The Tidynomicon

A Brief Introduction to R for People Who Count From Zero

Greg Wilson

Contents

Chapter 1

Introduction

Years ago, Patrick Burns wrote *The R Inferno*, a guide to R for those who think they are in hell. Upon first encountering the language after two decades of using Python, I thought Burns was an optimist—after all, hell has rules.

I have since realized that R does too, and that they are no more confusing or contradictory than those of other programming languages. They only appear so because R draws on a tradition unfamiliar to those of us raised with derivatives of C. Counting from one, copying data rather than modifying it, lazy evaluation: to quote the other bard, these are not mad, just differently sane.

Welcome, then, to a universe where the strange will become familiar, and everything familiar, strange. Welcome, thrice welcome, to R.

1.1 Who are these lessons for?

Andrzej completed a Master's in library science five years ago and has done data analysis for various school boards since then. He learned Python doing data science courses online, but has no formal training in programming. He just joined team that uses R and R Markdown to generate reports, and these lessons will show him how to translate his understanding of Python to R.

Padma has been building dashboards for a logistics company using Django and D3 while also doing systems administration and managing deployments. The company has just hired some data scientists who would like to rebuild some of her dashboards in Shiny. Padma isn't a statistician, but would like to learn enough about R to help the analysts and get their code into production.

Figure 1.1: Speak not of madness, oh you who count from zero.

1.2 How do I get started?

You will learn as much or more from the exercise in this book as from the lessons themselves. To start, you can create an account on rstudio.cloud, clone the tidynomicon project, and work in that. If you prefer to work on your own computer, you must install R and then install RStudio. We recommend that you do *not* use conda, brew, or other platform-specific package managers to do this, as they sometimes only install part of what you need. You will need additional software packages as we go along; each shall be named and summoned in due course.

Chapter 2

Simple Beginnings

We begin by introducing the basic elements of R. You will use these less often than you might expect, but they are the building blocks for the higher-level tools introduced in Chapter 3, and offer the comfort of familiarity. Where we feel comparisons would aid understanding, we provide short examples in Python.

2.1 Learning Objectives

- Name and describe R's atomic data types and create objects of those types.
- Explain what 'scalar' values actually are in R.
- Identify correct and incorrect variable names in R.
- Create vectors in R and index them to select single values, ranges of values, and selected values.
- Explain the difference between `NA` and `NULL` and correctly use tests for each.
- Explain the difference between a list and a vector.
- Explain the difference between indexing with `[` and with `[[`.
- Use `[` and `[[` correctly to extract elements and sub-structures from data structures in R.
- Create a named list in R.
- Access elements by name using both `[` and `$` notation.
- Correctly identify cases in which back-quoting is necessary when accessing elements via `$`.
- Create and index matrices in R.
- Create `for` loops and `if`/`else` statements in R.
- Explain why vectors cannot be used directly in conditional expressions and correctly use `all` and `any` to combine their values.
- Define functions taking a fixed number of named arguments and/or a variable number of arguments.

Figure 2.1: RStudio Console

- Explain what vectorization is and create vectorized equivalents of unnested loops containing simple conditional tests.

2.2 How do I say hello?

We begin with a traditional greeting. In Python, we write:

```
print("Hello, world!")
```

```
Hello, world!
```

We can run the equivalent R in the RStudio Console (Figure 2.1):

```
print("Hello, world!")
```

```
[1] "Hello, world!"
```

Python prints what we asked for, but what does the `[1]` in R's output signify? Is it perhaps something akin to a line number? Let's take a closer look by evaluating a couple of expressions without calling `print`:

```
'This is in single quotes.'
```

```
[1] "This is in single quotes."
```

```
"This is in double quotes."
```

```
[1] "This is in double quotes."
```

[1] doesn't appear to be a line number; let's ignore it for now and do a little more exploring.

> Note that R uses double quotes to display strings even when we give it a single-quoted string (which is no worse than Python using single quotes when we've given it doubles).

2.3 How do I add numbers?

In Python, we add numbers using `+`.

```
print(1 + 2 + 3)
```

```
6
```

We can check the type of the result using `type`, which tells us that the result `6` is an integer:

```
print(type(6))
```

```
<class 'int'>
```

What does R do?

```
1 + 2 + 3
```

```
[1] 6
```

```
typeof(6)
```

```
[1] "double"
```

R's type inspection function is called `typeof` rather than `type`, and it returns the type's name as a string. That's all fine, but it seems odd for integer addition to produce a double-precision floating-point result. Let's try an experiment:

```
typeof(6)
```

```
[1] "double"
```

Ah: by default, R represents numbers as floating-point values, even if they look like integers when written. We can force a literal value to be an integer by appending an upper-case `L` (which stands for "long integer"):

```
typeof(6L)
```

```
[1] "integer"
```

Arithmetic on integers does produce integers:

```
typeof(1L + 2L + 3L)
```

```
[1] "integer"
```

and if we want to convert a floating-point number to an integer we can do so:

```
typeof(as.integer(6))
```

```
[1] "integer"
```

But wait: what is that dot in `as.integer`'s name? Is there an object called `as` with a method called `integer`? The answer is "no": `.` is (usually) just another character in R; like the underscore `_`, it is used to make names more readable.

2.4 How do I store many numbers together?

The Elder Gods do not bother to learn most of our names because there are so many of us and we are so ephemeral. Similarly, we only give a handful of values in our programs their own names; we lump the rest together into lists, matrices, and more esoteric structure so that we too can create, manipulate, and dispose of multitudes with a single imperious command.

The most common such structure in Python is the list. We create lists using square brackets and assign a list to a variable using `=`. If the variable does not exist, it is created:

```
primes = [3, 5, 7, 11]
print(primes)
```

```
[3, 5, 7, 11]
```

Since assignment is a statement rather than an expression, it has no result, so Python does not display anything when this command is run.

The equivalent operation in R uses a function called `c`, which stands for "column" and which creates a vector:

```
primes <- c(3, 5, 7, 11)
primes
```

```
[1]  3  5  7 11
```

Assignment is done using a left-pointing arrow `<-` (though other forms exist, which we will discuss later). As in Python, assignment is a statement rather than an expression, so we enter the name of the newly-created variable to get R to display its value.

Now that we can create vectors in R, we can explain the errant `[1]` in our previous examples. To start, let's have a look at the lengths of various things in Python:

```
print(primes, len(primes))
```

```
[3, 5, 7, 11] 4
```

```
print(len(4))
```

```
Error in py_call_impl(callable, dots$args, dots$keywords): TypeError: object of type 'int' has no

Detailed traceback:
  File "<string>", line 1, in <module>
```

Fair enough: the length of a list is the number of elements it contains, and since a scalar like the integer 4 doesn't contain elements, it has no length. What of R's vectors?

```
length(primes)
```

```
[1] 4
```

Good—and numbers?

```
length(4)
```

```
[1] 1
```

That's surprising. Let's have a closer look:

```
typeof(primes)
```

```
[1] "double"
```

That's also unexpected: the type of the vector is the type of the elements it contains. This all becomes clear once we realize that *there are no scalars in R*. `4` is not a single lonely integer, but rather a vector of length one containing the value 4. When we display its value, the `[1]` that R prints is the index of its first value. We can prove this by creating and displaying a longer vector:

```
c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
  1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

```
 [1]  1  2  3  4  5  6  7  8  9 10  1  2  3  4  5  6  7  8  9 10  1  2  3
[24]  4  5  6  7  8  9 10  1  2  3  4  5  6  7  8  9 10
```

In order to help us find out way in our data, R automatically breaks long lines and displays the starting index of each line. These indices also show us that R counts from 1 as humans do, rather than from zero. (There are a great many myths about why programming languages do the latter. The truth is stranger than any fiction could be.)

2.5 How do I index a vector?

Python's rules for indexing are simple once you understand them (a statement which is also true of quantum mechanics and necromancy). To avoid confusing indices with values, let's create a list of color names and index that:

```
colors = ["eburnean", "glaucous", "wenge"]
print(colors[0])
```

```
eburnean
```

```
print(colors[2])
```

```
wenge
```

```
colors[3]
```

```
Error in py_call_impl(callable, dots$args, dots$keywords): IndexError: list index out

Detailed traceback:
  File "<string>", line 1, in <module>
```

```
print(colors[-1])
```

```
wenge
```

Indexing the equivalent vector in R with the indices 1 to 3 produces unsurprising results:

```
colors <- c("eburnean", "glaucous", "wenge")
colors[1]
```

```
[1] "eburnean"
```

```
colors[3]
```

```
[1] "wenge"
```

What happens if we go off the end?

```
colors[4]
```

```
[1] NA
```

R handles gaps in data using the special value `NA` (short for "not available"), and returns this value when we ask for a nonexistent element of a vector. But it does more than this—much more. In Python, a negative index counts backward from the end of a list. In R, we use a negative index to indicate a value that we don't want:

```
colors[-1]
```

```
[1] "glaucous" "wenge"
```

But wait. If every value in R is a vector, then when we use 1 or -1 as an index, we're actually using a vector to index another one. What happens if the index itself contains more than one value?

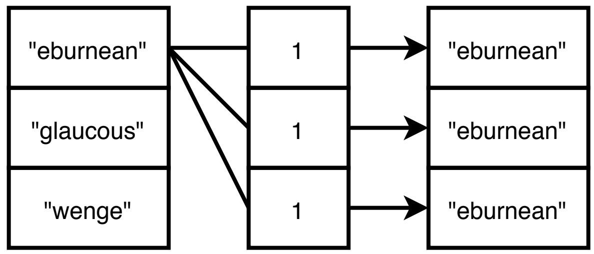


Figure 2.2: Pull Indexing

```
colors[1, 2]
```

```
Error in colors[1, 2]: incorrect number of dimensions
```

That didn't work because R interprets `[i, j]` as being row and column indices, and our vector has only one dimension. What if we create a vector with `c(...)` and use that as a subscript?

```
colors[c(3, 1, 2)]
```

```
[1] "wenge"    "eburnean" "glaucous"
```

That works, and allows us to repeat elements:

```
colors[c(1, 1, 1)]
```

```
[1] "eburnean" "eburnean" "eburnean"
```

Note that this is pull indexing, i.e., the value at location i in the index vector specifies which element of the source vector is being pulled into that location in the result vector (Figure 2.2).

We can also select out several elements:

```
colors[c(-1, -2)]
```

```
[1] "wenge"
```

But we cannot simultaneously select elements in (with positive indices) and out (with negative ones):

```
colors[c(1, -1)]
```

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

That error message is suggestive: what happens if we use 0 as an index?

```
colors[0]
```

```
character(0)
```

In order to understand this rather cryptic response, we can try calling the function `character` ourselves with a positive argument:

```
character(3)
```

```
[1] "" "" ""
```

Ah: `character(N)` constructs a vector of empty strings of the specified length. The expression `character(0)` presumably therefore means "an empty vector of type character". From this, we conclude that the index 0 doesn't correspond to any elements, so R gives us back something of the right type but with no content. As a check, let's try indexing with 0 and 1 together:

```
colors[c(0, 1)]
```

```
[1] "eburnean"
```

So when 0 is mixed with either positive or negative indices, it is ignored, which will undoubtedly lead to some puzzling bugs. What if in-bounds and out-of-bounds indices are mixed?

```
colors[c(1, 10)]
```

```
[1] "eburnean" NA
```

That is consistent with the behavior of single indices.

2.6 How do I create new vectors from old?

Modern Python encourages programmers to use list comprehensions instead of loops, i.e., to write:

```
original = [3, 5, 7, 9]
doubled = [2 * x for x in original]
print(doubled)
```

```
[6, 10, 14, 18]
```

instead of:

```
doubled = []
for x in original:
  doubled.append(2 * x)
print(doubled)
```

```
[6, 10, 14, 18]
```

If `original` is a NumPy array, we can shorten this to `2 * original`. R provides this capability in the language itself:

```
original <- c(3, 5, 7, 9)
doubled <- 2 * original
doubled
```

```
[1]  6 10 14 18
```

Modern R strongly encourages us to vectorize computations in this way, i.e., to do operations on whole vectors at once rather than looping over their contents. To aid this, all arithmetic operations work element by element on vectors:

```
tens <- c(10, 20, 30)
hundreds <- c(100, 200, 300)
tens + hundreds / (tens * hundreds)
```

```
[1] 10.10000 20.05000 30.03333
```

If two vectors of unequal length are used together, the elements of the shorter are recycled. This behaves sensibly if one of the vectors is a scalar—it is just re-used as many times as necessary:

```
hundreds + 5
```

```
[1] 105 205 305
```

If both vectors have several elements, the shorter is repeated as often as necessary. This works, but is so likely to lead to hard-to-find bugs that R produces a warning message:

```
thousands <- c(1000, 2000)
hundreds + thousands
```

```
Warning in hundreds + thousands: longer object length is not a multiple of
shorter object length
```

```
[1] 1100 2200 1300
```

R also provides vectorized alternatives to `if-else` statements. If we use a vector containing the logical (or Boolean) values `TRUE` and `FALSE` as an index, it selects elements corresponding to `TRUE` values:

```
colors # as a reminder
```

```
[1] "eburnean" "glaucous" "wenge"
```

```
colors[c(TRUE, FALSE, TRUE)]
```

```
[1] "eburnean" "wenge"
```

This is called logical indexing, though to the best of my knowledge illogical indexing is not provided as an alternative. The function `ifelse` uses this to do what its name suggests: select a value from one vector if a condition is `TRUE`, and a corresponding value from another vector if the condition is `FALSE`:

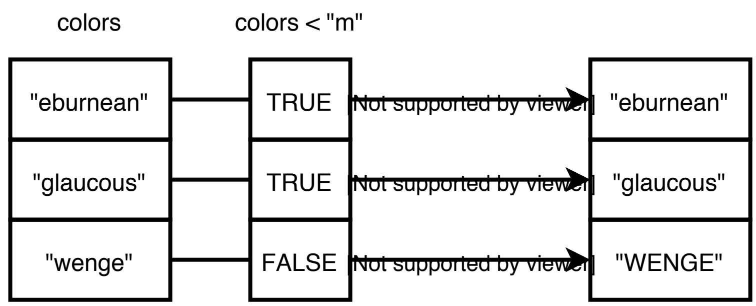


Figure 2.3: Vector Conditionals

```
before_letter_m <- colors < "m"
before_letter_m # to show the index
```

```
[1]  TRUE  TRUE FALSE
```

```
ifelse(before_letter_m, colors, c("comes", "after", "m"))
```

```
[1] "eburnean" "glaucous" "m"
```

All three vectors are of the same length, and the first (the condition) is usually constructed using the values of one or both of the other vectors:

```
ifelse(colors < "m", colors, toupper(colors))
```

```
[1] "eburnean" "glaucous" "WENGE"
```

2.7 How else does R represent the absence of data?

The special value `NA` means "there's supposed to be a value here but we don't know what it is." A different value, `NULL`, represents the absence of a vector. It is not the same as a vector of zero length, though testing that statement produces a rather odd result:

```
NULL == integer(0)
```

```
logical(0)
```

The safe way to test if something is `NULL` is to use the function `is.null`:

```
is.null(NULL)
```

```
[1] TRUE
```

Circling back, the safe way to test whether a value is `NA` is *not* to use direct comparison:

```
threshold <- 1.75
threshold == NA
```

```
[1] NA
```

The result is `NA` because if we don't know what the value is, we can't know if it's equal to `threshold` or not. Instead, we should always use the function `is.na`:

```
is.na(threshold)
```

```
[1] FALSE
```

```
is.na(NA)
```

```
[1] TRUE
```

2.8 How can I store a mix of different types of objects?

One of the things that newcomers to R often trip over is the various ways in which structures can be indexed. All of the following are legal:

```
thing[i]
thing[i, j]
thing[[i]]
thing[[i, j]]
thing$name
thing$"name"
```

but they can behave differently depending on what kind of thing `thing` is. To explain, we must first take a look at lists.

A list in R is a vector that can contain values of many different types. (The technical term for this is heterogeneous, in contrast with a homogeneous data structure that can only contain one type of value.) We'll use this list in our examples:

```
thing <- list("first", c(2, 20, 200), 3.3)
thing
```

```
[[1]]
[1] "first"

[[2]]
[1]   2  20 200

[[3]]
```

```
[1] 3.3
```

The output tells us that the first element of `thing` is a vector of one element, that the second is a vector of three elements, and the third is again a vector of one element; the major indices are shown in `[[…]]`, while the indices of the contained elements are shown in `[…]`. (Again, remember that `"first"` and 3.3 are actually vectors of length 1.)

> In keeping with R's conventions, we will henceforth use `[[` and `[` to refer to the two kinds of indexing rather than `[[…]]` and `[…]`.

2.9 What is the difference between `[` and `[[`?

The output above strongly suggests that we can get the elements of a list using `[[` (double square brackets):

```
thing[[1]]
```

```
[1] "first"
```

```
thing[[2]]
```

```
[1]   2  20 200
```

```
thing[[3]]
```

```
[1] 3.3
```

Let's have a look at the types of those three values:

```
typeof(thing[[1]])
```

```
[1] "character"
```

```
typeof(thing[[2]])
```

```
[1] "double"
```

```
typeof(thing[[3]])
```

```
[1] "double"
```

That seems sensible. Now, what do we get if we index single square brackets `[…]`?

```
thing[1]
```

```
[[1]]
[1] "first"
```

That looks like a list, not a vector—let's check:

```
typeof(thing[1])
```

```
[1] "list"
```

This shows the difference between `[[` and `[`: the former peels away a layer of data structure, returning only the sub-structure, while the latter gives us back a structure of the same type as the thing being indexed. Since a "scalar" is just a vector of length 1, there is no difference between `[[` and `[` when they are applied to vectors:

```
v <- c("first", "second", "third")
v[2]
```

```
[1] "second"
```

```
typeof(v[2])
```

```
[1] "character"
```

```
v[[2]]
```

```
[1] "second"
```

```
typeof(v[[2]])
```

```
[1] "character"
```

> **Flattening and Recursive Indexing**
>
> If a list is just a vector of objects, why do we need the function `list`? Why can't we create a list with `c("first", c(2, 20, 200), 30)`? The answer is that R flattens the arguments to `c`, so that `c(c(1, 2), c(3, 4))` produces `c(1, 2, 3, 4)`. It also does automatic type conversion: `c("first", c(2, 20, 200), 30)` produces a vector of character strings `c("first", "2", "20", "200", "30")`. This is helpful once you get used to it (which once again is true of both quantum mechanics and necromancy).
>
> Another "helpful, ish" behavior is that using `[[` with a list subsets recursively: if `thing <- list(a = list(b = list(c = list(d = 1))))`, then `thing[[c("a", "b", "c", "d")]]` selects the 1.

2.10 How can I access elements by name?

R allows us to name the elements in vectors and lists: if we assign `c(one = 1, two = 2, three = 3)` to `names`, then `names["two"]` is 2. We can use this to create a lookup table:

```
values <- c("m", "f", "nb", "f", "f", "m", "m")
lookup <- c(m = "Male", f = "Female", nb = "Non-binary")
lookup[values]
```

```
           m            f           nb            f            f
      "Male"     "Female" "Non-binary"     "Female"     "Female"
           m            m
      "Male"       "Male"
```

If the structure in question is a list rather than an atomic vector of numbers, characters, or logicals, we can use the syntax `lookup$m` instead of `lookup["m"]`:

```
lookup_list <- list(m = "Male", f = "Female", nb = "Non-binary")
lookup_list$m
```

```
[1] "Male"
```

We will explore this in more detail when we look at the tidyverse in Chapter 3, since that is where access-by-name is used most often. For now, simply note that if the name of an element isn't a legal variable name, we have to put it in backward quotes to use it with `$`:

```
another_list <- list("first field" = "F", "second field" = "S")
another_list$`first field`
```

```
[1] "F"
```

> If you have control, or at least the illusion thereof, choose names such as `first_field` that don't require back-quoting.

2.11 How can I create and index a matrix?

Matrices are frequently used in statistics, so R provides built-in support for them. After `a <- matrix(1:9, nrow = 3)`, `a` is a 3x3 matrix containing the values 1 through 9:

```
a <- matrix(1:9, nrow = 3)
a
```

```
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

Behind the scenes, a matrix is a vector with an attribute called `dim` that stores its dimensions:

```
dim(a)
```

```
[1] 3 3
```

`a[3, 3]` is a vector of length 1 containing the value 9 (again, "scalars" in R are actually vectors), while `a[1,]` is the vector `c(1, 4, 7)` (because we are selecting the first row of the matrix) and `a[,1]` is the vector `c(1, 2, 3)` (because we are selecting the first column of the matrix). Elements can still be accessed using a single index, which returns the value from that location in the underlying vector:

```
a[8]
```

```
[1] 8
```

2.12 How do I choose and repeat things?

We cherish the illusion of free will so much that we embed a pretense of it in our machines in the form of conditional statements using `if` and `else`. (Ironically, we then instruct those same machines to make the same decisions over and over. It's no wonder they sometimes appear mad…) For example, here is a snippet of Python that uses `for` and `if` to display the signs of the numbers in a list:

```
values = [-15, 0, 15]
for v in values:
    if v < 0:
        pos_neg = -1
    elif v == 0:
        pos_neg = 0
    else:
        pos_neg = 1
    print("The pos_neg of", v, "is", pos_neg)
```

```
The pos_neg of -15 is -1
The pos_neg of 0 is 0
The pos_neg of 15 is 1
```

```
print("The final value of v is", v)
```

```
The final value of v is 15
```

Its direct translation into R is:

```
values <- c(-15, 0, 15)
for (v in values) {
  if (v < 0) {
    pos_neg <- -1
  }
  else if (v == 0) {
    pos_neg <- 0
  }
  else {
```

```
    pos_neg <- 1
  }
  print(glue::glue("The sign of {v} is {pos_neg}"))
}
```

```
The sign of -15 is -1
The sign of 0 is 0
The sign of 15 is 1
```

```
print(glue::glue("The final value of v is {v}"))
```

```
The final value of v is 15
```

There are a few things to note here:

1. The parentheses in the loop header are required: we cannot simply write `for v in values`.
2. The curly braces around the body of the loop and around the bodies of the conditional branches are optional, since each contains only a single statement. However, they should always be there to help readability.
3. As in Python, the loop variable `v` persists after the loop is over.
4. `glue::glue` (the function `glue` from the library of the same name) interpolates variables into strings in sensible ways. We will load this library and use plain old `glue` in the explanations that follow. (Note that R uses `::` to get functions out of packages rather than Python's `.`.)
5. We have called our temporary variable `pos_neg` rather than `sign` so that we don't accidentally overwrite the rather useful built-in R function with the latter name. Name collisions of this sort are just as easy in R as they are in Python.

2.13 How can I vectorize loops and conditionals?

The example above is *not* how we should write R: everything in that snippet can and should be vectorized. The simplest way to do this is to use the aforementioned built-in function:

```
print(sign(values))
```

```
[1] -1  0  1
```

```
print(glue::glue("The sign of {values} is {sign(values)}"))
```

```
The sign of -15 is -1
The sign of 0 is 0
The sign of 15 is 1
```

But what if the function we want doesn't exist (or if we don't know what it's called)? In that case, the easiest approach is often to create a new vector whose values are derived from those of the vector we had and trust R to match up corresponding elements:

```
pos_neg <- dplyr::case_when(
  values <  0 ~ -1,
  values == 0 ~ 0,
  values >  0 ~ 1
)

print(glue::glue("The sign of {values} is {pos_neg}"))
```

```
The sign of -15 is -1
The sign of 0 is 0
The sign of 15 is 1
```

This solution makes use of `case_when`, which is a vectorized analog of `if/else if/else`. Each branch uses the `~` operator to combine a Boolean test on the left with a result on the right. We will see other uses for `~` in subsequent chapters.

2.14 How can I express a range of values?

`for` in R loops over the values in a vector, just as it does in Python. If we want to loop over the indices instead, we can use the function `seq_along`:

```
colors <- c("eburnean", "glaucous", "squamous", "wenge")
for (i in seq_along(colors)) {
  print(glue("The length of color {i} is {length(colors[i])}"))
}
```

```
The length of color 1 is 1
The length of color 2 is 1
The length of color 3 is 1
The length of color 4 is 1
```

This output makes no sense until we remember that every value is a vector, and that `length` returns the length of a vector, so that `length(colors[0])` is telling us that `colors[0]` contains one element. If we want the number of characters in the strings, we can use R's built-in `nchar` or the more modern function `stringr::str_length`:

```
for (i in seq_along(colors)) {
  print(glue("The length of color {i} is {stringr::str_length(colors[i])}"))
}
```

```
The length of color 1 is 8
The length of color 2 is 8
The length of color 3 is 8
```

```
The length of color 4 is 5
```

As you may already have guessed, `seq_along` returns a vector containing a sequence of integers:

```r
seq_along(colors)
```

```
[1] 1 2 3 4
```

Since sequences of this kind are used frequently, R lets us write them using range expressions:

```r
5:10
```

```
[1]  5  6  7  8  9 10
```

Their most common use is as indices to vectors:

```r
colors[2:3]
```

```
[1] "glaucous" "squamous"
```

We can similarly subtract a range of colors by index:

```r
colors[-1:-2]
```

```
[1] "squamous" "wenge"
```

However, R does not allow tripartite expressions of the form `start:end:step`. For that, we must use `seq`:

```r
seq(1, 10, 3)
```

```
[1]  1  4  7 10
```

This example also shows that ranges in R are inclusive at both ends, i.e., they run up to *and including* the upper bound. As is traditional among programming language advocates, people claim that this is more natural and then cite some supportive anecdote as if it were proof.

> **Repeating Things**
>
> The function `rep` repeats things, so `rep("a", 3)` is `c("a", "a", "a")`. If the second argument is a vector of the same length as the first, it specifies how many times each item in the first vector is to be repeated: `rep(c("a", "b"), c(2, 3))` is `c("a", "a", "b", "b", "b")`.

2.15 How can I use a vector in a conditional statement?

We cannot use a vector directly as a condition in an `if` statement:

```
numbers <- c(0, 1, 2)
if (numbers) {
  print("This should not work.")
}
```

```
Warning in if (numbers) {: the condition has length > 1 and only the first
element will be used
```

Instead, we must collapse the vector into a single logical value:

```
numbers <- c(0, 1, 2)
if (all(numbers >= 0)) {
  print("This, on the other hand, should work.")
}
```

```
[1] "This, on the other hand, should work."
```

The function `all` returns `TRUE` if every element in its argument is `TRUE`; it corresponds to a logical "and" of all its inputs. We can use a corresponding function `any` to check if at least one value is `TRUE`, which corresponds to a logical "or" across the whole input.

2.16 How do I create and call functions?

As we have already seen, we call functions in R much as we do in Python:

```
max(1, 3, 5) + min(1, 3, 5)
```

```
[1] 6
```

We define a new function using the `function` keyword. This creates the function; to name it, we must assign the newly-created function to a variable:

```
swap <- function(pair) {
  c(pair[2], pair[1])
}
swap(c("left", "right"))
```

```
[1] "right" "left"
```

As this example shows, the result of a function is the value of the last expression evaluated within it. A function can return a value earlier using the `return` function; we can use `return` for the final value as well, but most R programmers do not.

```
swap <- function(pair) {
  if (length(pair) != 2) {
    return(NULL) # This is very bad practice.
  }
  c(pair[2], pair[1])
```

```
}
swap(c("one"))
```

```
NULL
```

```
swap(c("left", "right"))
```

```
[1] "right" "left"
```

Returning `NULL` when our function's inputs are invalid as we have done above is foolhardy, as doing so means that `swap` can fail without telling us that it has done so. Consider:

```
NULL[1]                 # Try to access an element of the vector that does not exist.
```

```
NULL
```

```
values <- 5:10          # More than two values.
result <- swap(values)  # Attempting to swap the values produces NULL.
result[1]               # But we can operate on the result without error.
```

```
NULL
```

We will look at what we should do instead in Chapter 8.

2.17 How can I write a function that takes variable arguments?

If the number of arguments given to a function is not the number expected, R complains:

```
swap("one", "two", "three")
```

```
Error in swap("one", "two", "three"): unused arguments ("two", "three")
```

(Note that we are passing three separate values here, not a single vector containing three values.) If we want a function to handle a varying number of arguments, we represent the "extra" arguments with an ellipsis `...` (three dots), which serves the same purpose as Python's `*args`:

```
print_with_title <- function(title, ...) {
  print(glue("=={title}=="), paste(..., sep = "\n"))
}

print_with_title("to-do", "Monday", "Tuesday", "Wednesday")
```

```
==to-do==
Monday
Tuesday
```

```
Wednesday
```

> The function `paste` creates a string by combining its arguments with the specified separator.

R uses a special data structure to represent the extra arguments in `...`. If we want to work with those arguments one by one, we must explicitly convert `...` to a list:

```
add <- function(...) {
  result <- 0
  for (value in list(...)) {
    result <- result + value
  }
  result
}
add(1, 3, 5, 7)
```

```
[1] 16
```

2.18 How can I provide default values for arguments?

Like Python and most other modern programming languages, R lets us define default values for arguments and then pass arguments by name:

```
example <- function(first, second = "second", third = "third") {
  print(glue("first='{first}' second='{second}' third='{third}'"))
}

example("with just first")
```

```
first='with just first' second='second' third='third'
```

```
example("with first and second by position", "positional")
```

```
first='with first and second by position' second='positional' third='third'
```

```
example("with first and third by name", third = "by name")
```

```
first='with first and third by name' second='second' third='by name'
```

One caution: when you use a name in a function call, R ignores things that *aren't* functions when looking up the function. This means that the call to `orange()` in the code below produces 110 rather than an error because `purple(purple)` is interpreted as "pass the value 10 into the globally-defined function `purple`" rather than "try to call a function `10(10)`":

```r
purple <- function(x) x + 100
orange <- function() {
  purple <- 10
  purple(purple)
}
orange()
```

```
[1] 110
```

2.19 How can I hide the value that R returns?

If the value returned by a function isn't assigned to something, R displays it. Since this usually isn't what we want in library functions, we can use the function `invisible` to mark a value as "not to be printed" (though the value can still be assigned). For example, we can convert:

```r
something <- function(value) {
  10 * value
}
something(2)
```

```
[1] 20
```

to this:

```r
something <- function(value) {
  invisible(10 * value)
}
something(2)
```

The calculation is still being done, but the output is suppressed.

2.20 How can I assign to a global variable from inside a function?

The assignment operator `<<-` means "assign to a variable outside the current scope". As the example below shows, this means that what looks like creation of a new local variable can actually be modification of a global one:

```r
var <- "original value"

demonstrate <- function() {
  var <<- "new value"
}
```

```
demonstrate()
var
```

```
[1] "new value"
```

This should only and always be done with care: modern R strongly encourages a functional style of programming in which functions do not modify their input data, and *nobody* thinks that modifying global variables is a good idea any more.

2.21 Key Points

- Use `print(expression)` to print the value of a single expression.
- Variable names may include letters, digits, `.`, and `_`, but `.` should be avoided, as it sometimes has special meaning.
- R's atomic data types include logical, integer, double (also called numeric), and character.
- R stores collections in homogeneous vectors of atomic types, or in heterogeneous lists.
- 'Scalars' in R are actually vectors of length 1.
- Vectors and lists are created using the function `c(...)`.
- Vector indices from 1 to length(vector) select single elements.
- Negative indices to vectors deselect elements from the result.
- The index 0 on its own selects no elements, creating a vector or list of length 0.
- The expression `low:high` creates the vector of integers from `low` to `high` inclusive.
- Subscripting a vector with a vector of numbers selects the elements at those locations (possibly with repeats).
- Subscripting a vector with a vector of logicals selects elements where the indexing vector is `TRUE`.
- Values from short vectors (such as 'scalars') are repeated to match the lengths of longer vectors.
- The special value `NA` represents missing values, and (almost all) operations involving `NA` produce `NA`.
- The special values `NULL` represents a nonexistent vector, which is not the same as a vector of length 0.
- A list is a heterogeneous vector capable of storing values of any type (including other lists).
- Indexing with `[` returns a structure of the same type as the structure being indexed (e.g., returns a list when applied to a list).
- Indexing with `[[` strips away one level of structure (i.e., returns the indicated element without any wrapping).
- Use `list('name' = value, ...)` to name the elements of a list.
- Use either `L['name']` or `L$name` to access elements by name.
- Use back-quotes around the name with `$` notation if the name is not a

legal R variable name.

- Use `matrix(values, nrow = N)` to create a matrix with `N` rows containing the given values.
- Use `m[i, j]` to get the value at the i'th row and j'th column of a matrix.
- Use `m[i,]` to get a vector containing the values in the i'th row of a matrix.
- Use `m[,j]` to get a vector containing the values in the j'th column of a matrix.
- Use `for (loop_variable in collection){ ...body... }` to create a loop.
- Use `if (expression) { ...body... } else if (expression) { ...body... } else { ...body... }` to create conditionals.
- Expression conditions must have length 1; use `any(...)` and `all(...)` to collapse logical vectors to single values.
- Use `function(...arguments...) { ...body... }` to create a function.
- Use variable <- function(…arguments…) { …body… }‘ to create a function and give it a name.
- The body of a function can be a single expression or a block in curly braces.
- The last expression evaluated in a function is returned as its result.
- Use `return(expression)` to return a result early from a function.

Chapter 3

The Tidyverse

There is no point in becoming fluent in Enochian if you do not then call forth a Dweller Beneath at the time of the new moon. Similarly, there is no point learning a language designed for data manipulation if you do not then bend data to your will. This chapter therefore looks at how to do the things that R was summoned—er, designed—to do.

3.1 Learning Objectives

- Install and load packages in R.
- Read CSV data with R.
- Explain what a tibble is and how tibbles related to data frames and matrices.
- Describe how `read_csv` infers data types for columns in tabular datasets.
- Name and use three functions for inspects tibbles.
- Select subsets of tabular data using column names, scalar indices, ranges, and logical expressions.
- Explain the difference between indexing with `[` and with `[[`.
- Name and use four functions for calculating aggregate statistics on tabular data.
- Explain how these functions treat `NA` by default, and how to change that behavior.

