6   Australia   Oceania <tibble [12 × 4]>
#> # ... with 136 more rows
```

From Vectorized Functions

Some useful functions take an atomic vector and return a list. For example, in Chapter 11 you learned about `stringr::str_split()`, which takes a character vector and returns a list of character vectors. If you use that inside mutate, you'll get a list-column:

```
df <- tribble(
  ~x1,
  "a,b,c",
  "d,e,f,g"
)

df %>%
  mutate(x2 = stringr::str_split(x1, ","))
#> # A tibble: 2 × 2
#>       x1        x2
#>    <chr>    <list>
#> 1   a,b,c <chr [3]>
#> 2 d,e,f,g <chr [4]>
```

`unnest()` knows how to handle these lists of vectors:

```
df %>%
  mutate(x2 = stringr::str_split(x1, ",")) %>%
  unnest()
#> # A tibble: 7 × 2
#>       x1    x2
#>    <chr> <chr>
#> 1   a,b,c     a
```

```
#> 2   a,b,c     b
#> 3   a,b,c     c
#> 4 d,e,f,g     d
#> 5 d,e,f,g     e
#> 6 d,e,f,g     f
#> # ... with 1 more rows
```

(If you find yourself using this pattern a lot, make sure to check out `tidyr:separate_rows()`, which is a wrapper around this common pattern).

Another example of this pattern is using the `map()`, `map2()`, `pmap()` functions from **purrr**. For example, we could take the final example from "Invoking Different Functions" on page 334 and rewrite it to use `mutate()`:

```
sim <- tribble(
  ~f,      ~params,
  "runif", list(min = -1, max = -1),
  "rnorm", list(sd = 5),
  "rpois", list(lambda = 10)
)

sim %>%
  mutate(sims = invoke_map(f, params, n = 10))
#> # A tibble: 3 × 3
#>       f     params       sims
#>   <chr>     <list>     <list>
#> 1 runif <list [2]> <dbl [10]>
#> 2 rnorm <list [1]> <dbl [10]>
#> 3 rpois <list [1]> <int [10]>
```

Note that technically `sim` isn't homogeneous because it contains both double and integer vectors. However, this is unlikely to cause many problems since integers and doubles are both numeric vectors.

From Multivalued Summaries

One restriction of `summarize()` is that it only works with summary functions that return a single value. That means that you can't use it with functions like `quantile()` that return a vector of arbitrary length:

```
mtcars %>%
  group_by(cyl) %>%
  summarize(q = quantile(mpg))
#> Error in eval(expr, envir, enclos): expecting a single value
```

You can however, wrap the result in a list! This obeys the contract of `summarize()`, because each summary is now a list (a vector) of length 1:

```
mtcars %>%
  group_by(cyl) %>%
  summarize(q = list(quantile(mpg)))
#> # A tibble: 3 × 2
#>     cyl         q
#>   <dbl>    <list>
#> 1     4 <dbl [5]>
#> 2     6 <dbl [5]>
#> 3     8 <dbl [5]>
```

To make useful results with `unnest()`, you'll also need to capture the probabilities:

```
probs <- c(0.01, 0.25, 0.5, 0.75, 0.99)
mtcars %>%
  group_by(cyl) %>%
  summarize(p = list(probs), q = list(quantile(mpg, probs))) %>%
  unnest()
#> # A tibble: 15 × 3
#>     cyl     p     q
#>   <dbl> <dbl> <dbl>
#> 1     4  0.01  21.4
#> 2     4  0.25  22.8
#> 3     4  0.50  26.0
#> 4     4  0.75  30.4
#> 5     4  0.99  33.8
#> 6     6  0.01  17.8
#> # ... with 9 more rows
```

From a Named List

Data frames with list-columns provide a solution to a common problem: what do you do if you want to iterate over both the contents of a list and its elements? Instead of trying to jam everything into one object, it's often easier to make a data frame: one column can contain the elements, and one column can contain the list. An easy way to create such a data frame from a list is `tibble::enframe()`:

```
x <- list(
  a = 1:5,
  b = 3:4,
  c = 5:6
)
```

```
df <- enframe(x)
df
#> # A tibble: 3 × 2
#>    name     value
#>   <chr>    <list>
#> 1     a <int [5]>
#> 2     b <int [2]>
#> 3     c <int [2]>
```

The advantage of this structure is that it generalizes in a straightforward way—names are useful if you have a character vector of metadata, but don't help if you have other types of data, or multiple vectors.

Now if you want to iterate over names and values in parallel, you can use `map2()`:

```
df %>%
  mutate(
    smry = map2_chr(
      name,
      value,
      ~ stringr::str_c(.x, ": ", .y[1])
    )
  )
#> # A tibble: 3 × 3
#>    name     value  smry
#>   <chr>    <list> <chr>
#> 1     a <int [5]>  a: 1
#> 2     b <int [2]>  b: 3
#> 3     c <int [2]>  c: 5
```

Exercises

1. List all the functions that you can think of that take an atomic vector and return a list.
2. Brainstorm useful summary functions that, like `quantile()`, return multiple values.
3. What's missing in the following data frame? How does `quantile()` return that missing piece? Why isn't that helpful here?

```
mtcars %>%
  group_by(cyl) %>%
  summarize(q = list(quantile(mpg))) %>%
  unnest()
#> # A tibble: 15 × 2
#>     cyl     q
#>   <dbl> <dbl>
```

```
#> 1     4  21.4
#> 2     4  22.8
#> 3     4  26.0
#> 4     4  30.4
#> 5     4  33.9
#> 6     6  17.8
#> # ... with 9 more rows
```

4. What does this code do? Why might might it be useful?

```
mtcars %>%
  group_by(cyl) %>%
  summarize_each(funs(list))
```

Simplifying List-Columns

To apply the techniques of data manipulation and visualization you've learned in this book, you'll need to simplify the list-column back to a regular column (an atomic vector), or set of columns. The technique you'll use to collapse back down to a simpler structure depends on whether you want a single value per element, or multiple values:

- If you want a single value, use `mutate()` with `map_lgl()`, `map_int()`, `map_dbl()`, and `map_chr()` to create an atomic vector.
- If you want many values, use `unnest()` to convert list-columns back to regular columns, repeating the rows as many times as necessary.

These are described in more detail in the following sections.

List to Vector

If you can reduce your list column to an atomic vector then it will be a regular column. For example, you can always summarize an object with it's type and length, so this code will work regardless of what sort of list-column you have:

```
df <- tribble(
  ~x,
  letters[1:5],
  1:3,
  runif(5)
)
```

```
df %>% mutate(
  type = map_chr(x, typeof),
  length = map_int(x, length)
)
#> # A tibble: 3 × 3
#>           x      type length
#>      <list>     <chr>  <int>
#> 1 <chr [5]> character      5
#> 2 <int [3]>   integer      3
#> 3 <dbl [5]>    double      5
```

This is the same basic information that you get from the default `tbl` `print` method, but now you can use it for filtering. This is a useful technique if you have a heterogeneous list, and want to filter out the parts that aren't working for you.

Don't forget about the `map_*()` shortcuts—you can use `map_chr(x, "apple")` to extract the string stored in `apple` for each element of `x`. This is useful for pulling apart nested lists into regular columns. Use the `.null` argument to provide a value to use if the element is missing (instead of returning `NULL`):

```
df <- tribble(
  ~x,
  list(a = 1, b = 2),
  list(a = 2, c = 4)
)
df %>% mutate(
  a = map_dbl(x, "a"),
  b = map_dbl(x, "b", .null = NA_real_)
)
#> # A tibble: 2 × 3
#>           x     a     b
#>      <list> <dbl> <dbl>
#> 1 <list [2]>     1     2
#> 2 <list [2]>     2    NA
```

Unnesting

`unnest()` works by repeating the regular columns once for each element of the list-column. For example, in the following very simple example we repeat the first row four times (because there the first element of `y` has length four), and the second row once:

```
tibble(x = 1:2, y = list(1:4, 1)) %>% unnest(y)
#> # A tibble: 5 × 2
#>       x     y
#>   <int> <dbl>
```

```
#> 1     1     1
#> 2     1     2
#> 3     1     3
#> 4     1     4
#> 5     2     1
```

This means that you can't simultaneously unnest two columns that contain a different number of elements:

```
# Ok, because y and z have the same number of elements in
# every row
df1 <- tribble(
  ~x, ~y,           ~z,
   1, c("a", "b"), 1:2,
   2, "c",           3
)
df1
#> # A tibble: 2 × 3
#>       x         y         z
#>   <dbl>    <list>    <list>
#> 1     1 <chr [2]> <int [2]>
#> 2     2 <chr [1]> <dbl [1]>
df1 %>% unnest(y, z)
#> # A tibble: 3 × 3
#>       x     y     z
#>   <dbl> <chr> <dbl>
#> 1     1     a     1
#> 2     1     b     2
#> 3     2     c     3

# Doesn't work because y and z have different number of elements
df2 <- tribble(
  ~x, ~y,           ~z,
   1, "a",         1:2,
   2, c("b", "c"),   3
)
df2
#> # A tibble: 2 × 3
#>       x         y         z
#>   <dbl>    <list>    <list>
#> 1     1 <chr [1]> <int [2]>
#> 2     2 <chr [2]> <dbl [1]>
df2 %>% unnest(y, z)
#> Error: All nested columns must have
#> the same number of elements.
```

The same principle applies when unnesting list-columns of data frames. You can unnest multiple list-columns as long as all the data frames in each row have the same number of rows.

Exercises

1. Why might the `lengths()` function be useful for creating atomic vector columns from list-columns?
2. List the most common types of vector found in a data frame. What makes lists different?

Making Tidy Data with broom

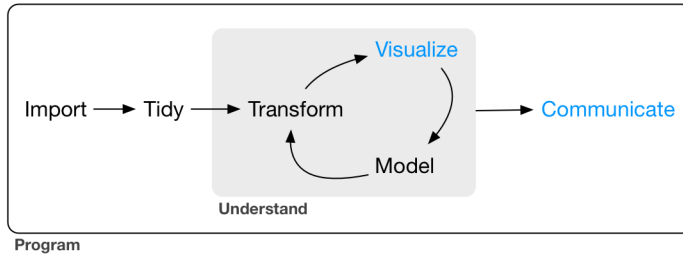
The **broom** package provides three general tools for turning models into tidy data frames:

- `broom::glance(model)` returns a row for each model. Each column gives a model summary: either a measure of model quality, or complexity, or a combination of the two.
- `broom::tidy(model)` returns a row for each coefficient in the model. Each column gives information about the estimate or its variability.
- `broom::augment(model, data)` returns a row for each row in `data`, adding extra values like residuals, and influence statistics.

Broom works with a wide variety of models produced by the most popular modelling packages. See *https://github.com/tidyverse/broom* for a list of currently supported models.

PART V
Communicate

So far, you've learned the tools to get your data into R, tidy it into a form convenient for analysis, and then understand your data through transformation, visualization, and modeling. However, it doesn't matter how great your analysis is unless you can explain it to others: you need to *communicate* your results.



Communication is the theme of the following four chapters:

- In Chapter 21, you will learn about R Markdown, a tool for integrating prose, code, and results. You can use R Markdown in notebook mode for analyst-to-analyst communication, and in report mode for analyst-to-decision-maker communication. Thanks to the power of R Markdown formats, you can even use the same document for both purposes.

- In Chapter 22, you will learn how to take your exploratory graphics and turn them into expository graphics, graphics that help the newcomer to your analysis understand what's going on as quickly and easily as possible.
- In Chapter 23, you'll learn a little about the many other varieties of outputs you can produce using R Markdown, including dashboards, websites, and books.
- We'll finish up with Chapter 24, where you'll learn about the "analysis notebook" and how to systematically record your successes and failures so that you can learn from them.

Unfortunately, these chapters focus mostly on the technical mechanics of communication, not the really hard problems of communicating your thoughts to other humans. However, there are lot of other great books about communication, which we'll point you to at the end of each chapter.

CHAPTER 21

R Markdown

Introduction

R Markdown provides a unified authoring framework for data science, combining your code, its results, and your prose commentary. R Markdown documents are fully reproducible and support dozens of output formats, like PDFs, Word files, slideshows, and more.

R Markdown files are designed to be used in three ways:

- For communicating to decision makers, who want to focus on the conclusions, not the code behind the analysis.
- For collaborating with other data scientists (including future you!), who are interested in both your conclusions, and how you reached them (i.e., the code).
- As an environment in which to *do* data science, as a modern day lab notebook where you can capture not only what you did, but also what you were thinking.

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

- R Markdown Cheat Sheet: available in the RStudio IDE under *Help → Cheatsheets → R Markdown Cheat Sheet*

- R Markdown Reference Guide: available in the RStudio IDE under *Help → Cheatsheets → R Markdown Reference Guide*

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

Prerequisites

You need the **rmarkdown** package, but you don't need to explicitly install it or load it, as RStudio automatically does both when needed.

R Markdown Basics

This is an R Markdown file, a plain-text file that has the extension *.Rmd*:

````
---
title: "Diamond sizes"
date: 2016-08-25
output: html_document
---

```{r setup, include = FALSE}
library(ggplot2)
library(dplyr)

smaller <- diamonds %>%
  filter(carat <= 2.5)
```

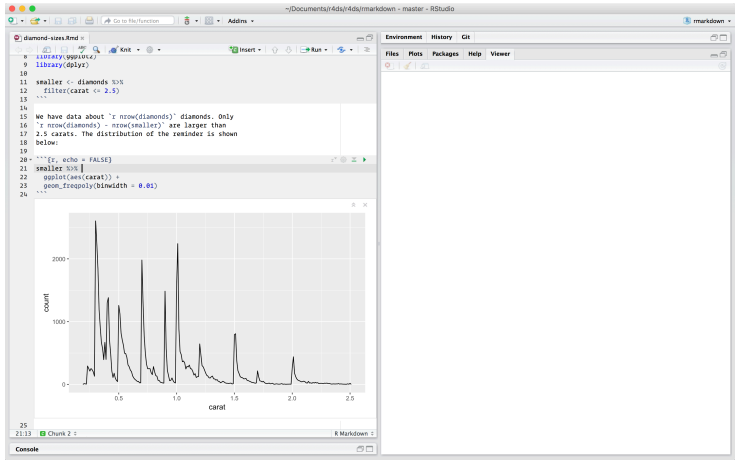
We have data about `r nrow(diamonds)` diamonds. Only
`r nrow(diamonds) - nrow(smaller)` are larger than
2.5 carats. The distribution of the remainder is shown
below:

```{r, echo = FALSE}
smaller %>%
  ggplot(aes(carat)) +
  geom_freqpoly(binwidth = 0.01)
```
````

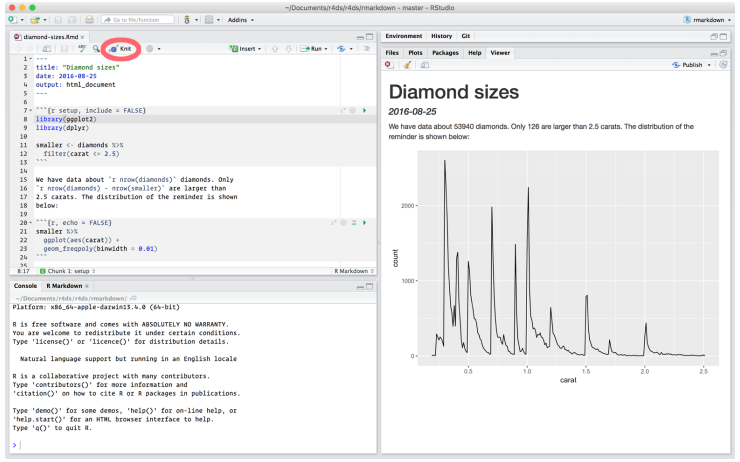
It contains three important types of content:

1. An (optional) *YAML header* surrounded by `---`s.
2. *Chunks* of R code surrounded by ```.
3. Text mixed with simple text formatting like `# heading` and `_italics_`.

When you open an *.Rmd*, you get a notebook interface where code and output are interleaved. You can run each code chunk by clicking the Run icon (it looks like a play button at the top of the chunk), or by pressing Cmd/Ctrl-Shift-Enter. RStudio executes the code and displays the results inline with the code:



To produce a complete report containing all text, code, and results, click "Knit" or press Cmd/Ctrl-Shift-K. You can also do this programmatically with `rmarkdown::render("1-example.Rmd")`. This will display the report in the viewer pane, and create a self-contained HTML file that you can share with others.



When you *knit* the document R Markdown sends the *.Rmd* file to **knitr**, which executes all of the code chunks and creates a new Markdown (*.md*) document that includes the code and its output. The Markdown file generated by **knitr** is then processed by **pandoc**, which is responsible for creating the finished file. The advantage of this two-step workflow is that you can create a very wide range of output formats, as you'll learn about in Chapter 23.

To get started with your own *.Rmd* file, select *File → New File → R Markdown…* in the menu bar. RStudio will launch a wizard that you can use to pre-populate your file with useful content that reminds you how the key features of R Markdown work.

The following sections dive into the three components of an R Markdown document in more detail: the Markdown text, the code chunks, and the YAML header.

Exercises

1. Create a new notebook using *File → New File → R Notebook*. Read the instructions. Practice running the chunks. Verify that you can modify the code, rerun it, and see modified output.
2. Create a new R Markdown document with *File → New File → R Markdown…* Knit it by clicking the appropriate button. Knit it by using the appropriate keyboard shortcut. Verify that you can modify the input and see the output update.
3. Compare and contrast the R Notebook and R Markdown files you created earlier. How are the outputs similar? How are they different? How are the inputs similar? How are they different? What happens if you copy the YAML header from one to the other?
4. Create one new R Markdown document for each of the three built-in formats: HTML, PDF, and Word. Knit each of the three documents. How does the output differ? How does the input

differ? (You may need to install LaTeX in order to build the PDF output—RStudio will prompt you if this is necessary.)

Text Formatting with Markdown

Prose in *.Rmd* files is written in Markdown, a lightweight set of conventions for formatting plain-text files. Markdown is designed to be easy to read and easy to write. It is also very easy to learn. The following guide shows how to use Pandoc's Markdown, a slightly extended version of Markdown that R Markdown understands:

```
Text formatting
------------------------------------------------------------

*italic*  or _italic_
**bold**   __bold__
`code`
superscript^2^ and subscript~2~

Headings
------------------------------------------------------------

# 1st Level Header

## 2nd Level Header

### 3rd Level Header

Lists
------------------------------------------------------------

*   Bulleted list item 1

*   Item 2

    * Item 2a

    * Item 2b

1.  Numbered list item 1

1.  Item 2. The numbers are incremented automatically in
the output.

Links and images
------------------------------------------------------------

<http://example.com>
```

```
[linked phrase](http://example.com)

![optional caption text](path/to/img.png)

Tables
------------------------------------------------------------

First Header  | Second Header
------------- | -------------
Content Cell  | Content Cell
Content Cell  | Content Cell
```

The best way to learn these is simply to try them out. It will take a few days, but soon they will become second nature, and you won't need to think about them. If you forget, you can get to a handy reference sheet with *Help → Markdown Quick Reference*.

Exercises

1. Practice what you've learned by creating a brief CV. The title should be your name, and you should include headings for (at least) education or employment. Each of the sections should include a bulleted list of jobs/degrees. Highlight the year in bold.
2. Using the R Markdown quick reference, figure out how to:
   a. Add a footnote.
   b. Add a horizontal rule.
   c. Add a block quote.
3. Copy and paste the contents of *diamond-sizes.Rmd* from *https://github.com/hadley/r4ds/tree/master/rmarkdown* into a local R Markdown document. Check that you can run it, then add text after the frequency polygon that describes its most striking features.

Code Chunks

To run code inside an R Markdown document, you need to insert a chunk. There are three ways to do so:

1. The keyboard shortcut Cmd/Ctrl-Alt-I
2. The "Insert" button icon in the editor toolbar
3. By manually typing the chunk delimiters ```` ```{r} ```` and ```` ``` ````

Obviously, I'd recommend you learn the keyboard shortcut. It will save you a lot of time in the long run!

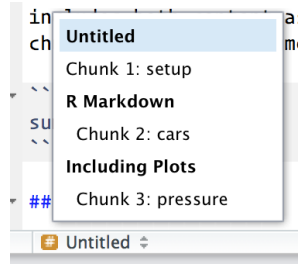
You can continue to run the code using the keyboard shortcut that by now (I hope!) you know and love: Cmd/Ctrl-Enter. However, chunks get a new keyboard shortcut: Cmd/Ctrl-Shift-Enter, which runs all the code in the chunk. Think of a chunk like a function. A chunk should be relatively self-contained, and focused around a single task.

The following sections describe the chunk header, which consists of ```` ```{r ```, followed by an optional chunk name, followed by comma-separated options, followed by `}`. Next comes your R code and the chunk end is indicated by a final ```` ``` ````.

Chunk Name

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

- You can more easily navigate to specific chunks using the drop-down code navigator in the bottom-left of the script editor:



- Graphics produced by the chunks will have useful names that make them easier to use elsewhere. More on that in "Other Important Options" on page 467.

- You can set up networks of cached chunks to avoid re-performing expensive computations on every run. More on that in a bit.

There is one chunk name that imbues special behavior: `setup`. When you're in a notebook mode, the chunk named `setup` will be run automatically once, before any other code is run.

Chunk Options

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

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

- `eval = FALSE` prevents code from being evaluated. (And obviously if the code is not run, no results will be generated.) This is useful for displaying example code, or for disabling a large block of code without commenting each line.
- `include = FALSE` runs the code, but doesn't show the code or results in the final document. Use this for setup code that you don't want cluttering your report.
- `echo = FALSE` prevents code, but not the results from appearing in the finished file. Use this when writing reports aimed at people who don't want to see the underlying R code.
- `message = FALSE` or `warning = FALSE` prevents messages or warnings from appearing in the finished file.
- `results = 'hide'` hides printed output; `fig.show = 'hide'` hides plots.
- `error = TRUE` causes the render to continue even if code returns an error. This is rarely something you'll want to include in the final version of your report, but can be very useful if you need to debug exactly what is going on inside your *.Rmd*. It's also useful if you're teaching R and want to deliberately include an error. The default, `error = FALSE`, causes knitting to fail if there is a single error in the document.

The following table summarizes which types of output each option suppressess:

Option	Run code	Show code	Output	Plots	Messages	Warnings
`eval = FALSE`	x		x	x	x	x
`include = FALSE`		x	x	x	x	x
`echo = FALSE`		x				
`results = "hide"`			x			
`fig.show = "hide"`				x		
`message = FALSE`					x	
`warning = FALSE`						x

Table

By default, R Markdown prints data frames and matrices as you'd see them in the console:

```
mtcars[1:5, 1:10]
#>                   mpg cyl disp  hp drat   wt qsec vs am gear
#> Mazda RX4         21.0   6  160 110 3.90 2.62 16.5  0  1    4
#> Mazda RX4 Wag     21.0   6  160 110 3.90 2.88 17.0  0  1    4
#> Datsun 710        22.8   4  108  93 3.85 2.32 18.6  1  1    4
#> Hornet 4 Drive    21.4   6  258 110 3.08 3.21 19.4  1  0    3
#> Hornet Sportabout 18.7   8  360 175 3.15 3.44 17.0  0  0    3
```

If you prefer that data be displayed with additional formatting you can use the `knitr::kable` function. The following code generates Table 21-1:

```
knitr::kable(
  mtcars[1:5, ],
  caption = "A knitr kable."
)
```

Table 21-1. A knitr kable

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160	110	3.90	2.62	16.5	0	1	4	4
Mazda RX4 Wag	21.0	6	160	110	3.90	2.88	17.0	0	1	4	4
Datsun 710	22.8	4	108	93	3.85	2.32	18.6	1	1	4	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.21	19.4	1	0	3	1
Hornet Sportabout	18.7	8	360	175	3.15	3.44	17.0	0	0	3	2

Read the documentation for `?knitr::kable` to see the other ways in which you can customize the table. For even deeper customization, consider the **xtable**, **stargazer**, **pander**, **tables**, and **ascii** packages. Each provides a set of tools for returning formatted tables from R code.

There is also a rich set of options for controlling how figures are embedded. You'll learn about these in "Saving Your Plots" on page 464.

Caching

Normally, each knit of a document starts from a completely clean slate. This is great for reproducibility, because it ensures that you've captured every important computation in code. However, it can be painful if you have some computations that take a long time. The solution is `cache = TRUE`. When set, this will save the output of the chunk to a specially named file on disk. On subsequent runs, **knitr** will check to see if the code has changed, and if it hasn't, it will reuse the cached results.

The caching system must be used with care, because by default it is based on the code only, not its dependencies. For example, here the `processed_data` chunk depends on the `raw_data` chunk:

````
```{r raw_data}
rawdata <- readr::read_csv("a_very_large_file.csv")
```

```{r processed_data, cached = TRUE}
processed_data <- rawdata %>%
  filter(!is.na(import_var)) %>%
  mutate(new_variable = complicated_transformation(x, y, z))
```
````

Caching the `processed_data` chunk means that it will get rerun if the **dplyr** pipeline is changed, but it won't get rerun if the `read_csv()` call changes. You can avoid that problem with the `dependson` chunk option:

````
```{r processed_data, cached = TRUE, dependson = "raw_data"}
processed_data <- rawdata %>%
  filter(!is.na(import_var)) %>%
  mutate(new_variable = complicated_transformation(x, y, z))
```
````

`dependson` should contain a character vector of *every* chunk that the cached chunk depends on. **knitr** will update the results for the cached chunk whenever it detects that one of its dependencies has changed.

Note that the chunks won't update if *a_very_large_file.csv* changes, because **knitr** caching only tracks changes within the *.Rmd* file. If you want to also track changes to that file you can use the `cache.extra` option. This is an arbitrary R expression that will invalidate the cache whenever it changes. A good function to use is `file.info()`: it returns a bunch of information about the file including when it was last modified. Then you can write:

````
```{r raw_data, cache.extra = file.info("a_very_large_file.csv")}
rawdata <- readr::read_csv("a_very_large_file.csv")
```
````

As your caching strategies get progressively more complicated, it's a good idea to regularly clear out all your caches with `knitr::clean_cache()`.

I've used the advice of David Robinson to name these chunks: each chunk is named after the primary object that it creates. This makes it easier to understand the `dependson` specification.

Global Options

As you work more with **knitr**, you will discover that some of the default chunk options don't fit your needs, and want to change them. You can do that by calling `knitr::opts_chunk$set()` in a code chunk. For example, when writing books and tutorials I set:

```
knitr::opts_chunk$set(
  comment = "#>",
  collapse = TRUE
)
```

This uses my preferred comment formatting, and ensures that the code and output are kept closely entwined. On the other hand, if you were preparing a report, you might set:

```
knitr::opts_chunk$set(
  echo = FALSE
)
```

That will hide the code by default, only showing the chunks you deliberately choose to show (with `echo = TRUE`). You might con-

sider setting `message = FALSE` and `warning = FALSE`, but that would make it harder to debug problems because you wouldn't see any messages in the final document.

Inline Code

There is one other way to embed R code into an R Markdown document: directly into the text, with: `` `r ` ``. This can be very useful if you mention properties of your data in the text. For example, in the example document I used at the start of the chapter I had:

> We have data about `` `r nrow(diamonds)` `` diamonds. Only `` `r nrow(diamonds) - nrow(smaller)` `` are larger than 2.5 carats. The distribution of the remainder is shown below:

When the report is knit, the results of these computations are inserted into the text:

> We have data about 53940 diamonds. Only 126 are larger than 2.5 carats. The distribution of the remainder is shown below:

When inserting numbers into text, `format()` is your friend. It allows you to set the number of `digits` so you don't print to a ridiculous degree of accuracy, and a `big.mark` to make numbers easier to read. I'll often combine these into a helper function:

```
comma <- function(x) format(x, digits = 2, big.mark = ",")
comma(3452345)
#> [1] "3,452,345"
comma(.12358124331)
#> [1] "0.12"
```

Exercises

1. Add a section that explores how diamond sizes vary by cut, color, and clarity. Assume you're writing a report for someone who doesn't know R, and instead of setting `echo = FALSE` on each chunk, set a global option.
2. Download *diamond-sizes.Rmd* from *https://github.com/hadley/r4ds/tree/master/rmarkdown*. Add a section that describes the largest 20 diamonds, including a table that displays their most important attributes.
3. Modify *diamonds-sizes.Rmd* to use `comma()` to produce nicely formatted output. Also include the percentage of diamonds that are larger than 2.5 carats.

4. Set up a network of chunks where `d` depends on `c` and `b`, and both `b` and `c` depend on `a`. Have each chunk print `lubridate::now()`, set `cache = TRUE`, then verify your understanding of caching.

Troubleshooting

Troubleshooting R Markdown documents can be challenging because you are no longer in an interactive R environment, and you will need to learn some new tricks. The first thing you should always try is to re-create the problem in an interactive session. Restart R, then "Run all chunks" (either from the the Code menu, under the Run region, or with the keyboard shortcut Ctrl-Alt-R). If you're lucky, that will re-create the problem, and you can figure out what's going on interactively.

If that doesn't help, there must be something different between your interactive environment and the R Markdown environment. You're going to need to systematically explore the options. The most common difference is the working directory: the working directory of an R Markdown document is the directory in which it lives. Check that the working directory is what you expect by including `getwd()` in a chunk.

Next, brainstorm all of the things that might cause the bug. You'll need to systematically check that they're the same in your R session and your R Markdown session. The easiest way to do that is to set `error = TRUE` on the chunk causing the problem, then use `print()` and `str()` to check that settings are as you expect.

YAML Header

You can control many other "whole document" settings by tweaking the parameters of the YAML header. You might wonder what YAML stands for: it's "yet another markup language," which is designed for representing hierarchical data in a way that's easy for humans to read and write. R Markdown uses it to control many details of the output. Here we'll discuss two: document parameters and bibliographies.

Parameters

R Markdown documents can include one or more parameters whose values can be set when you render the report. Parameters are useful when you want to re-render the same report with distinct values for various key inputs. For example, you might be producing sales reports per branch, exam results by student, or demographic summaries by country. To declare one or more parameters, use the `params` field.

This example use a `my_class` parameter to determine which class of cars to display:

````
---
output: html_document
params:
  my_class: "suv"
---

```{r setup, include = FALSE}
library(ggplot2)
library(dplyr)

class <- mpg %>% filter(class == params$my_class)
```

# Fuel economy for `r params$my_class`s

```{r, message = FALSE}
ggplot(class, aes(displ, hwy)) +
  geom_point() +
  geom_smooth(se = FALSE)
```
````

As you can see, parameters are available within the code chunks as a read-only list named `params`.

You can write atomic vectors directly into the YAML header. You can also run arbitrary R expressions by prefacing the parameter value with `!r`. This is a good way to specify date/time parameters:

```
params:
  start: !r lubridate::ymd("2015-01-01")
  snapshot: !r lubridate::ymd_hms("2015-01-01 12:30:00")
```

In RStudio, you can click the "Knit with Parameters" option in the Knit drop-down menu to set parameters, render, and preview the report in a single user-friendly step. You can customize the dialog by

setting other options in the header. See *http://bit.ly/ParamReports* for more details.

Alternatively, if you need to produce many such parameterized reports, you can call `rmarkdown::render()` with a list of `params`:

```
rmarkdown::render(
  "fuel-economy.Rmd",
  params = list(my_class = "suv")
)
```

This is particularly powerful in conjunction with `purrr:pwalk()`. The following example creates a report for each value of `class` found in `mpg`. First we create a data frame that has one row for each class, giving the `filename` of report and the `params` it should be given:

```
reports <- tibble(
  class = unique(mpg$class),
  filename = stringr::str_c("fuel-economy-", class, ".html"),
  params = purrr::map(class, ~ list(my_class = .))
)
reports
#> # A tibble: 7 × 3
#>     class                 filename     params
#>     <chr>                    <chr>     <list>
#> 1 compact fuel-economy-compact.html <list [1]>
#> 2 midsize fuel-economy-midsize.html <list [1]>
#> 3     suv     fuel-economy-suv.html <list [1]>
#> 4 2seater fuel-economy-2seater.html <list [1]>
#> 5 minivan fuel-economy-minivan.html <list [1]>
#> 6  pickup  fuel-economy-pickup.html <list [1]>
#> # ... with 1 more rows
```

Then we match the column names to the argument names of `render()`, and use **purrr**'s *parallel* walk to call `render()` once for each row:

```
reports %>%
  select(output_file = filename, params) %>%
  purrr::pwalk(rmarkdown::render, input = "fuel-economy.Rmd")
```

Bibliographies and Citations

Pandoc can automatically generate citations and a bibliography in a number of styles. To use this feature, specify a bibliography file using the `bibliography` field in your file's header. The field should contain a path from the directory that contains your *.Rmd* file to the file that contains the bibliography file:

```
bibliography: rmarkdown.bib
```

You can use many common bibliography formats including BibLaTeX, BibTeX, endnote, and medline.

To create a citation within your *.Rmd* file, use a key composed of *"@" and the citation identifier* from the bibliography file. Then place the citation in square brackets. Here are some examples:

```
Separate multiple citations with a `;`:
Blah blah [@smith04; @doe99].

You can add arbitrary comments inside the square brackets:
Blah blah [see @doe99, pp. 33-35; also @smith04, ch. 1].

Remove the square brackets to create an in-text citation:
@smith04 says blah, or @smith04 [p. 33] says blah.

Add a `-` before the citation to suppress the author's name:
Smith says blah [-@smith04].
```

When R Markdown renders your file, it will build and append a bibliography to the end of your document. The bibliography will contain each of the cited references from your bibliography file, but it will not contain a section heading. As a result it is common practice to end your file with a section header for the bibliography, such as `# References` or `# Bibliography`.

You can change the style of your citations and bibliography by referencing a CSL (citation style language) file to the `csl` field:

```
bibliography: rmarkdown.bib
csl: apa.csl
```

As with the bibliography field, your CSL file should contain a path to the file. Here I assume that the CSL file is in the same directory as the *.Rmd* file. A good place to find CSL style files for common bibliography styles is *http://github.com/citation-style-language/styles*.

Learning More

R Markdown is still relatively young, and is still growing rapidly. The best place to stay on top of innovations is the official R Markdown website: *http://rmarkdown.rstudio.com*.

There are two important topics that we haven't covered here: collaboration, and the details of accurately communicating your ideas to other humans. Collaboration is a vital part of modern data science,

and you can make your life much easier by using version control tools, like Git and GitHub. We recommend two free resources that will teach you about Git:

- "Happy Git with R": a user-friendly introduction to Git and GitHub from R users, by Jenny Bryan. The book is freely available online.
- The "Git and GitHub" chapter of *R Packages*, by Hadley. You can also read it for free online: *http://r-pkgs.had.co.nz/git.html*.

I have also not talked about what you should actually write in order to clearly communicate the results of your analysis. To improve your writing, I highly recommend reading either *Style: Lessons in Clarity and Grace* by Joseph M. Williams and Joseph Bizup, or *The Sense of Structure: Writing from the Reader's Perspective* by George Gopen. Both books will help you understand the structure of sentences and paragraphs, and give you the tools to make your writing more clear. (These books are rather expensive if purchased new, but they're used by many English classes so there are plenty of cheap secondhand copies). George Gopen also has a number of short articles on writing. They are aimed at lawyers, but almost everything applies to data scientists too.

CHAPTER 22

Graphics for Communication with ggplot2

Introduction

In Chapter 5, you learned how to use plots as tools for *exploration*. When you make exploratory plots, you know—even before looking—which variables the plot will display. You made each plot for a purpose, could quickly look at it, and then move on to the next plot. In the course of most analyses, you'll produce tens or hundreds of plots, most of which are immediately thrown away.

Now that you understand your data, you need to *communicate* your understanding to others. Your audience will likely not share your background knowledge and will not be deeply invested in the data. To help others quickly build up a good mental model of the data, you will need to invest considerable effort in making your plots as self-explanatory as possible. In this chapter, you'll learn some of the tools that **ggplot2** provides to do so.

This chapter focuses on the tools you need to create good graphics. I assume that you know what you want, and just need to know how to do it. For that reason, I highly recommend pairing this chapter with a good general visualization book. I particularly like *The Truthful Art*, by Albert Cairo. It doesn't teach the mechanics of creating visualizations, but instead focuses on what you need to think about in order to create effective graphics.

Prerequisites

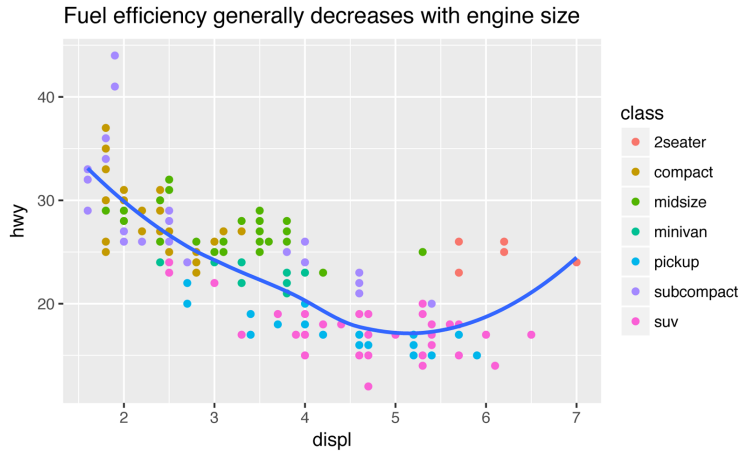
In this chapter, we'll focus once again on **ggplot2**. We'll also use a little **dplyr** for data manipulation, and a few **ggplot2** extension packages, including **ggrepel** and **viridis**. Rather than loading those extensions here, we'll refer to their functions explicitly, using the `::` notation. This will help make it clear which functions are built into **ggplot2**, and which come from other packages. Don't forget you'll need to install those packages with `install.packages()` if you don't already have them.