3.2 How do I read data?

We begin by looking at the file `results/infant_hiv.csv`, a tidied version of data on the percentage of infants born to women with HIV who received an HIV test themselves within two months of birth. The original data comes from the UNICEF site at https://data.unicef.org/resources/dataset/hiv-aids-statistical-tables/, and this file contains:

```
country,year,estimate,hi,lo
AFG,2009,NA,NA,NA
AFG,2010,NA,NA,NA
...
AFG,2017,NA,NA,NA
AGO,2009,NA,NA,NA
AGO,2010,0.03,0.04,0.02
AGO,2011,0.05,0.07,0.04
AGO,2012,0.06,0.08,0.05
...
ZWE,2016,0.71,0.88,0.62
ZWE,2017,0.65,0.81,0.57
```

The actual file has many more rows (and no ellipses). It uses `NA` to show missing data rather than (for example) `-`, a space, or a blank, and its values are interpreted as follows:

Header	Datatype	Description
country	char	ISO3 country code of country reporting data
year	integer	year CE for which data reported
estimate	double/NA	estimated percentage of measurement
hi	double/NA	high end of range
lo	double/NA	low end of range

We can load this data in Python like this:

```
import pandas as pd

infant_hiv = pd.read_csv('results/infant_hiv.csv')
print(infant_hiv)
```

```
     country  year  estimate    hi    lo
0        AFG  2009       NaN   NaN   NaN
1        AFG  2010       NaN   NaN   NaN
2        AFG  2011       NaN   NaN   NaN
3        AFG  2012       NaN   NaN   NaN
4        AFG  2013       NaN   NaN   NaN
5        AFG  2014       NaN   NaN   NaN
6        AFG  2015       NaN   NaN   NaN
7        AFG  2016       NaN   NaN   NaN
8        AFG  2017       NaN   NaN   NaN
...
```

The equivalent in R is to load the tidyverse collection of packages and then call the `read_csv` function. We will go through this in stages, since each produces output.

```
library(tidyverse)
```

```
Error in library(tidyverse) : there is no package called 'tidyverse'
```

Ah. We must install the tidyverse (but only need to do so once per machine):

```
install.packages("tidyverse")
```

At any time, we can call `sessionInfo` to find out what versions of which packages we have loaded, along with the version of R we're using and some other useful information:

```
sessionInfo()
```

```
R version 3.6.0 (2019-04-26)
Platform: x86_64-apple-darwin15.6.0 (64-bit)
Running under: macOS High Sierra 10.13.6

Matrix products: default
BLAS:   /Library/Frameworks/R.framework/Versions/3.6/Resources/lib/libRblas.0.dylib
LAPACK: /Library/Frameworks/R.framework/Versions/3.6/Resources/lib/libRlapack.dylib

locale:
[1] en_CA.UTF-8/en_CA.UTF-8/en_CA.UTF-8/C/en_CA.UTF-8/en_CA.UTF-8

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
 [1] kableExtra_1.1.0 here_0.1         glue_1.3.1       knitr_1.24
 [5] rlang_0.4.0      reticulate_1.12  forcats_0.4.0    stringr_1.4.0
 [9] dplyr_0.8.3      purrr_0.3.2      readr_1.3.1      tidyr_1.0.0
[13] tibble_2.1.3     ggplot2_3.2.1    tidyverse_1.2.1

loaded via a namespace (and not attached):
 [1] tidyselect_0.2.5  xfun_0.8          haven_2.1.1
 [4] lattice_0.20-38   colorspace_1.4-1  vctrs_0.2.0
 [7] generics_0.0.2    viridisLite_0.3.0 htmltools_0.3.6
[10] yaml_2.2.0        pillar_1.4.2      withr_2.1.2
[13] modelr_0.1.5      readxl_1.3.1      lifecycle_0.1.0
[16] munsell_0.5.0     gtable_0.3.0      cellranger_1.1.0
[19] rvest_0.3.4       evaluate_0.14     broom_0.5.2
[22] Rcpp_1.0.2        scales_1.0.0      backports_1.1.4
[25] webshot_0.5.1     jsonlite_1.6      hms_0.5.0
[28] digest_0.6.20     stringi_1.4.3     bookdown_0.11
```

```
[31] grid_3.6.0        rprojroot_1.3-2   cli_1.1.0
[34] tools_3.6.0       magrittr_1.5      lazyeval_0.2.2
[37] crayon_1.3.4      pkgconfig_2.0.2   zeallot_0.1.0
[40] Matrix_1.2-17     xml2_1.2.0        lubridate_1.7.4
[43] assertthat_0.2.1  rmarkdown_1.14    httr_1.4.1
[46] rstudioapi_0.10   R6_2.4.0          nlme_3.1-140
[49] compiler_3.6.0
```

We then load the library once per program:

```
library(tidyverse)
```

Note that we give `install.packages` a string to install, but simply give the name of the package we want to load to `library`.

Loading the tidyverse gives us eight packages. One of those, dplyr, defines two functions that mask standard functions in R with the same names. If we need the originals, we can always get them with their fully-qualified names `stats::filter` and `stats::lag`.

Once we have the tidyverse loaded, reading the file looks remarkably like reading the file:

```
infant_hiv <- read_csv('results/infant_hiv.csv')
```

```
Parsed with column specification:
cols(
  country = col_character(),
  year = col_double(),
  estimate = col_double(),
  hi = col_double(),
  lo = col_double()
)
```

R's `read_csv` tells us more about what it has done than Pandas does. In particular, it guesses the data types of columns based on the first thousand values and then tells us what types it has inferred. (In a better universe, people would habitually use the first *two* rows of their spreadsheets for name *and units*, but we do not live there.)

We can now look at what `read_csv` has produced.

```
infant_hiv
```

```
# A tibble: 1,728 x 5
   country  year estimate    hi    lo
   <chr>   <dbl>    <dbl> <dbl> <dbl>
 1 AFG      2009       NA    NA    NA
 2 AFG      2010       NA    NA    NA
 3 AFG      2011       NA    NA    NA
```

```
 4 AFG      2012       NA    NA    NA
 5 AFG      2013       NA    NA    NA
 6 AFG      2014       NA    NA    NA
 7 AFG      2015       NA    NA    NA
 8 AFG      2016       NA    NA    NA
 9 AFG      2017       NA    NA    NA
10 AGO      2009       NA    NA    NA
# ... with 1,718 more rows
```

This is a tibble, which is the tidyverse's enhanced version of R's `data.frame`. It organizes data into named columns, each having one value for each row. In statistical terms, the columns are the variables being observed and the rows are the actual observations.

3.3 How do I inspect data?

We often have a quick look at the content of a table to remind ourselves what it contains. Pandas does this using methods whose names are borrowed from the Unix shell's `head` and `tail` commands:

```
print(infant_hiv.head())
```

```
  country  year  estimate  hi  lo
0     AFG  2009       NaN NaN NaN
1     AFG  2010       NaN NaN NaN
2     AFG  2011       NaN NaN NaN
3     AFG  2012       NaN NaN NaN
4     AFG  2013       NaN NaN NaN
```

```
print(infant_hiv.tail())
```

```
     country  year  estimate    hi    lo
1723     ZWE  2013      0.57  0.70  0.49
1724     ZWE  2014      0.54  0.67  0.47
1725     ZWE  2015      0.59  0.73  0.51
1726     ZWE  2016      0.71  0.88  0.62
1727     ZWE  2017      0.65  0.81  0.57
```

R has similarly-named functions:

```
head(infant_hiv)
```

```
# A tibble: 6 x 5
  country  year estimate    hi    lo
  <chr>   <dbl>    <dbl> <dbl> <dbl>
1 AFG      2009       NA    NA    NA
2 AFG      2010       NA    NA    NA
3 AFG      2011       NA    NA    NA
```

```
4 AFG      2012       NA    NA    NA
5 AFG      2013       NA    NA    NA
6 AFG      2014       NA    NA    NA
```

```
tail(infant_hiv)
```

```
# A tibble: 6 x 5
  country  year estimate    hi    lo
  <chr>   <dbl>    <dbl> <dbl> <dbl>
1 ZWE      2012    0.38   0.47 0.33
2 ZWE      2013    0.570  0.7  0.49
3 ZWE      2014    0.54   0.67 0.47
4 ZWE      2015    0.59   0.73 0.51
5 ZWE      2016    0.71   0.88 0.62
6 ZWE      2017    0.65   0.81 0.570
```

Let's have a closer look at that last command's output:

```
tail(infant_hiv)
```

```
# A tibble: 6 x 5
  country  year estimate    hi    lo
  <chr>   <dbl>    <dbl> <dbl> <dbl>
1 ZWE      2012    0.38   0.47 0.33
2 ZWE      2013    0.570  0.7  0.49
3 ZWE      2014    0.54   0.67 0.47
4 ZWE      2015    0.59   0.73 0.51
5 ZWE      2016    0.71   0.88 0.62
6 ZWE      2017    0.65   0.81 0.570
```

Note that the row numbers printed by `tail` are relative to the output, not absolute to the table. This is different from Pandas, which retains the original row numbers.

What about overall information?

```
print(infant_hiv.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1728 entries, 0 to 1727
Data columns (total 5 columns):
country     1728 non-null object
year        1728 non-null int64
estimate    728 non-null float64
hi          728 non-null float64
lo          728 non-null float64
dtypes: float64(3), int64(1), object(1)
memory usage: 67.6+ KB
None
```

```
summary(infant_hiv)
```

```
   country              year         estimate           hi
 Length:1728        Min.   :2009   Min.   :0.000   Min.   :0.0000
 Class :character   1st Qu.:2011   1st Qu.:0.100   1st Qu.:0.1400
 Mode  :character   Median :2013   Median :0.340   Median :0.4350
                    Mean   :2013   Mean   :0.387   Mean   :0.4614
                    3rd Qu.:2015   3rd Qu.:0.620   3rd Qu.:0.7625
                    Max.   :2017   Max.   :0.950   Max.   :0.9500
                                   NA's   :1000    NA's   :1000
       lo
 Min.   :0.0000
 1st Qu.:0.0800
 Median :0.2600
 Mean   :0.3221
 3rd Qu.:0.5100
 Max.   :0.9500
 NA's   :1000
```

Your display of R's summary may or may not wrap, depending on how large a screen the older acolytes have allowed you.

3.4 How do I index rows and columns?

A Pandas DataFrame is a collection of series (also called columns), each containing the values of a single observed variable:

```
print(infant_hiv['estimate'])
```

```
0        NaN
1        NaN
2        NaN
3        NaN
4        NaN
5        NaN
6        NaN
7        NaN
8        NaN
9        NaN
10      0.03
11      0.05
12      0.06
13      0.15
14      0.10
15      0.06
16      0.01
```

```
17      0.01
18       NaN
19       NaN
...
```

We would get exactly the same output in Python with `infant_hiv.estimate`, i.e., with an attribute name rather than a string subscript. The same tricks work in R:

```
infant_hiv['estimate']
```

```
# A tibble: 1,728 x 1
   estimate
      <dbl>
 1       NA
 2       NA
 3       NA
 4       NA
 5       NA
 6       NA
 7       NA
 8       NA
 9       NA
10       NA
# ... with 1,718 more rows
```

However, R's `infant_hiv$estimate` provides all the data:

```
infant_hiv$estimate
```

```
   [1]   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA 0.03 0.05 0.06
  [14] 0.15 0.10 0.06 0.01 0.01   NA   NA   NA   NA   NA   NA   NA   NA
  [27]   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA
  [40]   NA   NA   NA   NA   NA   NA   NA   NA 0.13 0.12 0.12 0.52 0.53
  [53] 0.67 0.66   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA
  [66]   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA
  [79]   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA 0.26
  [92] 0.24 0.38 0.55 0.61 0.74 0.83 0.75 0.74   NA 0.10 0.10 0.11 0.18
 [105] 0.12 0.02 0.12 0.20   NA   NA   NA   NA   NA   NA   NA   NA   NA
 [118]   NA   NA 0.10 0.09 0.12 0.26 0.27 0.25 0.32 0.03 0.09 0.13 0.19
 [131] 0.25 0.30 0.28 0.15 0.16   NA 0.02 0.02 0.02 0.03 0.15 0.10 0.17
 [144] 0.14   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA
 [157]   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA
 [170]   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA 0.95 0.95
 [183] 0.95 0.95 0.95 0.95 0.80 0.95 0.87 0.77 0.75 0.72 0.51 0.55 0.50
 [196] 0.62 0.37 0.36 0.07 0.46 0.46 0.46 0.46 0.44 0.43 0.42 0.40 0.25
 [209] 0.25 0.46 0.25 0.45 0.45 0.46 0.46 0.45   NA   NA   NA   NA   NA
 [222]   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA
```

```
 [235]   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA 0.53 0.35 0.36
 [248] 0.48 0.41 0.45 0.47 0.50 0.01 0.01 0.07 0.05 0.03 0.09 0.12 0.21
...
```

Again, note that the boxed number on the left is the start index of that row.

What about single values? Remembering to count from zero from Python and as humans do for R, we have:

```
print(infant_hiv.estimate[11])
```

```
0.05
```

```
infant_hiv$estimate[12]
```

```
[1] 0.05
```

Ah—everything in R is a vector, so we get a vector of one value as an output rather than a single value.

```
print(len(infant_hiv.estimate[11]))
```

```
Error in py_call_impl(callable, dots$args, dots$keywords): TypeError: object of type 'numpy.float

Detailed traceback:
  File "<string>", line 1, in <module>
```

```
length(infant_hiv$estimate[12])
```

```
[1] 1
```

And yes, ranges work:

```
print(infant_hiv.estimate[5:15])
```

```
5      NaN
6      NaN
7      NaN
8      NaN
9      NaN
10    0.03
11    0.05
12    0.06
13    0.15
14    0.10
Name: estimate, dtype: float64
```

```
infant_hiv$estimate[6:15]
```

```
 [1]   NA   NA   NA   NA   NA 0.03 0.05 0.06 0.15 0.10
```

Note that the upper bound is the same, because it's inclusive in R and exclusive in Python. Note also that nothing prevents us from selecting a range of rows that spans several countries, which is why selecting by row number is usually a sign of innocence, insouciance, or desperation.

We can select by column number as well. Pandas uses the rather clumsy `object.iloc[rows, columns]` with the usual shortcut `:` for "entire range":

```
print(infant_hiv.iloc[:, 0])
```

```
0       AFG
1       AFG
2       AFG
3       AFG
4       AFG
5       AFG
6       AFG
7       AFG
8       AFG
9       AGO
10      AGO
11      AGO
12      AGO
13      AGO
14      AGO
15      AGO
16      AGO
17      AGO
18      AIA
19      AIA
...
```

Since this is a column, it can be indexed:

```
print(infant_hiv.iloc[:, 0][0])
```

```
AFG
```

In R, a single index is interpreted as the column index:

```
infant_hiv[1]
```

```
# A tibble: 1,728 x 1
   country
   <chr>
 1 AFG
 2 AFG
 3 AFG
 4 AFG
 5 AFG
 6 AFG
```

```
 7 AFG
 8 AFG
 9 AFG
10 AGO
# ... with 1,718 more rows
```

But notice that the output is not a vector, but another tibble (i.e., a table with N rows and one column). This means that adding another index does column-wise indexing on that tibble:

```
infant_hiv[1][1]
```

```
# A tibble: 1,728 x 1
   country
   <chr>
 1 AFG
 2 AFG
 3 AFG
 4 AFG
 5 AFG
 6 AFG
 7 AFG
 8 AFG
 9 AFG
10 AGO
# ... with 1,718 more rows
```

How then are we to get the first mention of Afghanistan? The answer is to use double square brackets to strip away one level of structure:

```
infant_hiv[[1]]
```

```
   [1] "AFG" "AFG" "AFG" "AFG" "AFG" "AFG" "AFG" "AFG" "AFG" "AGO" "AGO"
  [12] "AGO" "AGO" "AGO" "AGO" "AGO" "AGO" "AGO" "AIA" "AIA" "AIA" "AIA"
  [23] "AIA" "AIA" "AIA" "AIA" "AIA" "ALB" "ALB" "ALB" "ALB" "ALB" "ALB"
  [34] "ALB" "ALB" "ALB" "ARE" "ARE" "ARE" "ARE" "ARE" "ARE" "ARE" "ARE"
  [45] "ARE" "ARG" "ARG" "ARG" "ARG" "ARG" "ARG" "ARG" "ARG" "ARG" "ARM"
  [56] "ARM" "ARM" "ARM" "ARM" "ARM" "ARM" "ARM" "ARM" "ATG" "ATG" "ATG"
  [67] "ATG" "ATG" "ATG" "ATG" "ATG" "ATG" "AUS" "AUS" "AUS" "AUS" "AUS"
  [78] "AUS" "AUS" "AUS" "AUS" "AUT" "AUT" "AUT" "AUT" "AUT" "AUT" "AUT"
  [89] "AUT" "AUT" "AZE" "AZE" "AZE" "AZE" "AZE" "AZE" "AZE" "AZE" "AZE"
 [100] "BDI" "BDI" "BDI" "BDI" "BDI" "BDI" "BDI" "BDI" "BDI" "BEL" "BEL"
 [111] "BEL" "BEL" "BEL" "BEL" "BEL" "BEL" "BEL" "BEN" "BEN" "BEN" "BEN"
 [122] "BEN" "BEN" "BEN" "BEN" "BEN" "BFA" "BFA" "BFA" "BFA" "BFA" "BFA"
 [133] "BFA" "BFA" "BFA" "BGD" "BGD" "BGD" "BGD" "BGD" "BGD" "BGD" "BGD"
 [144] "BGD" "BGR" "BGR" "BGR" "BGR" "BGR" "BGR" "BGR" "BGR" "BGR" "BHR"
 [155] "BHR" "BHR" "BHR" "BHR" "BHR" "BHR" "BHR" "BHR" "BHS" "BHS" "BHS"
 [166] "BHS" "BHS" "BHS" "BHS" "BHS" "BHS" "BIH" "BIH" "BIH" "BIH" "BIH"
```

```
 [177] "BIH" "BIH" "BIH" "BIH" "BLR" "BLR" "BLR" "BLR" "BLR" "BLR" "BLR"
 [188] "BLR" "BLR" "BLZ" "BLZ" "BLZ" "BLZ" "BLZ" "BLZ" "BLZ" "BLZ" "BLZ"
 [199] "BOL" "BOL" "BOL" "BOL" "BOL" "BOL" "BOL" "BOL" "BOL" "BRA" "BRA"
 [210] "BRA" "BRA" "BRA" "BRA" "BRA" "BRA" "BRA" "BRB" "BRB" "BRB" "BRB"
...
```

This is now a plain old vector, so it can be indexed with single square brackets:

```
infant_hiv[[1]][1]
```

```
[1] "AFG"
```

But that too is a vector, so it can of course be indexed as well (for some value of "of course"):

```
infant_hiv[[1]][1][1]
```

```
[1] "AFG"
```

Thus, `data[1][[1]]` produces a tibble, then selects the first column vector from it, so it still gives us a vector. *This is not madness.* It is merely…differently sane.

> **Subsetting Data Frames**
>
> When we are working with data frames (including tibbles), subsetting with a single vector selects columns, not rows, because data frames are stored as lists of columns. This means that `df[1:2]` selects two columns from `df`. However, in `df[2:3, 1:2]`, the first index selects rows, while the second selects columns.

3.5 How do I calculate basic statistics?

What is the average estimate? We start by grabbing that column for convenience:

```
estimates = infant_hiv.estimate
print(len(estimates))
```

```
1728
```

```
print(estimates.mean())
```

```
0.3870192307692308
```

This translates almost directly to R:

```
estimates <- infant_hiv$estimate
length(estimates)
```

```
[1] 1728
```

```
mean(estimates)
```

```
[1] NA
```

The void is always there, waiting for us… Let's fix this in R first by telling `mean` to drop NAs:

```
mean(estimates, na.rm = TRUE)
```

```
[1] 0.3870192
```

and then try to get the statistically correct behavior in Pandas:

```
print(estimates.mean(skipna=False))
```

```
nan
```

Many functions in R use `na.rm` to control whether `NA`s are removed or not. (Remember, the `.` character is just another part of the name) R's default behavior is to leave `NA`s in, and then to include them in aggregate computations. Python's is to get rid of missing values early and work with what's left, which makes translating code from one language to the next much more interesting than it might otherwise be. But other than that, the statistics works the same way. In Python, we write:

```
print("min", estimates.min())
```

```
min 0.0
```

```
print("max", estimates.max())
```

```
max 0.95
```

```
print("std", estimates.std())
```

```
std 0.3034511074214113
```

and in R:

```
print(glue("min {min(estimates, na.rm = TRUE)}"))
```

```
min 0
```

```
print(glue("max {max(estimates, na.rm = TRUE)}"))
```

```
max 0.95
```

```
print(glue("sd {sd(estimates, na.rm = TRUE)}"))
```

```
sd 0.303451107421411
```

A good use of aggregation is to check the quality of the data. For example, we can ask if there are any records where some of the estimate, the low value, or the high value are missing, but not all of them:

```
print((infant_hiv.hi.isnull() != infant_hiv.lo.isnull()).any())
```

```
False
```

```
any(is.na(infant_hiv$hi) != is.na(infant_hiv$lo))
```

```
[1] FALSE
```

3.6 How do I filter data?

By "filtering", we mean "selecting records by value". As discussed in Chapter 2, the simplest approach is to use a vector of logical values to keep only the values corresponding to `TRUE`. In Python, this is:

```
maximal = estimates[estimates >= 0.95]
print(len(maximal))
```

```
52
```

And in R:

```
maximal <- estimates[estimates >= 0.95]
length(maximal)
```

```
[1] 1052
```

The difference is unexpected. Let's have a closer look at the result in Python:

```
print(maximal)
```

```
180     0.95
181     0.95
182     0.95
183     0.95
184     0.95
185     0.95
187     0.95
360     0.95
361     0.95
362     0.95
379     0.95
380     0.95
381     0.95
382     0.95
384     0.95
385     0.95
386     0.95
446     0.95
447     0.95
```

```
461      0.95
...
```

And in R:

```r
maximal
```

```
  [1]   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA
 [14]   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA
 [27]   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA
 [40]   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA
 [53]   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA
 [66]   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA
 [79]   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA
 [92]   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA
[105]   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA
[118]   NA   NA   NA   NA   NA   NA   NA 0.95 0.95 0.95 0.95 0.95 0.95
[131] 0.95   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA
[144]   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA
[157]   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA
[170]   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA
[183]   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA
[196]   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA
[209] 0.95 0.95 0.95 0.95 0.95 0.95 0.95 0.95 0.95 0.95   NA   NA   NA
[222]   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA
[235]   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA
[248]   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA
...
```

It appears that R has kept the unknown values in order to highlight just how little we know. More precisely, wherever there was an `NA` in the original data there is an `NA` in the logical vector and hence an `NA` in the final vector. Let us then turn to `which` to get a vector of indices at which a vector contains `TRUE`. This function does not return indices for `FALSE` or `NA`:

```r
which(estimates >= 0.95)
```

```
 [1]  181  182  183  184  185  186  188  361  362  363  380  381  382  383
[15]  385  386  387  447  448  462  793  794  795  796  797  798  911  912
[29]  955  956  957  958  959  960  961  962  963 1098 1107 1128 1429 1430
[43] 1462 1554 1604 1607 1625 1626 1627 1629 1708 1710
```

And as a quick check:

```r
length(which(estimates >= 0.95))
```

```
[1] 52
```

So now we can index our vector with the result of the `which`:

```r
maximal <- estimates[which(estimates >= 0.95)]
maximal
```

```
 [1] 0.95 0.95 0.95 0.95 0.95 0.95 0.95 0.95 0.95 0.95 0.95 0.95 0.95 0.95
[15] 0.95 0.95 0.95 0.95 0.95 0.95 0.95 0.95 0.95 0.95 0.95 0.95 0.95 0.95
[29] 0.95 0.95 0.95 0.95 0.95 0.95 0.95 0.95 0.95 0.95 0.95 0.95 0.95 0.95
[43] 0.95 0.95 0.95 0.95 0.95 0.95 0.95 0.95 0.95 0.95
```

But should we do this? Those `NA`s are important information, and should not be discarded so blithely. What we should *really* be doing is using the tools the tidyverse provides rather than clever indexing tricks. These behave consistently across a wide scale of problems and encourage use of patterns that make it easier for others to understand our programs.

3.7 How do I write tidy code?

The six basic data transformation operations in the tidyverse are:

- `filter`: choose observations (rows) by value(s)
- `arrange`: reorder rows
- `select`: choose variables (columns) by name
- `mutate`: derive new variables from existing ones
- `group_by`: define subsets of rows for further processing
- `summarize`: combine many values to create a single new value

`filter(tibble, ...criteria...)` keeps rows that pass all of the specified criteria:

```r
filter(infant_hiv, lo > 0.5)
```

```
# A tibble: 183 x 5
   country  year estimate    hi    lo
   <chr>   <dbl>    <dbl> <dbl> <dbl>
 1 ARG      2016     0.67  0.77  0.61
 2 ARG      2017     0.66  0.77  0.6
 3 AZE      2014     0.74  0.95  0.53
 4 AZE      2015     0.83  0.95  0.64
 5 AZE      2016     0.75  0.95  0.56
 6 AZE      2017     0.74  0.95  0.56
 7 BLR      2009     0.95  0.95  0.95
 8 BLR      2010     0.95  0.95  0.95
 9 BLR      2011     0.95  0.95  0.91
10 BLR      2012     0.95  0.95  0.95
# ... with 173 more rows
```

Notice that the expression is `lo > 0.5` rather than `"lo" > 0.5`. The latter expression would return the entire table because the string `"lo"` is greater than the number 0.5 everywhere.

But how is it that the name `lo` can be used on its own? It is the name of a column, but there is no variable called `lo`. The answer is that R uses lazy evaluation: function arguments aren't evaluated until they're needed, so the function `filter` actually gets the expression `lo > 0.5`, which allows it to check that there's a column called `lo` and then use it appropriately. It may seem strange at first, but it is much tidier than `filter(data, data$lo > 0.5)` or `filter(data, "lo > 0.5")`. We will explore lazy evaluation further in Chapter 6.

We can make data anlaysis code more readable by using the pipe operator `%>%`:

```
infant_hiv %>% filter(lo > 0.5)
```

```
# A tibble: 183 x 5
   country  year estimate    hi    lo
   <chr>   <dbl>    <dbl> <dbl> <dbl>
 1 ARG      2016     0.67  0.77  0.61
 2 ARG      2017     0.66  0.77  0.6
 3 AZE      2014     0.74  0.95  0.53
 4 AZE      2015     0.83  0.95  0.64
 5 AZE      2016     0.75  0.95  0.56
 6 AZE      2017     0.74  0.95  0.56
 7 BLR      2009     0.95  0.95  0.95
 8 BLR      2010     0.95  0.95  0.95
 9 BLR      2011     0.95  0.95  0.91
10 BLR      2012     0.95  0.95  0.95
# ... with 173 more rows
```

This may not seem like much of an improvement, but neither does a Unix pipe consisting of `cat filename.txt | head`. What about this?

```
filter(infant_hiv, (estimate != 0.95) & (lo > 0.5) & (hi <= (lo + 0.1)))
```

```
# A tibble: 1 x 5
  country  year estimate    hi    lo
  <chr>   <dbl>    <dbl> <dbl> <dbl>
1 TTO      2017     0.94  0.95  0.86
```

It uses the vectorized "and" operator `&` twice, and parsing the condition takes a human being at least a few seconds. Its pipelined equivalent is:

```
infant_hiv %>% filter(estimate != 0.95) %>% filter(lo > 0.5) %>% filter(hi <= (lo + 0.1))
```

```
# A tibble: 1 x 5
  country  year estimate    hi    lo
  <chr>   <dbl>    <dbl> <dbl> <dbl>
1 TTO      2017     0.94  0.95  0.86
```

Breaking the condition into stages like this often makes reading and testing much easier, and encourages incremental write-test-extend development. Let's increase the band from 10% to 20%, break the line the way the tidyverse style guide recommends to make the operations easier to spot, and order by `lo` in descending order:

```
infant_hiv %>%
  filter(estimate != 0.95) %>%
  filter(lo > 0.5) %>%
  filter(hi <= (lo + 0.2)) %>%
  arrange(desc(lo))
```

```
# A tibble: 55 x 5
   country  year estimate    hi    lo
   <chr>   <dbl>    <dbl> <dbl> <dbl>
 1 TTO      2017     0.94  0.95  0.86
 2 SWZ      2011     0.93  0.95  0.84
 3 CUB      2014     0.92  0.95  0.83
 4 TTO      2016     0.9   0.95  0.83
 5 CRI      2009     0.92  0.95  0.81
 6 CRI      2012     0.89  0.95  0.81
 7 NAM      2014     0.91  0.95  0.81
 8 URY      2016     0.9   0.95  0.81
 9 ZMB      2014     0.91  0.95  0.81
10 KAZ      2015     0.84  0.95  0.8
# ... with 45 more rows
```

We can now select the three columns we care about:

```
infant_hiv %>%
  filter(estimate != 0.95) %>%
  filter(lo > 0.5) %>%
  filter(hi <= (lo + 0.2)) %>%
  arrange(desc(lo)) %>%
  select(year, lo, hi)
```

```
# A tibble: 55 x 3
    year    lo    hi
   <dbl> <dbl> <dbl>
 1  2017  0.86  0.95
 2  2011  0.84  0.95
 3  2014  0.83  0.95
 4  2016  0.83  0.95
 5  2009  0.81  0.95
 6  2012  0.81  0.95
 7  2014  0.81  0.95
 8  2016  0.81  0.95
 9  2014  0.81  0.95
10  2015  0.8   0.95
```

```
# ... with 45 more rows
```

Once again, we are using the unquoted column names `year`, `lo`, and `hi` and letting R's lazy evaluation take care of the details for us.

Rather than selecting these three columns, we can select *out* the columns we're not interested in by negating their names. This leaves the columns that are kept in their original order, rather than putting `lo` before `hi`, which won't matter if we later select by name, but *will* if we ever want to select by position:

```
infant_hiv %>%
  filter(estimate != 0.95) %>%
  filter(lo > 0.5) %>%
  filter(hi <= (lo + 0.2)) %>%
  arrange(desc(lo)) %>%
  select(-country, -estimate)
```

```
# A tibble: 55 x 3
    year    hi    lo
   <dbl> <dbl> <dbl>
 1  2017  0.95  0.86
 2  2011  0.95  0.84
 3  2014  0.95  0.83
 4  2016  0.95  0.83
 5  2009  0.95  0.81
 6  2012  0.95  0.81
 7  2014  0.95  0.81
 8  2016  0.95  0.81
 9  2014  0.95  0.81
10  2015  0.95  0.8
# ... with 45 more rows
```

Giddy with power, we now add a column containing the difference between the low and high values. This can be done using either `mutate`, which adds new columns to the end of an existing tibble, or with `transmute`, which creates a new tibble containing only the columns we explicitly ask for. (There is also a function `rename` which simply renames columns.) Since we want to keep `hi` and `lo`, we decide to use `mutate`:

```
infant_hiv %>%
  filter(estimate != 0.95) %>%
  filter(lo > 0.5) %>%
  filter(hi <= (lo + 0.2)) %>%
  arrange(desc(lo)) %>%
  select(-country, -estimate) %>%
  mutate(difference = hi - lo)
```

```
# A tibble: 55 x 4
    year    hi    lo difference
```

```
   <dbl> <dbl> <dbl>      <dbl>
 1  2017  0.95  0.86     0.0900
 2  2011  0.95  0.84     0.110
 3  2014  0.95  0.83     0.12
 4  2016  0.95  0.83     0.12
 5  2009  0.95  0.81     0.140
 6  2012  0.95  0.81     0.140
 7  2014  0.95  0.81     0.140
 8  2016  0.95  0.81     0.140
 9  2014  0.95  0.81     0.140
10  2015  0.95  0.8      0.150
# ... with 45 more rows
```

Does the difference between high and low estimates vary by year? To answer that question, we use `group_by` to group records by value and then `summarize` to aggregate within groups. We might as well get rid of the `arrange` and `select` calls in our pipeline at this point, since we're not using them, and count how many records contributed to each aggregation using `n()`:

```
infant_hiv %>%
  filter(estimate != 0.95) %>%
  filter(lo > 0.5) %>%
  filter(hi <= (lo + 0.2)) %>%
  mutate(difference = hi - lo) %>%
  group_by(year) %>%
  summarize(count = n(), ave_diff = mean(year))
```

```
# A tibble: 9 x 3
   year count ave_diff
  <dbl> <int>    <dbl>
1  2009     3     2009
2  2010     3     2010
3  2011     5     2011
4  2012     5     2012
5  2013     6     2013
6  2014    10     2014
7  2015     6     2015
8  2016    10     2016
9  2017     7     2017
```

How might we do this with Pandas? One approach is to use a single multi-part `.query` to select data and store the result in a variable so that we can refer to the `hi` and `lo` columns twice without repeating the filtering expression. We then group by year and aggregate, again using strings for column names:

```
data = pd.read_csv('results/infant_hiv.csv')
data = data.query('(estimate != 0.95) & (lo > 0.5) & (hi <= (lo + 0.2))')
data = data.assign(difference = (data.hi - data.lo))
```

```
grouped = data.groupby('year').agg({'difference' : {'ave_diff' : 'mean', 'count' : 'count'}})
```

```
//anaconda3/lib/python3.7/site-packages/pandas/core/groupby/generic.py:1315: FutureWarning: using
  return super(DataFrameGroupBy, self).aggregate(arg, *args, **kwargs)
```

```
print(grouped)
```

```
     difference
       ave_diff count
year
2009   0.170000     3
2010   0.186667     3
2011   0.168000     5
2012   0.186000     5
2013   0.183333     6
2014   0.168000    10
2015   0.161667     6
2016   0.166000    10
2017   0.152857     7
```

There are other ways to tackle this problem with Pandas, but the tidyverse approach produces code that I find more readable.

3.8 How do I model my data?

Tidying up data can be as calming and rewarding in the same way as knitting or rearranging the specimen jars on the shelves in your dining room-stroke-laboratory. Eventually, though, people want to do some statistics. The simplest tool for this in R is `lm`, which stands for "linear model". Given a formula and a data set, it calculates coefficients to fit that formula to that data:

```
lm(estimate ~ lo, data = infant_hiv)
```

```

Call:
lm(formula = estimate ~ lo, data = infant_hiv)

Coefficients:
(Intercept)           lo
     0.0421       1.0707
```

This is telling us that `estimate` is more-or-less equal to `0.0421 + 1.0707 * lo`. The `~` symbol is used to separate the left and right sides of the equation, and as with all things tidyverse, lazy evaluation allows us to use variable names directly. In fact, it lets us write much more complex formulas involving functions of multiple variables. For example, we can regress `estimate` against the square roots of `lo` and `hi` (though there is no sound statistical reason to do so):

```r
lm(estimate ~ sqrt(lo) + sqrt(hi), data = infant_hiv)
```

```

Call:
lm(formula = estimate ~ sqrt(lo) + sqrt(hi), data = infant_hiv)

Coefficients:
(Intercept)     sqrt(lo)     sqrt(hi)
    -0.2225       0.6177       0.4814
```

One important thing to note here is the way that `+` is overloaded in formulas. The formula `estimate ~ lo + hi` does *not* mean "regress `estimate` against the sum of `lo` and `hi`", but rather, "regress `estimate` against the two variables `lo` and `hi`":

```r
lm(estimate ~ lo + hi, data = infant_hiv)
```

```

Call:
lm(formula = estimate ~ lo + hi, data = infant_hiv)

Coefficients:
(Intercept)           lo           hi
   -0.01327      0.42979      0.56752
```

If we want to regress `estimate` against the average of `lo` and `hi` (i.e., regress `estimate` against a single calculated variable instead of against two variables) we need to create a temporary column:

```r
infant_hiv %>%
  mutate(ave_lo_hi = (lo + hi)/2) %>%
  lm(estimate ~ ave_lo_hi, data = .)
```

```

Call:
lm(formula = estimate ~ ave_lo_hi, data = .)

Coefficients:
(Intercept)    ave_lo_hi
   -0.00897      1.01080
```

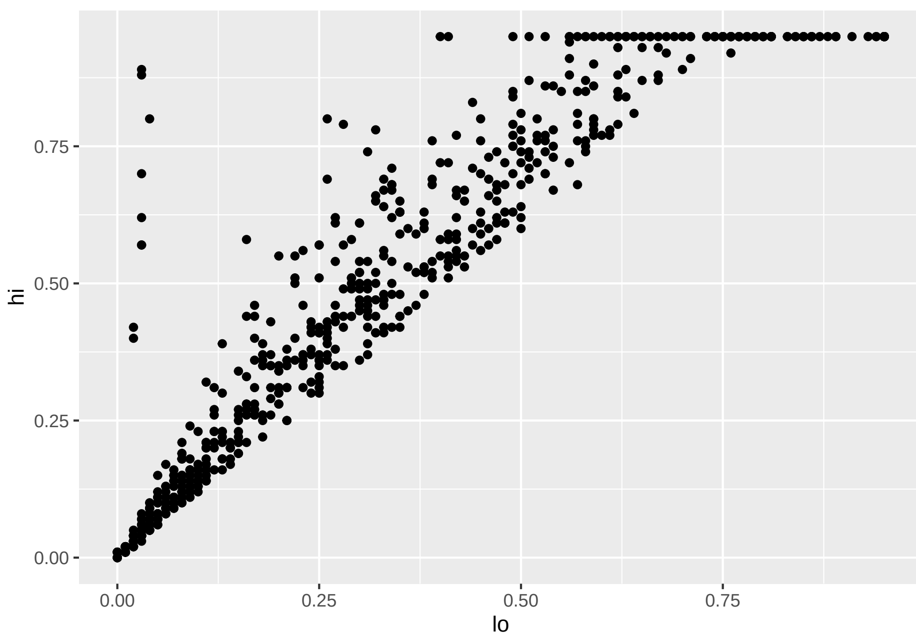
Here, the call to `lm` is using the variable `.` to mean "the data coming in from the previous stage of the pipeline". Most of the functions in the tidyverse use this convention so that data can be passed to a function that expects it in a position other than the first.