```
library(tidyverse)
```

Label

The easiest place to start when turning an exploratory graphic into an expository graphic is with good labels. You add labels with the `labs()` function. This example adds a plot title:

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  geom_smooth(se = FALSE) +
  labs(
   title = paste(
     "Fuel efficiency generally decreases with"
     "engine size"
  )
```

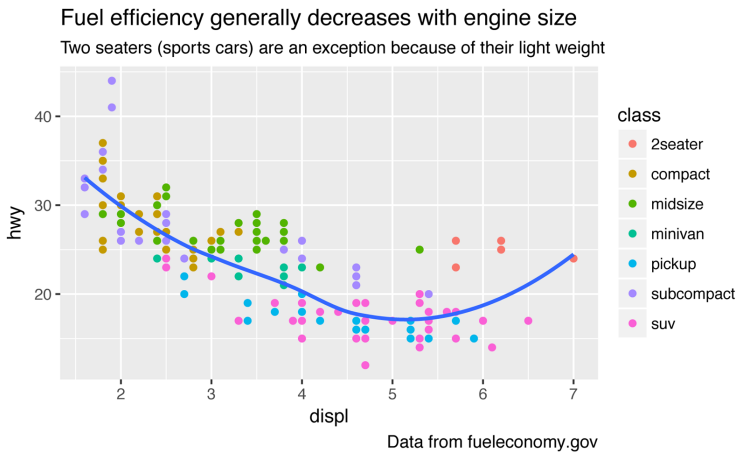


The purpose of a plot title is to summarize the main finding. Avoid titles that just describe what the plot is, e.g., "A scatterplot of engine displacement vs. fuel economy."

If you need to add more text, there are two other useful labels that you can use in **ggplot2** 2.2.0 and above (which should be available by the time you're reading this book):

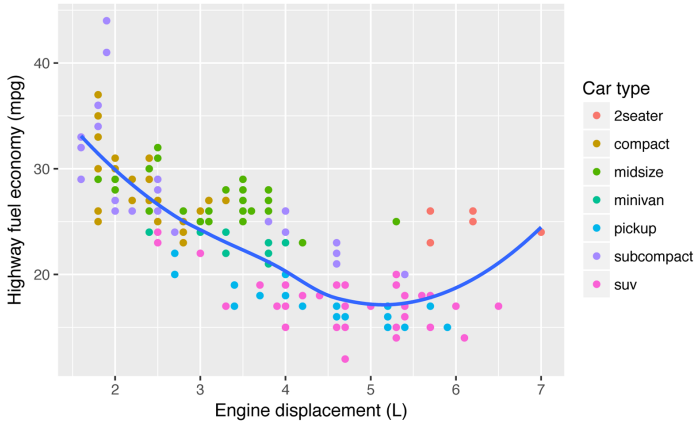
- `subtitle` adds additional detail in a smaller font beneath the title.
- `caption` adds text at the bottom right of the plot, often used to describe the source of the data:

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  geom_smooth(se = FALSE) +
  labs(
    title = paste(
      "Fuel efficiency generally decreases with"
      "engine size",
    )
    subtitle = paste(
      "Two seaters (sports cars) are an exception"
      "because of their light weight",
    )
    caption = "Data from fueleconomy.gov"
  )
```



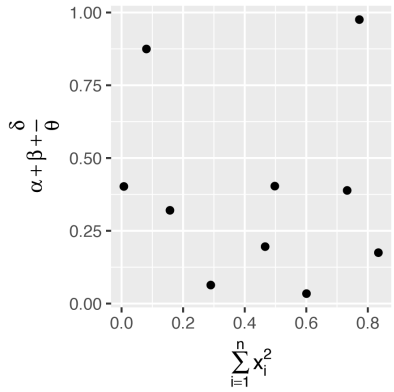
You can also use `labs()` to replace the axis and legend titles. It's usually a good idea to replace short variable names with more detailed descriptions, and to include the units:

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  geom_smooth(se = FALSE) +
  labs(
    x = "Engine displacement (L)",
    y = "Highway fuel economy (mpg)",
    colour = "Car type"
  )
```



It's possible to use mathematical equations instead of text strings. Just switch `""` out for `quote()` and read about the available options in `?plotmath`:

```
df <- tibble(
  x = runif(10),
  y = runif(10)
)
ggplot(df, aes(x, y)) +
  geom_point() +
  labs(
    x = quote(sum(x[i] ^ 2, i == 1, n)),
    y = quote(alpha + beta + frac(delta, theta))
  )
```



Exercises

1. Create one plot on the fuel economy data with customized `title`, `subtitle`, `caption`, `x`, `y`, and `colour` labels.
2. The `geom_smooth()` is somewhat misleading because the `hwy` for large engines is skewed upwards due to the inclusion of lightweight sports cars with big engines. Use your modeling tools to fit and display a better model.
3. Take an exploratory graphic that you've created in the last month, and add informative titles to make it easier for others to understand.

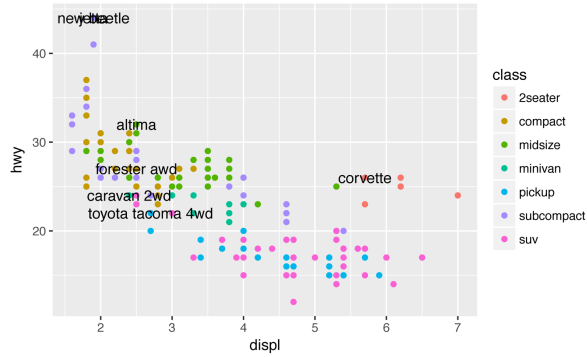
Annotations

In addition to labeling major components of your plot, it's often useful to label individual observations or groups of observations. The first tool you have at your disposal is `geom_text()`. `geom_text()` is similar to `geom_point()`, but it has an additional aesthetic: `label`. This makes it possible to add textual labels to your plots.

There are two possible sources of labels. First, you might have a tibble that provides labels. The following plot isn't terribly useful, but it illustrates a useful approach—pull out the most efficient car in each class with **dplyr**, and then label it on the plot:

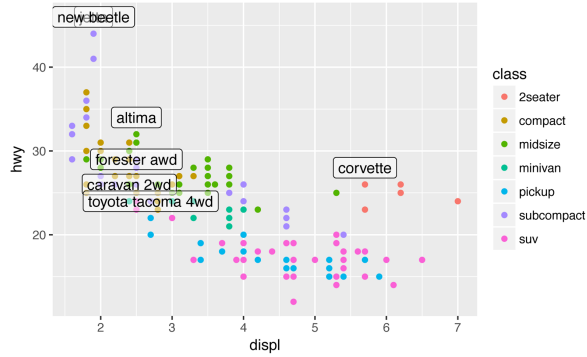
```
best_in_class <- mpg %>%
  group_by(class) %>%
  filter(row_number(desc(hwy)) == 1)
```

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  geom_text(aes(label = model), data = best_in_class)
```



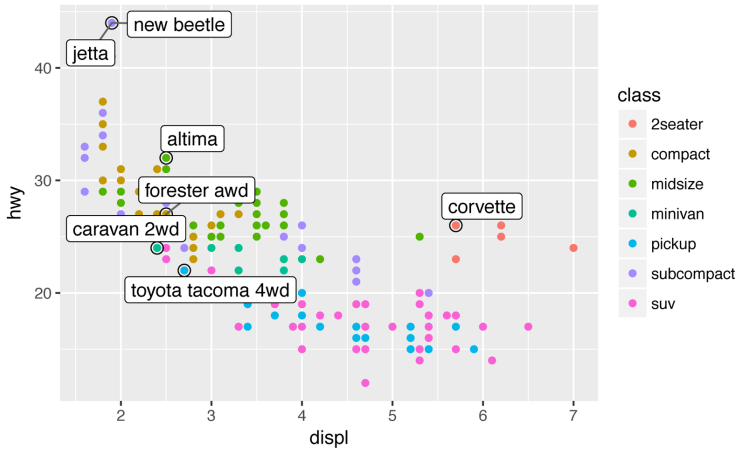
This is hard to read because the labels overlap with each other, and with the points. We can make things a little better by switching to `geom_label()`, which draws a rectangle behind the text. We also use the `nudge_y` parameter to move the labels slightly above the corresponding points:

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  geom_label(
    aes(label = model),
    data = best_in_class,
    nudge_y = 2,
    alpha = 0.5
  )
```



That helps a bit, but if you look closely in the top lefthand corner, you'll notice that there are two labels practically on top of each other. This happens because the highway mileage and displacement for the best cars in the compact and subcompact categories are exactly the same. There's no way that we can fix these by applying the same transformation for every label. Instead, we can use the **ggrepel** package by Kamil Slowikowski. This useful package will automatically adjust labels so that they don't overlap:

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  geom_point(size = 3, shape = 1, data = best_in_class) +
  ggrepel::geom_label_repel(
    aes(label = model),
    data = best_in_class
  )
```

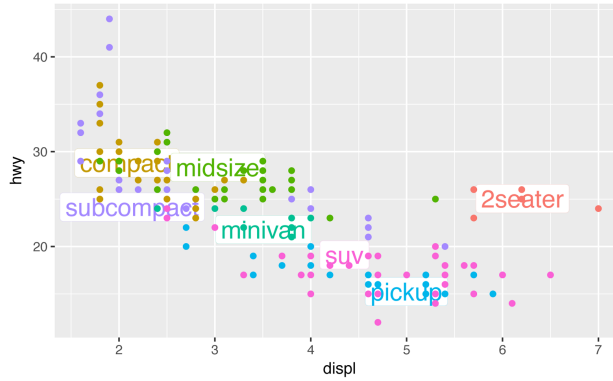


Note another handy technique used here: I added a second layer of large, hollow points to highlight the points that I've labeled.

You can sometimes use the same idea to replace the legend with labels placed directly on the plot. It's not wonderful for this plot, but it isn't too bad. (`theme(legend.position = "none")` turns the legend off—we'll talk about it more shortly.)

```
class_avg <- mpg %>%
  group_by(class) %>%
  summarize(
    displ = median(displ),
    hwy = median(hwy)
  )
```

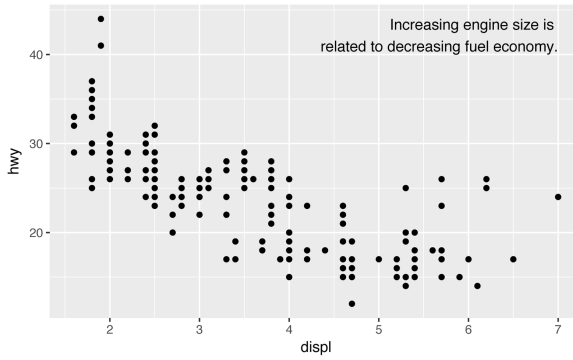
```
ggplot(mpg, aes(displ, hwy, color = class)) +
  ggrepel::geom_label_repel(aes(label = class),
    data = class_avg,
    size = 6,
    label.size = 0,
    segment.color = NA
  ) +
  geom_point() +
  theme(legend.position = "none")
```



Alternatively, you might just want to add a single label to the plot, but you'll still need to create a data frame. Often, you want the label in the corner of the plot, so it's convenient to create a new data frame using `summarize()` to compute the maximum values of `x` and `y`:

```
label <- mpg %>%
  summarize(
    displ = max(displ),
    hwy = max(hwy),
    label = paste(
      "Increasing engine size is \nrelated to"
      "decreasing fuel economy."
    )
  )

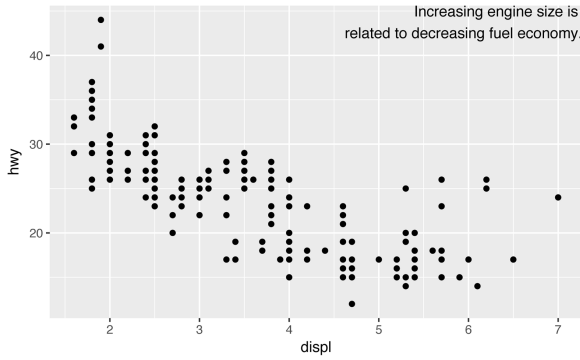
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  geom_text(
    aes(label = label),
    data = label,
    vjust = "top",
    hjust = "right"
  )
```



If you want to place the text exactly on the borders of the plot, you can use `+Inf` and `-Inf`. Since we're no longer computing the positions from `mpg`, we can use `tibble()` to create the data frame:

```
label <- tibble(
  displ = Inf,
  hwy = Inf,
  label = paste(
    "Increasing engine size is \nrelated to"
    "decreasing fuel economy."
  )
)

ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  geom_text(
    aes(label = label),
    data = label,
    vjust = "top",
    hjust = "right"
  )
```



In these examples, I manually broke the label up into lines using `"\n"`. Another approach is to use `stringr::str_wrap()` to automatically add line breaks, given the number of characters you want per line:

```
"Increasing engine size related to decreasing fuel economy." %>%
  stringr::str_wrap(width = 40) %>%
  writeLines()
#> Increasing engine size is related to
#> decreasing fuel economy.
```

Note the use of `hjust` and `vjust` to control the alignment of the label. Figure 22-1 shows all nine possible combinations.

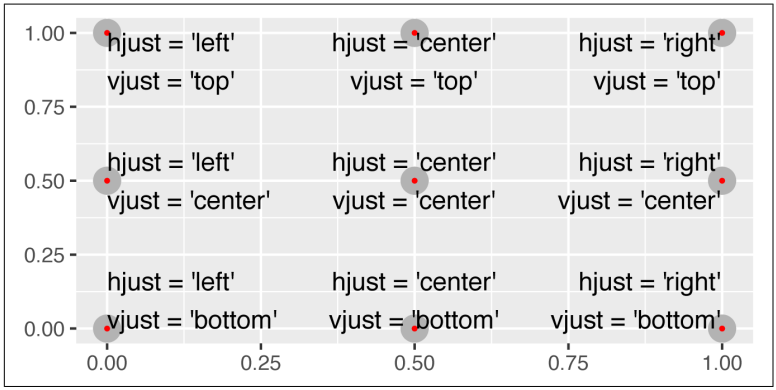


Figure 22-1. All nine combinations of hjust and vjust

Remember, in addition to `geom_text()`, you have many other geoms in **ggplot2** available to help annotate your plot. A few ideas:

- Use `geom_hline()` and `geom_vline()` to add reference lines. I often make them thick (`size = 2`) and white (`color = white`), and draw them underneath the primary data layer. That makes them easy to see, without drawing attention away from the data.
- Use `geom_rect()` to draw a rectangle around points of interest. The boundaries of the rectangle are defined by the `xmin`, `xmax`, `ymin`, and `ymax` aesthetics.
- Use `geom_segment()` with the `arrow` argument to draw attention to a point with an arrow. Use the `x` and `y` aesthetics to define the starting location, and `xend` and `yend` to define the end location.

The only limit is your imagination (and your patience with positioning annotations to be aesthetically pleasing)!

Exercises

1. Use `geom_text()` with infinite positions to place text at the four corners of the plot.
2. Read the documentation for `annotate()`. How can you use it to add a text label to a plot without having to create a tibble?
3. How do labels with `geom_text()` interact with faceting? How can you add a label to a single facet? How can you put a different label in each facet? (Hint: think about the underlying data.)
4. What arguments to `geom_label()` control the appearance of the background box?
5. What are the four arguments to `arrow()`? How do they work? Create a series of plots that demonstrate the most important options.

Scales

The third way you can make your plot better for communication is to adjust the scales. Scales control the mapping from data values to things that you can perceive. Normally, **ggplot2** automatically adds scales for you. For example, when you type:

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class))
```

ggplot2 automatically adds default scales behind the scenes:

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  scale_x_continuous() +
  scale_y_continuous() +
  scale_color_discrete()
```

Note the naming scheme for scales: `scale_` followed by the name of the aesthetic, then `_`, then the name of the scale. The default scales are named according to the type of variable they align with: continuous, discrete, datetime, or date. There are lots of nondefault scales, which you'll learn about next.

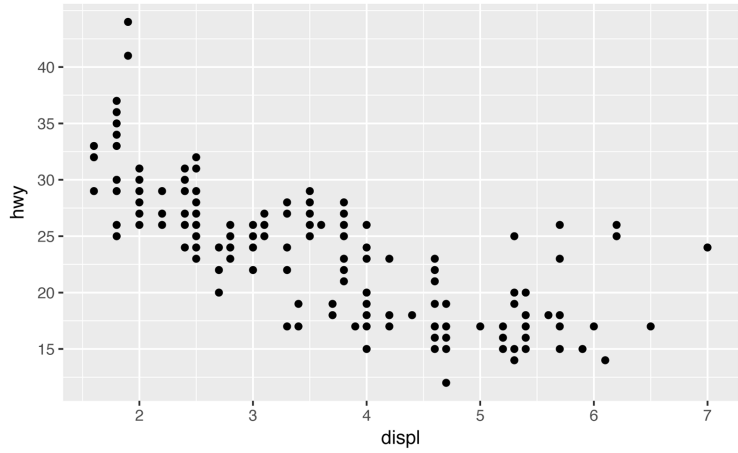
The default scales have been carefully chosen to do a good job for a wide range of inputs. Nevertheless, you might want to override the defaults for two reasons:

- You might want to tweak some of the parameters of the default scale. This allows you to do things like change the breaks on the axes, or the key labels on the legend.
- You might want to replace the scale altogether, and use a completely different algorithm. Often you can do better than the default because you know more about the data.

Axis Ticks and Legend Keys

There are two primary arguments that affect the appearance of the ticks on the axes and the keys on the legend: `breaks` and `labels`. `breaks` controls the position of the ticks, or the values associated with the keys. `labels` controls the text label associated with each tick/key. The most common use of `breaks` is to override the default choice:

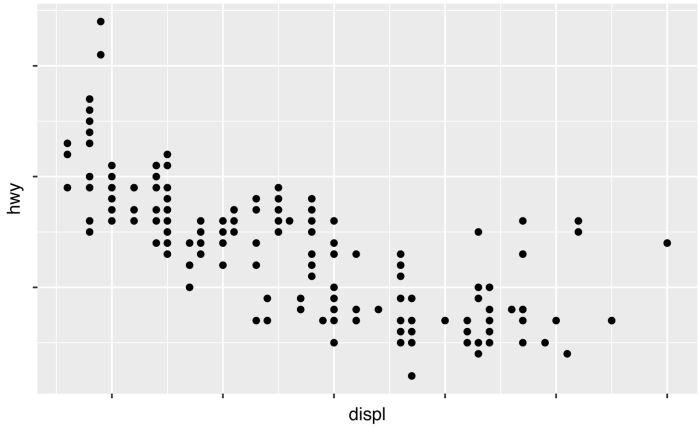
```
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  scale_y_continuous(breaks = seq(15, 40, by = 5))
```



You can use `labels` in the same way (a character vector the same length as `breaks`), but you can also set it to `NULL` to suppress the

labels altogether. This is useful for maps, or for publishing plots where you can't share the absolute numbers:

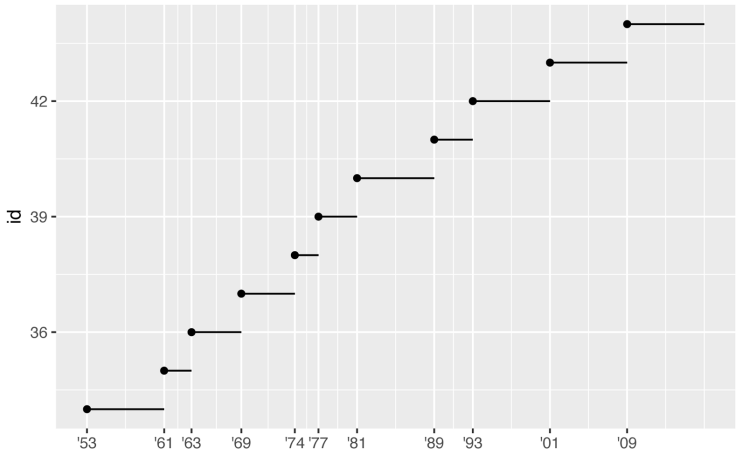
```
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  scale_x_continuous(labels = NULL) +
  scale_y_continuous(labels = NULL)
```



You can also use `breaks` and `labels` to control the appearance of legends. Collectively axes and legends are called *guides*. Axes are used for the `x` and `y` aesthetics; legends are used for everything else.

Another use of `breaks` is when you have relatively few data points and want to highlight exactly where the observations occur. For example, take this plot that shows when each US president started and ended their term:

```
presidential %>%
  mutate(id = 33 + row_number()) %>%
  ggplot(aes(start, id)) +
    geom_point() +
    geom_segment(aes(xend = end, yend = id)) +
    scale_x_date(
      NULL,
      breaks = presidential$start,
      date_labels = "'%y"
    )
```



Note that the specification of breaks and labels for date and datetime scales is a little different:

- `date_labels` takes a format specification, in the same form as `parse_datetime()`.
- `date_breaks` (not shown here) takes a string like "2 days" or "1 month".

Legend Layout

You will most often use `breaks` and `labels` to tweak the axes. While they both also work for legends, there are a few other techniques you are more likely to use.

To control the overall position of the legend, you need to use a `theme()` setting. We'll come back to themes at the end of the chapter, but in brief, they control the nondata parts of the plot. The theme setting `legend.position` controls where the legend is drawn:

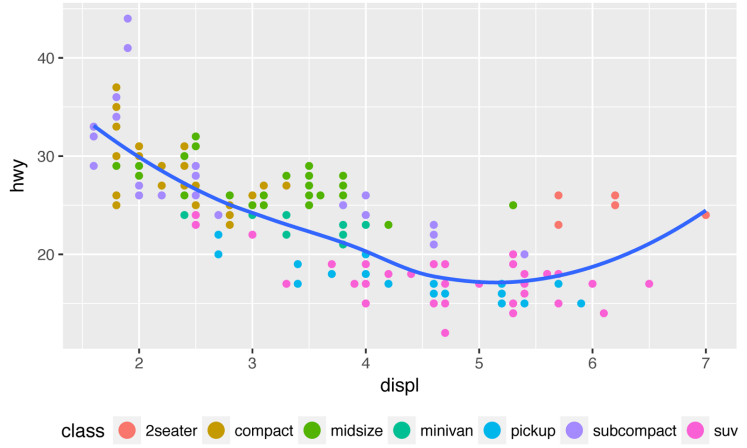
```
base <- ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class))

base + theme(legend.position = "left")
base + theme(legend.position = "top")
base + theme(legend.position = "bottom")
base + theme(legend.position = "right") # the default
```

You can also use `legend.position = "none"` to suppress the display of the legend altogether.

To control the display of individual legends, use `guides()` along with `guide_legend()` or `guide_colorbar()`. The following example shows two important settings: controlling the number of rows the legend uses with `nrow`, and overriding one of the aesthetics to make the points bigger. This is particularly useful if you have used a low `alpha` to display many points on a plot:

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  geom_smooth(se = FALSE) +
  theme(legend.position = "bottom") +
  guides(
    color = guide_legend(
      nrow = 1,
      override.aes = list(size = 4)
    )
  )
#> `geom_smooth()` using method = 'loess'
```



Replacing a Scale

Instead of just tweaking the details a little, you can replace the scale altogether. There are two types of scales you're most likely to want to switch out: continuous position scales and color scales. Fortunately, the same principles apply to all the other aesthetics, so once you've mastered position and color, you'll be able to quickly pick up other scale replacements.

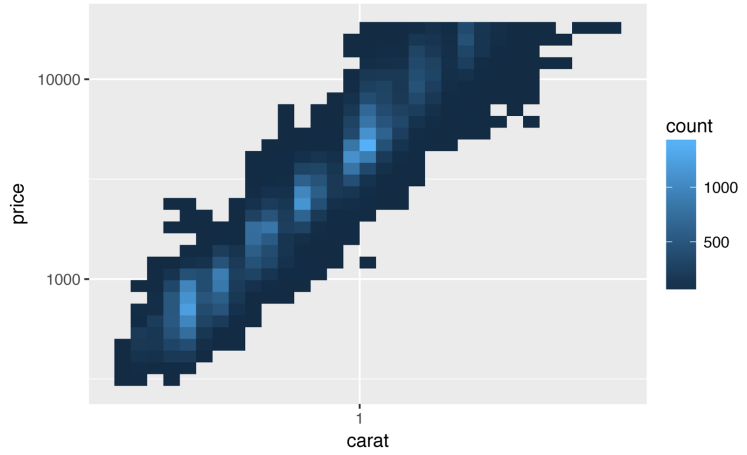
It's very useful to plot transformations of your variable. For example, as we've seen in "Why Are Low-Quality Diamonds More Expensive?" on page 376, it's easier to see the precise relationship between `carat` and `price` if we log-transform them:

```
ggplot(diamonds, aes(carat, price)) +
  geom_bin2d()

ggplot(diamonds, aes(log10(carat), log10(price))) +
  geom_bin2d()
```

However, the disadvantage of this transformation is that the axes are now labeled with the transformed values, making it hard to interpret the plot. Instead of doing the transformation in the aesthetic mapping, we can instead do it with the scale. This is visually identical, except the axes are labeled on the original data scale:

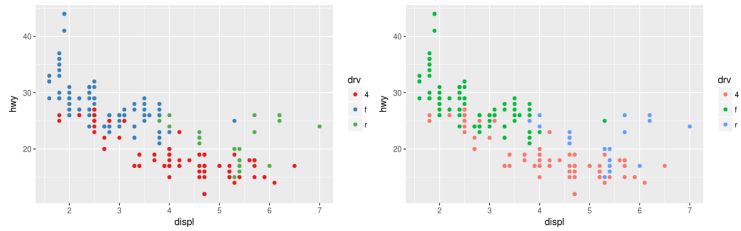
```
ggplot(diamonds, aes(carat, price)) +
  geom_bin2d() +
  scale_x_log10() +
  scale_y_log10()
```



Another scale that is frequently customized is color. The default categorical scale picks colors that are evenly spaced around the color wheel. Useful alternatives are the ColorBrewer scales, which have been hand-tuned to work better for people with common types of color blindness. The following two plots look similar, but there is enough difference in the shades of red and green that the dots on the right can be distinguished even by people with red-green color blindness:

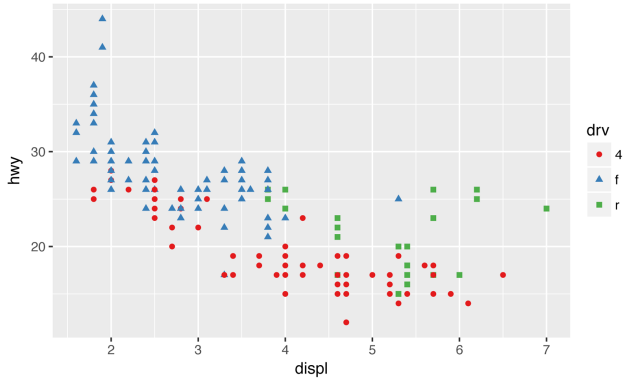
```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = drv))

ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = drv)) +
  scale_color_brewer(palette = "Set1")
```



Don't forget simpler techniques. If there are just a few colors, you can add a redundant shape mapping. This will also help ensure your plot is interpretable in black and white:

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = drv, shape = drv)) +
  scale_color_brewer(palette = "Set1")
```

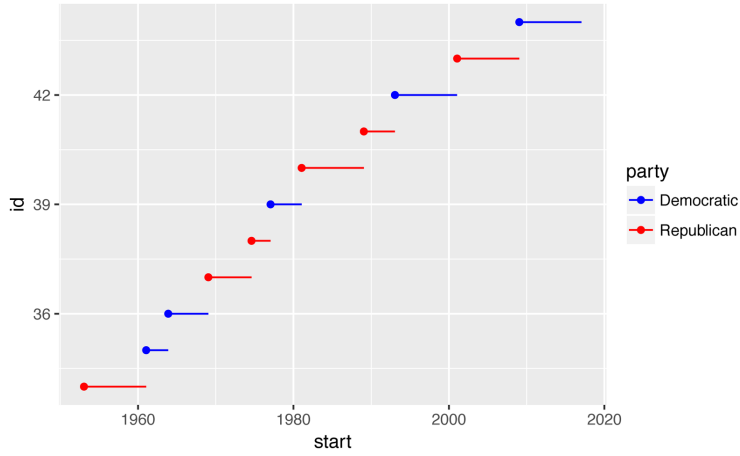


The ColorBrewer scales are documented online at *http://colorbrewer2.org/* and made available in R via the **RColorBrewer** package, by Erich Neuwirth. Figure 22-2 shows the complete list of all palettes. The sequential (top) and diverging (bottom) palettes are particularly useful if your categorical values are ordered, or have a "middle." This often arises if you've used `cut()` to make a continuous variable into a categorical variable.

Figure 22-2. All ColorBrewer scales

When you have a predefined mapping between values and colors, use `scale_color_manual()`. For example, if we map presidential party to color, we want to use the standard mapping of red for Republicans and blue for Democrats:

```
presidential %>%
  mutate(id = 33 + row_number()) %>%
  ggplot(aes(start, id, color = party)) +
    geom_point() +
    geom_segment(aes(xend = end, yend = id)) +
    scale_colour_manual(
      values = c(Republican = "red", Democratic = "blue")
    )
```



For continuous color, you can use the built-in `scale_color_gradient()` or `scale_fill_gradient()`. If you have a diverging scale, you can use `scale_color_gradient2()`. That allows you to give, for example, positive and negative values different colors. That's sometimes also useful if you want to distinguish points above or below the mean.

Another option is `scale_color_viridis()` provided by the **viridis** package. It's a continuous analog of the categorical ColorBrewer scales. The designers, Nathaniel Smith and Stéfan van der Walt, carefully tailored a continuous color scheme that has good perceptual properties. Here's an example from the **viridis** vignette:

```
df <- tibble(
  x = rnorm(10000),
  y = rnorm(10000)
```

```
)
ggplot(df, aes(x, y)) +
  geom_hex() +
  coord_fixed()
#> Loading required package: methods

ggplot(df, aes(x, y)) +
  geom_hex() +
  viridis::scale_fill_viridis() +
  coord_fixed()
```

Note that all color scales come in two varieties: `scale_color_x()` and `scale_fill_x()` for the `color` and `fill` aesthetics, respectively (the color scales are available in both UK and US spellings).

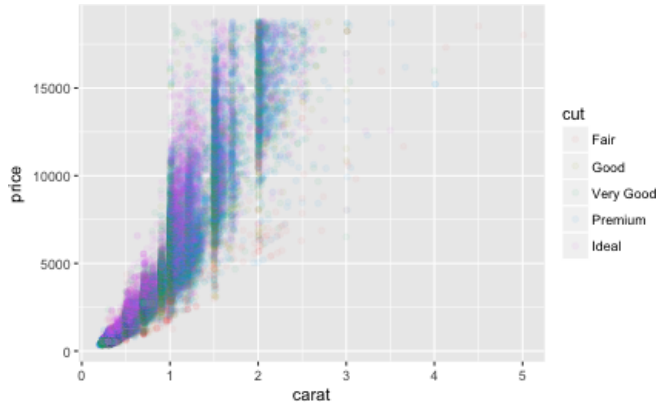
Exercises

1. Why doesn't the following code override the default scale?

   ```
   ggplot(df, aes(x, y)) +
     geom_hex() +
     scale_color_gradient(low = "white", high = "red") +
     coord_fixed()
   ```

2. What is the first argument to every scale? How does it compare to `labs()`?
3. Change the display of the presidential terms by:
   a. Combining the two variants shown above.
   b. Improving the display of the y-axis.
   c. Labeling each term with the name of the president.
   d. Adding informative plot labels.
   e. Placing breaks every four years (this is trickier than it seems!).
4. Use `override.aes` to make the legend on the following plot easier to see:

   ```
   ggplot(diamonds, aes(carat, price)) +
     geom_point(aes(color = cut), alpha = 1/20)
   ```



Zooming

There are three ways to control the plot limits:

- Adjusting what data is plotted
- Setting the limits in each scale
- Setting `xlim` and `ylim` in `coord_cartesian()`

To zoom in on a region of the plot, it's generally best to use `coord_cartesian()`. Compare the following two plots:

```
ggplot(mpg, mapping = aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  geom_smooth() +
  coord_cartesian(xlim = c(5, 7), ylim = c(10, 30))

mpg %>%
  filter(displ >= 5, displ <= 7, hwy >= 10, hwy <= 30) %>%
  ggplot(aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  geom_smooth()
```

You can also set the limits on individual scales. Reducing the limits is basically equivalent to subsetting the data. It is generally more useful if you want *expand* the limits, for example, to match scales across different plots. For example, if we extract two classes of cars and plot them separately, it's difficult to compare the plots because all three scales (the x-axis, the y-axis, and the color aesthetic) have different ranges:

```
suv <- mpg %>% filter(class == "suv")
compact <- mpg %>% filter(class == "compact")

ggplot(suv, aes(displ, hwy, color = drv)) +
  geom_point()

ggplot(compact, aes(displ, hwy, color = drv)) +
  geom_point()
```

One way to overcome this problem is to share scales across multiple plots, training the scales with the `limits` of the full data:

```
x_scale <- scale_x_continuous(limits = range(mpg$displ))
y_scale <- scale_y_continuous(limits = range(mpg$hwy))
col_scale <- scale_color_discrete(limits = unique(mpg$drv))

ggplot(suv, aes(displ, hwy, color = drv)) +
  geom_point() +
  x_scale +
  y_scale +
  col_scale

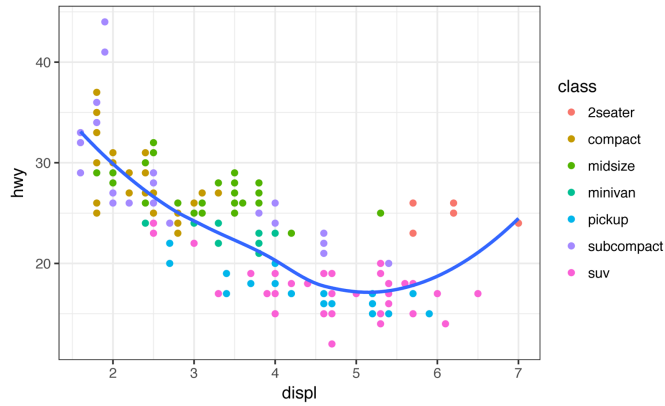
ggplot(compact, aes(displ, hwy, color = drv)) +
  geom_point() +
  x_scale +
  y_scale +
  col_scale
```

In this particular case, you could have simply used faceting, but this technique is useful more generally if, for instance, you want to spread plots over multiple pages of a report.

Themes

Finally, you can customize the nondata elements of your plot with a theme:

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  geom_smooth(se = FALSE) +
  theme_bw()
```



ggplot2 includes eight themes by default, as shown in Figure 22-3. Many more are included in add-on packages like **ggthemes**, by Jeffrey Arnold.

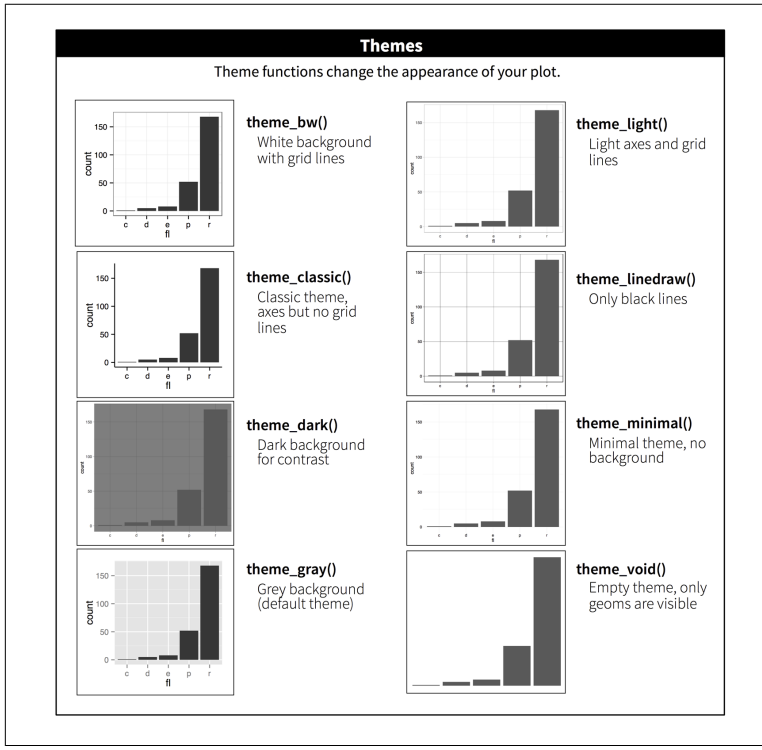


Figure 22-3. The eight themes built into ggplot2

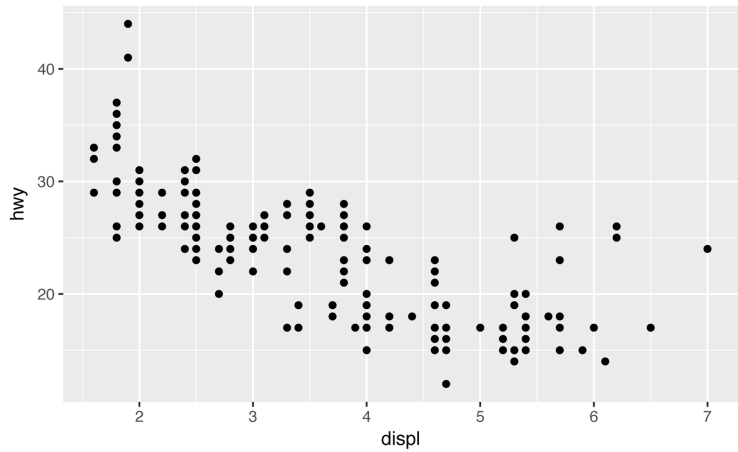
Many people wonder why the default theme has a gray background. This was a deliberate choice because it puts the data forward while still making the grid lines visible. The white grid lines are visible (which is important because they significantly aid position judgments), but they have little visual impact and we can easily tune them out. The gray background gives the plot a similar typographic color to the text, ensuring that the graphics fit in with the flow of a document without jumping out with a bright white background. Finally, the gray background creates a continuous field of color, which ensures that the plot is perceived as a single visual entity.