3.9 How do I create a plot?

Human being always want to see the previously unseen, though they are not always glad to have done so. The most popular tool for doing this in R is `ggplot2`, which implements and extends the patterns for creating charts described in Wilkinson (2005). Every chart it creates has a geometry that controls how data is displayed and a mapping that controls how values are represented geometrically. For example, these lines of code create a scatter plot showing the relationship between `lo` and `hi` values in the infant HIV data:

```
ggplot(infant_hiv) + geom_point(mapping = aes(x = lo, y = hi))
```

```
Warning: Removed 1000 rows containing missing values (geom_point).
```

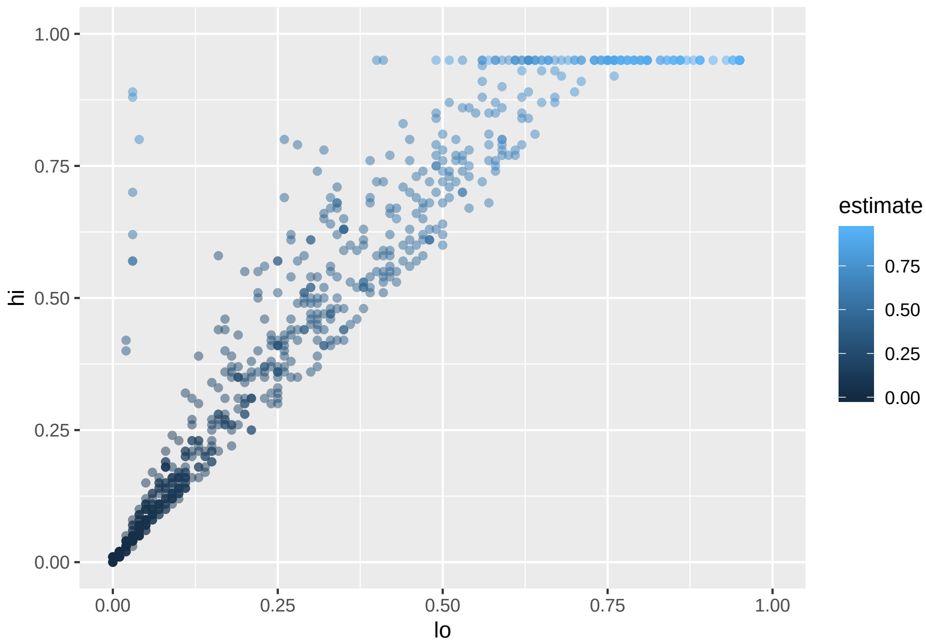


Looking more closely:

- The function `ggplot` creates an object to represent the chart with `infant_hiv` as the underlying data.
- `geom_point` specifies the geometry we want (points).
- Its `mapping` argument is assigned an aesthetic that specifies `lo` is to be used as the `x` coordinate and `hi` is to be used as the `y` coordinate.
- The elements of the chart are combined with `+` rather than `%>%` for historical reasons.

Let's create a slightly more appealing plot by dropping NAs, making the points semi-transparent, and colorizing them according to the value of `estimate`:

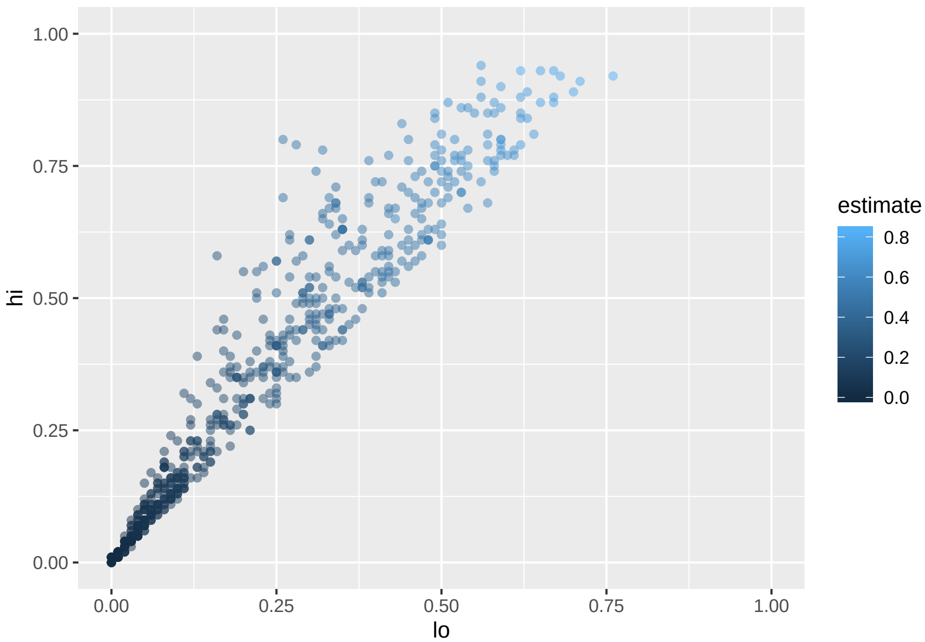
```
infant_hiv %>%
  drop_na() %>%
  ggplot(mapping = aes(x = lo, y = hi, color = estimate)) +
  geom_point(alpha = 0.5) +
  xlim(0.0, 1.0) + ylim(0.0, 1.0)
```



We set the transparency `alpha` outside the aesthetic because its value is constant for all points. If we set it inside `aes(...)`, we would be telling ggplot2 to set the transparency according to the value of the data. We specify the limits to the axes manually with `xlim` and `ylim` to ensure that ggplot2 includes the upper bounds: without this, all of the data would be shown, but the upper label "1.00" would be omitted.

This plot immediately shows us that we have some outliers. There are far more values with `hi` equal to 0.95 than it seems there ought to be, and there are eight points running up the left margin that seem troubling as well. Let's create a new tibble that doesn't have these:

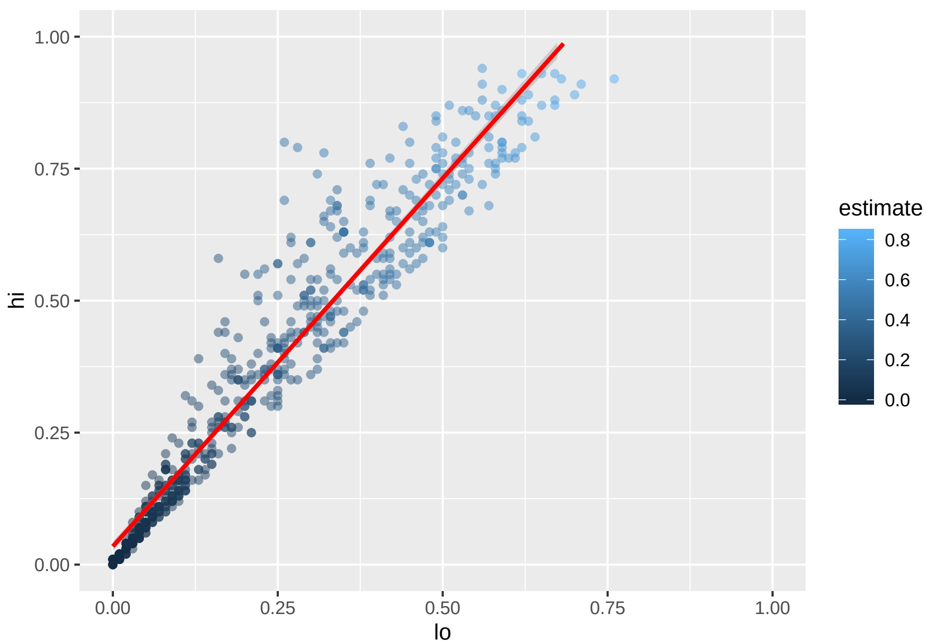
```
infant_hiv %>%
  drop_na() %>%
  filter(hi != 0.95) %>%
  filter(!((lo < 0.10) & (hi > 0.25))) %>%
  ggplot(mapping = aes(x = lo, y = hi, color = estimate)) +
  geom_point(alpha = 0.5) +
  xlim(0.0, 1.0) + ylim(0.0, 1.0)
```



We can add the fitted curve by including another geometry called `geom_smooth`:

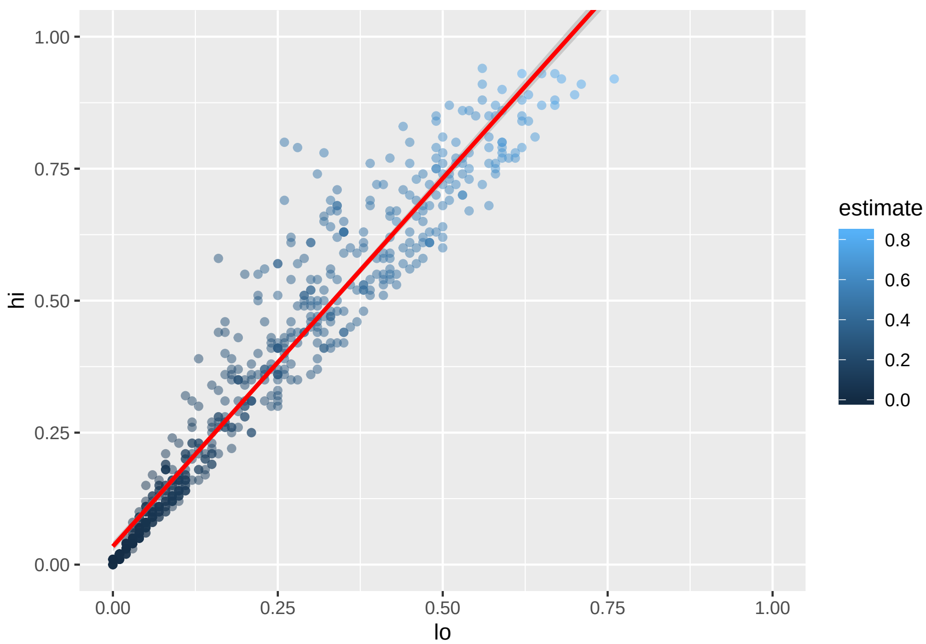
```
infant_hiv %>%
  drop_na() %>%
  filter(hi != 0.95) %>%
  filter(!((lo < 0.10) & (hi > 0.25))) %>%
  ggplot(mapping = aes(x = lo, y = hi)) +
  geom_point(mapping = aes(color = estimate), alpha = 0.5) +
  geom_smooth(method = lm, color = 'red') +
  xlim(0.0, 1.0) + ylim(0.0, 1.0)
```

```
Warning: Removed 8 rows containing missing values (geom_smooth).
```



But wait: why is this complaining about missing values? Some online searches lead to the discovery that `geom_smooth` adds virtual points to the data for plotting purposes, some of which lie outside the range of the actual data, and that setting `xlim` and `ylim` then truncates these. (Remember, R is differently sane…) The safe way to control the range of the data is to add a call to `coord_cartesian`, which effectively zooms in on a region of interest:

```
infant_hiv %>%
  drop_na() %>%
  filter(hi != 0.95) %>%
  filter(!((lo < 0.10) & (hi > 0.25))) %>%
  ggplot(mapping = aes(x = lo, y = hi)) +
  geom_point(mapping = aes(color = estimate), alpha = 0.5) +
  geom_smooth(method = lm, color = 'red') +
  coord_cartesian(xlim = c(0.0, 1.0), ylim = c(0.0, 1.0))
```



3.10 Do I need more practice with the tidyverse?

Absolutely: open a fresh file and begin by loading the tidyverse and the here package used to construct paths:

```
library(tidyverse)
library(here)
```

Next, use `here::here` to construct a path to a file and `readr::read_csv` to read that file:

```
path = here::here("data", "person.csv")
person <- readr::read_csv(path)
```

```
Parsed with column specification:
cols(
  person_id = col_character(),
  personal_name = col_character(),
  family_name = col_character()
)
```

We don't need to write out fully-qualified names—`here` and `read_csv` will do—but we will use them to make it easier to see what comes from where.

Next, have a look at the tibble `person`, which contains some basic information about a group of foolhardy scientists who ventured into the Antarctic in the 1920s and 1930s in search of things best left undisturbed:

```
person
```

```
# A tibble: 5 x 3
  person_id personal_name family_name
  <chr>     <chr>         <chr>
1 dyer      William       Dyer
2 pb        Frank         Pabodie
3 lake      Anderson      Lake
4 roe       Valentina     Roerich
5 danforth  Frank         Danforth
```

How many rows and columns does this tibble contain?

```
nrow(person)
```

```
[1] 5
```

```
ncol(person)
```

```
[1] 3
```

(These names don't have a package prefix because they are built in.) Let's show that information in a slightly nicer way using `glue` to insert values into a string and `print` to display the result:

```
print(glue::glue("person has {nrow(person)} rows and {ncol(person)} columns"))
```

```
person has 5 rows and 3 columns
```

If we want to display several values, we can use the function `paste` to combine the elements of a vector. `colnames` gives us the names of a tibble's columns, and `paste`'s `collapse` argument tells the function to use a single space to separate concatenated values:

```
print(glue::glue("person columns are {paste(colnames(person), collapse = ' ')}"))
```

```
person columns are person_id personal_name family_name
```

Time for some data manipulation. Let's get everyone's family and personal names:

```
dplyr::select(person, family_name, personal_name)
```

```
# A tibble: 5 x 2
  family_name personal_name
  <chr>       <chr>
1 Dyer        William
2 Pabodie     Frank
3 Lake        Anderson
```

```
4 Roerich     Valentina
5 Danforth    Frank
```

and then filter that list to keep only those that come in the first half of the alphabet:

```
dplyr::select(person, family_name, personal_name) %>%
  dplyr::filter(family_name < "N")
```

```
# A tibble: 3 x 2
  family_name personal_name
  <chr>       <chr>
1 Dyer        William
2 Lake        Anderson
3 Danforth    Frank
```

It would be more consistent to rewrite this as:

```
person %>%
  dplyr::select(family_name, personal_name) %>%
  dplyr::filter(family_name < "N")
```

```
# A tibble: 3 x 2
  family_name personal_name
  <chr>       <chr>
1 Dyer        William
2 Lake        Anderson
3 Danforth    Frank
```

It's easy to add a column that records the lengths of family names:

```
person %>%
  dplyr::mutate(name_length = stringr::str_length(family_name))
```

```
# A tibble: 5 x 4
  person_id personal_name family_name name_length
  <chr>     <chr>         <chr>             <int>
1 dyer      William       Dyer                  4
2 pb        Frank         Pabodie               7
3 lake      Anderson      Lake                  4
4 roe       Valentina     Roerich               7
5 danforth  Frank         Danforth              8
```

and then arrange in descending order:

```
person %>%
  dplyr::mutate(name_length = stringr::str_length(family_name)) %>%
  dplyr::arrange(dplyr::desc(name_length))
```

```
# A tibble: 5 x 4
```

```
  person_id personal_name family_name name_length
  <chr>     <chr>         <chr>             <int>
1 danforth  Frank         Danforth              8
2 pb        Frank         Pabodie               7
3 roe       Valentina     Roerich               7
4 dyer      William       Dyer                  4
5 lake      Anderson      Lake                  4
```

3.11 Key Points

- `install.packages('name')` installs packages.
- `library(name)` (without quoting the name) loads a package.
- `library(tidyverse)` loads the entire collection of tidyverse libraries at once.
- `read_csv(filename)` reads CSV files that use the string 'NA' to represent missing values.
- `read_csv` infers each column's data types based on the first thousand values it reads.
- A tibble is the tidyverse's version of a data frame, which represents tabular data.
- `head(tibble)` and `tail(tibble)` inspect the first and last few rows of a tibble.
- `summary(tibble)` displays a summary of a tibble's structure and values.
- `tibble$column` selects a column from a tibble, returning a vector as a result.
- `tibble['column']` selects a column from a tibble, returning a tibble as a result.
- `tibble[,c]` selects column `c` from a tibble, returning a tibble as a result.
- `tibble[r,]` selects row `r` from a tibble, returning a tibble as a result.
- Use ranges and logical vectors as indices to select multiple rows/columns or specific rows/columns from a tibble.
- `tibble[[c]]` selects column `c` from a tibble, returning a vector as a result.
- `min(...)`, `mean(...)`, `max(...)`, and `std(...)` calculates the minimum, mean, maximum, and standard deviation of data.
- These aggregate functions include `NA`s in their calculations, and so will produce `NA` if the input data contains any.
- Use `func(data, na.rm = TRUE)` to remove `NA`s from data before calculations are done (but make sure this is statistically justified).
- `filter(tibble, condition)` selects rows from a tibble that pass a logical test on their values.
- `arrange(tibble, column)` or `arrange(desc(column))` arrange rows according to values in a column (the latter in descending order).
- `select(tibble, column, column, ...)` selects columns from a tibble.
- `select(tibble, -column)` selects *out* a column from a tibble.
- `mutate(tibble, name = expression, name = expression, ...)`

adds new columns to a tibble using values from existing columns.
- `group_by(tibble, column, column, ...)` groups rows that have the same values in the specified columns.
- `summarize(tibble, name = expression, name = expression)` aggregates tibble values (by groups if the rows have been grouped).
- `tibble %>% function(arguments)` performs the same operation as `function(tibble, arguments)`.
- Use `%>%` to create pipelines in which the left side of each `%>%` becomes the first argument of the next stage.

Chapter 4

Creating Reports

Every serious scholar dreams of recording their knowledge in a leather-bound tome that will lie forgotten and dusty on the shelves of an out-of-the-way library at a small college with a dubious reputation until someone naïve enough to believe that they can control forces beyond mortal ken stumbles upon it and unleashes ravenous horrors to prey upon the sanity of the innocent. Sadly, in these diminished times we must settle for dry expositions of trivia typeset in two columns using a demure font and sequentially-numbered citations.

But there is yet hope. One of R's greatest strengths is a package called `knitr` that translates documents written in a format called R Markdown into HTML, PDF, and e-books. R Markdown files are a kind of programmable document; authors can interleave prose with chunks of R (or other languages) and `knitr` will run that code as it processes the document to create tables and diagrams. To aid this, the RStudio IDE includes tools to create new documents, insert and run code chunks, preview documents' structure and output, and much more.

That's the good news. The bad news is that Markdown isn't a standard: it's more like a set of ad hoc implementations flying in loose formation. While basic elements like headings and links are more or less the same in R Markdown as they are in (for example) GitHub Flavored Markdown, people who have used other dialects may trip over small differences.

This chapter starts by introducing Markdown and publishing workflow, then shows how to embed code and customize reports. We close by showing how to publish reports on GitHub and Netlify, which both offer free hosting for small websites.

4.1 Learning Objectives

- Create a Markdown file that includes headings, links, and external images.

- Compile that file to produce an HTML page.
- Customize the page via its YAML header.
- Add R code chunks to a page.
- Format tabular output programmatically.
- Share common startup code between pages.
- Create and customize parameterized documents.
- Publish pages on GitHub by putting generated HTML in the `docs` directory.
- Publish pages by copying them to Netlify.

4.2 How can I create and preview a simple page?

To begin, create a file called `first.Rmd` and add the following text to it:

```
## Methods {-}

Something *really important*.

An [external link](https://tidynomicon.tech).

[Another link][rstudio]

<!-- link table below -->

[rstudio]: https://rstudio.com
```

This shows several key features of Markdown:

- A level-1 heading (`h1` in HTML) is put on a line starting with `#`. A level-2 heading (`h2`) uses two of these and so on.
- Putting `{-}` immediately after the heading title suppresses numbering. Without this, our examples would all be included in this book's table of contents, which isn't what we want. (This is one of R Markdown's extensions to Markdown.)
- Paragraphs are separated by blank lines.
- Text can be put in single asterisks for *italics* or double asterisk for **bold**.
- To create a link, put the visible text inside square brackets and the URL inside parentheses immediately after it. This is the reverse of HTML's order, which puts the URL inside the opening `a` tag *before* the contained text that is displayed.
- Links can also be written by putting text inside square brackets and an identifier immediately after it, also in square brackets. Those identifiers can then be associated with links in a table at the bottom of the page.
- Comments are written as they are in HTML.

Here's what the HTML corresponding to our simple Markdown document looks like:

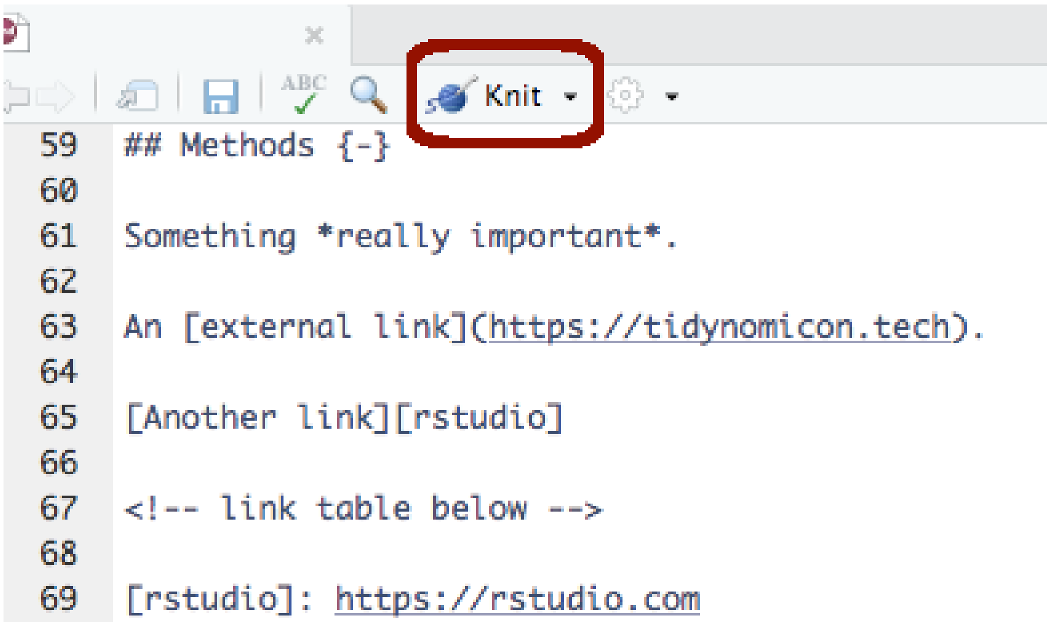


Figure 4.1: The 'knit' Button

Methods

Something *really important*.

An external link.

Another link

To preview it, go to the mini-toolbar at the top of your document in the RStudio IDE and click "knit": to call the appropriate function from `knitr` (or a function from one of the libraries built on top of it for generating books or slides).

4.3 How can I run code and include its output in a page?

If this is all R Markdown could do, it would be nothing more than an idiosyncratic way to create HTML pages. What makes it powerful is the ability to include code chunks that are evaluated as the document is knit, and whose output is included in the final page. Put this in a file called `second.Rmd`:

```
Displaying the colors:

```{r}
```
```

```
colors <- c('red', 'green', 'blue')
colors
```
```

The triple back-quotes mark the start and end of a block of code; putting `{r}` immediately after the back-quotes at the start tells `knitr` to run the code and include its output in the generated page, which therefore looks like this:

Displaying the colors:

We can put any code we want inside code blocks. We don't have to execute it all at once: the `Code` pulldown in RStudio's main menu offers a variety of ways to run regions of code. The IDE also gives us a keyboard shortcut to insert a new code chunk, so there really is no excuse for not making notes as we go along.

We can control execution and formatting by putting options inside the curly braces at the start of the code block:

- `{r label}` gives the chunk a label that we can cross-reference. Labels must be unique within documents, just like the `id` attributes of HTML elements.
- `{r include=FALSE}` tells `knitr` to run the code but *not* to include either the code or its output in the finished document. While the option name is confusing—the code is actually included in processing—this is handy when we have setup code that loads libraries or does other things that our readers probably don't care about.
- `{r eval=FALSE}` displays the code but doesn't run it, and is often used for tutorials like this one.
- `{r echo=FALSE}` hides the code but includes the output. This is most often used for displaying static images as we will see below.

These options can be combined by separating them with commas. In particular, it's good style to give every chunk a unique label, so a document might look like this:

````
My Thesis {-}

```{r setup, include=FALSE}
# Load tidyverse but don't display messages.
library(tidyverse)
```

```{r read-data, message=FALSE}
earthquakes <- read_csv('earthquakes.csv')
```

A profound quotation to set the scene.
````
````

````
And then some analysis:

```{r calculate-depth-by-magnitude}
depth_by_magnitude <- earthquakes %>%
  mutate(round_mag = round(Magnitude)) %>%
  group_by(round_mag) %>%
  summarize(depth = mean(Depth_Km))
depth_by_magnitude
```

Now let's visualize that:

```{r plot-depth-by-magnitude}
depth_by_magnitude %>%
  ggplot() +
  geom_point(mapping = aes(x = round_mag, y = depth))
```
````

In order:

- The document title is a level-1 header with suppressed numbering.
- The first code chunk is called `setup` and neither it nor its output are included in the output page.
- The second chunk is called `read-data`. It is shown in the output, but its output is not.
- There is then a (very) short paragraph.
- The third code chunk calculates the mean depth by rounded magnitude. Both the code and its output are included; the output is just R's textual display of the `depth_by_magnitude` table.
- After an even shorter paragraph, there is another named chunk whose output is a plot rather than text. `knitr` runs `ggplot2` to create the plot and includes it in the page.

When this page is knit, the result is:

My Thesis

A profound quotation to set the scene. And then some analysis:

Now let's visualize that:

4.4 How can I format tables in a page?

Tables are the undemonstrative yet reliable foundation on which data science is built. While they are not as showy as their graphical counterparts, they permit closer scrutiny, and are accessible both to people with visual challenges and to the machines whose inevitable triumph over us shall usher in an agorithmic age free of superstition and mercy.

The simplest way to format tables is to use `knitr::kable`:

```
earthquakes %>%
  head(5) %>%
  kable()
```

Time	Latitude	Longitude	Depth_Km	Magnitude
2016-08-24 03:36:32	42.6983	13.2335	8.1	6.0
2016-08-24 03:37:26	42.7123	13.2533	9.0	4.5
2016-08-24 03:40:46	42.7647	13.1723	9.7	3.8
2016-08-24 03:41:38	42.7803	13.1683	9.7	3.9
2016-08-24 03:42:07	42.7798	13.1575	9.7	3.6

Our output is more attractive if we install and load the `kableExtra` package and use it to style the table. We must call its functions *after* we call `kable()`, just as we call the styling functions for plots after `ggplot()`. Below, we select four columns from our earthquake data and format them as a narrow table with two decimal places for latitude and longitude, one for magnitude and depth, and some multi-column headers:

```
earthquakes %>%
  select(lat = Latitude, long = Longitude,
         mag = Magnitude, depth = Depth_Km) %>%
  head(5) %>%
  kable(digits = c(2, 2, 1, 1)) %>%
  kable_styling(full_width = FALSE) %>%
  add_header_above(c('Location' = 2, 'Details' = 2))
```

Location		Details	
lat	long	mag	depth
42.70	13.23	6.0	8.1
42.71	13.25	4.5	9.0
42.76	13.17	3.8	9.7
42.78	13.17	3.9	9.7
42.78	13.16	3.6	9.7

4.5 How can I share code between pages?

If you are working on several related reports, you may want to share some code between them. The best way to do this with R Markdown is to put that code in a separate `.R` file and then load that at the start of each document using the `source` function. For example, all of the chapters in this book begin with:

```
```{r setup, include=FALSE}
source('common.R')
```
```

```
```
```

The chunk is named `setup`, and neither it nor its output are displayed. All the chunk does is load and run `common.R`, which contains the following lines:

```
library(tidyverse)
library(reticulate)
library(rlang)
library(knitr)

knitr::opts_knit$set(width = 69)
```

The first few load libraries that various chapters depend on; the last one tells `knitr` to set the line width option to 69 characters.

4.6 How can I parameterize documents?

`knitr` has many other options besides line width, and the tools built on top of it, like Blogdown and Bookdown, have many (many) more. Rather than calling a function to set them, you can and should add a header to each document. If we use `File...New File...RMarkdown` to create a new R Markdown file, its header looks like this:

```
---
title: "fourth"
author: "Greg Wilson"
date: "18/09/2019"
output: html_document
---
```

1. The header starts with exactly three dashes on a line of their own and ends the same way. A common mistake is to forget the closing dashes; another is to use too many or too few, or to include whitespace in the line.
2. The content of the header is formatted using YAML, which stands for "Yet Another Markup Language". In its simplest form it contains key-value pairs: the keys are words, the values can be numbers, quoted strings, or a variety of other things, and the two are separated by a comma.

This header tells `knitr` what the document's title is, who its author is, when it was created (which really ought to be written as an ISO-formatted date, but worse sins await us), and what output format we want by default. When we knit the document, `knitr` reads the header but does *not* include it in the output. Instead, its values control `knitr`'s operation (e.g., select HTML as the output format) or are inserted into the document itself (e.g., the title).

Let's edit the YAML header so that it looks like this:
```

```
---
title: "fourth"
author: "Greg Wilson"
date: "2019-09-18"
output:
  html_document:
    theme: united
    toc: true
---
```

1. The date is now in an unambiguous, sortable format. This doesn't impact our document, but makes us feel better.
2. We have added two sub-keys under `html_document` (which we have made a sub-key of `output` so that we can nest things beneath it). The first tells `knitr` to use the `united` theme, which gives us a different set of fonts and margins. The second tells it to create a table of contents at the start of the document with links to all of the section headers.

YAML can be quite complicated to understand. Luckily, a package called `ymlthis` is being developed to create and check files' headers. Its documentation and capabilities are both steadily growing, it's a great way to experiment with new or obscure options.

But YAML can do more than control the way `knitr` processes the document: we can also use it to create parameterized reports.

````
---
title: "Fifth Report"
params:
  country: Canada
---

This report looks at defenstration rates in `r params$country`.

```{r load-data}
data <- read_csv(here::here('data', glue(params$country, '.csv')))
```
````

This document's YAML header contains the key `params`, under which is a sub-key for each parameter we want to create. When the document is knit, these parameters are put in a named list called `params` and can be referred to like any other variable. If we want to display it inline, we use a back-ticked code fragment that starts with the letter 'r'; if we want to use it in a fenced code block, it's no different from any other variable.

Parameters don't have to be single values: they can, for example, be lists of mysterious ailments whose inexorable spread you are vainly trying to halt. Parameters can also be provided on the command line:

```
Rscript -e "rmarkdown::render('fifth.Rmd', params=list(country='Lesotho'))"
```

will create a page called `fifth.html` that reports defenestration rates in Lesotho.

4.7 How can I publish pages on GitHub?

FIXME

4.8 How can I publish pages on Netlify?

FIXME

4.9 Key Points

- FIXME: key points for R Markdown

Chapter 5

Creating Packages

Data is not born tidy. We must cleanse it to make it serve our needs. The previous chapter gave us the tools; here, we will see how to apply them and how to make our work usable by others.

5.1 Learning Objectives

- Describe and use the `read_csv` function.
- Describe and use the `str_replace` function.
- Describe and use the `is.numeric` and `as.numeric` functions.
- Describe and use the `map` function and its kin.
- Describe and use pre-allocation to capture the results of loops.
- Describe the three things an R package can contain.
- Explain how R code in a package is distributed and one implication of this.
- Explain the purpose of the `DESCRIPTION`, `NAMESPACE` and `.Rbuildignore` files in an R project.
- Explain what should be put in the `R`, `data`, `man`, and `tests` directories of an R project.
- Describe and use specially-formatted comments with roxygen2 to document a package.
- Use `@export` and `@import` directives correctly in roxygen2 documentation.
- Add a dataset to an R package.
- Use functions from external libraries inside a package in a hygienic way.
- Rewrite references to bare column names to satisfy R's packaging checks.
- Correctly document the package as a whole and the datasets it contains.

5.2 What is our starting point?

Here is a sample of data from the original data set `data/infant_hiv.csv`, where ... shows values elided to make the segment readable:

```
"Early Infant Diagnosis: Percentage of infants born to women living with HIV...",,,,,,
,,2009,,,2010,,,2011,,,2012,,,2013,,,2014,,,2015,,,2016,,,2017,,,
ISO3,Countries,Estimate,hi,lo,Estimate,hi,lo,Estimate,hi,lo,Estimate,hi,lo,...
AFG,Afghanistan,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,
ALB,Albania,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,
DZA,Algeria,-,-,-,-,-,-,38%,42%,35%,23%,25%,21%,55%,60%,50%,27%,30%,25%,23%,25%,21%,33
AGO,Angola,-,-,-,3%,4%,2%,5%,7%,4%,6%,8%,5%,15%,20%,12%,10%,14%,8%,6%,8%,5%,1%,2%,1%,1
... many more rows ...
YEM,Yemen,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,
ZMB,Zambia,59%,70%,53%,27%,32%,24%,70%,84%,63%,74%,88%,67%,64%,76%,57%,91%,>95%,81%,43
ZWE,Zimbabwe,-,-,-,12%,15%,10%,23%,28%,20%,38%,47%,33%,57%,70%,49%,54%,67%,47%,59%,73%
,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
,,2009,,,2010,,,2011,,,2012,,,2013,,,2014,,,2015,,,2016,,,2017,,,
,,Estimate,hi,lo,Estimate,hi,lo,Estimate,hi,lo,Estimate,hi,lo,...
Region,East Asia and the Pacific,25%,30%,22%,35%,42%,29%,30%,37%,26%,32%,38%,27%,28%,3
,Eastern and Southern Africa,23%,29%,20%,44%,57%,37%,48%,62%,40%,54%,69%,46%,51%,65%,4
,Eastern Europe and Central Asia,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-
... several more rows ...
,Sub-Saharan Africa,16%,22%,13%,34%,46%,28%,37%,50%,30%,43%,57%,35%,41%,54%,33%,50%,66
,Global,17%,23%,13%,33%,45%,27%,36%,49%,29%,41%,55%,34%,40%,53%,32%,48%,64%,39%,49%,64
,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
Indicator definition: Percentage of infants born to women living with HIV... ,,,,,,,,,
Note: Data are not available if country did not submit data...,,,,,,,,,,,,,,,,,,,,,,,,,,
Data source: Global AIDS Monitoring 2018 and UNAIDS 2018 estimates,,,,,,,,,,,,,,,,,,,,,
"For more information on this indicator, please visit the guidance:...",,,,,,,,,,,,,,
"For more information on the data, visit data.unicef.org",,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
```

This is a mess—no, more than that, it is an affront to decency. There are comments mixed with data, values' actual indices have to be synthesized by combining column headings from two rows (two thirds of which have to be carried forward from previous columns), and so on. We want to create the tidy data found in `results/infant_hiv.csv`:

```
country,year,estimate,hi,lo
AFG,2009,NA,NA,NA
AFG,2010,NA,NA,NA
AFG,2011,NA,NA,NA
AFG,2012,NA,NA,NA
...
ZWE,2016,0.71,0.88,0.62
ZWE,2017,0.65,0.81,0.57
```

5.3 How do I convert values to numbers?

We begin by reading the data into a tibble:

```
raw <- read_csv("data/infant_hiv.csv")
```

```
Warning: Missing column names filled in: 'X2' [2], 'X3' [3], 'X4' [4],
'X5' [5], 'X6' [6], 'X7' [7], 'X8' [8], 'X9' [9], 'X10' [10], 'X11' [11],
'X12' [12], 'X13' [13], 'X14' [14], 'X15' [15], 'X16' [16], 'X17' [17],
'X18' [18], 'X19' [19], 'X20' [20], 'X21' [21], 'X22' [22], 'X23' [23],
'X24' [24], 'X25' [25], 'X26' [26], 'X27' [27], 'X28' [28], 'X29' [29],
'X30' [30]
```

```
Parsed with column specification:
cols(
  .default = col_character(),
  X30 = col_logical()
)
```

```
See spec(...) for full column specifications.
```

```
head(raw)
```

```
# A tibble: 6 x 30
  `Early Infant D~ X2    X3    X4    X5    X6    X7    X8    X9    X10
  <chr>            <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr>
1 <NA>             <NA>  2009  <NA>  <NA>  2010  <NA>  <NA>  2011  <NA>
2 ISO3             Coun~ Esti~ hi    lo    Esti~ hi    lo    Esti~ hi
3 AFG              Afgh~ -     -     -     -     -     -     -     -
4 ALB              Alba~ -     -     -     -     -     -     -     -
5 DZA              Alge~ -     -     -     -     -     -     38%   42%
6 AGO              Ango~ -     -     -     3%    4%    2%    5%    7%
# ... with 20 more variables: X11 <chr>, X12 <chr>, X13 <chr>, X14 <chr>,
#   X15 <chr>, X16 <chr>, X17 <chr>, X18 <chr>, X19 <chr>, X20 <chr>,
#   X21 <chr>, X22 <chr>, X23 <chr>, X24 <chr>, X25 <chr>, X26 <chr>,
#   X27 <chr>, X28 <chr>, X29 <chr>, X30 <lgl>
```

All right: R isn't able to infer column names, so it uses the entire first comment string as a very long column name and then makes up names for the other columns. Looking at the file, the second row has years (spaced at three-column intervals) and the column after that has the ISO3 country code, the country's name, and then "Estimate", "hi", and "lo" repeated for every year. We are going to have to combine what's in the second and third rows, so we're going to have to do some work no matter which we skip or keep. Since we want the ISO3 code and the country name, let's skip the first two rows.

```
raw <- read_csv("data/infant_hiv.csv", skip = 2)
```

```
Warning: Missing column names filled in: 'X30' [30]
```

```
Warning: Duplicated column names deduplicated: 'Estimate' =>
'Estimate_1' [6], 'hi' => 'hi_1' [7], 'lo' => 'lo_1' [8], 'Estimate' =>
'Estimate_2' [9], 'hi' => 'hi_2' [10], 'lo' => 'lo_2' [11], 'Estimate' =>
'Estimate_3' [12], 'hi' => 'hi_3' [13], 'lo' => 'lo_3' [14], 'Estimate' =>
'Estimate_4' [15], 'hi' => 'hi_4' [16], 'lo' => 'lo_4' [17], 'Estimate' =>
'Estimate_5' [18], 'hi' => 'hi_5' [19], 'lo' => 'lo_5' [20], 'Estimate' =>
'Estimate_6' [21], 'hi' => 'hi_6' [22], 'lo' => 'lo_6' [23], 'Estimate' =>
'Estimate_7' [24], 'hi' => 'hi_7' [25], 'lo' => 'lo_7' [26], 'Estimate' =>
'Estimate_8' [27], 'hi' => 'hi_8' [28], 'lo' => 'lo_8' [29]
```

```
Parsed with column specification:
cols(
  .default = col_character(),
  X30 = col_logical()
)
```

```
See spec(...) for full column specifications.
```

```r
head(raw)
```

```
# A tibble: 6 x 30
  ISO3  Countries Estimate hi    lo    Estimate_1 hi_1  lo_1  Estimate_2
  <chr> <chr>     <chr>    <chr> <chr> <chr>      <chr> <chr> <chr>
1 AFG   Afghanis~ -        -     -     -          -     -     -
2 ALB   Albania   -        -     -     -          -     -     -
3 DZA   Algeria   -        -     -     -          -     -     38%
4 AGO   Angola    -        -     -     3%         4%    2%    5%
5 AIA   Anguilla  -        -     -     -          -     -     -
6 ATG   Antigua ~ -        -     -     -          -     -     -
# ... with 21 more variables: hi_2 <chr>, lo_2 <chr>, Estimate_3 <chr>,
#   hi_3 <chr>, lo_3 <chr>, Estimate_4 <chr>, hi_4 <chr>, lo_4 <chr>,
#   Estimate_5 <chr>, hi_5 <chr>, lo_5 <chr>, Estimate_6 <chr>,
#   hi_6 <chr>, lo_6 <chr>, Estimate_7 <chr>, hi_7 <chr>, lo_7 <chr>,
#   Estimate_8 <chr>, hi_8 <chr>, lo_8 <chr>, X30 <lgl>
```

That's a bit of an improvement, but why are all the columns `character` instead of numbers? This happens because:

1. our CSV file uses `-` (a single dash) to show missing data, and
2. all of our numbers end with `%`, which means those values actually *are* character strings.

We will tackle the first problem by setting `na = c("-")` in our `read_csv` call (since we should never do ourselves what a library function will do for us):

```r
raw <- read_csv("data/infant_hiv.csv", skip = 2, na = c("-"))
```

```
Warning: Missing column names filled in: 'X30' [30]
```

```
Warning: Duplicated column names deduplicated: 'Estimate' =>
```

```
'Estimate_1' [6], 'hi' => 'hi_1' [7], 'lo' => 'lo_1' [8], 'Estimate' =>
'Estimate_2' [9], 'hi' => 'hi_2' [10], 'lo' => 'lo_2' [11], 'Estimate' =>
'Estimate_3' [12], 'hi' => 'hi_3' [13], 'lo' => 'lo_3' [14], 'Estimate' =>
'Estimate_4' [15], 'hi' => 'hi_4' [16], 'lo' => 'lo_4' [17], 'Estimate' =>
'Estimate_5' [18], 'hi' => 'hi_5' [19], 'lo' => 'lo_5' [20], 'Estimate' =>
'Estimate_6' [21], 'hi' => 'hi_6' [22], 'lo' => 'lo_6' [23], 'Estimate' =>
'Estimate_7' [24], 'hi' => 'hi_7' [25], 'lo' => 'lo_7' [26], 'Estimate' =>
'Estimate_8' [27], 'hi' => 'hi_8' [28], 'lo' => 'lo_8' [29]
```

```
Parsed with column specification:
cols(
  .default = col_character(),
  X30 = col_logical()
)
```

```
See spec(...) for full column specifications.
```

```r
head(raw)
```

```
# A tibble: 6 x 30
  ISO3  Countries Estimate hi    lo    Estimate_1 hi_1  lo_1  Estimate_2
  <chr> <chr>     <chr>    <chr> <chr> <chr>      <chr> <chr> <chr>
1 AFG   Afghanis~ <NA>     <NA>  <NA>  <NA>       <NA>  <NA>  <NA>
2 ALB   Albania   <NA>     <NA>  <NA>  <NA>       <NA>  <NA>  <NA>
3 DZA   Algeria   <NA>     <NA>  <NA>  <NA>       <NA>  <NA>  38%
4 AGO   Angola    <NA>     <NA>  <NA>  3%         4%    2%    5%
5 AIA   Anguilla  <NA>     <NA>  <NA>  <NA>       <NA>  <NA>  <NA>
6 ATG   Antigua ~ <NA>     <NA>  <NA>  <NA>       <NA>  <NA>  <NA>
# ... with 21 more variables: hi_2 <chr>, lo_2 <chr>, Estimate_3 <chr>,
#   hi_3 <chr>, lo_3 <chr>, Estimate_4 <chr>, hi_4 <chr>, lo_4 <chr>,
#   Estimate_5 <chr>, hi_5 <chr>, lo_5 <chr>, Estimate_6 <chr>,
#   hi_6 <chr>, lo_6 <chr>, Estimate_7 <chr>, hi_7 <chr>, lo_7 <chr>,
#   Estimate_8 <chr>, hi_8 <chr>, lo_8 <chr>, X30 <lgl>
```

That's progress. We now need to strip the percentage signs and convert what's left to numeric values. To simplify our lives, let's get the `ISO3` and `Countries` columns out of the way. We will save the ISO3 values for later use (and because it will illustrate a point about data hygiene that we want to make later, but which we don't want to reveal just yet). Rather than typing out the names of all the columns we want to keep in the call to `filter`, we subtract the ones we want to discard:

```r
raw <- read_csv("data/infant_hiv.csv", skip = 2, na = c("-"))
```

```
Warning: Missing column names filled in: 'X30' [30]
```

```
Warning: Duplicated column names deduplicated: 'Estimate' =>
'Estimate_1' [6], 'hi' => 'hi_1' [7], 'lo' => 'lo_1' [8], 'Estimate' =>
'Estimate_2' [9], 'hi' => 'hi_2' [10], 'lo' => 'lo_2' [11], 'Estimate' =>
```

```
'Estimate_3' [12], 'hi' => 'hi_3' [13], 'lo' => 'lo_3' [14], 'Estimate' =>
'Estimate_4' [15], 'hi' => 'hi_4' [16], 'lo' => 'lo_4' [17], 'Estimate' =>
'Estimate_5' [18], 'hi' => 'hi_5' [19], 'lo' => 'lo_5' [20], 'Estimate' =>
'Estimate_6' [21], 'hi' => 'hi_6' [22], 'lo' => 'lo_6' [23], 'Estimate' =>
'Estimate_7' [24], 'hi' => 'hi_7' [25], 'lo' => 'lo_7' [26], 'Estimate' =>
'Estimate_8' [27], 'hi' => 'hi_8' [28], 'lo' => 'lo_8' [29]

Parsed with column specification:
cols(
  .default = col_character(),
  X30 = col_logical()
)

See spec(...) for full column specifications.
```

```
countries <- raw$ISO3
body <- raw %>%
  filter(-ISO3, -Countries)
```

```
Error in -ISO3: invalid argument to unary operator
```

In the Hollywood version of this lesson, we would sigh heavily at this point as we realize that we should have called `select`, not `filter`. Once we make that change, we can move forward once again:

```
raw <- read_csv("data/infant_hiv.csv", skip = 2, na = c("-"))
```

```
Warning: Missing column names filled in: 'X30' [30]

Warning: Duplicated column names deduplicated: 'Estimate' =>
'Estimate_1' [6], 'hi' => 'hi_1' [7], 'lo' => 'lo_1' [8], 'Estimate' =>
'Estimate_2' [9], 'hi' => 'hi_2' [10], 'lo' => 'lo_2' [11], 'Estimate' =>
'Estimate_3' [12], 'hi' => 'hi_3' [13], 'lo' => 'lo_3' [14], 'Estimate' =>
'Estimate_4' [15], 'hi' => 'hi_4' [16], 'lo' => 'lo_4' [17], 'Estimate' =>
'Estimate_5' [18], 'hi' => 'hi_5' [19], 'lo' => 'lo_5' [20], 'Estimate' =>
'Estimate_6' [21], 'hi' => 'hi_6' [22], 'lo' => 'lo_6' [23], 'Estimate' =>
'Estimate_7' [24], 'hi' => 'hi_7' [25], 'lo' => 'lo_7' [26], 'Estimate' =>
'Estimate_8' [27], 'hi' => 'hi_8' [28], 'lo' => 'lo_8' [29]

Parsed with column specification:
cols(
  .default = col_character(),
  X30 = col_logical()
)

See spec(...) for full column specifications.
```

```
countries <- raw$ISO3
body <- raw %>%
  select(-ISO3, -Countries)
```

```r
head(body)
```

```
# A tibble: 6 x 28
  Estimate hi    lo    Estimate_1 hi_1  lo_1  Estimate_2 hi_2  lo_2 
  <chr>    <chr> <chr> <chr>      <chr> <chr> <chr>      <chr> <chr>
1 <NA>     <NA>  <NA>  <NA>       <NA>  <NA>  <NA>       <NA>  <NA> 
2 <NA>     <NA>  <NA>  <NA>       <NA>  <NA>  <NA>       <NA>  <NA> 
3 <NA>     <NA>  <NA>  <NA>       <NA>  <NA>  38%        42%   35%  
4 <NA>     <NA>  <NA>  3%         4%    2%    5%         7%    4%   
5 <NA>     <NA>  <NA>  <NA>       <NA>  <NA>  <NA>       <NA>  <NA> 
6 <NA>     <NA>  <NA>  <NA>       <NA>  <NA>  <NA>       <NA>  <NA> 
# ... with 19 more variables: Estimate_3 <chr>, hi_3 <chr>, lo_3 <chr>,
#   Estimate_4 <chr>, hi_4 <chr>, lo_4 <chr>, Estimate_5 <chr>,
#   hi_5 <chr>, lo_5 <chr>, Estimate_6 <chr>, hi_6 <chr>, lo_6 <chr>,
#   Estimate_7 <chr>, hi_7 <chr>, lo_7 <chr>, Estimate_8 <chr>,
#   hi_8 <chr>, lo_8 <chr>, X30 <lgl>
```

But wait. Weren't there some aggregate lines of data at the end of our input? What happened to them?

```r
tail(countries, n = 25)
```

```
 [1] "YEM"
 [2] "ZMB"
 [3] "ZWE"
 [4] ""
 [5] ""
 [6] ""
 [7] "Region"
 [8] ""
 [9] ""
[10] ""
[11] ""
[12] ""
[13] ""
[14] ""
[15] ""
[16] "Super-region"
[17] ""
[18] ""
[19] ""
[20] ""
[21] "Indicator definition: Percentage of infants born to women living with HIV receiving a virol
[22] "Note: Data are not available if country did not submit data to Global AIDS Monitoring or if
[23] "Data source: Global AIDS Monitoring 2018 and UNAIDS 2018 estimates"
[24] "For more information on this indicator, please visit the guidance: http://www.unaids.org/si
[25] "For more information on the data, visit data.unicef.org"
```

Once again the actor playing our part on screen sighs heavily. How are we to trim this? Since there is only one file, we can open the file with an editor or spreadsheet program, scroll down, check the line number, and slice there. This is a very bad idea if we're planning to use this script on other files—we should instead look for the first blank line or the entry for Zimbabwe or something like that—but let's revisit the problem once we have our data in place.

```
num_rows <- 192
raw <- read_csv("data/infant_hiv.csv", skip = 2, na = c("-"))
```

```
Warning: Missing column names filled in: 'X30' [30]
```

```
Warning: Duplicated column names deduplicated: 'Estimate' =>
'Estimate_1' [6], 'hi' => 'hi_1' [7], 'lo' => 'lo_1' [8], 'Estimate' =>
'Estimate_2' [9], 'hi' => 'hi_2' [10], 'lo' => 'lo_2' [11], 'Estimate' =>
'Estimate_3' [12], 'hi' => 'hi_3' [13], 'lo' => 'lo_3' [14], 'Estimate' =>
'Estimate_4' [15], 'hi' => 'hi_4' [16], 'lo' => 'lo_4' [17], 'Estimate' =>
'Estimate_5' [18], 'hi' => 'hi_5' [19], 'lo' => 'lo_5' [20], 'Estimate' =>
'Estimate_6' [21], 'hi' => 'hi_6' [22], 'lo' => 'lo_6' [23], 'Estimate' =>
'Estimate_7' [24], 'hi' => 'hi_7' [25], 'lo' => 'lo_7' [26], 'Estimate' =>
'Estimate_8' [27], 'hi' => 'hi_8' [28], 'lo' => 'lo_8' [29]
```

```
Parsed with column specification:
cols(
  .default = col_character(),
  X30 = col_logical()
)
```

```
See spec(...) for full column specifications.
```

```
sliced <- slice(raw, 1:num_rows)
countries <- sliced$ISO3
tail(countries, n = 5)
```

```
[1] "VEN" "VNM" "YEM" "ZMB" "ZWE"
```

Notice that we're counting rows *not including* the two we're skipping, which means that the 192 in the call to `slice` above corresponds to row 195 of our original data: 195, not 194, because we're using the first row of unskipped data as headers and yes, you are in fact making that faint whimpering sound you now hear. You will hear it often when dealing with real-world data…

Notice also that we are slicing, *then* extracting the column containing the countries. In an earlier version of this lesson we peeled off the ISO3 country codes, sliced that vector, and then wondered why our main table still had unwanted data at the end. Vigilance, my friends—vigilance shall be our watchword, and in light of that, we shall first test our plan for converting our strings to numbers:

```
fixture <- c(NA, "1%", "10%", "100%")
result <- as.numeric(str_replace(fixture, "%", "")) / 100
```

```
result
```

```
[1]   NA 0.01 0.10 1.00
```

And as a further check:

```
is.numeric(result)
```

```
[1] TRUE
```

The function `is.numeric` is `TRUE` for both `NA` and actual numbers, so it is doing the right thing here, and so are we. Our updated conversion script is now:

```
raw <- read_csv("data/infant_hiv.csv", skip = 2, na = c("-"))
```

```
Warning: Missing column names filled in: 'X30' [30]
```

```
Warning: Duplicated column names deduplicated: 'Estimate' =>
'Estimate_1' [6], 'hi' => 'hi_1' [7], 'lo' => 'lo_1' [8], 'Estimate' =>
'Estimate_2' [9], 'hi' => 'hi_2' [10], 'lo' => 'lo_2' [11], 'Estimate' =>
'Estimate_3' [12], 'hi' => 'hi_3' [13], 'lo' => 'lo_3' [14], 'Estimate' =>
'Estimate_4' [15], 'hi' => 'hi_4' [16], 'lo' => 'lo_4' [17], 'Estimate' =>
'Estimate_5' [18], 'hi' => 'hi_5' [19], 'lo' => 'lo_5' [20], 'Estimate' =>
'Estimate_6' [21], 'hi' => 'hi_6' [22], 'lo' => 'lo_6' [23], 'Estimate' =>
'Estimate_7' [24], 'hi' => 'hi_7' [25], 'lo' => 'lo_7' [26], 'Estimate' =>
'Estimate_8' [27], 'hi' => 'hi_8' [28], 'lo' => 'lo_8' [29]
```

```
Parsed with column specification:
cols(
  .default = col_character(),
  X30 = col_logical()
)
```

```
See spec(...) for full column specifications.
```

```
sliced <- slice(raw, 1:192)
countries <- sliced$ISO3
body <- raw %>%
  select(-ISO3, -Countries)
numbers <- as.numeric(str_replace(body, "%", "")) / 100
```

```
Warning in stri_replace_first_regex(string, pattern,
fix_replacement(replacement), : argument is not an atomic vector; coercing
```

```
Warning: NAs introduced by coercion
```

```
is.numeric(numbers)
```

```
[1] TRUE
```

Bother. It appears that `str_replace` expects an atomic vector rather than a tibble. It worked for our test case because that was a character vector, but tibbles have more structure than that.

The second complaint is that `NA`s were introduced, which is troubling because we didn't get a complaint when we had actual `NA`s in our data. However, `is.numeric` tells us that all of our results are numbers. Let's take a closer look:

```
is_tibble(body)
```

```
[1] TRUE
```

```
is_tibble(numbers)
```

```
[1] FALSE
```

Perdition. After browsing the data, we realize that some entries are `">95%"`, i.e., there is a greater-than sign as well as a percentage in the text. We will need to regularize those before we do any conversions.

Before that, however, let's see if we can get rid of the percent signs. The obvious way is is to use `str_replace(body, "%", "")`, but that doesn't work: `str_replace` works on vectors, but a tibble is a list of vectors. Instead, we can use a higher-order function called `map` to apply the function `str_replace` to each column in turn to get rid of the percent signs:

```
raw <- read_csv("data/infant_hiv.csv", skip = 2, na = c("-"))
```

```
Warning: Missing column names filled in: 'X30' [30]
```

```
Warning: Duplicated column names deduplicated: 'Estimate' =>
'Estimate_1' [6], 'hi' => 'hi_1' [7], 'lo' => 'lo_1' [8], 'Estimate' =>
'Estimate_2' [9], 'hi' => 'hi_2' [10], 'lo' => 'lo_2' [11], 'Estimate' =>
'Estimate_3' [12], 'hi' => 'hi_3' [13], 'lo' => 'lo_3' [14], 'Estimate' =>
'Estimate_4' [15], 'hi' => 'hi_4' [16], 'lo' => 'lo_4' [17], 'Estimate' =>
'Estimate_5' [18], 'hi' => 'hi_5' [19], 'lo' => 'lo_5' [20], 'Estimate' =>
'Estimate_6' [21], 'hi' => 'hi_6' [22], 'lo' => 'lo_6' [23], 'Estimate' =>
'Estimate_7' [24], 'hi' => 'hi_7' [25], 'lo' => 'lo_7' [26], 'Estimate' =>
'Estimate_8' [27], 'hi' => 'hi_8' [28], 'lo' => 'lo_8' [29]
```

```
Parsed with column specification:
cols(
  .default = col_character(),
  X30 = col_logical()
)
```

```
See spec(...) for full column specifications.
```

```
sliced <- slice(raw, 1:192)
countries <- sliced$ISO3
body <- raw %>%
  select(-ISO3, -Countries)
```

```
trimmed <- map(body, str_replace, pattern = "%", replacement = "")
head(trimmed)
```

```
$Estimate
  [1] NA         NA         NA         NA         NA         NA
  [7] NA         NA         NA         NA         "26"       NA
 [13] NA         NA         NA         ">95"      NA         "77"
 [19] NA         NA         "7"        NA         NA         "25"
 [25] NA         NA         "3"        NA         ">95"      NA
 [31] "27"       NA         "1"        NA         NA         NA
 [37] "5"        NA         "8"        NA         "92"       NA
 [43] NA         "83"       NA         NA         NA         NA
 [49] NA         NA         NA         "28"       "1"        "4"
 [55] NA         NA         NA         NA         "4"        NA
 [61] NA         NA         NA         NA         "61"       NA
 [67] NA         NA         NA         NA         NA         NA
 [73] NA         NA         "61"       NA         NA         NA
 [79] NA         "2"        NA         NA         NA         NA
 [85] NA         NA         NA         ">95"      NA         NA
 [91] NA         NA         NA         NA         NA         "43"
 [97] "5"        NA         NA         NA         NA         NA
[103] "37"       NA         "8"        NA         NA         NA
[109] NA         NA         NA         NA         NA         "2"
[115] NA         NA         NA         NA         "2"        NA
[121] NA         "50"       NA         "4"        NA         NA
[127] NA         "1"        NA         NA         NA         NA
[133] NA         NA         "1"        NA         NA         NA
[139] ">95"      NA         NA         "58"       NA         NA
[145] NA         NA         NA         NA         "11"       NA
[151] NA         NA         NA         NA         NA         NA
[157] NA         NA         NA         NA         NA         NA
[163] "9"        NA         NA         NA         NA         "1"
[169] NA         NA         NA         "7"        NA         NA
[175] NA         NA         NA         NA         "8"        "78"
[181] NA         NA         "13"       NA         NA         "0"
[187] NA         NA         NA         NA         "59"       NA
[193] ""         "2009"     "Estimate" "25"       "23"       NA
[199] "24"       "2"        NA         "1"        "8"        NA
[205] "7"        "72"       "16"       "17"       ""         ""
[211] ""         ""         ""         ""

$hi
  [1] NA     NA     NA     NA     NA     NA     NA     NA     NA     NA     "35"
...
```

Perdition once again. The problem now is that `map` produces a raw list as output. The function we want is `map_dfr`, which maps a function across the rows of a tibble and returns a tibble as a result. (There is a corresponding function `map_dfc` that maps a function across columns.)

```
raw <- read_csv("data/infant_hiv.csv", skip = 2, na = c("-"))
```

```
Warning: Missing column names filled in: 'X30' [30]

Warning: Duplicated column names deduplicated: 'Estimate' =>
'Estimate_1' [6], 'hi' => 'hi_1' [7], 'lo' => 'lo_1' [8], 'Estimate' =>
'Estimate_2' [9], 'hi' => 'hi_2' [10], 'lo' => 'lo_2' [11], 'Estimate' =>
'Estimate_3' [12], 'hi' => 'hi_3' [13], 'lo' => 'lo_3' [14], 'Estimate' =>
'Estimate_4' [15], 'hi' => 'hi_4' [16], 'lo' => 'lo_4' [17], 'Estimate' =>
'Estimate_5' [18], 'hi' => 'hi_5' [19], 'lo' => 'lo_5' [20], 'Estimate' =>
'Estimate_6' [21], 'hi' => 'hi_6' [22], 'lo' => 'lo_6' [23], 'Estimate' =>
'Estimate_7' [24], 'hi' => 'hi_7' [25], 'lo' => 'lo_7' [26], 'Estimate' =>
'Estimate_8' [27], 'hi' => 'hi_8' [28], 'lo' => 'lo_8' [29]

Parsed with column specification:
cols(
  .default = col_character(),
  X30 = col_logical()
)

See spec(...) for full column specifications.
```

```
sliced <- slice(raw, 1:192)
countries <- sliced$ISO3
body <- raw %>%
  select(-ISO3, -Countries)
trimmed <- map_dfr(body, str_replace, pattern = "%", replacement = "")
head(trimmed)
```

```
# A tibble: 6 x 28
  Estimate hi    lo    Estimate_1 hi_1  lo_1  Estimate_2 hi_2  lo_2
  <chr>    <chr> <chr> <chr>      <chr> <chr> <chr>      <chr> <chr>
1 <NA>     <NA>  <NA>  <NA>       <NA>  <NA>  <NA>       <NA>  <NA>
2 <NA>     <NA>  <NA>  <NA>       <NA>  <NA>  <NA>       <NA>  <NA>
3 <NA>     <NA>  <NA>  <NA>       <NA>  <NA>  38         42    35
4 <NA>     <NA>  <NA>  3          4     2     5          7     4
5 <NA>     <NA>  <NA>  <NA>       <NA>  <NA>  <NA>       <NA>  <NA>
6 <NA>     <NA>  <NA>  <NA>       <NA>  <NA>  <NA>       <NA>  <NA>
# ... with 19 more variables: Estimate_3 <chr>, hi_3 <chr>, lo_3 <chr>,
#   Estimate_4 <chr>, hi_4 <chr>, lo_4 <chr>, Estimate_5 <chr>,
#   hi_5 <chr>, lo_5 <chr>, Estimate_6 <chr>, hi_6 <chr>, lo_6 <chr>,
#   Estimate_7 <chr>, hi_7 <chr>, lo_7 <chr>, Estimate_8 <chr>,
#   hi_8 <chr>, lo_8 <chr>, X30 <chr>
```

Now to tackle those `">95%"` values. It turns out that `str_replace` uses regular expressions, not just direct string matches, so we can get rid of the `>` at the same time as we get rid of the `%`. We will check by looking at the first `Estimate` column, which earlier inspection informed us had at least one `">95%"` in it:

```
raw <- read_csv("data/infant_hiv.csv", skip = 2, na = c("-"))
```

```
Warning: Missing column names filled in: 'X30' [30]

Warning: Duplicated column names deduplicated: 'Estimate' =>
'Estimate_1' [6], 'hi' => 'hi_1' [7], 'lo' => 'lo_1' [8], 'Estimate' =>
'Estimate_2' [9], 'hi' => 'hi_2' [10], 'lo' => 'lo_2' [11], 'Estimate' =>
'Estimate_3' [12], 'hi' => 'hi_3' [13], 'lo' => 'lo_3' [14], 'Estimate' =>
'Estimate_4' [15], 'hi' => 'hi_4' [16], 'lo' => 'lo_4' [17], 'Estimate' =>
'Estimate_5' [18], 'hi' => 'hi_5' [19], 'lo' => 'lo_5' [20], 'Estimate' =>
'Estimate_6' [21], 'hi' => 'hi_6' [22], 'lo' => 'lo_6' [23], 'Estimate' =>
'Estimate_7' [24], 'hi' => 'hi_7' [25], 'lo' => 'lo_7' [26], 'Estimate' =>
'Estimate_8' [27], 'hi' => 'hi_8' [28], 'lo' => 'lo_8' [29]

Parsed with column specification:
cols(
  .default = col_character(),
  X30 = col_logical()
)

See spec(...) for full column specifications.
```

```
sliced <- slice(raw, 1:192)
countries <- sliced$ISO3
body <- raw %>%
  select(-ISO3, -Countries)
trimmed <- map_dfr(body, str_replace, pattern = ">?(\\d+)%", replacement = "\\1")
trimmed$Estimate
```

```
  [1] NA         NA         NA         NA         NA         NA
  [7] NA         NA         NA         NA         "26"       NA
 [13] NA         NA         NA         "95"       NA         "77"
 [19] NA         NA         "7"        NA         NA         "25"
 [25] NA         NA         "3"        NA         "95"       NA
 [31] "27"       NA         "1"        NA         NA         NA
 [37] "5"        NA         "8"        NA         "92"       NA
 [43] NA         "83"       NA         NA         NA         NA
 [49] NA         NA         NA         "28"       "1"        "4"
 [55] NA         NA         NA         NA         "4"        NA
 [61] NA         NA         NA         NA         "61"       NA
 [67] NA         NA         NA         NA         NA         NA
 [73] NA         NA         "61"       NA         NA         NA
 [79] NA         "2"        NA         NA         NA         NA
 [85] NA         NA         NA         "95"       NA         NA
 [91] NA         NA         NA         NA         NA         "43"
```

```
 [97] "5"        NA         NA         NA         NA         NA
[103] "37"       NA         "8"        NA         NA         NA
[109] NA         NA         NA         NA         NA         "2"
[115] NA         NA         NA         NA         "2"        NA
...
```

Excellent. We can now use `map_dfr` to convert the columns to numeric percentages using an anonymous function that we define inside the `map_dfr` call itself:

```r
raw <- read_csv("data/infant_hiv.csv", skip = 2, na = c("-"))
```

```
Warning: Missing column names filled in: 'X30' [30]

Warning: Duplicated column names deduplicated: 'Estimate' =>
'Estimate_1' [6], 'hi' => 'hi_1' [7], 'lo' => 'lo_1' [8], 'Estimate' =>
'Estimate_2' [9], 'hi' => 'hi_2' [10], 'lo' => 'lo_2' [11], 'Estimate' =>
'Estimate_3' [12], 'hi' => 'hi_3' [13], 'lo' => 'lo_3' [14], 'Estimate' =>
'Estimate_4' [15], 'hi' => 'hi_4' [16], 'lo' => 'lo_4' [17], 'Estimate' =>
'Estimate_5' [18], 'hi' => 'hi_5' [19], 'lo' => 'lo_5' [20], 'Estimate' =>
'Estimate_6' [21], 'hi' => 'hi_6' [22], 'lo' => 'lo_6' [23], 'Estimate' =>
'Estimate_7' [24], 'hi' => 'hi_7' [25], 'lo' => 'lo_7' [26], 'Estimate' =>
'Estimate_8' [27], 'hi' => 'hi_8' [28], 'lo' => 'lo_8' [29]

Parsed with column specification:
cols(
  .default = col_character(),
  X30 = col_logical()
)

See spec(...) for full column specifications.
```

```r
sliced <- slice(raw, 1:192)
countries <- sliced$ISO3
body <- raw %>%
  select(-ISO3, -Countries)
trimmed <- map_dfr(body, str_replace, pattern = ">?(\\d+)%", replacement = "\\1")
percents <- map_dfr(trimmed, function(col) as.numeric(col) / 100)
```

```
Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion
```

```
Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion
```

```r
head(percents)
```

```
# A tibble: 6 x 28
  Estimate    hi    lo Estimate_1  hi_1  lo_1 Estimate_2  hi_2  lo_2
     <dbl> <dbl> <dbl>      <dbl> <dbl> <dbl>      <dbl> <dbl> <dbl>
1       NA    NA    NA      NA    NA    NA         NA    NA    NA
2       NA    NA    NA      NA    NA    NA         NA    NA    NA
3       NA    NA    NA      NA    NA    NA          0.38  0.42  0.35
4       NA    NA    NA       0.03  0.04  0.02       0.05  0.07  0.04
5       NA    NA    NA      NA    NA    NA         NA    NA    NA
6       NA    NA    NA      NA    NA    NA         NA    NA    NA
# ... with 19 more variables: Estimate_3 <dbl>, hi_3 <dbl>, lo_3 <dbl>,
#   Estimate_4 <dbl>, hi_4 <dbl>, lo_4 <dbl>, Estimate_5 <dbl>,
#   hi_5 <dbl>, lo_5 <dbl>, Estimate_6 <dbl>, hi_6 <dbl>, lo_6 <dbl>,
#   Estimate_7 <dbl>, hi_7 <dbl>, lo_7 <dbl>, Estimate_8 <dbl>,
#   hi_8 <dbl>, lo_8 <dbl>, X30 <dbl>
```