It's also possible to control individual components of each theme, like the size and color of the font used for the y-axis. Unfortunately, this level of detail is outside the scope of this book, so you'll need to read the **ggplot2** book for the full details. You can also create your own themes, if you are trying to match a particular corporate or journal style.

Saving Your Plots

There are two main ways to get your plots out of R and into your final write-up: `ggsave()` and **knitr**. `ggsave()` will save the most recent plot to disk:

```
ggplot(mpg, aes(displ, hwy)) + geom_point()
```




```
ggsave("my-plot.pdf")
#> Saving 6 x 3.71 in image
```

If you don't specify the `width` and `height` they will be taken from the dimensions of the current plotting device. For reproducible code, you'll want to specify them.

Generally, however, I think you should be assembling your final reports using R Markdown, so I want to focus on the important code chunk options that you should know about for graphics. You can learn more about `ggsave()` in the documentation.

Figure Sizing

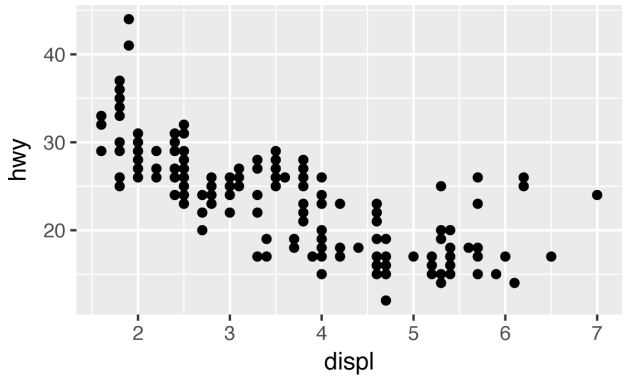
The biggest challenge of graphics in R Markdown is getting your figures the right size and shape. There are five main options that control figure sizing: `fig.width`, `fig.height`, `fig.asp`, `out.width`, and `out.height`. Image sizing is challenging because there are two sizes (the size of the figure created by R and the size at which it is inserted in the output document), and multiple ways of specifying the size (i.e., height, width, and aspect ratio: pick two of three).

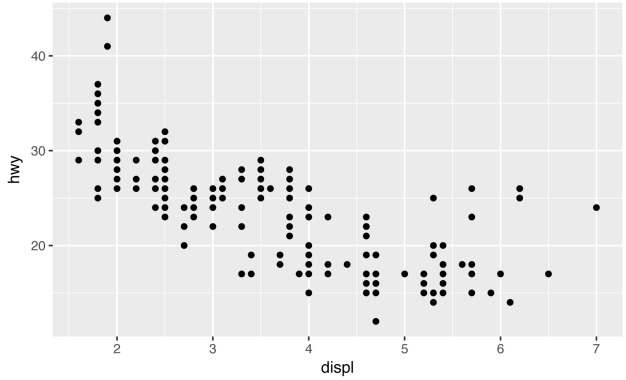
I only ever use three of the five options:

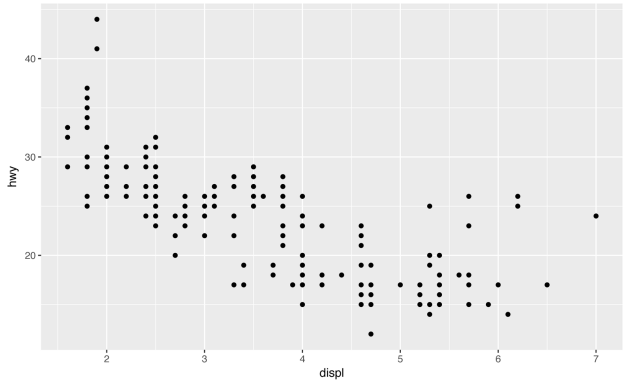
- I find it most aesthetically pleasing for plots to have a consistent width. To enforce this, I set `fig.width = 6` (6") and `fig.asp = 0.618` (the golden ratio) in the defaults. Then in individual chunks, I only adjust `fig.asp`.
- I control the output size with `out.width` and set it to a percentage of the line width). I default to `out.width = "70%"` and `fig.align = "center"`. That give plots room to breathe, without taking up too much space.
- To put multiple plots in a single row I set the `out.width` to `50%` for two plots, `33%` for three plots, or `25%` to four plots, and set `fig.align = "default"`. Depending on what I'm trying to illustrate (e.g., show data or show plot variations), I'll also tweak `fig.width`, as discussed next.

If you find that you're having to squint to read the text in your plot, you need to tweak `fig.width`. If `fig.width` is larger than the size the figure is rendered in the final doc, the text will be too small; if `fig.width` is smaller, the text will be too big. You'll often need to do a little experimentation to figure out the right ratio between the `fig.width` and the eventual width in your document. To illustrate

the principle, the following three plots have `fig.width` of 4, 6, and 8, respectively:







If you want to make sure the font size is consistent across all your figures, whenever you set `out.width`, you'll also need to adjust `fig.width` to maintain the same ratio with your default `out.width`. For example, if your default `fig.width` is 6 and `out.width` is 0.7, when you set `out.width = "50%"` you'll need to set `fig.width` to 4.3 (6 * 0.5 / 0.7).

Other Important Options

When mingling code and text, like I do in this book, I recommend setting `fig.show = "hold"` so that plots are shown after the code. This has the pleasant side effect of forcing you to break up large blocks of code with their explanations.

To add a caption to the plot, use `fig.cap`. In R Markdown this will change the figure from inline to "floating."

If you're producing PDF output, the default graphics type is PDF. This is a good default because PDFs are high-quality vector graphics. However, they can produce very large and slow plots if you are displaying thousands of points. In that case, set `dev = "png"` to force the use of PNGs. They are slightly lower quality, but will be much more compact.

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

Learning More

The absolute best place to learn more is the **ggplot2** book: *ggplot2: Elegant graphics for data analysis*. It goes into much more depth about the underlying theory, and has many more examples of how to combine the individual pieces to solve practical problems. Unfortunately, the book is not available online for free, although you can find the source code at *https://github.com/hadley/ggplot2-book*.

Another great resource is the **ggplot2** extensions guide. This site lists many of the packages that extend **ggplot2** with new geoms and scales. It's a great place to start if you're trying to do something that seems hard with **ggplot2**.

CHAPTER 23

R Markdown Formats

Introduction

So far you've seen R Markdown used to produce HTML documents. This chapter gives a brief overview of some of the many other types of output you can produce with R Markdown. There are two ways to set the output of a document:

1. Permanently, by modifying the the YAML header:

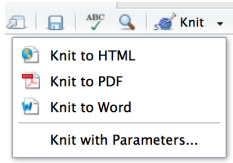
   ```
   title: "Viridis Demo"
   output: html_document
   ```

2. Transiently, by calling `rmarkdown::render()` by hand:

   ```
   rmarkdown::render(
     "diamond-sizes.Rmd",
     output_format = "word_document"
   )
   ```

   This is useful if you want to programmatically produce multiple types of output.

RStudio's knit button renders a file to the first format listed in its `output` field. You can render to additional formats by clicking the drop-down menu beside the knit button.



Output Options

Each output format is associated with an R function. You can either write `foo` or `pkg::foo`. If you omit `pkg`, the default is assumed to be **rmarkdown**. It's important to know the name of the function that makes the output because that's where you get help. For example, to figure out what parameters you can set with `html_document`, look at `?rmarkdown:html_document()`.

To override the default parameter values, you need to use an expanded `output` field. For example, if you wanted to render an `html_document` with a floating table of contents, you'd use:

```
output:
  html_document:
    toc: true
    toc_float: true
```

You can even render to multiple outputs by supplying a list of formats:

```
output:
  html_document:
    toc: true
    toc_float: true
  pdf_document: default
```

Note the special syntax if you don't want to override any of the default options.

Documents

The previous chapter focused on the default `html_document` output. There are number of basic variations on that theme, generating different types of documents:

- `pdf_document` makes a PDF with LaTeX (an open source document layout system), which you'll need to install. RStudio will prompt you if you don't already have it.

- `word_document` for Microsoft Word documents (*.docx*).
- `odt_document` for OpenDocument Text documents (*.odt*).
- `rtf_document` for Rich Text Format (*.rtf*) documents.
- `md_document` for a Markdown document. This isn't typically useful by itself, but you might use it if, for example, your corporate CMS or lab wiki uses Markdown.
- `github_document` is a tailored version of `md_document` designed for sharing on GitHub.

Remember, when generating a document to share with decision makers, you can turn off the default display of code by setting global options in the setup chunk:

```
knitr::opts_chunk$set(echo = FALSE)
```

For `html_documents` another option is to make the code chunks hidden by default, but visible with a click:

```
output:
  html_document:
    code_folding: hide
```

Notebooks

A notebook, `html_notebook`, is a variation on an `html_document`. The rendered outputs are very similar, but the purpose is different. An `html_document` is focused on communicating with decision makers, while a notebook is focused on collaborating with other data scientists. These different purposes lead to using the HTML output in different ways. Both HTML outputs will contain the fully rendered output, but the notebook also contains the full source code. That means you can use the *.nb.html* generated by the notebook in two ways:

- You can view it in a web browser, and see the rendered output. Unlike `html_document`, this rendering always includes an embedded copy of the source code that generated it.
- You can edit it in RStudio. When you open an *.nb.html* file, RStudio will automatically re-create the *.Rmd* file that generated it. In the future, you can also include supporting files (e.g., *.csv* data files), which will be automatically extracted when needed.

Emailing *.nb.html* files is a simple way to share analyses with your colleagues. But things will get painful as soon as they want to make changes. If this starts to happen, it's a good time to learn Git and GitHub. Learning Git and GitHub is definitely painful at first, but the collaboration payoff is huge. As mentioned earlier, Git and GitHub are outside the scope of the book, but there's one tip that's useful if you're already using them: use both `html_notebook` and `github_document` outputs:

```
output:
  html_notebook: default
  github_document: default
```

`html_notebook` gives you a local preview, and a file that you can share via email. `github_document` creates a minimal MD file that you can check into Git. You can easily see how the results of your analysis (not just the code) change over time, and GitHub will render it for you nicely online.

Presentations

You can also use R Markdown to produce presentations. You get less visual control than with a tool like Keynote or PowerPoint, but automatically inserting the results of your R code into a presentation can save a huge amount of time. Presentations work by dividing your content into slides, with a new slide beginning at each first (`#`) or second (`##`) level header. You can also insert a horizontal rule (`***`) to create a new slide without a header.

R Markdown comes with three presentations formats built in:

`ioslides_presentation`
: HTML presentation with ioslides.

`slidy_presentation`
: HTML presentation with W3C Slidy.

`beamer_presentation`
: PDF presentation with LaTeX Beamer.

Two other popular formats are provided by packages:

`revealjs::revealjs_presentation`
: HTML presentation with reveal.js. Requires the **revealjs** package.

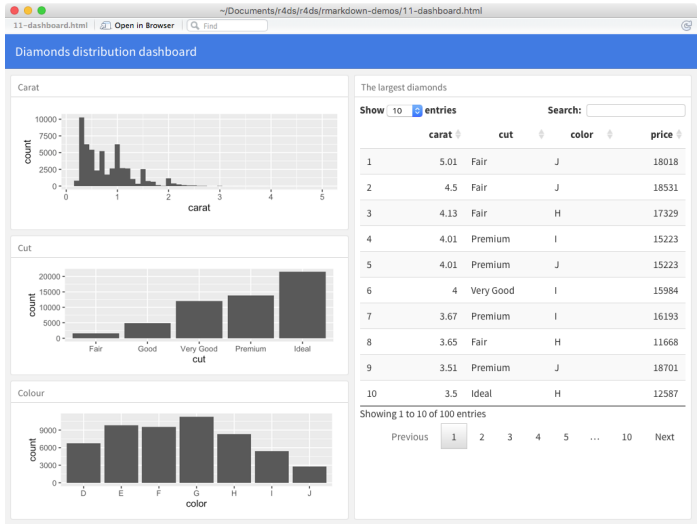
`rmdshower`
: Provides a wrapper around the **shower** presentation engine.