27 warnings is rather a lot, so let's see what running `warnings()` produces right after the `as.numeric` call:

```r
raw <- read_csv("data/infant_hiv.csv", skip = 2, na = c("-"))
```

```
Warning: Missing column names filled in: 'X30' [30]

Warning: Duplicated column names deduplicated: 'Estimate' =>
'Estimate_1' [6], 'hi' => 'hi_1' [7], 'lo' => 'lo_1' [8], 'Estimate' =>
'Estimate_2' [9], 'hi' => 'hi_2' [10], 'lo' => 'lo_2' [11], 'Estimate' =>
'Estimate_3' [12], 'hi' => 'hi_3' [13], 'lo' => 'lo_3' [14], 'Estimate' =>
'Estimate_4' [15], 'hi' => 'hi_4' [16], 'lo' => 'lo_4' [17], 'Estimate' =>
'Estimate_5' [18], 'hi' => 'hi_5' [19], 'lo' => 'lo_5' [20], 'Estimate' =>
'Estimate_6' [21], 'hi' => 'hi_6' [22], 'lo' => 'lo_6' [23], 'Estimate' =>
'Estimate_7' [24], 'hi' => 'hi_7' [25], 'lo' => 'lo_7' [26], 'Estimate' =>
'Estimate_8' [27], 'hi' => 'hi_8' [28], 'lo' => 'lo_8' [29]

Parsed with column specification:
cols(
  .default = col_character(),
  X30 = col_logical()
)

See spec(...) for full column specifications.
```

```r
sliced <- slice(raw, 1:192)
countries <- sliced$ISO3
body <- raw %>%
  select(-ISO3, -Countries)
trimmed <- map_dfr(body, str_replace, pattern = ">?(\\d+)%", replacement = "\\1")
percents <- map_dfr(trimmed, function(col) as.numeric(col) / 100)
```

```
Warning in .f(.x[[i]], ...): NAs introduced by coercion
```

```
Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion
```

```
Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion

Warning in .f(.x[[i]], ...): NAs introduced by coercion
```

```
warnings()
```

Something is still not right. The first `Estimates` column looks all right, so let's have a look at the second column:

```
trimmed$hi
```

```
  [1] NA   NA   NA   NA   NA   NA   NA   NA   NA   NA   "35" NA   NA   NA  
 [15] NA   "95" NA   "89" NA   NA   "10" NA   NA   "35" NA   NA   "5"  NA  
 [29] "95" NA   "36" NA   "1"  NA   NA   NA   "6"  NA   "12" NA   "95" NA  
 [43] NA   "95" NA   NA   NA   NA   NA   NA   NA   "36" "1"  "4"  NA   NA  
 [57] NA   NA   "6"  NA   NA   NA   NA   NA   "77" NA   NA   NA   NA   NA  
 [71] NA   NA   NA   NA   "74" NA   NA   NA   NA   "2"  NA   NA   NA   NA  
 [85] NA   NA   NA   "95" NA   NA   NA   NA   NA   NA   NA   "53" "7"  NA  
 [99] NA   NA   NA   NA   "44" NA   "9"  NA   NA   NA   NA   NA   NA   NA  
[113] NA   "2"  NA   NA   NA   NA   "2"  NA   NA   "69" NA   "7"  NA   NA  
[127] NA   "1"  NA   NA   NA   NA   NA   NA   "1"  NA   NA   NA   "95" NA  
[141] NA   "75" NA   NA   NA   NA   NA   NA   "13" NA   NA   NA   NA   NA  
[155] NA   NA   NA   NA   NA   NA   NA   NA   "11" NA   NA   NA   NA   "1" 
[169] NA   NA   NA   "12" NA   NA   NA   NA   NA   NA   "9"  "95" NA   NA  
[183] "16" NA   NA   "1"  NA   NA   NA   NA   "70" NA   ""   ""   "hi" "30"
[197] "29" NA   "32" "2"  NA   "2"  "12" NA   "9"  "89" "22" "23" ""   ""  
[211] ""   ""   ""   ""  
```

Where are the empty strings toward the end of `trimmed$hi` coming from? Let's backtrack by examining the `hi` column of each of our intermediate variables interactively in the console…

…and there's our bug. We are creating a variable called `sliced` that has only the rows we care about, but then using the full table in `raw` to create `body`. It's a simple mistake, and one that could easily have slipped by us. Here is our revised script:

```
raw <- read_csv("data/infant_hiv.csv", skip = 2, na = c("-"))
```

```
Warning: Missing column names filled in: 'X30' [30]

Warning: Duplicated column names deduplicated: 'Estimate' =>
'Estimate_1' [6], 'hi' => 'hi_1' [7], 'lo' => 'lo_1' [8], 'Estimate' =>
'Estimate_2' [9], 'hi' => 'hi_2' [10], 'lo' => 'lo_2' [11], 'Estimate' =>
'Estimate_3' [12], 'hi' => 'hi_3' [13], 'lo' => 'lo_3' [14], 'Estimate' =>
'Estimate_4' [15], 'hi' => 'hi_4' [16], 'lo' => 'lo_4' [17], 'Estimate' =>
'Estimate_5' [18], 'hi' => 'hi_5' [19], 'lo' => 'lo_5' [20], 'Estimate' =>
```

```
'Estimate_6' [21], 'hi' => 'hi_6' [22], 'lo' => 'lo_6' [23], 'Estimate' =>
'Estimate_7' [24], 'hi' => 'hi_7' [25], 'lo' => 'lo_7' [26], 'Estimate' =>
'Estimate_8' [27], 'hi' => 'hi_8' [28], 'lo' => 'lo_8' [29]
```

```
Parsed with column specification:
cols(
  .default = col_character(),
  X30 = col_logical()
)
```

```
See spec(...) for full column specifications.
```

```r
sliced <- slice(raw, 1:192)
countries <- sliced$ISO3
body <- sliced %>%
  select(-ISO3, -Countries)
trimmed <- map_dfr(body, str_replace, pattern = ">?(\\d+)%", replacement = "\\1")
percents <- map_dfr(trimmed, function(col) as.numeric(col) / 100)
```

and here are the checks on the head:

```r
head(percents)
```

```
# A tibble: 6 x 28
  Estimate    hi    lo Estimate_1  hi_1  lo_1 Estimate_2  hi_2  lo_2
     <dbl> <dbl> <dbl>      <dbl> <dbl> <dbl>      <dbl> <dbl> <dbl>
1       NA    NA    NA      NA    NA    NA         NA    NA    NA
2       NA    NA    NA      NA    NA    NA         NA    NA    NA
3       NA    NA    NA      NA    NA    NA          0.38  0.42  0.35
4       NA    NA    NA       0.03  0.04  0.02       0.05  0.07  0.04
5       NA    NA    NA      NA    NA    NA         NA    NA    NA
6       NA    NA    NA      NA    NA    NA         NA    NA    NA
# ... with 19 more variables: Estimate_3 <dbl>, hi_3 <dbl>, lo_3 <dbl>,
#   Estimate_4 <dbl>, hi_4 <dbl>, lo_4 <dbl>, Estimate_5 <dbl>,
#   hi_5 <dbl>, lo_5 <dbl>, Estimate_6 <dbl>, hi_6 <dbl>, lo_6 <dbl>,
#   Estimate_7 <dbl>, hi_7 <dbl>, lo_7 <dbl>, Estimate_8 <dbl>,
#   hi_8 <dbl>, lo_8 <dbl>, X30 <dbl>
```

and tail:

```r
tail(percents)
```

```
# A tibble: 6 x 28
  Estimate    hi    lo Estimate_1  hi_1  lo_1 Estimate_2  hi_2  lo_2
     <dbl> <dbl> <dbl>      <dbl> <dbl> <dbl>      <dbl> <dbl> <dbl>
1    NA      NA    NA         NA    NA    NA         NA    NA    NA
2    NA      NA    NA         NA    NA    NA         NA    NA    NA
3    NA      NA    NA         NA    NA    NA          0.31  0.37  0.26
4    NA      NA    NA         NA    NA    NA         NA    NA    NA
```

```
5     0.59   0.7  0.53       0.27  0.32  0.24       0.7   0.84  0.63
6    NA     NA   NA          0.12  0.15  0.1        0.23  0.28  0.2
# ... with 19 more variables: Estimate_3 <dbl>, hi_3 <dbl>, lo_3 <dbl>,
#   Estimate_4 <dbl>, hi_4 <dbl>, lo_4 <dbl>, Estimate_5 <dbl>,
#   hi_5 <dbl>, lo_5 <dbl>, Estimate_6 <dbl>, hi_6 <dbl>, lo_6 <dbl>,
#   Estimate_7 <dbl>, hi_7 <dbl>, lo_7 <dbl>, Estimate_8 <dbl>,
#   hi_8 <dbl>, lo_8 <dbl>, X30 <dbl>
```

Comparing this to the raw data file convinces us that yes, we are now converting the percentages properly, which means we are halfway home.

5.4 How do I reorganize the columns?

We now have numeric values in `percents` and corresponding ISO3 codes in `countries`. What we do *not* have is tidy data: countries are not associated with records, years are not recorded at all, and the column headers for `percents` have mostly been manufactured for us by R. We must now sew these parts together like Dr. Frankenstein's trusty assistant Igor (who, like so many lab assistants, did most of the actual work but was given only crumbs of credit).

We could write a loop to grab three columns at a time and relabel them, but a more concise solution makes use of a pair of functions called `pivot_longer` and `separate`. `pivot_longer` takes multiple columns and collapses them into two, one of which holds a key and the other of which holds a value. To show how it works, let's create a small tibble by hand using the function `tribble`. The first few arguments use `~` as a prefix operator to define columns names, and all of the other values are then put into a tibble with those columns:

```r
small <- tribble(
  ~ISO, ~est, ~hi, ~lo,
  'ABC', 0.25, 0.3, 0.2,
  'DEF', 0.55, 0.6, 0.5
)
small
```

```
# A tibble: 2 x 4
  ISO     est    hi    lo
  <chr> <dbl> <dbl> <dbl>
1 ABC    0.25   0.3   0.2
2 DEF    0.55   0.6   0.5
```

and then rearrange the data in `est`, `hi`, and `lo`:

```r
small %>%
  pivot_longer(cols = c(est, hi, lo), names_to = "kind", values_to = "reported")
```

```
# A tibble: 6 x 3
  ISO   kind  reported
```

```
  <chr> <chr>    <dbl>
1 ABC   est       0.25
2 ABC   hi        0.3
3 ABC   lo        0.2
4 DEF   est       0.55
5 DEF   hi        0.6
6 DEF   lo        0.5
```

The `cols` parameter tells `pivot_longer` which columns to rearrange. The new column `names_to` gets the old column titles (in our case, `est`, `hi`, and `lo`), while the new column `values_to` gets the values. The result is a table which is longer and narrower than the original, which is what inspired the function's name. (Previous versions of the tidyverse called this function `gather`, but users reported that they found the name confusing.)

The other tool we need to rearrange our data is `separate`, which splits one column into two. For example, if we have the year and the heading type in a single column:

```
single <- tribble(
  ~combined, ~value,
  '2009-est', 123,
  '2009-hi',  456,
  '2009-lo',  789,
  '2010-est', 987,
  '2010-hi',  654,
  '2010-lo',  321
)
single
```

```
# A tibble: 6 x 2
  combined value
  <chr>    <dbl>
1 2009-est   123
2 2009-hi    456
3 2009-lo    789
4 2010-est   987
5 2010-hi    654
6 2010-lo    321
```

we can get the year and the heading into separate columns by separating on the `-` character:

```
single %>%
  separate(combined, sep = "-", c("year", "kind"))
```

```
# A tibble: 6 x 3
  year  kind  value
```

```
  <chr> <chr> <dbl>
1 2009  est     123
2 2009  hi      456
3 2009  lo      789
4 2010  est     987
5 2010  hi      654
6 2010  lo      321
```

Our strategy is therefore going to be:

1. Replace the double column headers in the existing data with a single header that combines the year with the kind.
2. Gather the data so that the year-kind values are in a single column.
3. Split that column.

We've seen the tools we need for the second and third step; the first involves a little bit of list manipulation. Let's start by repeating `"est"`, `"hi"`, and `"lo"` as many times as we need them:

```
num_years <- 1 + 2017 - 2009
kinds <- rep(c("est", "hi", "lo"), num_years)
kinds
```

```
 [1] "est" "hi"  "lo"  "est" "hi"  "lo"  "est" "hi"  "lo"  "est" "hi"
[12] "lo"  "est" "hi"  "lo"  "est" "hi"  "lo"  "est" "hi"  "lo"  "est"
[23] "hi"  "lo"  "est" "hi"  "lo"
```

As you can probably guess from its name, `rep` repeats things a specified number of times, and as noted previously, a vector of vectors is flattened into a single vector, so what an innocent might expect to be `c(c('est', 'hi', 'lo'), c('est', 'hi', 'lo))` automatically becomes `c('est', 'hi', 'lo', 'est', 'hi', 'lo)`.

What about the years? We want to wind up with:

```
c("2009", "2009" "2009", "2010", "2010", "2010", ...)
```

i.e., with each year repeated three times. `rep` won't do this, but we can get there with `map`:

```
years <- map(2009:2017, rep, 3)
years
```

```
[[1]]
[1] 2009 2009 2009

[[2]]
[1] 2010 2010 2010

[[3]]
[1] 2011 2011 2011
```

```
[[4]]
[1] 2012 2012 2012

[[5]]
[1] 2013 2013 2013

[[6]]
[1] 2014 2014 2014

[[7]]
[1] 2015 2015 2015

[[8]]
[1] 2016 2016 2016

[[9]]
[1] 2017 2017 2017
```

That's almost right, but `map` hasn't flattened the list for us. Luckily, we can use `unlist` to do that:

```
years <- map(2009:2017, rep, 3) %>% unlist()
years
```

```
 [1] 2009 2009 2009 2010 2010 2010 2011 2011 2011 2012 2012 2012 2013 2013
[15] 2013 2014 2014 2014 2015 2015 2015 2016 2016 2016 2017 2017 2017
```

We can now combine the years and kinds by pasting the two vectors together with `"-"` as a separator:

```
headers <- paste(years, kinds, sep = "-")
headers
```

```
 [1] "2009-est" "2009-hi"  "2009-lo"  "2010-est" "2010-hi"  "2010-lo"
 [7] "2011-est" "2011-hi"  "2011-lo"  "2012-est" "2012-hi"  "2012-lo"
[13] "2013-est" "2013-hi"  "2013-lo"  "2014-est" "2014-hi"  "2014-lo"
[19] "2015-est" "2015-hi"  "2015-lo"  "2016-est" "2016-hi"  "2016-lo"
[25] "2017-est" "2017-hi"  "2017-lo"
```

Remember, everything in R is a vector and most functions are vectorized, so if we give `paste` two vectors to combine, it will paste corresponding elements together and give us a vector result.

Let's use this to relabel the columns of `percents` (which holds our data without the ISO country codes):

```
names(percents) <- headers
```

```
Warning: The `names` must have length 28, not 27.
```

```
This warning is displayed once per session.
```

```r
percents
```

```
# A tibble: 192 x 28
   `2009-est` `2009-hi` `2009-lo` `2010-est` `2010-hi` `2010-lo` `2011-est`
        <dbl>     <dbl>     <dbl>      <dbl>     <dbl>     <dbl>      <dbl>
 1         NA        NA        NA      NA        NA        NA         NA
 2         NA        NA        NA      NA        NA        NA         NA
 3         NA        NA        NA      NA        NA        NA           0.38
 4         NA        NA        NA       0.03      0.04      0.02        0.05
 5         NA        NA        NA      NA        NA        NA         NA
 6         NA        NA        NA      NA        NA        NA         NA
 7         NA        NA        NA      NA        NA        NA           0.13
 8         NA        NA        NA      NA        NA        NA         NA
 9         NA        NA        NA      NA        NA        NA         NA
10         NA        NA        NA      NA        NA        NA         NA
# ... with 182 more rows, and 21 more variables: `2011-hi` <dbl>,
#   `2011-lo` <dbl>, `2012-est` <dbl>, `2012-hi` <dbl>, `2012-lo` <dbl>,
#   `2013-est` <dbl>, `2013-hi` <dbl>, `2013-lo` <dbl>, `2014-est` <dbl>,
#   `2014-hi` <dbl>, `2014-lo` <dbl>, `2015-est` <dbl>, `2015-hi` <dbl>,
#   `2015-lo` <dbl>, `2016-est` <dbl>, `2016-hi` <dbl>, `2016-lo` <dbl>,
#   `2017-est` <dbl>, `2017-hi` <dbl>, `2017-lo` <dbl>, NA <dbl>
```

This example shows that `names(table)` doesn't just give us a list of column names: it gives us something we can assign to when we want to rename those columns. This example also shows us that `percents` has the wrong number of columns. Inspecting the tibble in the console, we see that the last column is full of NAs:

```r
percents[, ncol(percents)]
```

```
# A tibble: 192 x 1
      NA
   <dbl>
 1    NA
 2    NA
 3    NA
 4    NA
 5    NA
 6    NA
 7    NA
 8    NA
 9    NA
10    NA
# ... with 182 more rows
```

```r
all(is.na(percents[,ncol(percents)]))
```

```
[1] TRUE
```

Let's relabel our data again and then drop the empty column. (There are other ways to do this, but I find steps easier to read after the fact this way.)

```r
headers <- c(headers, "empty")
names(percents) <- headers
percents <- select(percents, -empty)
percents
```

```
# A tibble: 192 x 27
   `2009-est` `2009-hi` `2009-lo` `2010-est` `2010-hi` `2010-lo` `2011-est`
        <dbl>     <dbl>     <dbl>      <dbl>     <dbl>     <dbl>      <dbl>
 1         NA        NA        NA      NA        NA        NA         NA
 2         NA        NA        NA      NA        NA        NA         NA
 3         NA        NA        NA      NA        NA        NA          0.38
 4         NA        NA        NA       0.03      0.04      0.02       0.05
 5         NA        NA        NA      NA        NA        NA         NA
 6         NA        NA        NA      NA        NA        NA         NA
 7         NA        NA        NA      NA        NA        NA          0.13
 8         NA        NA        NA      NA        NA        NA         NA
 9         NA        NA        NA      NA        NA        NA         NA
10         NA        NA        NA      NA        NA        NA         NA
# ... with 182 more rows, and 20 more variables: `2011-hi` <dbl>,
#   `2011-lo` <dbl>, `2012-est` <dbl>, `2012-hi` <dbl>, `2012-lo` <dbl>,
#   `2013-est` <dbl>, `2013-hi` <dbl>, `2013-lo` <dbl>, `2014-est` <dbl>,
#   `2014-hi` <dbl>, `2014-lo` <dbl>, `2015-est` <dbl>, `2015-hi` <dbl>,
#   `2015-lo` <dbl>, `2016-est` <dbl>, `2016-hi` <dbl>, `2016-lo` <dbl>,
#   `2017-est` <dbl>, `2017-hi` <dbl>, `2017-lo` <dbl>
```

It's time to put the country codes back on the table, move the year and kind from column headers to a column with `pivot_longer`, and then split that column with `separate`:

```r
final <- percents %>%
  mutate(country = countries) %>%
  pivot_longer(-country, names_to = "year_kind", values_to = "reported") %>%
  separate(year_kind, c("year", "kind"), sep = "-")
final
```

```
# A tibble: 5,184 x 4
   country year  kind  reported
   <chr>   <chr> <chr>    <dbl>
 1 AFG     2009  est         NA
 2 AFG     2009  hi          NA
 3 AFG     2009  lo          NA
```

```
 4 AFG     2010  est         NA
 5 AFG     2010  hi          NA
 6 AFG     2010  lo          NA
 7 AFG     2011  est         NA
 8 AFG     2011  hi          NA
 9 AFG     2011  lo          NA
10 AFG     2012  est         NA
# ... with 5,174 more rows
```

Here's everything in one function:

```
clean_infant_hiv <- function(filename, num_rows) {
  # Read raw data.
  raw <- read_csv(filename, skip = 2, na = c("-")) %>%
    slice(1:num_rows)

  # Save the country names to reattach later.
  countries <- raw$ISO3

  # Convert data values to percentages.
  percents <- raw %>%
    select(-ISO3, -Countries) %>%
    slice(1:num_rows) %>%
    map_dfr(str_replace, pattern = ">?(\\d+)%", replacement = "\\1") %>%
    map_dfr(function(col) as.numeric(col) / 100)

  # Change the headers on the percentages.
  num_years <- 1 + 2017 - 2009
  kinds <- rep(c("est", "hi", "lo"), num_years)
  years <- map(2009:2017, rep, 3) %>% unlist()
  headers <- c(paste(years, kinds, sep = "-"), "empty")
  names(percents) <- headers

  # Stitch everything back together.
  percents %>%
    mutate(country = countries) %>%
    pivot_longer(-country, names_to = "year_kind", values_to = "reported") %>%
    separate(year_kind, c("year", "kind"), sep = "-")
}

clean_infant_hiv("data/infant_hiv.csv", 192)
```

```
Warning: Missing column names filled in: 'X30' [30]
```

```
Warning: Duplicated column names deduplicated: 'Estimate' =>
'Estimate_1' [6], 'hi' => 'hi_1' [7], 'lo' => 'lo_1' [8], 'Estimate' =>
'Estimate_2' [9], 'hi' => 'hi_2' [10], 'lo' => 'lo_2' [11], 'Estimate' =>
```

```
'Estimate_3' [12], 'hi' => 'hi_3' [13], 'lo' => 'lo_3' [14], 'Estimate' =>
'Estimate_4' [15], 'hi' => 'hi_4' [16], 'lo' => 'lo_4' [17], 'Estimate' =>
'Estimate_5' [18], 'hi' => 'hi_5' [19], 'lo' => 'lo_5' [20], 'Estimate' =>
'Estimate_6' [21], 'hi' => 'hi_6' [22], 'lo' => 'lo_6' [23], 'Estimate' =>
'Estimate_7' [24], 'hi' => 'hi_7' [25], 'lo' => 'lo_7' [26], 'Estimate' =>
'Estimate_8' [27], 'hi' => 'hi_8' [28], 'lo' => 'lo_8' [29]
```

```
Parsed with column specification:
cols(
  .default = col_character(),
  X30 = col_logical()
)
```

```
See spec(...) for full column specifications.
```

```
Warning: Expected 2 pieces. Missing pieces filled with `NA` in 192 rows
[28, 56, 84, 112, 140, 168, 196, 224, 252, 280, 308, 336, 364, 392, 420,
448, 476, 504, 532, 560, ...].
```

```
# A tibble: 5,376 x 4
   country year  kind  reported
   <chr>   <chr> <chr>    <dbl>
 1 AFG     2009  est         NA
 2 AFG     2009  hi          NA
 3 AFG     2009  lo          NA
 4 AFG     2010  est         NA
 5 AFG     2010  hi          NA
 6 AFG     2010  lo          NA
 7 AFG     2011  est         NA
 8 AFG     2011  hi          NA
 9 AFG     2011  lo          NA
10 AFG     2012  est         NA
# ... with 5,366 more rows
```

We're done, and we have learned a lot of R, but what we have also learned is that we make mistakes, and that those mistakes can easily slip past us. It would be hubris to believe that we will not make more as we continue to clean this data. What will guide us safely through these dark caverns and back into the light of day?

The answer is testing. We must test our assumptions, test our code, test our very *being* if we are to advance. R provides tools for this purpose, but in order to use them, we must venture into the greater realm of packaging in R.

5.5 How do I create a package?

The more software you write, the more you realize that a programming language is mostly a way to build and combine software packages. Every widely-used language now has an online repository from which people can download and install packages, and sharing ours is a great way to contribute to the community that has helped us on our journey.

5.5.1 CRAN and Alternatives

CRAN, the Comprehensive R Archive Network, is the best place to find the packages you need. CRAN's famously strict rules ensure that packages run for everyone, but also makes package development a little more onerous than it might be. You can also share packages directly from GitHub, which many people do while packages are still in development. We will explore this in more detail below.

We cannot turn this tutorial into an R package because we're building it as a website, not as a package. Instead, we will create an R package called `unicefdata` to hold cleaned-up copies of some HIV/AIDS data and maternal health data from UNICEF.

An R package must contain the following files:

- The text file `DESCRIPTION` (with no suffix) describes what the package does, who wrote it, and what other packages it requires to run. We will edit its contents as we go along.
- `NAMESPACE`, (whose name also has no suffix) contains the names of everything exported from the package (i.e., everything that is visible to the outside world). As we will see, we should leave its management in the hands of RStudio and the `devtools` package we will meet below.
- Just as `.gitignore` tells Git what files in a project to ignore, `.Rbuildignore` tells R which files to include or not include in the package.
- All of the R source for our package must go in a directory called `R`; subdirectories below this are not allowed.
- As you would expect from its name, the optional `data` directory contains any data we have put in our package. In order for it to be loadable as part of the package, the data must be saved in R's custom `.rda` format. We will see how to do this below.
- Manual pages go in the `man` directory. The bad news is that they have to be in a sort-of-LaTeX format that is only a bit less obscure than the runes inscribed on the ancient dagger your colleague brought back from her latest archeological dig. The good news is, we can embed Markdown comments in our source code and use a tool called `roxygen2` to extract them and translate them into the format that R packages require.
- The `tests` directory holds the package's unit tests. It should contain files with names like test_some_feature.R, which should in turn contain

functions named test_something_specific. We'll have a closer look at these in Chapter 8.

We can type all of this in if we want, but R has a very useful package called `usethis` to help us create and maintain packages. To use it, we load `usethis` in the console with `library(usethis)` and use `usethis::create_package` with the path to the new package directory as an argument:

```r
usethis::create_package('~/unicefdata')
```

```
  Creating '/Users/gvwilson/unicefdata/'
  Setting active project to '/Users/gvwilson/unicefdata'
  Creating 'R/'
  Writing 'DESCRIPTION'
Package: unicefdata
Title: What the Package Does (One Line, Title Case)
Version: 0.0.0.9000
Authors@R (parsed):
    * First Last <first.last@example.com> [aut, cre] (<https://orcid.org/YOUR-ORCID-ID>)
Description: What the package does (one paragraph).
License: What license it uses
Encoding: UTF-8
LazyData: true
  Writing 'NAMESPACE'
  Writing 'unicefdata.Rproj'
  Adding '.Rproj.user' to '.gitignore'
  Adding '^unicefdata\\.Rproj$', '^\\.Rproj\\.user$' to '.Rbuildignore'
  Opening '/Users/gvwilson/unicefdata/' in new RStudio session
  Setting active project to '<no active project>'
```

Every well-behaved package should have a README file, a license, and a Code of Conduct, so we will ask `usethis` to add those *in the RStudio session that just opened up* (rather than in the one in which this tutorial is being written, and yes, imprecations were uttered upon making that mistake for the second time):

```r
usethis::use_readme_md()
```

```
  Setting active project to '/Users/gvwilson/unicefdata'
  Writing 'README.md'
  Modify 'README.md'
```

```r
usethis::use_mit_license(name="UNICEF Data")
```

```
  Setting License field in DESCRIPTION to 'MIT + file LICENSE'
  Writing 'LICENSE.md'
  Adding '^LICENSE\\.md$' to '.Rbuildignore'
  Writing 'LICENSE'
```

`use_mit_license` creates two files: `LICENSE` and `LICENSE.md`. The rules for R packages require the former, but GitHub expects the latter.

```r
usethis::use_code_of_conduct()
```

```
Setting active project to '/Users/gvwilson/tidynomicon'
Don't forget to describe the code of conduct in your README:
 Please note that the 'placeholder' project is released with a
 [Contributor Code of Conduct](CODE_OF_CONDUCT.md).
 By contributing to this project, you agree to abide by its terms.
```

```
Writing 'CODE_OF_CONDUCT.md'
Adding '^CODE_OF_CONDUCT\\.md$' to '.Rbuildignore'
Don't forget to describe the code of conduct in your README:
 Please note that the 'unicefdata' project is released with a
 [Contributor Code of Conduct](CODE_OF_CONDUCT.md).
 By contributing to this project, you agree to abide by its terms.
 [Copied to clipboard]
```

We then edit `README.md` to be:

```
# unicefdata

unicefdata is a small R data package created for tutorial purposes.
See `data/README.md` for the provenance of the original data.

## Installation

You can install unicefdata from GitHub with `devtools::install_github("gvwilson/unicef
```

and similarly edit `DESCRIPTION` so that it contains:

```
Package: unicefdata
Title: Small UNICEF Dataset for Tutorial Purposes
Version: 0.0.0.9000
Authors@R:
    person(given = "Greg",
           family = "Wilson",
           role = c("aut", "cre"),
           email = "gvwilson@third-bit.com",
           comment = c(ORCID = "0000-0001-8659-8979"))
Description: This package demonstrates how to share small datasets in R.
License: MIT + file LICENSE
Encoding: UTF-8
LazyData: true
```

We can now go to the `Build` tab in RStudio and run `Check` to make sure our empty package is judged sane by our strict, yet impartial, machine.

We can now put the function we wrote to clean up the infant HIV data in a file called `R/clean_infant_hiv.R` either by using `File...New` in RStudio or by running `usethis::use_r('clean_infant_hiv.R')` (which always creates the file in the `R` directory). We do *not* include the line that actually runs the function, since we don't want that to happen every time this file is loaded. We also fix the number of valid rows inside the function rather than passing it as a parameter, since it's highly unlikely that users will know or guess the value 192:

```
clean_infant_hiv <- function(filename) {
  # Indexes into the specific file.
  header_rows <- 2
  num_rows <- 192
  first_year <- 2009
  last_year <- 2017

  # Read raw data.
  raw <- read_csv(filename, skip = header_rows, na = c("-")) %>%
    slice(1:num_rows)

  # Save the country names to reattach later.
  countries <- raw$ISO3

  # Convert data values to percentages.
  percents <- raw %>%
    select(-ISO3, -Countries) %>%
    slice(1:num_rows) %>%
    map_dfr(str_replace, pattern = ">?(\\d+)%", replacement = "\\1") %>%
    map_dfr(function(col) as.numeric(col) / 100)

  # Change the headers on the percentages.
  num_years <- 1 + last_year - first_year
  kinds <- rep(c("est", "hi", "lo"), num_years)
  years <- map(first_year:last_year, rep, 3) %>% unlist()
  headers <- c(paste(years, kinds, sep = "-"), "empty")
  names(percents) <- headers

  # Stitch everything back together.
  percents %>%
    mutate(country = countries) %>%
    pivot_longer(-country, names_to = "year_kind", values_to = "reported") %>%
    separate(year_kind, c("year", "kind"), sep = "-")
}
```

5.6 How can I document the contents of a package?

`Build...Check` runs a lot more checks now because we have some actual code for it to look at. It also produces some warnings:

```
  R CMD check results                  unicefdata 0.0.0.9000
Duration: 19.5s

  checking for missing documentation entries ... WARNING
  Undocumented code objects:
    'infant_hiv'
  All user-level objects in a package should have documentation entries.
  See chapter 'Writing R documentation files' in the 'Writing R
  Extensions' manual.

  checking R code for possible problems ... NOTE
  infant_hiv: no visible global function definition for '%>%'
  infant_hiv: no visible global function definition for 'read_csv'
  infant_hiv: no visible global function definition for 'slice'
  infant_hiv: no visible global function definition for 'select'
  infant_hiv: no visible binding for global variable 'ISO3'
  infant_hiv: no visible binding for global variable 'Countries'
  infant_hiv: no visible global function definition for 'map_dfr'
  infant_hiv: no visible binding for global variable 'str_replace'
  infant_hiv: no visible global function definition for 'map'
  infant_hiv: no visible global function definition for 'mutate'
  infant_hiv: no visible global function definition for 'gather'
  infant_hiv: no visible binding for global variable 'country'
  infant_hiv: no visible global function definition for 'separate'
  infant_hiv: no visible binding for global variable 'year_kind'
  Undefined global functions or variables:
    %>% Countries ISO3 country gather map map_dfr mutate read_csv select
    separate slice str_replace year_kind

0 errors   | 1 warning   | 1 note
Error: R CMD check found WARNINGs
Execution halted
```

A little documentation seems like a fair request. For this, we turn to Hadley Wickham's *R Packages* and Karl Broman's "R package primer" for advice on writing roxygen2 documentation. We then return to our source file and prefix our existing code with this:

```r
#' Tidy up the infant HIV data set.
#'
#' @param filename path to source file
```

```r
#'
#' @return a tibble of tidy data
#'
#' @export

infant_hiv <- function(filename) {
  …all the code from before…
}
```

roxygen2 processes comment lines that start with `#'` (hash followed by single quote). Putting a comment block right before a function associates that documentation with that function, so here we are saying that:

- the function has a single parameter called `filename`;
- it returns a tibble of tidy data; and
- we want it exported (i.e., we want it to be visible outside the package).

Our function is now documented, but when we run `Check`, we still get a warning. After a bit more searching and experimentation, we discover that we need to load the `devtools` package and run `devtools::document()` in the console to regenerate documentation—it isn't done automatically.

```r
devtools::document()
```

```
Updating unicefdata documentation
Updating roxygen version in /Users/gvwilson/unicefdata/DESCRIPTION
Writing NAMESPACE
Loading unicefdata
Writing NAMESPACE
Writing clean_infant_hiv.Rd
```

Another check confirms that our function is now documented. `NAMESPACE` now contains:

```
# Generated by roxygen2: do not edit by hand

export(infant_hiv)
```

The `export` directive signals that we want `infant_hiv` to be visible outside the package, and the comment helpfully reminds us that we shouldn't edit this file ourselves, but should instead trust our tools to do the work for us. As for `man/clean_infant_hiv.Rd`, it shows us more clearly than mere words ever could why we want to use `roxygen2`:

```
% Generated by roxygen2: do not edit by hand
% Please edit documentation in R/clean_infant_hiv.R
\name{infant_hiv}
\alias{infant_hiv}
\title{Tidy up the infant HIV data set.}
```

```
\usage{
infant_hiv(filename)
}
\arguments{
\item{filename}{path to source file}
}
\value{
a tibble of tidy data
}
\description{
Tidy up the infant HIV data set.
}
```

5.7 How can my package import what it needs?

Running the build again still gives us undefined function warnings for `read_csv`, `%>%`, and many others. The reason is that R packages are distributed as compiled bytecode, *not* as source code (which is how Python does it). When a package is built, R loads and checks the code, then saves the corresponding instructions. Our R files should therefore define functions, not run commands immediately, because if they do the latter, those commands will be executed every time the script loads, which is probably not what users will want.

As a side effect, this means that if a package uses `load(something)`, then that `load` command is executed *while the package is being compiled*, and *not* while the compiled package is being loaded by a user after distribution. Thus, this simple and rather pointless "package":

```r
library(stringr)

sr <- function(text, pattern, replacement) {
  str_replace(text, pattern, replacement)
}
```

probably won't work when it's loaded by a user, because `stringr` may not be in memory on the user's machine at the time `str_replace` is called.

How then can our packages use libraries? One way is to add import directives to the documentation for our functions to tell R what we depend on:

```r
#' @import dplyr
#' @importFrom magrittr %>%
```

The safer way is to use fully-qualified names such as `stringr::str_replace` every time we call a function defined somewhere outside our package, as in:

```r
  percents %>%
    dplyr::mutate(country = countries) %>%
```

```
    tidyr::pivot_longer(cols = c(est, hi, lo), names_to = "kind", values_to = "reported")
    tidyr::separate(year_kind, c("year", "kind"))
```

This changes the error to one that is slightly more confusing:

```
  R CMD check results                  unicefdata 0.0.0.9000
Duration: 21.3s

  checking dependencies in R code ... WARNING
  '::' or ':::' imports not declared from:
    ‘dplyr’ ‘purrr’ ‘reader’ ‘stringr’ ‘tidyr’

  checking R code for possible problems ... NOTE
  infant_hiv: no visible global function definition for ‘%>%’
  infant_hiv: no visible binding for global variable ‘ISO3’
  infant_hiv: no visible binding for global variable ‘Countries’
  infant_hiv: no visible binding for global variable ‘purrr’
  infant_hiv: no visible global function definition for ‘map’
  infant_hiv: no visible binding for global variable ‘country’
  infant_hiv: no visible binding for global variable ‘year_kind’
  Undefined global functions or variables:
    %>% Countries ISO3 country map purrr year_kind
```

More searching, more experimentation, and finally we add this to the `DESCRIPTION` file:

```
Imports:
    readr (>= 1.1.0),
    dplyr (>= 0.7.0),
    magrittr (>= 1.5.0),
    purrr (>= 0.2.0),
    rlang (>= 0.3.0),
    stringr (>= 1.3.0),
    tidyr (>= 0.8.3)
```

The `Imports` field in `DESCRIPTION` actually has nothing to do with importing functions; it just ensures that those packages are installed when this package is. As for the version numbers in parentheses, we got those by running `packageVersion("readr")` and similar commands inside RStudio and then rounding off.

But that is still not enough, because the check still complains about `%>%`. Luckily, others have ventured into this poorly-lit basement before us and lived to tell the tale. `usethis::use_pipe()` at the console creates a file called `R/utils-pipe.R` containing:

```
#' Pipe operator
#'
```

```
#' See \code{magrittr::\link[magrittr:pipe]{\%>\%}} for details.
#'
#' @name %>%
#' @rdname pipe
#' @keywords internal
#' @export
#' @importFrom magrittr %>%
#' @usage lhs \%>\% rhs
NULL
```

```
NULL
```

which is all the documentation we need to satisfy the check.

All right: are we done now? No, we are not:

```
checking R code for possible problems ... NOTE
tidy_infant_hiv: no visible binding for global variable 'ISO3'
tidy_infant_hiv: no visible binding for global variable 'Countries'
tidy_infant_hiv: no visible binding for global variable 'country'
tidy_infant_hiv: no visible binding for global variable 'year'
```

This is annoying but understandable. When the package builder is checking our code, it has no idea what columns are going to be in our data frames, so it has no way to know if `ISO3` or `Countries` will cause a problem. However, this is just a `NOTE`, not an `ERROR`, so we can try running "Build…Install and Restart" to build our package, re-start our R session (so that memory is clean), and load our newly-created package, and then run `infant_hiv("~/tidynomicon/data/infant_hiv.csv")`.

Our data loads, so we return to the problem of "variables" that are actually column names. A bit more searching online tells us to add this to the documentation block for our function:

```
#' @importFrom rlang .data
```

and then modify the calls that use naked column names to look like:

```
    dplyr::select(-.data$ISO3, -.data$Countries)
```

What is this `.data` that we have invoked? Typing `?rlang::.data` gives us the answer: it is a pronoun that allows us to be explicit when we refer to an object inside the data. Adding this—i.e., being explicity that `country` is a column of `.data` rather than an undefined variable—finally (finally) gives us a clean build.

5.8 How can I add data to a package?

But we are not done, because we are never *truly* done, any more than we are ever truly safe. We still need to add our cleaned-up data to our package and document the package as a whole. There are three steps to this.

First, we put the raw data file into `inst/extdata/infant_hiv.csv` Data that *isn't* meant to be loaded directly into are should go in `inst/extdata`. The first part of the directory name, `inst`, is short for "install": when the package is installed, everything in this directory is bumped up a level and put in the installation directory. Thus, the installation directory will get a sub-directory called `extdata` (for "external data"), and that can hold whatever we want.

Next, we use `clean_infant_hiv` to put a tidy version of this data in a variable called `infant_hiv`, then call `usethis::use_data(infant_hiv)` to store the tibble in `data/infant_hiv.rda`. We *must* save the data as `.rda`, not as (for example) `.rds` or `.csv`; only `.rda` will be automatically loaded as part of the project. (We can write this file using `save` if we want, but `usethis::use_data` automatically uses the right format and location.)

We now create a file called `R/infant_hiv.R` to hold documentation about the dataset:

```
#' Tidied infant HIV data.
#'
#' This tidy data is derived from the `infant_hiv.csv` file, which in turn is
#' derived from an Excel spreadsheet provided by UNICEF - see the README.md file
#' in the raw data directory for details.
#'
#' @format A data frame
#' \describe{
#'   \item{country}{Country reporting (ISO3 code)}
#'   \item{kind}{Type of report (low, estimate, high)}
#'   \item{reported}{Value reported (may be NA)}
#'   \item{year}{Year reported}
#' }
"infant_hiv"
```

Everything except the last line is a roxygen2 comment block that describes the data in plain language, then uses some tags and directives to document its format and fields. (Note that we have also documented our data in `inst/extdata/README.md`, but that focuses on the format and meaning of the raw data, not the cleaned-up version.)

The last line is the string `"infant_hiv"`, i.e., the name of the dataset. We will create one placeholder R file like this for each of our datasets, and each will have that dataset's name as the thing being documented.

Let's run a check:

```
Warning: package needs dependence on R (>= 2.10)
```

That's easy enough to fix—we just add another section to `DESCRIPTION` to specify the version of R we depend on:

```
Depends:
    R (>= 2.10)
```

and voilà, a clean build.

We use a similar trick to document the package as a whole: we create a file `R/unicefdata.R` (i.e., a file with exactly the same name as the package) and put this in it:

```
#' Clean up and share some data from UNICEF on infant HIV rates.
#'
#' @author Greg Wilson, \email{gvwilson@third-bit.com}
#' @docType package
#' @name unicefdata
NULL
```

That's right: to document the entire package, we document `NULL`, which is one of the few times R uses call-by-value. (That's a fairly clumsy joke, but honestly, who among us is at our best at times like these?)

5.9 Key Points

- Develop data-cleaning scripts one step at a time, checking intermediate results carefully.
- Use `read_csv` to read CSV-formatted tabular data into a tibble.
- Use the `skip` and `na` parameters of `read_csv` to skip rows and interpret certain values as `NA`.
- Use `str_replace` to replace portions of strings that match patterns with new strings.
- Use `is.numeric` to test if a value is a number and `as.numeric` to convert it to a number.
- Use `map` to apply a function to every element of a vector in turn.
- Use `map_dfc` and `map_dfr` to map functions across the columns and rows of a tibble.
- Pre-allocate storage in a list for each result from a loop and fill it in rather than repeatedly extending the list.
- An R package can contain code, data, and documentation.
- R code is distributed as compiled bytecode in packages, not as source.
- R packages are almost always distributed through CRAN, the Comprehensive R Archive Network.
- Most of a project's metadata goes in a file called `DESCRIPTION`.
- Metadata related to imports and exports goes in a file called `NAMESPACE`.
- Add patterns to a file called `.Rbuildignore` to ignore files or directories when building a project.
- All source code for a package must go in the `R` sub-directory.
- `library` calls in a package's source code will *not* be executed as the package is loaded after distribution.

- Data can be included in a package by putting it in the `data` sub-directory.
- Data must be in `.rda` format in order to be loaded as part of a package.
- Data in other formats can be put in the `inst/extdata` directory, and will be installed when the package is installed.
- Add comments starting with `#'` to an R file to document functions.
- Use roxygen2 to extract these comments to create manual pages in the `man` directory.
- Use `@export` directives in roxygen2 comment blocks to make functions visible outside a package.
- Add required libraries to the `Imports` section of the `DESCRIPTION` file to indicate that your package depends on them.
- Use `package::function` to access externally-defined functions inside a package.
- Alternatively, add `@import` directives to roxygen2 comment blocks to make external functions available inside the package.
- Import `.data` from `rlang` and use `.data$column` to refer to columns instead of using bare column names.
- Create a file called R/package.R and document `NULL` to document the package as a whole.
- Create a file called R/dataset.R and document the string 'dataset' to document a dataset.

Chapter 6

Non-Standard Evaluation

The biggest difference between R and Python is not where R starts counting, but its use of lazy evaluation. Nothing in R truly makes sense until we understand how this works.

6.1 Learning Objectives

- Trace the order of evaluation in function calls.
- Explain what environments and expressions are and how they relate to one another.
- Justify the author's use of ASCII art in the second decade of the 21st Century.

6.2 How does Python evaluate function calls?

Let's start by looking at a small Python program and its output:

```
def ones_func(ones_arg):
    return ones_arg + " ones"

def tens_func(tens_arg):
    return ones_func(tens_arg + " tens")

initial = "start"
final = tens_func(initial + " more")
print(final)
```

```
start more tens ones
```

When we call `tens_func` we pass it `initial + " more"`; since `initial` has

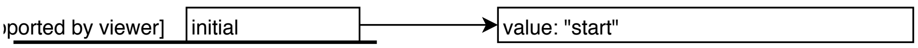


Figure 6.1: Python Step 1

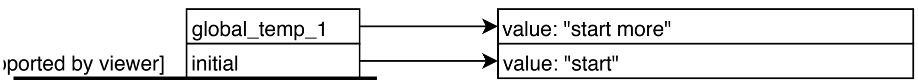


Figure 6.2: Python Step 2

just been assigned the value `"start"`, that's the same as calling `tens_func` with `"start more"`. `tens_func` then calls `ones_func` with `"start more tens"`, and `ones_func` returns `"start more tens ones"`. But there's more going on here than that two-sentence summary suggests. Let's spell out the steps:

```
def ones_func(ones_arg):
    ones_temp_1 = ones_arg + " ones"
    return ones_temp_1

def tens_func(tens_arg):
    tens_temp_1 = tens_arg + " tens"
    tens_temp_2 = ones_func(tens_temp_1)
    return tens_temp_2

initial = "start"
global_temp_1 = initial + " more"
final = tens_func(global_temp_1)
print(final)
```

```
start more tens ones
```

Step 1: we assign `"start"` to `initial` at the global level:

Step 2: we ask Python to call `tens_func(initial + "more")`, so it creates a temporary variable to hold the result of the concatenation *before* calling `tens_func`:

Step 3: Python creates a new stack frame to hold the call to `tens_func`:

Note that `tens_arg` points to the same thing in memory as `global_temp_1`, since Python passes everything by reference.

Step 4: we ask Python to call `ones_func(tens_arg + " tens")`, so it creates another temporary variable:

Step 5: Python creates a new stack frame to manage the call to `ones_func`:

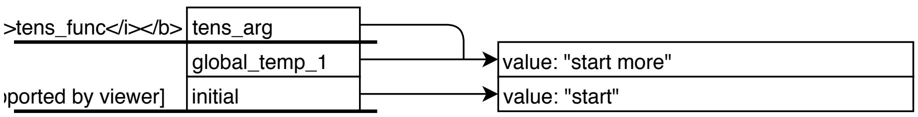


Figure 6.3: Python Step 3

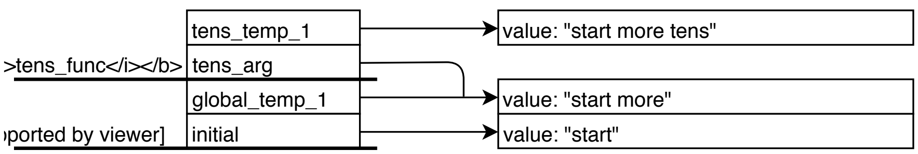


Figure 6.4: Python Step 4

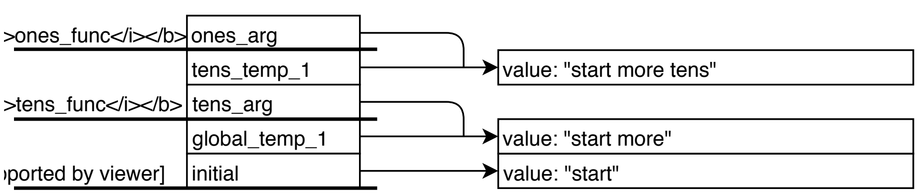


Figure 6.5: Python Step 5

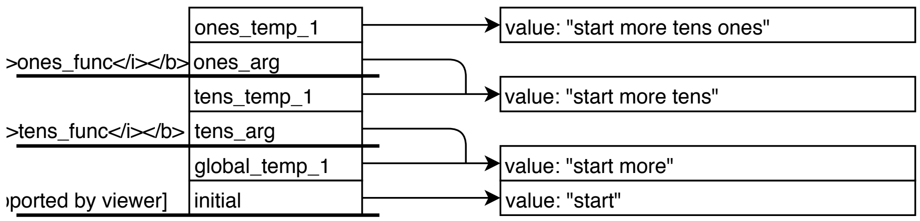


Figure 6.6: Python Step 6

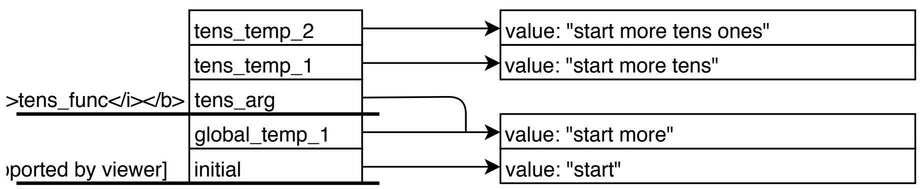


Figure 6.7: Python Step 7

Step 6: Python creates a temporary variable to hold `ones_arg + "ones"`:

Step 7: Python returns from `ones_func` and puts its result in yet another temporary variable in `tens_func`:

Step 8: Python returns from `tens_func` and puts that call's result in `final`:

The most important thing here is that Python evaluates expressions *before* it calls functions, and passes the results of those evaluations to the functions. This is called eager evaluation, and is what most widely-used programming languages do.

6.3 How does R evaluate function calls?

In contrast, R uses lazy evaluation. Here's an R program that's roughly equivalent to the Python shown above:

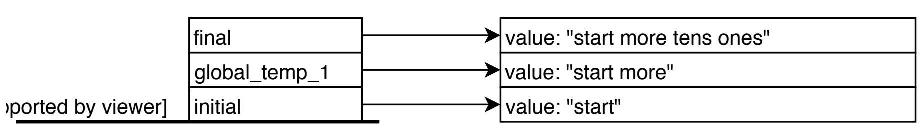


Figure 6.8: Python Step 8

```
ones_func <- function(ones_arg) {
  paste(ones_arg, "ones")
}

tens_func <- function(tens_arg) {
  ones_func(paste(tens_arg, "tens"))
}

initial <- "start"
final <- tens_func(paste(initial, "more"))
print(final)
```

```
[1] "start more tens ones"
```

And here it is with the intermediate steps spelled out in a syntax I just made up:

```
ones_func <- function(ones_arg) {
  ones_arg.RESOLVE(@tens_func@, paste(tens_arg, "tens"), "start more tens")
  ones_temp_1 <- paste(ones_arg, "ones")
  return(ones_temp_1)
}

tens_func <- function(tens_arg) {
  tens_arg.RESOLVE(@global@, paste(initial, "more"), "start more")
  tens_temp_1 <- PROMISE(@tens_func@, paste(tens_arg, "tens"), ____)
  tens_temp_2 <- ones_func(paste(tens_temp_1))
  return(tens_temp_2)
}

initial <- "start"
global_temp_1 <- PROMISE(@global@, paste(initial, "more"), ____)
final <- tens_func(global_temp_1)
print(final)
```

While the original code looked much like our Python, the evaluation trace is very different, and hinges on the fact that *an expression in a programming language can be represented as a data structure.*

> **What's an Expression?**
>
> An expression is anything that has a value. The simplest expressions are literal values like the number 1, the string `"stuff"`, and the Boolean `TRUE`. A variable like `least` is also an expression: its value is whatever the variable currently refers to.
>
> Complex expressions are built out of simpler expressions: `1 + 2` is an expression that uses `+` to combine 1 and 2, while the expression

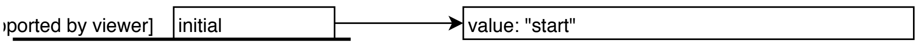


Figure 6.9: R Step 1

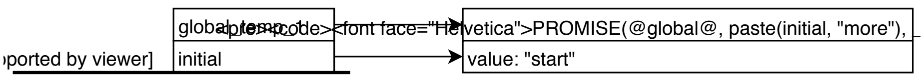


Figure 6.10: R Step 2

`c(10, 20, 30)` uses the function `c` to create a vector out of the values 10, 20, 30. Expressions are often drawn as trees like this:

```
    +
   / \
  1   2
```

When Python (or R, or any other language) reads a program, it parses the text and builds trees like the one shown above to represent what the program is supposed to do. Processing that data structure to find its value is called evaluating the expression.

Most modern languages allow us to build trees ourselves, either by concatenating strings to create program text and then asking the language to parse the result:

```
left <- '1'
right <- '2'
op <- '+'
combined <- paste(left, op, right)
tree <- parse(text = combined)
```

or by calling functions. The function-based approach is safer and more flexible; see Wickham (2019) for details.

Step 1: we assign "start" to `initial` in the global environment:

Step 2: we ask R to call `tens_func(initial + "more")`, so it creates a promise to hold:

- the environment we're in (which I'm surrounding with `@`),
- the expression we're passing to the function, and
- the value of that expression (which I'm showing as `____`, since it's initially empty).

and in Step 3, passes that into `tens_func`:

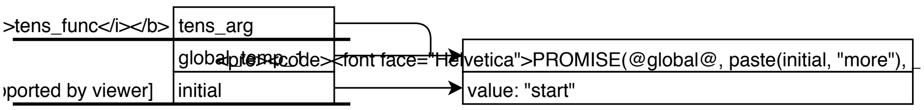


Figure 6.11: R Step 3

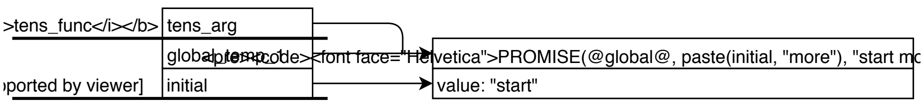


Figure 6.12: (#fig:r-step-4)R Step 4

Crucially, the promise in `tens_func` remembers that it was created in the global environment: it will eventually need a value for `initial`, so it needs to know where to look to find the right one.

Step 4: since the very next thing we ask for is `paste(tens_arg, "tens")`, R needs a value for `tens_arg`. To get it, R evaluates the promise that `tens_arg` refers to:

This evaluation happens *after* `tens_func` has been called, not before as in Python, which is why this scheme is called "lazy" evaluation. Once a promise has been resolved, R uses its value, and that value never changes.

Steps 5: `tens_func` wants to call `ones_func`, so R creates another promise to record what's being passed into `ones_func`:

Step 6: R calls `ones_func`, binding the newly-created promise to `ones_arg` as it does so:

Step 7: R needs a value for `ones_arg` to pass to `paste`, so it resolves the promise:

Step 8: `ones_func` uses `paste` to concatenate strings:

Step 9: `ones_func` returns:

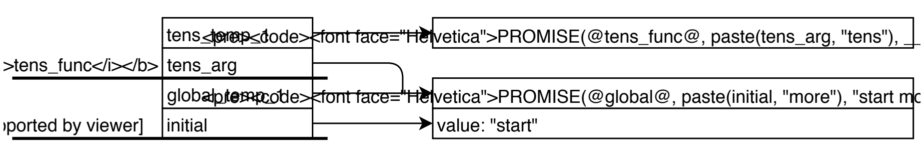


Figure 6.13: R Step 5

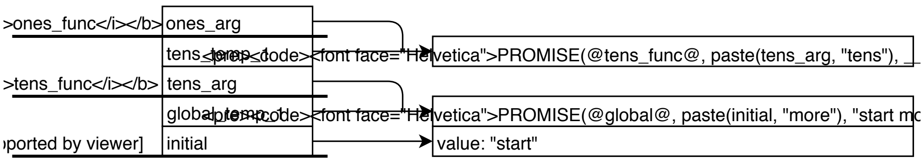


Figure 6.14: R Step 6

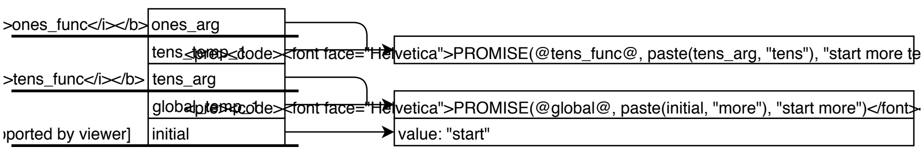


Figure 6.15: R Step 7

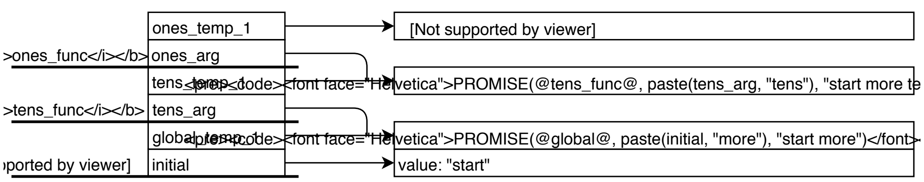


Figure 6.16: R Step 8

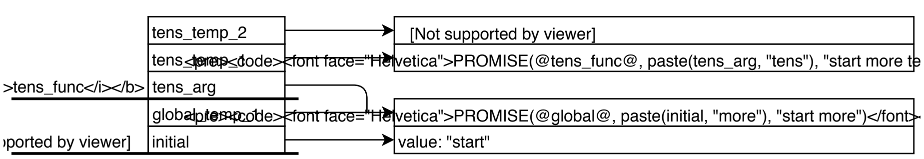


Figure 6.17: R Step 9

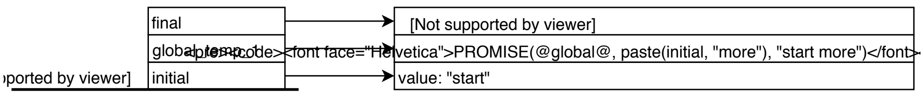


Figure 6.18: R Step 10

Step 10: `tens_func` returns:

We got the same answer as we did in Python, but in a significantly different way. Each time we passed something into a function, R created a promise to record what it was and where it came from, and then resolved the promise when the value was needed. R *always* does this—if we call:

```
sign(2)
```

then behind the scenes, R is creating a promise and passing it to `sign`, where it is automatically resolved to get the number 2 when its value is needed. (If I wanted to be thorough, I would have shown the promises passed into `paste` at each stage of execution above.)

6.4 Why is lazy evaluation useful?

R's lazy evaluation seems pointless if it always produces the same answer as Python's eager evaluation, but it doesn't have to. To see how powerful lazy evaluation can be, let's create an expression of our own:

```
my_expr <- expr(red)
```

Displaying the value of `my_expr` isn't very exciting:

```
my_expr
```

```
red
```

but what kind of thing is it?

```
typeof(my_expr)
```

```
[1] "symbol"
```

A symbol is a kind of expression. It is not a string (though strings can be converted to symbols and symbols to strings) nor is it a value—not yet. If we try to get the value it refers to, R displays an error message:

```
eval(my_expr)
```

```
Error in eval(my_expr): object 'red' not found
```

We haven't created a variable called `red`, so R cannot evaluate an expression that asks for it.

But what if we create such a variable now and then re-evaluate the expression?

```
red <- "this is red"
eval(my_expr)
```

```
[1] "this is red"
```

More usefully, what if we create something that has a value for `red`:

```
color_data <- tribble(
  ~red, ~green,
     1,     10,
     2,     20
)
color_data
```

```
# A tibble: 2 x 2
    red green
  <dbl> <dbl>
1     1    10
2     2    20
```

and then ask R to evaluate our expression in the context of that tibble:

```
eval(my_expr, color_data)
```

```
[1] 1 2
```

When we do this, `eval` looks for definitions of variables in the data structure we've given it—in this case, the tibble `color_data`. Since that tibble has a column called `red`, `eval(my_expr, color_data)` gives us that column.

This may not seem life-changing yet, but being able to pass expressions around and evaluate them in contexts of our choosing allows us to seem very clever indeed. For example, let's create another expression:

```
add_red_green <- expr(red + green)
typeof(add_red_green)
```

```
[1] "language"
```

The type of `add_red_green` is `language` rather than `symbol` because it contains more than just a single symbol, but it's still an expression, so we can evaluate it in the context of our data frame:

```
eval(add_red_green, color_data)
```

```
[1] 11 22
```

Still not convinced? Have a look at this function:

```
run_many_checks <- function(data, ...) {
  conditions <- list(...)
  checks <- vector("list", length(conditions))
  for (i in seq_along(conditions)) {
    checks[[i]] <- eval(conditions[[i]], data)
  }
  checks
}
```

`run_many_checks` takes a tibble and some logical expressions, evaluates each expression in turn, and returns a list of results:

```
run_many_checks(color_data, expr(0 < red), expr(red < green))
```

```
[[1]]
[1] TRUE TRUE

[[2]]
[1] TRUE TRUE
```

We can take it one step further and simply report whether all the checks passed or not:

```
run_all_checks <- function(data, ...) {
  conditions <- list(...)
  checks <- vector("logical", length(conditions))
  for (i in seq_along(conditions)) {
    checks[[i]] <- all(eval(conditions[[i]], data))
  }
  all(checks)
}

run_all_checks(color_data, expr(0 < red), expr(red < green))
```

```
[1] TRUE
```

This is cool, but typing `expr(...)` over and over is kind of clumsy. It also seems superfluous, since we know that arguments aren't evaluated before they're passed into functions. Can we get rid of this and write something that does this?

```
run_all_checks(color_data, 0 < red, red < green)
```

The answer is going to be "yes", but it's going to take a bit of work.

> **Square Brackets... Why'd It Have to Be Square Brackets?**
>
> Before we go there, a word (or code snippet) of warning. The first version of `run_many_checks` essentially did this:

```
conditions <- list(expr(red < green))
eval(conditions[1], color_data)
```

What I did wrong was use `[` instead of `[[`, which meant that `conditions[1]` was not an expression—it was a list containing a single expression. It turns out that evaluating a list containing an expression produces a list of expressions rather than an error, which is so helpful that it only took me an hour to figure out my mistake.

6.5 What is tidy evaluation?

Our goal is to write something that looks like it belongs in the tidyverse. We want to be able to write this:

```
check_all(color_data, 0 < red, red < green)
```

without calling `expr` to quote our expressions explicitly. For simplicity's sake, our first attempt only handles a single expression:

```
check_naive <- function(data, test) {
  eval(test, data)
}
```

When we try it, it fails:

```
check_naive(color_data, red != green)
```

```
Error in eval(test, data): object 'green' not found
```

This actually makes sense: by the time we reach the call to `eval`, `test` refers to a promise that represents the value of `red != green` in the global environment. Promises are not expressions—each promise contains an expression, but it also contains an environment and a copy of the expression's value (if it has ever been calculated). As a result, when R sees the call to `eval` inside `check_naive` it automatically tries to resolve the promise that contains `left != right`, and fails because there are no variables with those names in the global environment.

So how can we get the expression out of the promise without triggering evaluation? One way is to use a function called `substitute`:

```
check_using_substitute <- function(data, test) {
  subst_test <- substitute(test)
  eval(subst_test, data)
}

check_using_substitute(color_data, red != green)
```

```
[1] TRUE TRUE
```

However, `substitute` is frowned upon because it does one thing when called interactively on the command line and something else when called inside a function. Instead, we can use a function called `enquo` from the rlang package. `enquo` returns an object called a quosure that contains only an unevaluated expression and an environment:

```
check_using_enquo <- function(data, test) {
  q_test <- enquo(test)
  eval(q_test, data)
}

check_using_enquo(color_data, red != green)
```

```
<quosure>
expr: ^red != green
env:  global
```

Ah: a quosure is a structured object, so evaluating it just gives it back to us in the same way that evaluating `2` or `"hello"` would. What we want to `eval` is the expression inside the quosure, which we can get using `quo_get_expr`:

```
check_using_quo_get_expr <- function(data, test) {
  q_test <- enquo(test)
  eval(quo_get_expr(q_test), data)
}

check_using_quo_get_expr(list(left = 1, right = 2), left != right)
```

```
[1] TRUE
```

Enquoting and evaluating expressions is done so often in the tidyverse that rlang provides a shortcut called `{{..}}`:

```
max_of_var <- function(data, the_var) {
  data %>%
    group_by({{ the_var }}) %>%
    summarize(maximum = max({{ the_var }}))
}

max_of_var(color_data, red)
```

```
# A tibble: 2 x 2
    red maximum
  <dbl>   <dbl>
1     1       1
2     2       2
```

We can use this to write a function `run_two_checks` that runs two checks on some data:

```
run_two_checks <- function(data, first_check, second_check) {
  first_result <- data %>%
    transmute(temp = {{ first_check }}) %>%
    pull(temp) %>%
    all()
  second_result <- data %>%
    transmute(temp = {{ second_check }})%>%
    pull(temp) %>% all()
  first_result && second_result
}

run_two_checks(color_data, 0 < red, red < green)
```

```
[1] TRUE
```

That's much easier to follow than a bunch of `enquo` and `eval` calls, but what if we want to handle an arbitrary number of checks? Our first attempt is this:

```
new_colors <- tribble(
  ~yellow, ~violet,
  1,       10,
  2,       20
)
run_all_checks <- function(data, ...) {
  conditions <- list(...)
  result = TRUE
  for (i in seq_along(conditions)) {
    cond = conditions[[i]]
    result <- result && data %>% transmute(temp = {{cond}}) %>% pull(temp) %>% all()
  }
  result
}

run_all_checks(new_colors, 0 < yellow, violet < yellow)
```

```
Error in run_all_checks(new_colors, 0 < yellow, violet < yellow): object 'yellow' not
```

This code fails because the call to `list(...)` tries to evaluate the expressions in `...` when adding them to the list. What we need to use instead is `enquos`, which does what `enquo` does but on `...`:

```
run_all_checks <- function(data, ...) {
  conditions <- enquos(...)
  result = TRUE
  for (i in seq_along(conditions)) {
    cond = conditions[[i]]
    result <- result && data %>%
```

```
      transmute(temp = {{ cond }}) %>%
      pull(temp) %>%
      all()
  }
  result
}

run_all_checks(new_colors, 0 < yellow, violet < yellow)
```

```
[1] FALSE
```

6.6 What if I truly desire to venture into the depths?

We will occasionally need to go one level deeper in tidy evaluation. Our first attempt (which only handles a single test) is going to fail on purpose to demonstrate a common mistake:

```
check_without_quoting_test <- function(data, test) {
  data %>% transmute(result = test) %>% pull(result) %>% all()
}
check_without_quoting_test(color_data, yellow < violet)
```

```
Error: object 'yellow' not found
```

That failed because we're not enquoting the test. Let's modify it the code to enquote and then pass in the expression:

```
check_without_quoting_test <- function(data, test) {
  q_test <- enquo(test)
  x_test <- quo_get_expr(q_test)
  data %>% transmute(result = x_test) %>% pull(result) %>% all()
}
check_without_quoting_test(new_colors, yellow < violet)
```

```
Error: Column `result` is of unsupported type quoted call
```

Damn—we thought this one had a chance. The problem is that when we say `result = x_test`, what actually gets passed into `transmute` is a promise containing an expression. Somehow, we need to prevent R from doing that promise wrapping.

This brings us to `enquo`'s partner `!!`, which we can use to splice the expression in a quosure into a function call. `!!` is pronounced "bang bang" or "oh hell", depending on how your day is going. It only works in contexts like function calls where R is automatically quoting things for us, but if we use it then, it does exactly what we want:

```
check_using_bangbang <- function(data, test) {
  q_test <- enquo(test)
  data %>% transmute(result = !!q_test) %>% pull(result) %>% all()
}
check_using_bangbang(new_colors, yellow < violet)
```

```
[1] TRUE
```

We are almost in a state of grace. The two rules we must follow are:

1. Use `enquo` to enquote every argument that contains an unevaluated expression.
2. Use `!!` when passing each of those arguments into a tidyverse function.

```
check_all <- function(data, ...) {
  tests <- enquos(...)
  result <- TRUE
  for (t in tests) {
    result <- result && data %>%
      transmute(result = !!t) %>%
      pull(result) %>%
      all()
  }
  result
}

check_all(new_colors, 0 < yellow, yellow < violet)
```

```
[1] TRUE
```

And just to make sure that it fails when it's supposed to:

```
check_all(new_colors, yellow == violet)
```

```
[1] FALSE
```

Backing up a bit, `!!` works because there are two kinds of functions in R: evaluating functions and quoting functions. Evaluating functions take arguments as values—they're what most of us are used to working with. Quoting functions, on the other hand, aren't passed the values of expressions, but the expressions themselves. When we write `color_data$red`, the `$` function is being passed `color_data` and the quoted expression `red`. This is why we can't use variables as field names with `$`:

```
the_string_red <- "red"
color_data$the_string_red
```

```
Warning: Unknown or uninitialised column: 'the_string_red'.
```

```
NULL
```

The square bracket operators `[` and `[[`, on the other hand, are evaluating functions, so we can give them a variable containing a column name and get either a single-column tibble:

```
color_data[the_string_red]      # single square brackets
```

```
# A tibble: 2 x 1
    red
  <dbl>
1     1
2     2
```

or a naked vector:

```
color_data[[the_string_red]]    # double square brackets
```

```
[1] 1 2
```

6.7 Is it worth it?

Delayed evaluation and quoting are confusing for two reasons:

1. They expose machinery that most programmers have never had to deal with before (and might not even have known existed). It's rather like learning to drive an automatic transmission and then switching to a manual one—all of a sudden you have to worry about a gear shift and a clutch.
2. R's built-in tools don't behave as consistently as they could, and the core functions provided by the tidverse as alternatives use variations on a small number of names: `quo`, `quote`, and `enquo` might all appear on the same page.

That said, being able to pass column names to functions without wrapping them in strings is very useful, and many powerful tools (such as using formulas in models) rely on taking unevaluated expressions apart and rearranging them. If you would like to know more, or check that what you now think you understand is accurate, this tutorial is a good next step.

6.8 Key Points

- R uses lazy evaluation: expressions are evaluated when their values are needed, not before.
- Use `expr` to create an expression without evaluating it.
- Use `eval` to evaluate an expression in the context of some data.
- Use `enquo` to create a quosure containing an unevaluated expression and its environment.
- Use `quo_get_expr` to get the expression out of a quosure.
- Use `!!` to splice the expression in a quosure into a function call.

Chapter 7

Intellectual Debt

We have accumulated some intellectual debt in the previous lessons, and we should clear this burden from our conscience before we go on to new topics.

7.1 Learning Objectives

- Explain what the formula operator ~ was created for and what other uses it has.
- Describe and use `.`, `.x`, `.y`,..1,..2‘, and other convenience parameters.
- Define copy-on-modify and explain its use in R.

7.2 Why shouldn’t I use `setwd`?

Because reasons.

But…

No. Use the here package instead to create paths that are relative to your current location:

```
print(glue('here by itself: {here()}'))
```

```
here by itself: /Users/gvwilson/tidynomicon
```

```
print(glue('here("book.bib"): {here("book.bib")}'))
```

```
here("book.bib"): /Users/gvwilson/tidynomicon/book.bib
```

```
print(glue('here("etc", "common.R"): {here("etc", "common.R")}'))
```

```
here("etc", "common.R"): /Users/gvwilson/tidynomicon/etc/common.R
```

7.3 What the hell are factors?

Another feature of R that doesn't have an exact analog in Python is factors. In statistics, a factor is a categorical variable such as "flavor", which can be "vanilla", "chocolate", "strawberry", or "mustard". Factors can be represented as strings, but storing the same string many times wastes space and is inefficient (since comparing strings takes longer than comparing numbers). R therefore stores each string once and gives it with a numeric key, so that internally, "mustard" is the number 4 in the lookup table for "flavor", but is presented as "mustard" rather than 4.