Dashboards

Dashboards are a useful way to communicate large amounts of information visually and quickly. **flexdashboard** makes it particularly easy to create dashboards using R Markdown and a convention for how the headers affect the layout:

- Each level 1 header (#) begins a new page in the dashboard.
- Each level 2 header (##) begins a new column.
- Each level 3 header (###) begins a new row.

For example, you can produce this dashboard:



Using this code:

```
---
title: "Diamonds distribution dashboard"
output: flexdashboard::flex_dashboard
---

```{r setup, include = FALSE}
library(ggplot2)
library(dplyr)
```
```

````
knitr::opts_chunk$set(fig.width = 5, fig.asp = 1/3)
```

## Column 1

### Carat

```{r}
ggplot(diamonds, aes(carat)) + geom_histogram(binwidth = 0.1)
```

### Cut

```{r}
ggplot(diamonds, aes(cut)) + geom_bar()
```

### Color

```{r}
ggplot(diamonds, aes(color)) + geom_bar()
```

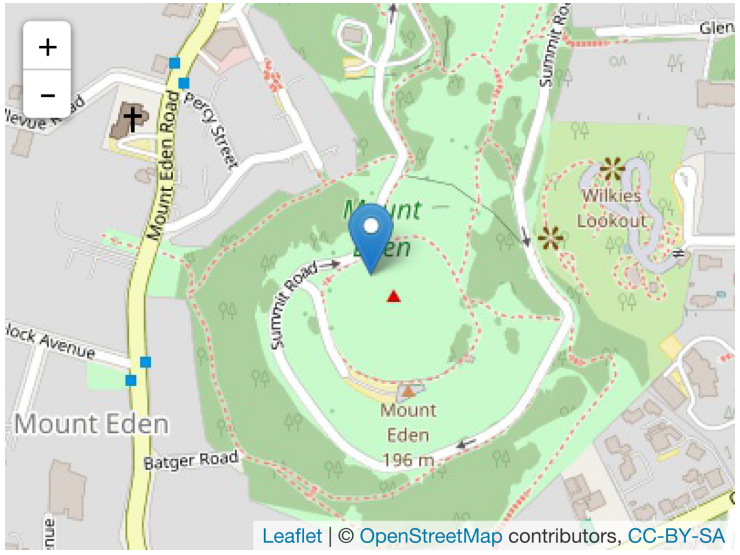
## Column 2

### The largest diamonds

```{r}
diamonds %>%
  arrange(desc(carat)) %>%
  head(100) %>%
  select(carat, cut, color, price) %>%
  DT::datatable()
```
````

flexdashboard also provides simple tools for creating sidebars, tabsets, value boxes, and gauges. To learn more about **flexdashboard** visit *http://rmarkdown.rstudio.com/flexdashboard/*.

Interactivity

Any HTML format (document, notebook, presentation, or dashboard) can contain interactive components.

htmlwidgets

HTML is an interactive format, and you can take advantage of that interactivity with *htmlwidgets*, R functions that produce interactive HTML visualizations. For example, take the following *leaflet* map. If
````

you're viewing this page on the web, you can drag the map around, zoom in and out, etc. You obviously can't do that on a book, so **rmarkdown** automatically inserts a static screenshot for you:

```
library(leaflet)
leaflet() %>%
  setView(174.764, -36.877, zoom = 16) %>%
  addTiles() %>%
  addMarkers(174.764, -36.877, popup = "Maungawhau")
```



The great thing about htmlwidgets is that you don't need to know anything about HTML or JavaScript to use them. All the details are wrapped inside the package, so you don't need to worry about it.

There are many packages that provide htmlwidgets, including:

- **dygraphs** for interactive time series visualizations.
- **DT** for interactive tables.
- **rthreejs** for interactive 3D plots.
- **DiagrammeR** for diagrams (like flow charts and simple node-link diagrams).

To learn more about htmlwidgets and see a more complete list of packages that provide them, visit *http://www.htmlwidgets.org/*.

Shiny

htmlwidgets provide *client-side* interactivity—all the interactivity happens in the browser, independently of R. On one hand, that's great because you can distribute the HTML file without any connection to R. However, that fundamentally limits what you can do to things that have been implemented in HTML and JavaScript. An alternative approach is to use **Shiny**, a package that allows you to create interactivity using R code, not JavaScript.

To call **Shiny** code from an R Markdown document, add `runtime: shiny` to the header:

```
title: "Shiny Web App"
output: html_document
runtime: shiny
```

Then you can use the "input" functions to add interactive components to the document:

```
library(shiny)

textInput("name", "What is your name?")
numericInput("age", "How old are you?", NA, min = 0, max = 150)
```

You can then refer to the values with `input$name` and `input$age`, and the code that uses them will be automatically rerun whenever they change.

I can't show you a live **Shiny** app here because **Shiny** interactions occur on the *server side*. This means you can write interactive apps without knowing JavaScript, but it means that you need a server to run it on. This introduces a logistical issue: **Shiny** apps need a **Shiny** server to be run online. When you run **Shiny** apps on your own computer, **Shiny** automatically sets up a **Shiny** server for you, but you need a public-facing **Shiny** server if you want to publish this sort of interactivity online. That's the fundamental trade-off of **Shiny**: you can do anything in a **Shiny** document that you can do in R, but it requires someone to be running R.

Learn more about **Shiny** at *http://shiny.rstudio.com/*.

Websites

With a little additional infrastructure you can use R Markdown to generate a complete website:

- Put your *.Rmd* files in a single directory. *index.Rmd* will become the home page.
- Add a YAML file named *_site.yml* that provides the navigation for the site. For example:

```
name: "my-website"
navbar:
  title: "My Website"
  left:
    - text: "Home"
      href: index.html
    - text: "Viridis Colors"
      href: 1-example.html
    - text: "Terrain Colors"
      href: 3-inline.html
```

Execute `rmarkdown::render_site()` to build *_site*, a directory of files ready to deploy as a standalone static website, or if you use an RStudio Project for your website directory. RStudio will add a Build tab to the IDE that you can use to build and preview your site.

Read more at *http://bit.ly/RMarkdownWebsites*.

Other Formats

Other packages provide even more output formats:

- The **bookdown** package makes it easy to write books, like this one. To learn more, read *Authoring Books with R Markdown*, by Yihui Xie, which is, of course, written in bookdown. Visit *http://www.bookdown.org* to see other bookdown books written by the wider R community.
- The **prettydoc** package provides lightweight document formats with a range of attractive themes.
- The **rticles** package compiles a selection of formats tailored for specific scientific journals.

See *http://rmarkdown.rstudio.com/formats.html* for a list of even more formats. You can also create your own by following the instructions at *http://bit.ly/CreatingNewFormats*.

Learning More

To learn more about effective communication in these different formats I recommend the following resources:

- To improve your presentation skills, I recommend *Presentation Patterns* by Neal Ford, Matthew McCollough, and Nathaniel Schutta. It provides a set of effective patterns (both low- and high-level) that you can apply to improve your presentations.
- If you give academic talks, I recommend reading the *Leek group guide to giving talks*.
- I haven't taken it myself, but I've heard good things about Matt McGarrity's online course on public speaking.
- If you are creating a lot of dashboards, make sure to read Stephen Few's *Information Dashboard Design: The Effective Visual Communication of Data*. It will help you create dashboards that are truly useful, not just pretty to look at.
- Effectively communicating your ideas often benefits from some knowledge of graphic design. *The Non-Designer's Design Book* is a great place to start.

CHAPTER 24
R Markdown Workflow

Earlier, we discussed a basic workflow for capturing your R code where you work interactively in the *console*, then capture what works in the *script editor*. R Markdown brings together the console and the script editor, blurring the lines between interactive exploration and long-term code capture. You can rapidly iterate within a chunk, editing and re-executing with Cmd/Ctrl-Shift-Enter. When you're happy, you move on and start a new chunk.

R Markdown is also important because it so tightly integrates prose and code. This makes it a great *analysis notebook* because it lets you develop code and record your thoughts. An analysis notebook shares many of the same goals as a classic lab notebook in the physical sciences. It:

- Records what you did and why you did it. Regardless of how great your memory is, if you don't record what you do, there will come a time when you have forgotten important details. Write them down so you don't forget!
- Supports rigorous thinking. You are more likely to come up with a strong analysis if you record your thoughts as you go, and continue to reflect on them. This also saves you time when you eventually write up your analysis to share with others.
- Helps others understand your work. It is rare to do data analysis by yourself, and you'll often be working as part of a team. A lab notebook helps you share not only what you've done, but why you did it with your colleagues or lab mates.

Much of the good advice about using lab notebooks effectively can also be translated to analysis notebooks. I've drawn on my own experiences and Colin Purrington's advice on lab notebooks (*http://colinpurrington.com/tips/lab-notebooks*) to come up with the following tips:

- Ensure each notebook has a descriptive title, an evocative filename, and a first paragraph that briefly describes the aims of the analysis.
- Use the YAML header date field to record the date you started working on the notebook:

  ```
  date: 2016-08-23
  ```

  Use ISO8601 YYYY-MM-DD format so that's there no ambiguity. Use it even if you don't normally write dates that way!
- If you spend a lot of time on an analysis idea and it turns out to be a dead end, don't delete it! Write up a brief note about why it failed and leave it in the notebook. That will help you avoid going down the same dead end when you come back to the analysis in the future.
- Generally, you're better off doing data entry outside of R. But if you do need to record a small snippet of data, clearly lay it out using `tibble::tribble()`.
- If you discover an error in a data file, never modify it directly, but instead write code to correct the value. Explain why you made the fix.
- Before you finish for the day, make sure you can knit the notebook (if you're using caching, make sure to clear the caches). That will let you fix any problems while the code is still fresh in your mind.
- If you want your code to be reproducible in the long run (i.e., so you can come back to run it next month or next year), you'll need to track the versions of the packages that your code uses. A rigorous approach is to use **packrat**, which stores packages in your project directory, or **checkpoint**, which will reinstall packages available on a specified date. A quick and dirty hack is to include a chunk that runs `sessionInfo()`—that won't let you easily re-create your packages as they are today, but at least you'll know what they were.

- You are going to create many, many, many analysis notebooks over the course of your career. How are you going to organize them so you can find them again in the future? I recommend storing them in individual projects, and coming up with a good naming scheme.

Index

E

F

G

H

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

About the Authors

Hadley Wickham is Chief Scientist at RStudio and a member of the R Foundation. He builds tools (both computational and cognitive) that make data science easier, faster, and more fun. His work includes packages for data science (the tidyverse: ggplot2, dplyr, tidyr, purrr, readr, ...), and principled software development (roxygen2, testthat, devtools). He is also a writer, educator, and frequent speaker promoting the use of R for data science. Learn more on his website, *http://hadley.nz*.

Garrett Grolemund is a statistician, teacher, and R developer who works for RStudio. He wrote the well-known lubridate R package and is the author of *Hands-On Programming with R* (O'Reilly).

Garrett is a popular R instructor at *DataCamp.com* and *oreilly.com/safari*, and has been invited to teach R and Data Science at many companies, including Google, eBay, Roche, and more. At RStudio, Garrett develops webinars, workshops, and an acclaimed series of cheat sheets for R.

Colophon

The animal on the cover of *R for Data Science* is the kakapo (*Strigops habroptilus*). Also known as the owl parrot, the kakapo is a large flightless bird native to New Zealand. Adult kakapos can grow up to 64 centimeters in height and 4 kilograms in weight. Their feathers are generally yellow and green, although there is significant variation between individuals. Kakapos are nocturnal and use their robust sense of smell to navigate at night. Although they cannot fly, kakapos have strong legs that enable them to run and climb much better than most birds.

The name kakapo comes from the language of the native Maori people of New Zealand. Kakapos were an important part of Maori culture, both as a food source and as a part of Maori mythology. Kakapo skin and feathers were also used to make cloaks and capes.

Due to the introduction of predators to New Zealand during European colonization, kakapos are now critically endangered, with less than 200 individuals currently living. The government of New Zealand has been actively attempting to revive the kakapo population by providing special conservation zones on three predator-free islands.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to *animals.oreilly.com*.

The cover image is from *Wood's Animate Creations*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.