This is useful, but brings with it some problems:

1. On the statistical side, it encourages people to put messy reality into tidy but misleading boxes. For example, it's unfortunately still common for forms to require people to identify themselves as either "male" or "female", which is scientifically incorrect. Similarly, census forms that ask questions about racial or ethnic identity often leave people scratching their heads, since they don't belong to any of the categories offered.
2. On the computational side, some functions in R automatically convert strings to factors by default. This makes sense when working with statistical data—in most cases, a column in which the same strings are repeated many times is categorical—but it is usually not the right choice in other situations. This has surprised enough people the years that the tidyverse goes the other way and only creates factors when asked to.

Let's work through a small example. Suppose we've read a CSV file and wound up with this table:

```
raw <- tribble(
  ~person, ~flavor, ~ranking,
  "Lhawang", "strawberry", 1.7,
  "Lhawang", "chocolate",  2.5,
  "Lhawang", "mustard",    0.2,
  "Khadee",  "strawberry", 2.1,
  "Khadee", "chocolate",   2.4,
  "Khadee", "vanilla",     3.9,
  "Haddad", "strawberry",  1.8,
  "Haddad", "vanilla",     2.1
)
raw
```

```
# A tibble: 8 x 3
  person  flavor     ranking
  <chr>   <chr>        <dbl>
1 Lhawang strawberry     1.7
2 Lhawang chocolate      2.5
3 Lhawang mustard        0.2
```

```
4 Khadee  strawberry     2.1
5 Khadee  chocolate      2.4
6 Khadee  vanilla        3.9
7 Haddad  strawberry     1.8
8 Haddad  vanilla        2.1
```

Let's aggregate using flavor values so that we can check our factor-based aggregating later:

```
raw %>%
  group_by(flavor) %>%
  summarize(number = n(), average = mean(ranking))
```

```
# A tibble: 4 x 3
  flavor     number average
  <chr>       <int>   <dbl>
1 chocolate       2    2.45
2 mustard         1    0.2
3 strawberry      3    1.87
4 vanilla         2    3
```

It probably doesn't make sense to turn `person` into factors, since names are actually character strings, but `flavor` is a good candidate:

```
raw <- mutate_at(raw, vars(flavor), as.factor)
raw
```

```
# A tibble: 8 x 3
  person  flavor     ranking
  <chr>   <fct>        <dbl>
1 Lhawang strawberry     1.7
2 Lhawang chocolate      2.5
3 Lhawang mustard        0.2
4 Khadee  strawberry     2.1
5 Khadee  chocolate      2.4
6 Khadee  vanilla        3.9
7 Haddad  strawberry     1.8
8 Haddad  vanilla        2.1
```

We can still aggregate as we did before:

```
raw %>%
  group_by(flavor) %>%
  summarize(number = n(), average = mean(ranking))
```

```
# A tibble: 4 x 3
  flavor     number average
  <fct>       <int>   <dbl>
1 chocolate       2    2.45
```

```
2 mustard        1    0.2
3 strawberry     3    1.87
4 vanilla        2    3
```

We can also impose an ordering on the factor's elements:

```
raw <- raw %>%
  mutate(flavor = fct_relevel(flavor, "chocolate", "strawberry", "vanilla", "mustard"))
raw
```

```
# A tibble: 8 x 3
  person  flavor     ranking
  <chr>   <fct>        <dbl>
1 Lhawang strawberry     1.7
2 Lhawang chocolate      2.5
3 Lhawang mustard        0.2
4 Khadee  strawberry     2.1
5 Khadee  chocolate      2.4
6 Khadee  vanilla        3.9
7 Haddad  strawberry     1.8
8 Haddad  vanilla        2.1
```

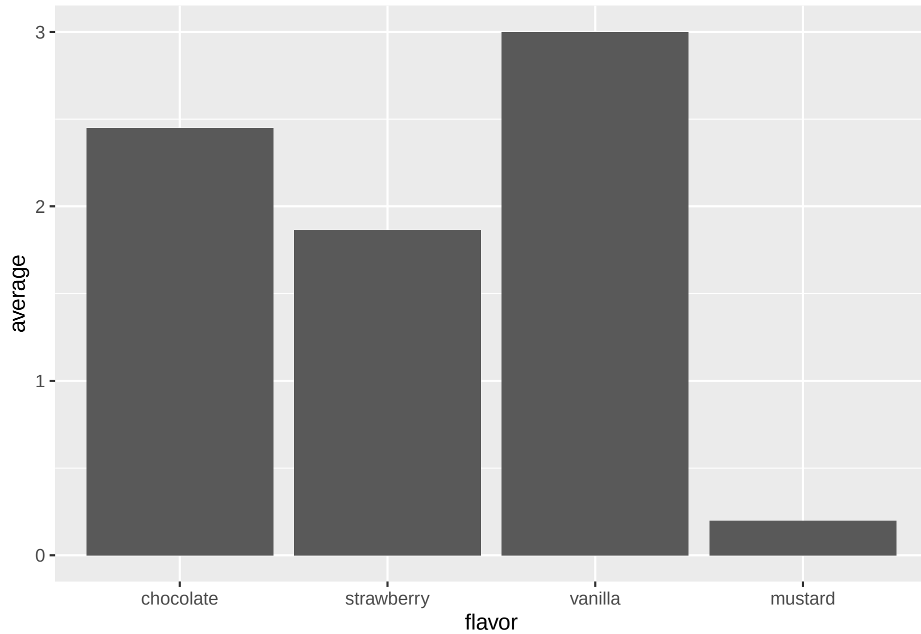
This changes the order in which they are displayed after grouping:

```
raw %>%
  group_by(flavor) %>%
  summarize(number = n(), average = mean(ranking))
```

```
# A tibble: 4 x 3
  flavor     number average
  <fct>       <int>   <dbl>
1 chocolate       2    2.45
2 strawberry      3    1.87
3 vanilla         2    3
4 mustard         1    0.2
```

And also changes the order of bars in a bar chart:

```
raw %>%
  group_by(flavor) %>%
  summarize(number = n(), average = mean(ranking)) %>%
  ggplot(mapping = aes(x = flavor, y = average)) +
  geom_col()
```



To learn more about how factors work and how to use them when analyzing categorical data, please see this paper by McNamara and Horton.

7.4 How do I refer to various arguments in a pipeline?

When we put a function in a pipeline using `%>%`, that operator calls the function with the incoming data as the first argument, so `data %>% func(arg)` is the same as `func(data, arg)`. This is fine when we want the incoming data to be the first argument, but what if we want it to be second? Or third?

One possibility is to save the result so far in a temporary variable and then start a second pipe:

```
data <- tribble(
  ~left, ~right,
  1,     NA,
  2,     20
)

empties <- data %>%
  pmap_lgl(function(...) {
    args <- list(...)
    any(is.na(args))
  })
```

```
data %>%
  transmute(id = row_number()) %>%
  filter(empties) %>%
  pull(id)
```

```
[1] 1
```

This builds a logical vector `empties` with as many entries as `data` has rows, then filters data according to which of the entries in the vector are `TRUE`.

A better practice is to use the parameter name `.`, which means "the incoming data". In some functions (e.g., a two-argument function being used in `map`) we can also use `.x` and `.y` for the first and second arguments, and for more arguments, we can use `..1`, `..2`, and so on (with two dots at the front):

```
data %>%
  pmap_lgl(function(...) {
    args <- list(...)
    any(is.na(args))
  }) %>%
  tibble(empty = .) %>%
  mutate(id = row_number()) %>%
  filter(empty) %>%
  pull(id)
```

```
[1] 1
```

In this model, we create the logical vector, then turn it into a tibble with one column called `empty` (which is what `empty = .` does in `tibble`'s constructor). After that, we add another column with row numbers, filter, and pull out the row numbers.

And while we're here: `row_number` doesn't do what its name suggests. We're better off using `rowid_to_column`:

```
data %>%
  rowid_to_column()
```

```
# A tibble: 2 x 3
  rowid  left right
  <int> <dbl> <dbl>
1     1     1    NA
2     2     2    20
```

7.5 I thought you said that R encouraged functional programming?

I did. Here is a function that reads a file and returns one of its columns:

```
col_from_file <- function(filename, colname) {
  dat <- readr::read_csv(filename)
  dat[colname]
}

person_filename <- here::here("data", "person.csv")
col_from_file(person_filename, "family_name")
```

```
# A tibble: 5 x 1
  family_name
  <chr>
1 Dyer
2 Pabodie
3 Lake
4 Roerich
5 Danforth
```

Note that the column name *must* be passed as a quoted string; Chapter 6 will show us how to pass unquoted column names.

We might occasionally want to allow the user to specify what values in the file are to be considered NAs. This small addition allows us to do that, while keeping the empty string and the string `"NA"` as defaults:

```
col_from_file <- function(filename, colname, na = c("", "NA")) {
  dat <- readr::read_csv(filename, na = na)
  dat[colname]
}

col_from_file(person_filename, "family_name", c("Dyer"))
```

```
# A tibble: 5 x 1
  family_name
  <chr>
1 <NA>
2 Pabodie
3 Lake
4 Roerich
5 Danforth
```

We can also allow the user to specify any number of columns by capturing "extra" parameters in `...` and passing that value directly to `dplyr::select`:

```
cols_from_file <- function(filename, ..., na = c("", "NA")) {
  readr::read_csv(filename, na = na) %>%
    dplyr::select(...)
}

cols_from_file(person_filename, personal_name, family_name)
```

```
# A tibble: 5 x 2
  personal_name family_name
  <chr>         <chr>
1 William       Dyer
2 Frank         Pabodie
3 Anderson      Lake
4 Valentina     Roerich
5 Frank         Danforth
```

Now that we can create functions, we can use the tools in the `purrr` library to wield them. `purrr::map` applies a function to each value in a vector in turn and returns a list:

```
is_long_name <- function(name) {
  stringr::str_length(name) > 4
}

person <- read_csv(here::here("data", "person.csv"))
```

```
Parsed with column specification:
cols(
  person_id = col_character(),
  personal_name = col_character(),
  family_name = col_character()
)
```

```
purrr::map(person$family_name, is_long_name)
```

```
[[1]]
[1] FALSE

[[2]]
[1] TRUE

[[3]]
[1] FALSE

[[4]]
[1] TRUE
```

```
[[5]]
[1] TRUE
```

For small calculations, we will define the function where it is used—this is sometimes called an anonymous function since it isn't given a name. We will also use `purrr::map_lgl` so that the result of the call is a logical vector rather than a list. Similarly-named functions will give us numbers, character strings, and so on:

```
purrr::map_lgl(person$family_name,
               function(name) stringr::str_length(name) > 4)
```

```
[1] FALSE  TRUE FALSE  TRUE  TRUE
```

Little functions like this are so common that `purrr` allows us to use write them as formulas using the `~ operator with`.x' as a shorthand for the value from the vector being processed:

```
purrr::map_chr(person$family_name, ~ stringr::str_to_upper(.x))
```

```
[1] "DYER"     "PABODIE"  "LAKE"     "ROERICH"  "DANFORTH"
```

Other functions in `purrr` let us work on two vectors at once:

```
purrr::map2_chr(person$personal_name,
                person$family_name,
                ~ stringr::str_c(.y, .x, sep = '_'))
```

```
[1] "Dyer_William"      "Pabodie_Frank"     "Lake_Anderson"
[4] "Roerich_Valentina" "Danforth_Frank"
```

If we need to collapse the result to a single value (e.g., to use in `if`) we have `purrr::some` and `purrr::every`:

```
purrr::every(person$personal_name, ~ .x > 'M')
```

```
[1] FALSE
```

7.5.1 Modify specific elements of a list:

```
purrr::modify_at(person$personal_name, c(2, 4), stringr::str_to_upper)
```

```
[1] "William"   "FRANK"     "Anderson"  "VALENTINA" "Frank"
```

Use `modify_if` to upper-case names that are greater than "M".

7.5.2 Create an acronym:

```
purrr::reduce(person$personal_name, ~stringr::str_c(.x, stringr::str_sub(.y, 1, 1)), .init = "")
```

```
[1] "WFAVF"
```

Explain why using `stringr::str_c(stringr::str_sub(.x, 1, 1), stringr::str_sub(.y, 1, 1))` doesn't work.

7.5.3 Create intermediate values:

```
purrr::accumulate(person$personal_name, ~stringr::str_c(.x, stringr::str_sub(.y, 1, 1)
```

```
[1] ""      "W"     "WF"    "WFA"   "WFAV"  "WFAVF"
```

Modify this so that the initial empty string isn't in the final result.

7.6 How does R give the appearance of immutable data?

Another feature of R that can surprise the unwary is its use of copy-on-modify to make data appear immutable (a jargon term meaning "cannot be changed after creation"). If two or more variables refer to the same data and that data is updated via one variable, R automatically makes a copy of the data so that the other variable's value doesn't change. Here's a simple example:

```
first <- c("red", "green", "blue")
second <- first
print(glue("before modification, first is {paste(first, collapse='-')} and second is {p
```

```
before modification, first is red-green-blue and second is red-green-blue
```

```
first[[1]] <- "sulphurous"
print(glue("after modification, first is {paste(first, collapse='-')} and second is {pa
```

```
after modification, first is sulphurous-green-blue and second is red-green-blue
```

This is true of nested structures as well:

```
first <- tribble(
  ~left, ~right,
  101,   202,
  303,   404)
second <- first
first$left[[1]] <- 999
print("first after modification")
```

```
[1] "first after modification"
```

```
first
```

```
# A tibble: 2 x 2
   left right
  <dbl> <dbl>
1   999   202
```

```
2   303   404
```

```r
print("second after modification")
```

```
[1] "second after modification"
```

```r
second
```

```
# A tibble: 2 x 2
   left right
  <dbl> <dbl>
1   101   202
2   303   404
```

In this case, the entire `left` column of `first` has been replaced: tibbles (and data frames) are stored as lists of vectors, so changing any value in a column triggers construction of a new column vector.

We can watch this happen using the `tracemem` function, which shows us where objects live in the computer's memory:

```r
first <- tribble(
  ~left, ~right,
  101,   202,
  303,   404
)
tracemem(first)
```

```
[1] "<0x7fe9097f4a08>"
```

```r
first$left[[1]] <- 999
```

```
tracemem[0x7fe9097f4a08 -> 0x7fe90976f9c8]: eval eval withVisible withCallingHandlers handle timi
tracemem[0x7fe90976f9c8 -> 0x7fe90976fa48]: eval eval withVisible withCallingHandlers handle timi
tracemem[0x7fe90976fa48 -> 0x7fe90976fac8]: $<-.data.frame $<- eval eval withVisible withCallingH
tracemem[0x7fe90976fac8 -> 0x7fe908d3c008]: $<-.data.frame $<- eval eval withVisible withCallingH
```

```r
untracemem(first)
```

This rather cryptic output tell us the address of the tibble, then notifies us of changes to the tibble and its contents. We can accomplish something a little more readable using `pryr::address` (i.e., the `address` function from the pryr package):

```r
left <- first$left # alias
print(glue("left column is initially at {pryr::address(left)}"))
```

```
Registered S3 method overwritten by 'pryr':
  method      from
  print.bytes Rcpp
```

```
left column is initially at 0x7fe90976fa08
```

```
first$left[[2]] <- 888
print(glue("after modification, the original column is still at {pryr::address(left)}"))
```

```
after modification, the original column is still at 0x7fe90976fa08
```

```
temp <- first$left # another alias
print(glue("but the first column is at {pryr::address(temp)}"))
```

```
but the first column is at 0x7fe9098f5a48
```

(We need to use the alias `temp` because `address(first$left)` doesn't work: the argument to `address` needs to be a variable name.)

R's copy-on-modify semantics is particularly important when writing functions. If we modify an argument inside a function, that modification isn't visible to the caller, so even functions that appear to modify structures usually don't. ("Usually", because there are exceptions, but we must stray off the path to find them.)

7.7 What else should I worry about?

Ralph Waldo Emerson once wrote, "A foolish consistency is the hobgoblin of little minds." Here, then, are few of the hobgoblins I've encountered on my journey through R.

7.7.1 The `order` function

The function `order` generates indices to pull values into place rather than push them, i.e., `order(x)[i]` is the index in `x` of the element that belongs at location `i`. For example:

```
bases <- c("g", "c", "t", "a")
order(bases)
```

```
[1] 4 2 1 3
```

shows that the value at location 4 (the `"a"`) belongs in the first spot of the vector; it does *not* mean that the value in the first location (the `"g"`) belongs in location 4. This convention means that `something[order(something)]` does the right thing:

```
bases[order(bases)]
```

```
[1] "a" "c" "g" "t"
```

7.7.2 One of a set of values

The function `one_of` is a handy way to specify several values for matching without complicated Boolean conditionals. For example, `gather(data, key =`

"year", value = "cases", one_of(c("1999", "2000"))) collects data for the years 1999 and 2000.

7.7.3 | and & are not the same as || and &&

Let's try some experiments:

```
TRUE_TRUE <- c(TRUE, TRUE)
TRUE_FALSE <- c(TRUE, FALSE)
FALSE_TRUE <- c(FALSE, TRUE)
print(glue("TRUE_TRUE &  TRUE_FALSE: {paste(TRUE_TRUE &  TRUE_FALSE, collapse = ' ')}"))
```

```
TRUE_TRUE &  TRUE_FALSE: TRUE FALSE
```

```
print(glue("TRUE_TRUE &  FALSE_TRUE: {paste(TRUE_TRUE &  FALSE_TRUE, collapse = ' ')}"))
```

```
TRUE_TRUE &  FALSE_TRUE: FALSE TRUE
```

```
print(glue("TRUE_TRUE && TRUE_FALSE: {paste(TRUE_TRUE && TRUE_FALSE, collapse = ' ')}"))
```

```
TRUE_TRUE && TRUE_FALSE: TRUE
```

```
print(glue("TRUE_TRUE && FALSE_TRUE: {paste(TRUE_TRUE && FALSE_TRUE, collapse = ' ')}"))
```

```
TRUE_TRUE && FALSE_TRUE: FALSE
```

The difference is that & always returns a vector result after doing element-by-element conjunction, while && returns a scalar result. This means that & is almost always what we want to use when working with data.

7.7.4 Functions and columns

There is a function called n. It's not the same thing as a column called n. I only made this mistake a dozen times.

```
data <- tribble(
  ~a, ~n,
  1,  10,
  2,  20
)
data %>% summarize(total = sum(n))
```

```
# A tibble: 1 x 1
  total
  <dbl>
1    30
```

```
data %>% summarize(total = sum(n()))
```

```
# A tibble: 1 x 1
  total
```

```
  <int>
1     2
```

7.8 Key Points

- Don't use `setwd`.
- The formula operator `~` delays evaluation of its operand or operands.
- `~` was created to allow users to pass formulas into functions, but is used more generally to delay evaluation.
- Some tidyverse functions define `.` to be the whole data, `.x` and `.y` to be the first and second arguments, and `..N` to be the N'th argument.
- These convenience parameters are primarily used when the data being passed to a pipelined function needs to go somewhere other than in the first parameter's slot.
- 'Copy-on-modify' means that data is aliased until something attempts to modify it, at which point it duplicated, so that data always appears to be unchanged.

Chapter 8

Testing and Error Handling

Novices write code and pray that it works. Experienced programmers know that prayer alone is not enough, and take steps to protect what little sanity they have left. This chapter looks at the tools R gives us for doing this.

8.1 Learning Objectives

- Name and describe the three levels of error handling in R.
- Handle an otherwise-fatal error in a function call in R.
- Create unit tests in R.
- Create unit tests for an R package.

8.2 How does R handle errors?

Python programs handle errors by raising and catching exceptions:

```
values = [-1, 0, 1]
for i in range(4):
    try:
        reciprocal = 1/values[i]
        print("index {} value {} reciprocal {}".format(i, values[i], reciprocal))
    except ZeroDivisionError:
        print("index {} value {} ZeroDivisionError".format(i, values[i]))
    except Exception as e:
        print("index{} some other Exception: {}".format(i, e))
```

```
index 0 value -1 reciprocal -1.0
index 1 value 0 ZeroDivisionError
index 2 value 1 reciprocal 1.0
index3 some other Exception: list index out of range
```

R draws on a different tradition. We say that the operation signals a condition that some other piece of code then handles. These things are all simpler to do using the rlang library, so we begin by loading that:

In order of increasing severity, the three built-in kinds of conditions are messages, warnings, and errors. (There are also interrupts, which are generated by the user pressing Ctrl-C to stop an operation, but we will ignore those for the sake of brevity.) We can signal conditions of these kinds using the functions `message`, `warning`, and `stop`, each of which takes an error message as a parameter:

```
message("This is a message.")
```

```
This is a message.
```

```
warning("This is a warning.\n")
```

```
Warning: This is a warning.
```

```
stop("This is an error.")
```

```
Error in eval(expr, envir, enclos): This is an error.
```

Note that we have to supply our own line ending for warnings but not for the other two cases. Note also that there are very few situations in which a warning is appropriate: if something has truly gone wrong then we should stop, but otherwise we should not distract users from more pressing concerns.

The bluntest of instruments for handling errors is to ignore them. If a statement is wrapped in the function `try` then errors that occur in it are still reported, but execution continues. Compare this:

```
attemptWithoutTry <- function(left, right){
  temp <- left + right
  "result" # returned
}
result <- attemptWithoutTry(1, "two")
```

```
Error in left + right: non-numeric argument to binary operator
```

```
cat("result is", result)
```

```
Error in cat("result is", result): object 'result' not found
```

with this:

```
attemptUsingTry <- function(left, right){
  temp <- try(left + right)
  "value returned" # returned
}
result <- attemptUsingTry(1, "two")
```

```
Error in left + right : non-numeric argument to binary operator
```

```
cat("result is", result)
```

```
result is value returned
```

We can suppress error messages from `try` by setting `silent` to `TRUE`:

```
attemptUsingTryQuietly <- function(left, right){
  temp <- try(left + right, silent = TRUE)
  "result" # returned
}
result <- attemptUsingTryQuietly(1, "two")
cat("result is", result)
```

```
result is result
```

Do not do this, lest you one day find yourself lost in a silent hellscape.

Should you more sensibly wish to handle conditions rather than ignore them, you may invoke `tryCatch`. We begin by raising an error explicitly:

```
tryCatch(
  stop("our message"),
  error = function(cnd) print(glue("error object is {cnd}"))
)
```

```
error object is Error in doTryCatch(return(expr), name, parentenv, handler): our message
```

We can now run a function that would otherwise blow up:

```
tryCatch(
  attemptWithoutTry(1, "two"),
  error = function(cnd) print(glue("error object is {cnd}"))
)
```

```
error object is Error in left + right: non-numeric argument to binary operator
```

We can also handle non-fatal errors using `withCallingHandlers`, and define new types of conditions, but this is done less often in day-to-day R code than in Python: see *Advanced R* or this tutorial for details.

8.3 What should I know about testing in general?

In keeping with common programming practice, we have left testing until the last possible moment. The standard testing library for R is testthat, which shares many features with Python's unittest and other unit testing libraries:

1. Each test consists of a single function that tests a single property or behavior of the system.

2. Tests are collected into files with prescribed names that can be found by a test runner.
3. Shared setup and teardown steps are put in functions of their own.

Let's load it and write our first test:

```
library(testthat)
```

```

Attaching package: 'testthat'

The following objects are masked from 'package:rlang':

    is_false, is_null, is_true

The following object is masked from 'package:dplyr':

    matches

The following object is masked from 'package:purrr':

    is_null

The following object is masked from 'package:tidyr':

    matches
```

```
test_that("Zero equals itself", {expect_equal(0, 0)})
```

As is conventional with unit testing libraries, no news is good news: if a test passes, it doesn't produce output because it doesn't need our attention. Let's try something that ought to fail:

```
test_that("Zero equals one", {expect_equal(0, 1)})
```

```
Error: Test failed: 'Zero equals one'
* 0 not equal to 1.
1/1 mismatches
[1] 0 - 1 == -1
```

Good: we can draw some comfort from the fact that Those Beyond have not yet changed the fundamental rules of arithmetic. But what are the curly braces around `expect_equal` for? The answer is that they create a code block for `test_that` to run. We can run `expect_equal` on its own:

```
expect_equal(0, 1)
```

```
Error: 0 not equal to 1.
1/1 mismatches
[1] 0 - 1 == -1
```

but that doesn't produce a summary of how many tests passed or failed. Passing a block of code to `test_that` also allows us to check several things in one test:

```
test_that("Testing two things", {
  expect_equal(0, 0)
  expect_equal(0, 1)
})
```

```
Error: Test failed: 'Testing two things'
* 0 not equal to 1.
1/1 mismatches
[1] 0 - 1 == -1
```

A block of code is *not* the same thing as an anonymous function, which is why running this block of code does nothing—the "test" defines a function but doesn't actually call it:

```
test_that("Using an anonymous function", function() {
  print("In our anonymous function")
  expect_equal(0, 1)
})
```

8.4 How should I organize my tests?

Running blocks of tests by hand is a bad practice. Instead, we should put related tests in files and then put those files in a directory called `tests/testthat`. We can then run some or all of those tests with a single command.

To start, let's create `tests/testthat/test_example.R`:

```
library(testthat)
context("Demonstrating the testing library")

test_that("Testing a number with itself", {
  expect_equal(0, 0)
  expect_equal(-1, -1)
  expect_equal(Inf, Inf)
})

test_that("Testing different numbers", {
  expect_equal(0, 1)
})

test_that("Testing with a tolerance", {
  expect_equal(0, 0.01, tolerance = 0.05, scale = 1)
  expect_equal(0, 0.01, tolerance = 0.005, scale = 1)
})
```

The first line loads the testthat package, which gives us our tools. The call to `context` on the second line gives this set of tests a name for reporting purposes. After that, we add as many calls to `test_that` as we want, each with a name and a block of code. We can now run this file from within RStudio:

```
test_dir("tests/testthat")
```

```
v |  OK F W S | Context

/ |   0       | Skipping rows correctly
x |   0 5     | Skipping rows correctly
---------------------------------------------------------------------------
test_determine_skip_rows_a.R:9: failure: The right row is found when there are header r
`result` not equal to 2.
Lengths differ: 0 is not 1

test_determine_skip_rows_a.R:14: failure: The right row is found when there are header
`result` not equal to 3.
Lengths differ: 0 is not 1

test_determine_skip_rows_a.R:19: failure: The right row is found when there are no head
`result` not equal to 0.
Lengths differ: 0 is not 1

test_determine_skip_rows_a.R:23: failure: No row is found when 'iso3' isn't present
`determine_skip_rows("a1,a2\nb1,b1\n")` did not throw an error.

test_determine_skip_rows_a.R:28: failure: No row is found when 'iso3' is in the wrong p
`determine_skip_rows("stuff,iso3\n")` did not throw an error.
---------------------------------------------------------------------------

/ |   0       | Skipping rows correctly
v |   5       | Skipping rows correctly

/ |   0       | Demonstrating the testing library
x |   4 2     | Demonstrating the testing library
---------------------------------------------------------------------------
test_example.R:11: failure: Testing different numbers
0 not equal to 1.
1/1 mismatches
[1] 0 - 1 == -1

test_example.R:16: failure: Testing with a tolerance
0 not equal to 0.01.
1/1 mismatches
[1] 0 - 0.01 == -0.01
```

```
------------------------------------------------------------------------------

/ |   0       | Finding empty rows
x |   1 2     | Finding empty rows
------------------------------------------------------------------------------
test_find_empty_a.R:9: failure: A single non-empty row is not mistakenly detected
`result` not equal to NULL.
Types not compatible: integer is not NULL

test_find_empty_a.R:14: failure: Half-empty rows are not mistakenly detected
`result` not equal to NULL.
Types not compatible: integer is not NULL
------------------------------------------------------------------------------

/ |   0       | Finding empty rows
v |   3       | Finding empty rows

/ |   0       | Testing properties of tibbles
v |   1   1   | Testing properties of tibbles
------------------------------------------------------------------------------
test_tibble.R:6: warning: Tibble columns are given the name 'value'
`as.tibble()` is deprecated, use `as_tibble()` (but mind the new semantics).
This warning is displayed once per session.
------------------------------------------------------------------------------

== Results ===================================================================
Duration: 0.2 s

OK:       14
Failed:   9
Warnings: 1
Skipped:  0
```

Care is needed when interpreting these results. There are four `test_that` calls, but eight actual checks, and the number of successes and failures is counted by recording the latter, not the former.

What then is the purpose of `test_that`? Why not just use `expect_equal` and its kin, such as `expect_true`, `expect_false`, `expect_length`, and so on? The answer is that it allows us to do one operation and then check several things afterward. Let's create another file called `tests/testthat/test_tibble.R`:

```r
library(tidyverse)
library(testthat)
context("Testing properties of tibbles")

test_that("Tibble columns are given the name 'value'", {
```

```
  t <- c(TRUE, FALSE) %>% as.tibble()
  expect_equal(names(t), "value")
})
```

(We don't actually have to call our test files `test_something.R`, but `test_dir` and the rest of R's testing infrastructure expect us to. Similarly, we don't have to put them in a `tests` directory, but gibbering incoherence will ensue if we do not.) Now let's run all of our tests:

```
test_dir("tests/testthat")
```

```
v |  OK F W S | Context

/ |   0       | Skipping rows correctly
x |   0 5     | Skipping rows correctly
---------------------------------------------------------------------------
test_determine_skip_rows_a.R:9: failure: The right row is found when there are header r
`result` not equal to 2.
Lengths differ: 0 is not 1

test_determine_skip_rows_a.R:14: failure: The right row is found when there are header
`result` not equal to 3.
Lengths differ: 0 is not 1

test_determine_skip_rows_a.R:19: failure: The right row is found when there are no head
`result` not equal to 0.
Lengths differ: 0 is not 1

test_determine_skip_rows_a.R:23: failure: No row is found when 'iso3' isn't present
`determine_skip_rows("a1,a2\nb1,b1\n")` did not throw an error.

test_determine_skip_rows_a.R:28: failure: No row is found when 'iso3' is in the wrong p
`determine_skip_rows("stuff,iso3\n")` did not throw an error.
---------------------------------------------------------------------------

/ |   0       | Skipping rows correctly
v |   5       | Skipping rows correctly

/ |   0       | Demonstrating the testing library
x |   4 2     | Demonstrating the testing library
---------------------------------------------------------------------------
test_example.R:11: failure: Testing different numbers
0 not equal to 1.
1/1 mismatches
[1] 0 - 1 == -1
```

```
test_example.R:16: failure: Testing with a tolerance
0 not equal to 0.01.
1/1 mismatches
[1] 0 - 0.01 == -0.01
---------------------------------------------------------------------------

/ |   0       | Finding empty rows
x |   1 2     | Finding empty rows
---------------------------------------------------------------------------
test_find_empty_a.R:9: failure: A single non-empty row is not mistakenly detected
`result` not equal to NULL.
Types not compatible: integer is not NULL

test_find_empty_a.R:14: failure: Half-empty rows are not mistakenly detected
`result` not equal to NULL.
Types not compatible: integer is not NULL
---------------------------------------------------------------------------

/ |   0       | Finding empty rows
v |   3       | Finding empty rows

/ |   0       | Testing properties of tibbles
v |   1       | Testing properties of tibbles

== Results ================================================================
Duration: 0.1 s

OK:       14
Failed:   9
Warnings: 0
Skipped:  0
```

That's rather a lot of output. Happily, we can provide a `filter` argument to `test_dir`:

```r
test_dir("tests/testthat", filter = "test_tibble.R")
```

```
Error in test_files(paths, reporter = reporter, env = env, stop_on_failure = stop_on_failure, : N
```

Ah. It turns out that `filter` is applied to filenames *after* the leading `test_` and the trailing `.R` have been removed. Let's try again:

```r
test_dir("tests/testthat", filter = "tibble")
```

```
v |  OK F W S | Context

/ |   0       | Testing properties of tibbles
v |   1       | Testing properties of tibbles
```

```
== Results =================================================================
OK:       1
Failed:   0
Warnings: 0
Skipped:  0
```

That's better, and it illustrates our earlier point about the importance of following conventions.

8.5 How can I write a few simple tests?

To give ourselves something to test, let's create a file called `scripts/find_empty_01.R` containing a single function `find_empty_rows` to identy all the empty rows in a CSV file. Our first implementation is:

```
find_empty_rows <- function(source) {
  data <- read_csv(source)
  empty <- data %>%
    pmap(function(...) {
      args <- list(...)
      all(is.na(args) | (args == ""))
    })
  data %>%
    transmute(id = row_number()) %>%
    filter(as.logical(empty)) %>%
    pull(id)
}
```

This is complex enough to merit line-by-line exegesis:

1. Define the function with one argument `source`, whence we shall read.
2. Read tabular data from that source and assign the resulting tibble to `data`.
3. Begin a pipeline that will assign something to the variable `empty`.
    1. Use `pmap` to map a function across each row of the tibble. Since we don't know how many columns are in each row, we use `...` to take any number of arguments.
    2. Convert the variable number of arguments to a list.
    3. Check to see if all of those arguments are either `NA` or the empty string.
    4. Close the mapped function's definition.
4. Start another pipeline. Its result isn't assigned to a variable, so whatever it produces will be the value returned by `find_empty_rows`.
    1. Construct a tibble that contains only the row numbers of the original table in a column called `id`.
    2. Filter those row numbers to keep only those corresponding to rows that were entirely empty. The `as.logical` call inside `filter` is

needed because the value returned by `pmap` (which we stored in `empty`) is a list, not a logical vector.
3. Use `pull` to get the one column we want from the filtered tibble as a vector.

There is a lot going on here, particularly if you are new to R (as I am at the time of writing) and needed help to figure out that `pmap` is the function this problem wants. But now that we have it, we can do this:

```
source("scripts/find_empty_01.R")
find_empty_rows("a,b\n1,2\n,\n5,6")
```

The `source` function reads R code from the given source. Using this inside an R Markdown file is usually a bad idea, since the generated HTML or PDF won't show readers what code we loaded and ran. On the other hand, if we are creating command-line tools for use on clusters or in other batch processing modes, and are careful to display the code in a nearby block, the stain on our soul is excusable.

The more interesting part of this example is the call to `find_empty_rows`. Instead of giving it the name of a file, we have given it the text of the CSV we want parsed. This string is passed to `read_csv`, which (according to documentation that only took us 15 minutes to realize we had already seen) interprets its first argument as a filename *or* as the actual text to be parsed if it contains a newline character. This allows us to write put the test fixture right there in the code as a literal string, which experience shows is to understand and maintain than having test data in separate files.

Our function seems to work, but we can make it more pipelinesque:

```
find_empty_rows <- function(source) {
  read_csv(source) %>%
    pmap_lgl(function(...) {
      args <- list(...)
      all(is.na(args) | (args == ""))
    }) %>%
    tibble(empty = .) %>%
    mutate(id = row_number()) %>%
    filter(empty) %>%
    pull(id)
}
```

Going line by line once again:

1. Define a function with one argument called `source`, from which we shall once again read.
2. Read from that source to fill the pipeline.
3. Map our test for emptiness across each row, returning a logical vector as a result. (`pmap_lgl` is a derivative of `pmap` that always casts its result to

logical. Similar functions like `pmap_dbl` return vectors of other types; and many other tidyverse functions also have strongly-typed variants.)

4. Turn that logical vector into a single-column tibble, giving that column the name "empty". We explain the use of `.` below.
5. Add a second column with row numbers.
6. Discard rows that aren't empty.
7. Return a vector of the remaining row IDs.

> **Wat?**
>
> Buried in the middle of the pipe shown above is the expression:
>
> `tibble(empty = .)`
>
> Quoting from *Advanced R*, "The function arguments look a little quirky but allow you to refer to `.` for one argument functions, `.x` and `.y.` for two argument functions, and `..1`, `..2`, `..3`, etc, for functions with an arbitrary number of arguments." In other words, `.` in tidyverse functions usually means "whatever is on the left side of the `%>%` operator", i.e., whatever would normally be passed as the function's first argument. Without this, we have no easy way to give the sole column of our newly-constructed tibble a name.

Here's our first batch of tests:

```
library(tidyverse)
library(testthat)
context("Finding empty rows")

source("../../scripts/find_empty_02.R")

test_that("A single non-empty row is not mistakenly detected", {
  result <- find_empty_rows("a\n1")
  expect_equal(result, NULL)
})

test_that("Half-empty rows are not mistakenly detected", {
  result <- find_empty_rows("a,b\n,2")
  expect_equal(result, NULL)
})

test_that("An empty row in the middle is found", {
  result <- find_empty_rows("a,b\n1,2\n,\n5,6")
  expect_equal(result, c(2L))
})
```

And here's what happens when we run this file with `test_dir`:

```r
test_dir("tests/testthat", "find_empty_a")
```

```
v |  OK F W S | Context

/ |   0       | Finding empty rows
x |   1 2     | Finding empty rows
---------------------------------------------------------------------------
test_find_empty_a.R:9: failure: A single non-empty row is not mistakenly detected
`result` not equal to NULL.
Types not compatible: integer is not NULL

test_find_empty_a.R:14: failure: Half-empty rows are not mistakenly detected
`result` not equal to NULL.
Types not compatible: integer is not NULL
---------------------------------------------------------------------------

== Results ================================================================
OK:       1
Failed:   2
Warnings: 0
Skipped:  0
```

This is perplexing: we expected that if there were no empty rows, our function would return `NULL`. Let's look more closely:

```r
find_empty_rows("a\n1")
```

```
integer(0)
```

Ah: our function is returning an integer vector of zero length rather than `NULL`. Let's have a closer look at the properties of this strange beast:

```r
print(glue("integer(0) equal to NULL? {is.null(integer(0))}"))
```

```
integer(0) equal to NULL? FALSE
```

```r
print(glue("any(logical(0))? {any(logical(0))}"))
```

```
any(logical(0))? FALSE
```

```r
print(glue("all(logical(0))? {all(logical(0))}"))
```

```
all(logical(0))? TRUE
```

All right. If we compare `c(1L, 2L)` to `NULL`, we expect `c(FALSE, FALSE)`, so it's reasonable to get a zero-length logical vector as a result when we compare `NULL` to an integer vector with no elements. The fact that `any` of an empty logical vector is `FALSE` isn't really surprising either—none of the elements are `TRUE`, so it would be hard to say that any of them are. `all` of an empty vector being `TRUE` is unexpected, though. The reasoning is apparently that none of the (nonexistent) elements are `FALSE`, but honestly, at this point we are veering dangerously close to JavaScript Logic, so we will accept this result for what it is and move on.

So what *should* our function return when there aren't any empty rows: `NULL` or `integer(0)`? After a bit of thought, we decide on the latter, which means it's the tests that we need to rewrite, not the code:

```
library(tidyverse)
library(testthat)
context("Finding empty rows")

source("../../scripts/find_empty_02.R")

test_that("A single non-empty row is not mistakenly detected", {
  result <- find_empty_rows("a\n1")
  expect_equal(result, integer(0))
})

test_that("Half-empty rows are not mistakenly detected", {
  result <- find_empty_rows("a,b\n,2")
  expect_equal(result, integer(0))
})

test_that("An empty row in the middle is found", {
  result <- find_empty_rows("a,b\n1,2\n,\n5,6")
  expect_equal(result, c(2L))
})
```

And here's what happens when we run this file with `test_dir`:

```
test_dir("tests/testthat", "find_empty_b")
```

```
v |  OK F W S | Context

/ |   0       | Finding empty rows
v |   3       | Finding empty rows

== Results =================================================================
OK:       3
Failed:   0
Warnings: 0
Skipped:  0
```

8.6 How can I check data transformation?

People normally write unit tests for the code in packages, not to check the steps taken to clean up particular datasets, but the latter are just as useful as the former. To illustrate, we have been given several more CSV files to clean up. The first, `at_health_facilities.csv`, shows the percentage of births at health facilities by country, year, and mother's age. It comes from the same UNICEF website as our previous data, but has a different set of problems. Here are its first few lines:

```
,,GLOBAL DATABASES,,,,,,,,,,,,,
,,[data.unicef.org],,,,,,,,,,,,,
,,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,
Indicator:,Delivered in health facilities,,,,,,,,,,,,,,
Unit:,Percentage,,,,,,,,,,,,,,
,,,,Mother's age,,,,,,,,,,,
iso3,Country/areas,year,Total ,age 15-17,age 18-19,age less than 20,age more than 20,age 20-34,ag
AFG,Afghanistan,2010,   33 ,    25 ,    29 ,    28 ,    31 ,    31 ,    31 ,MICS,2010,,,,
ALB,Albania,2005,   98 ,    100 ,   96 ,    97 ,    98 ,    99 ,    92 ,MICS,2005,,,,
ALB,Albania,2008,   98 ,    94 ,    98 ,    97 ,    98 ,    98 ,    99 ,DHS,2008,,,,
...
```

and its last:

```
ZWE,Zimbabwe,2005,  66 ,    64 ,    64 ,    64 ,    67 ,    69 ,    53 ,DHS,2005,,,,
ZWE,Zimbabwe,2009,  58 ,    49 ,    59 ,    55 ,    59 ,    60 ,    52 ,MICS,2009,,,,
ZWE,Zimbabwe,2010,  64 ,    56 ,    66 ,    62 ,    64 ,    65 ,    60 ,DHS,2010,,,,
ZWE,Zimbabwe,2014,  80 ,    82 ,    82 ,    82 ,    79 ,    80 ,    77 ,MICS,2014,,,,
,,,,,,,,,,,,,,,
Definition:,Percentage of births delivered in a health facility.,,,,,,,,,,,,,,
,"The indicator refers to women who had a live birth in a recent time period, generally two years
,,,,,,,,,,,,,,,
Note:,"Database include reanalyzed data from DHS and MICS, using a reference period of two years
,Includes surveys which microdata were available as of April 2016. ,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,
Source:,"UNICEF global databases 2016 based on DHS, MICS .",,,,,,,,,,,,,,
,,,,,,,,,,,,,,,
Contact us:,data@unicef.org,,,,,,,,,,,,,,
```

There are two other files in this collection called `c_sections.csv` and `skilled_attendant_at_birth.csv`, which are the number of Caesarean sections and the number of births where a midwife or other trained practitioner was present. All three datasets have been exported from the same Excel spreadsheet; rather than writing a separate script for each, we should create a tool that will handle them all.

At first glance, the problems we need to solve to do this are:

1. Each file may have a different number of header rows (by inspection, two of the files have 7 and one has 8), so we should infer this number from the file.
2. Each file may contain a different number of records, so our tool should select rows by content rather than by absolute row number.
3. The files appear to have the same column names (for which we give thanks), but we should check this in case someone tries to use our function with a dataset that doesn't.

These three requirements will make our program significantly more complicated, so we should tackle each with its own testable function.

8.6.1 How can I reorganize code to make it more testable?

The data we care about comes after the row with `iso3`, `Country/areas`, and other column headers, so the simplest way to figure out how many rows to skip is to read the data, look for this row, and discard everything above it. The simplest way to do *that* is to read the file once to find the number of header rows, then read it again, discarding that number of rows. It's inefficient, but for a dataset this size, simplicity beats performance.

Here's our first try:

```r
read_csv("data/at_health_facilities.csv") %>%
  select(check = 1) %>%
  mutate(id = row_number()) %>%
  filter(check == "iso3") %>%
  select(id) %>%
  first()
```

```
Warning: Missing column names filled in: 'X1' [1], 'X2' [2], 'X4' [4],
'X5' [5], 'X6' [6], 'X7' [7], 'X8' [8], 'X9' [9], 'X10' [10], 'X11' [11],
'X12' [12], 'X13' [13], 'X14' [14], 'X15' [15], 'X16' [16]
```

```
Parsed with column specification:
cols(
  X1 = col_character(),
  X2 = col_character(),
  `GLOBAL DATABASES` = col_character(),
  X4 = col_character(),
  X5 = col_character(),
  X6 = col_character(),
  X7 = col_character(),
  X8 = col_character(),
  X9 = col_character(),
  X10 = col_character(),
  X11 = col_character(),
  X12 = col_character(),
```

```
  X13 = col_logical(),
  X14 = col_logical(),
  X15 = col_logical(),
  X16 = col_logical()
)
```

```
[1] 7
```

Ignoring the messages about missing column names, this tells us that `iso3` appears in row 7 of our data, which is *almost* true: it's actually in row 8, because `read_csv` has interpreted the first row of the raw CSV data as a header. On the bright side, that means we can immediately use this value as the `skip` parameter to the next `read_csv` call.

How do we test this code? Easy: we turn it into a function, tell that function to stop if it can't find `iso3` in the data, and write some unit tests. The function is:

```
library(tidyverse)

determine_skip_rows <- function(src_path) {
  read_csv(src_path) %>%
    select(check = 1) %>%
    mutate(id = row_number()) %>%
    filter(check == "iso3") %>%
    select(id) %>%
    first()
}
```

We can then call `usethis::use_testthat()` to set up some testing infrastructure, including the directory `tests/testthat` and a script called `tests/testthat.R` that will run all our tests when we want to check the integrity of our project. Once we have done that we can put these five tests in `tests/testthat/test_determine_skip_rows.R`:

```
library(tidyverse)
library(testthat)
context("Skipping rows correctly")

source("../../scripts/determine_skip_rows_a.R")

test_that("The right row is found when there are header rows", {
  result <- determine_skip_rows("a1,a2\nb1,b2\nis03,stuff\nc1,c2\n")
  expect_equal(result, 2)
})

test_that("The right row is found when there are header rows and blank lines", {
  result <- determine_skip_rows("a1,a2\nb1,b2\n,\nis03,stuff\nc1,c2\n,\n")
```

```
  expect_equal(result, 3)
})

test_that("The right row is found when there are no header rows to discard", {
  result <- determine_skip_rows("iso3,stuff\nc1,c2\n")
  expect_equal(result, 0)
})

test_that("No row is found when 'iso3' isn't present", {
  expect_error(determine_skip_rows("a1,a2\nb1,b1\n"),
               "No start row found")
})

test_that("No row is found when 'iso3' is in the wrong place", {
  expect_error(determine_skip_rows("stuff,iso3\n"),
               "No start row found")
})
```

and run it:

```
test_dir("tests/testthat", "determine_skip_rows_a")
```

```
v |  OK F W S | Context

/ |   0       | Skipping rows correctly
x |   0 5     | Skipping rows correctly
---------------------------------------------------------------------------
test_determine_skip_rows_a.R:9: failure: The right row is found when there are header r
`result` not equal to 2.
Lengths differ: 0 is not 1

test_determine_skip_rows_a.R:14: failure: The right row is found when there are header 
`result` not equal to 3.
Lengths differ: 0 is not 1

test_determine_skip_rows_a.R:19: failure: The right row is found when there are no head
`result` not equal to 0.
Lengths differ: 0 is not 1

test_determine_skip_rows_a.R:23: failure: No row is found when 'iso3' isn't present
`determine_skip_rows("a1,a2\nb1,b1\n")` did not throw an error.

test_determine_skip_rows_a.R:28: failure: No row is found when 'iso3' is in the wrong p
`determine_skip_rows("stuff,iso3\n")` did not throw an error.
---------------------------------------------------------------------------
```

```
== Results ===============================================================
OK:       0
Failed:   5
Warnings: 0
Skipped:  0
```

That's right: all five fail. The first problem is that we have written `is03` (with a digit `0` instead of a letter `o`) in the first two tests. If we fix that and re-run the tests, they pass; what about the other three?

1. When there are no rows to skip, our function is returning `integer(0)` instead of 0 because the row with `iso3` is being used as headers.
2. When `iso3` isn't found at all, the function is returning `integer(0)` rather than stopping.

Here is a more robust version of the function:

```r
library(tidyverse)

determine_skip_rows <- function(src_path) {
  data <- read_csv(src_path)
  if (names(data)[1] == "iso3") {
    return(0)
  }
  result <- data %>%
    select(check = 1) %>%
    mutate(id = row_number()) %>%
    filter(check == "iso3") %>%
    select(id) %>%
    first()
  if (length(result) == 0) {
    stop("No start row found in", src_path)
  }
  result
}
```

And here are the results:

```r
test_dir("tests/testthat", "determine_skip_rows_b")
```

```
v |  OK F W S | Context

/ |   0       | Skipping rows correctly
v |   5       | Skipping rows correctly

== Results ===============================================================
OK:       5
Failed:   0
Warnings: 0
```

```
Skipped:  0
```

Our tests still aren't checking anything statistical, but without trustworthy data, our statistics will be meaningless. Tests like these allow our future selves to focus on making new mistakes instead of repeating old ones.

8.7 Key Points

- Operations signal conditions in R when errors occur.
- The three built-in levels of conditions are messages, warnings, and errors.
- Programs can signal these themselves using the functions `message`, `warning`, and `stop`.
- Operations can be placed in a call to the function `try` to suppress errors, but this is a bad idea.
- Operations can be placed in a call to the function `tryCatch` to handle errors.
- Use testthat to write unit tests for R.
- Put unit tests for an R package in the `tests/testthat` directory.
- Put tests in files called test_group.R and call them test_something.
- Use `test_dir` to run tests from a particular that match a pattern.
- Write tests for data transformation steps as well as library functions.

Chapter 9

Advanced Topics

FIXME

9.1 Learning Objectives

- Use `reticulate` to share data between R and Python.
- Use `reticulate` to call Python functions from R code and vice versa.
- Run Python scripts directly from R programs.
- Correctly identify the most commonly used object-oriented programming system in R.
- Explain what attributes R and correctly set and query objects' attributes, class, and dimensions.
- Explain how to define a new method for a class.
- Describe and implement the three functions that should be written for any user-defined class.
- FIXME: databases

9.2 How can I use Python with R?

You can put Python code in R Markdown documents:

```
print("Hello R")
```

```
Hello R
```

but how can those chunks interact with your R and vice versa? The answer is a package called reticulate that provides two-way communication between Python and R. To use it, run `install.packages("reticulate")`. By default, it uses the system-default Python:

```
Sys.which("python")
```

```
                 python
"/anaconda3/bin/python"
```

but you can configure it to use different versions, or to use `virtualenv` or a Conda environment—see the document for details.

> If you want to run the Pythonic bits of code we present as well as the R, run `install.packages("reticulate")` and then set the `RETICULATE_PYTHON` environment variable to point at the version of Python you want to use *before* you launch RStudio. This is necessary because you may have a system-installed version somewhere like `/usr/bin/python` and a conda-managed version in `~/anaconda3/bin/python`.

9.2.1 How can I access data across languages?

The most common way to use reticulate is to do some calculations in Python and then use the results in R or vice versa. To show how this works, let's read our infant HIV data into a Pandas data frame:

```
import pandas
data = pandas.read_csv('results/infant_hiv.csv')
print(data.head())
```

```
  country  year  estimate  hi  lo
0     AFG  2009       NaN NaN NaN
1     AFG  2010       NaN NaN NaN
2     AFG  2011       NaN NaN NaN
3     AFG  2012       NaN NaN NaN
4     AFG  2013       NaN NaN NaN
```

All of our Python variables are available in our R session as part of the `py` object, so `py$data` is our data frame inside a chunk of R code:

```
library(reticulate)
head(py$data)
```

```
  country year estimate  hi  lo
1     AFG 2009      NaN NaN NaN
2     AFG 2010      NaN NaN NaN
3     AFG 2011      NaN NaN NaN
4     AFG 2012      NaN NaN NaN
5     AFG 2013      NaN NaN NaN
6     AFG 2014      NaN NaN NaN
```

reticulate handles type conversions automatically, though there are a few tricky cases: for example, the number `9` is a float in R, so if you want an integer in

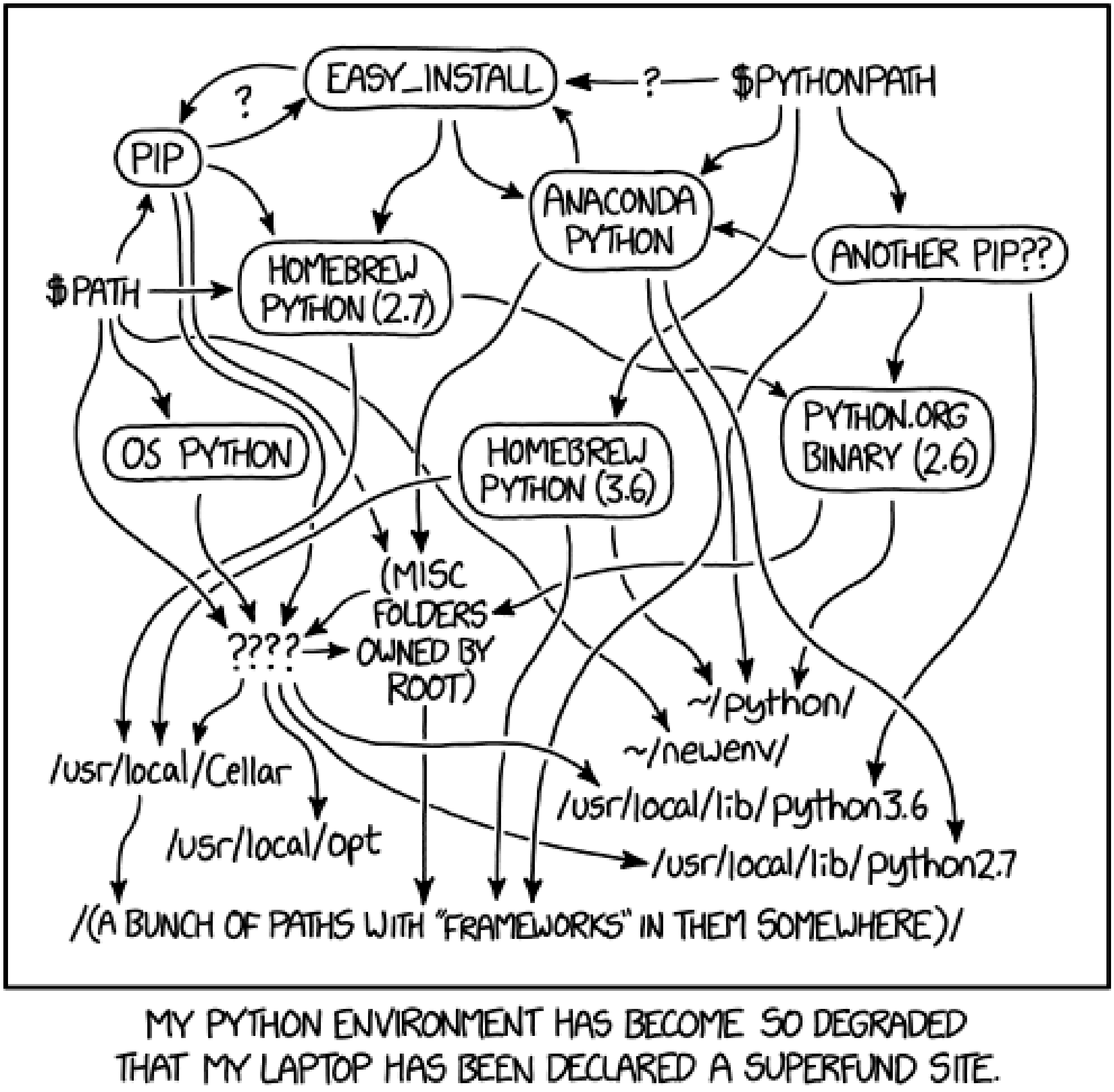


Figure 9.1: XKCD on Python Environments (from https://xkcd.com/1987/)

Python, you have to add the trailing `L` (for "long") and write it `9L`.

On the other hand, reticulate translates between 0-based and 1-based indexing. Suppose we create a character vector in R:

```
elements = c('hydrogen', 'helium', 'lithium', 'beryllium')
```

Hydrogen is in position 1 in R:

```
elements[1]
```

```
[1] "hydrogen"
```

but position 0 in Python:

```
print(r.elements[0])
```

```
hydrogen
```

Note our use of the object `r` in our Python code: just `py$whatever` gives us access to Python objects in R, `r.whatever` gives us access to R objects in Python.

9.2.2 How can I call functions across languages?

We don't have to run Python code, store values in a variable, and then access that variable from R: we can call the Python directly (or vice versa). For example, we can use Python's random number generator in R as follows:

```
pyrand <- import("random")
pyrand$gauss(0, 1)
```

```
[1] -0.7512465
```

(There's no reason to do this—R's random number generator is just as strong—but it illustrates the point.)

We can also source Python scripts. For example, suppose that `countries.py` contains this function:

```
#!/usr/bin/env python

import pandas as pd

def get_countries(filename):
    data = pd.read_csv(filename)
    return data.country.unique()
```

We can run that script using `source_python`:

```
source_python('countries.py')
```

There is no output because all the script did was define a function. By default, that function and all other top-level variables defined in the script are now available in R:

```
get_countries('results/infant_hiv.csv')
```

```
  [1] "AFG" "AGO" "AIA" "ALB" "ARE" "ARG" "ARM" "ATG" "AUS" "AUT" "AZE"
 [12] "BDI" "BEL" "BEN" "BFA" "BGD" "BGR" "BHR" "BHS" "BIH" "BLR" "BLZ"
 [23] "BOL" "BRA" "BRB" "BRN" "BTN" "BWA" "CAF" "CAN" "CHE" "CHL" "CHN"
 [34] "CIV" "CMR" "COD" "COG" "COK" "COL" "COM" "CPV" "CRI" "CUB" "CYP"
 [45] "CZE" "DEU" "DJI" "DMA" "DNK" "DOM" "DZA" "ECU" "EGY" "ERI" "ESP"
 [56] "EST" "ETH" "FIN" "FJI" "FRA" "FSM" "GAB" "GBR" "GEO" "GHA" "GIN"
 [67] "GMB" "GNB" "GNQ" "GRC" "GRD" "GTM" "GUY" "HND" "HRV" "HTI" "HUN"
 [78] "IDN" "IND" "IRL" "IRN" "IRQ" "ISL" "ISR" "ITA" "JAM" "JOR" "JPN"
 [89] "KAZ" "KEN" "KGZ" "KHM" "KIR" "KNA" "KOR" "LAO" "LBN" "LBR" "LBY"
[100] "LCA" "LKA" "LSO" "LTU" "LUX" "LVA" "MAR" "MDA" "MDG" "MDV" "MEX"
[111] "MHL" "MKD" "MLI" "MLT" "MMR" "MNE" "MNG" "MOZ" "MRT" "MUS" "MWI"
[122] "MYS" "NAM" "NER" "NGA" "NIC" "NIU" "NLD" "NOR" "NPL" "NRU" "NZL"
[133] "OMN" "PAK" "PAN" "PER" "PHL" "PLW" "PNG" "POL" "PRK" "PRT" "PRY"
[144] "PSE" "ROU" "RUS" "RWA" "SAU" "SDN" "SEN" "SGP" "SLB" "SLE" "SLV"
[155] "SOM" "SRB" "SSD" "STP" "SUR" "SVK" "SVN" "SWE" "SWZ" "SYC" "SYR"
[166] "TCD" "TGO" "THA" "TJK" "TKM" "TLS" "TON" "TTO" "TUN" "TUR" "TUV"
[177] "TZA" "UGA" "UKR" "UNK" "URY" "USA" "UZB" "VCT" "VEN" "VNM" "VUT"
[188] "WSM" "YEM" "ZAF" "ZMB" "ZWE"
```

There is one small pothole in this. When the script is run, the special Python variable `__name__` is set to `'__main__'"'`, i.e., the script thinks it is being called from the command line. If it includes a conditional block to handle command-line arguments like this:

```
if __name__ == '__main__':
    input_file, output_files = sys.argv[1], sys.argv[2:]
    main(input_file, output_files)
```

then that block will be executed, but will fail because `sys.argv` won't include anything.

9.3 How does object-oriented programming work in R?

Programmers spend a great deal of their time trying to create order out of chaos, and the rest of their time inventing new ways to create more chaos. Object-oriented programming serves both needs well: it allows good software designers to create marvels, and less conscientious or less experienced ones to manufacture horrors.

R has not one, not two, but at least three different frameworks for object-oriented programming. By far the most widely used is S3 (because it was first introduced with Version 3 of S, the language from which R is derived). Unlike the approaches used in Python and similarly pedestrian languages, S3 does not require users to define classes. Instead, they add attributes to data, then write specialized versions of generic functions to process data identified by those attributes. Since attributes can be used in other ways as well, we will start by exploring them.

9.3.1 What are attributes?

Let's begin by creating a matrix containing the first few hundreds:

```
values <- 100 * 1:9 # creates c(100, 200, ..., 900)
m <- matrix(values, nrow = 3, ncol = 3)
m
```

```
     [,1] [,2] [,3]
[1,]  100  400  700
[2,]  200  500  800
[3,]  300  600  900
```

Behind the scenes, R continues to store our nine values as a vector. However, it adds an attribute called `class` to the vector to identify it as a matrix:

```
class(m)
```

```
[1] "matrix"
```

and another attribute called `dim` to store its dimensions as a 2-element vector:

```
dim(m)
```

```
[1] 3 3
```

An object's attributes are simply a set of name-value pairs. We can find out what attributes are present using `attributes` and show or set individual attributes using `attr`:

```
attr(m, "prospects") <- "dismal"
attributes(m)
```

```
$dim
[1] 3 3

$prospects
[1] "dismal"
```

What are the type and attributes of a tibble?

```
t <- tribble(
  ~a, ~b,
  1, 2,
```

```
  3, 4)
typeof(t)
```

```
[1] "list"
```

```
attributes(t)
```

```
$names
[1] "a" "b"

$row.names
[1] 1 2

$class
[1] "tbl_df"     "tbl"        "data.frame"
```

This tells us that a tibble is stored as a list (the first line of output), and that it has an attribute called `names` that stores the names of its columns, another called `row.names` that stores the names of its rows (a feature we should ignore), and three classes. These classes tell R what functions to search for when we are (for example) asking for the length of a tibble (which is the number of rows it contains):

```
length(t)
```

```
[1] 2
```

9.3.2 How are classes represented?

To show how classes and generic functions work together, let's customize the way that 2D coordinates are converted to strings. First, we create two coordinate vectors:

```
first <- c(0.5, 0.7)
class(first) <- "two_d"
print(first)
```

```
[1] 0.5 0.7
attr(,"class")
[1] "two_d"
```

```
second <- c(1.3, 3.1)
class(second) <- "two_d"
print(second)
```

```
[1] 1.3 3.1
attr(,"class")
[1] "two_d"
```

Separately, we define the behavior of `toString` for such objects:

```
toString.two_d <- function(obj){
  paste0("<", obj[1], ", ", obj[2], ">")
}
toString(first)
```

```
[1] "<0.5, 0.7>"
```

```
toString(second)
```

```
[1] "<1.3, 3.1>"
```

S3's protocol is simple: given a function F and an object of class C, S3 looks for a function named F.C. If it doesn't find one, it looks at the object's next class (assuming it has more than one); once its user-assigned classes are exhausted, it uses whatever function the system has defined for its base type (in this case, character vector). We can trace this process by importing the sloop package and calling `s3_dispatch`:

```
library(sloop)
s3_dispatch(toString(first))
```

```
=> toString.two_d
 * toString.default
```

Compare this with calling `toString` on a plain old character vector:

```
s3_dispatch(toString(c(7.1, 7.2)))
```

```
   toString.double
   toString.numeric
=> toString.default
```

The specialized functions associated with a generic function like `toString` are called methods. Unlike languages that require methods to be defined all together as part of a class, S3 allows us to add methods when and as we see fit. But that doesn't mean we should: minds confined to three dimensions of space and one of time are simply not capable of comprehending complex class hierarchies. Instead, we should always write three functions that work together for a class like `two_d`:

- A constructor called `new_two_d` that creates objects of our class.
- An optional validator called `validate_two_d` that checks the consistency and correctness of an object's values.
- An optional helper, simply called `two_d`, that most users will call to create and validate objects.

The constructor's first argument should always be the base object (in our case, the two-element vector). It should also have one argument for each attribute the object is to have, if any. Unlike matrices, our 2D points don't have any extra arguments, so our constructor needs no extra arguments. Crucially, the constructor checks the type of its arguments to ensure that the object has at least some chance of being valid.

```
new_two_d <- function(coordinates){
  stopifnot(is.numeric(coordinates))
  class(coordinates) <- "two_d"
  coordinates
}

example <- new_two_d(c(4.4, -2.2))
toString(example)
```

```
[1] "<4.4, -2.2>"
```

Validators are only needed when checks on data correctness and consistency are expensive. For example, if we were to define a class to represent sorted vectors, checking that each element is no less than its predecessor could take a long time for very long vectors. To illustrate this, we will check that we have exactly two coordinates; in real code, we would probably include this (inexpensive) check in the constructor.

```
validate_two_d <- function(coordinates) {
  stopifnot(length(coordinates) == 2)
  stopifnot(class(coordinates) == "two_d")
}

validate_two_d(example)    # should succeed silently
validate_two_d(c(1, 3))    # should fail
```

```
Error in validate_two_d(c(1, 3)): class(coordinates) == "two_d" is not TRUE
```

```
validate_two_d(c(2, 2, 2)) # should also fail
```

```
Error in validate_two_d(c(2, 2, 2)): length(coordinates) == 2 is not TRUE
```

The third and final function in our trio provides a user-friendly way to construct objects of our new class. It should call the constructor and the validator (if one exists), but should also provide a richer set of defaults, better error messages, and so on. To illustrate this, we shall allow the user to provide either one argument (which must be a two-element vector) or two (which must each be numeric):

```
two_d <- function(...){
  args <- list(...)
  if (length(args) == 1) {
    args <- args[[1]]    # extract original value
  }
  else if (length(args) == 2) {
    args <- unlist(args) # convert list to vector
```

```
  }
  result <- new_two_d(args)
  validate_two_d(result)
  result
}

here <- two_d(10.1, 11.2)
toString(here)
```

```
[1] "<10.1, 11.2>"
```

```
there <- two_d(c(15.6, 16.7))
toString(there)
```

```
[1] "<15.6, 16.7>"
```

9.3.3 How does inheritance work?

We said above that an object can have more than one class, and that S3 searches the classes in order when it wants to find a method to call. Methods can also trigger invocation of other methods explicitly in order to supplement, rather than replace, the behavior of other classes. To show how this works, we shall look at that classic of object-oriented design: shapes. (The safe kind, of course, not those whose non-Euclidean angles have placed such intolerable stress on the minds of so many of our colleagues over the years.) We start by defining a `polygon` class:

```
new_polygon <- function(coords, name) {
  points <- map(coords, two_d)
  class(points) <- "polygon"
  attr(points, "name") <- name
  points
}

toString.polygon <- function(poly) {
  paste0(attr(poly, "name"), ": ", paste0(map(poly, toString), collapse = ", "))
}

right <- new_polygon(list(c(0, 0), c(1, 0), c(0, 1)), "triangle")
toString(right)
```

```
[1] "triangle: <0, 0>, <1, 0>, <0, 1>"
```

Now we will add colored shapes:

```
new_colored_polygon <- function(coords, name, color) {
  object <- new_polygon(coords, name)
  attr(object, "color") <- color
```

```
  class(object) <- c("colored_polygon", class(object))
  object
}

pinkish <- new_colored_polygon(list(c(0, 0), c(1, 0), c(1, 1)), "triangle", "roseate")
class(pinkish)
```

```
[1] "colored_polygon" "polygon"
```

```
toString(pinkish)
```

```
[1] "triangle: <0, 0>, <1, 0>, <1, 1>"
```

So far so good: since we have not defined a method to handle colored polygons specifically, we get the behavior for a regular polygon. Let's add another method that supplements the behavior of the existing method:

```
toString.colored_polygon <- function(poly) {
  paste0(toString.polygon(poly), "+ color = ", attr(poly, "color"))
}

toString(pinkish)
```

```
[1] "triangle: <0, 0>, <1, 0>, <1, 1>+ color = roseate"
```

In practice, we will almost always place all of the methods associated with a class in the same file as its constructor, validator, and helper. The time has finally come for us to explore projects and packages.

9.4 How can I write web applications in R?

R has this awesome gnarly web programming framework called Shiny. It uses ~~sympathetic magic~~ ~~quantum entanglement~~ reactive variables to update the application's interface when data changes. You should, like, totally check it out.

9.5 How can I work with relational databases in R?

Data frames and database tables go together as naturally as chocolate and the tears of our fallen foes. As in Python and other languages, there is a standard interface for connecting to and querying relational databases; each database is then supported by a package that implements that interface. This doesn't completely hide the differences between databases—we must still worry about the quirks of various SQL dialects—but it does keep the R side of things simple.

This tutorial uses the SQLite database and the RSQLite interface package. The former is included with the latter, so `install.packages("RSQLite")` will give you everything you need. We assume that you already speak enough SQL to get yourself into trouble; if you do not, this tutorial is a good place to start.

9.5.1 How can I get data from a database?

Suppose we have a small database in `data/example.db` containing survey data salvaged from a series of doomed expeditions to the Antarctic in the 1920s and 1930s. The database contains four tables:

Person: people who took readings.

person_id	personal	family
dyer	William	Dyer
pb	Frank	Pabodie
lake	Anderson	Lake
roe	Valentina	Roerich
danforth	Frank	Danforth

Site: locations where readings were taken.

site_id	lat	long
DR-1	-49.85	-128.57
DR-3	-47.15	-126.72
MSK-4	-48.87	-123.4

Visited: when readings were taken at specific sites.

visit_id	site_id	dated
619	DR-1	1927-02-08
622	DR-1	1927-02-10
734	DR-3	1930-01-07
735	DR-3	1930-01-12
751	DR-3	1930-02-26
752	DR-3	-null-
837	MSK-4	1932-01-14
844	DR-1	1932-03-22

Measurements: the actual readings.

visit_id	visitor	quantity	reading
619	dyer	rad	9.82

visit_id	visitor	quantity	reading
619	dyer	sal	0.13
622	dyer	rad	7.8
622	dyer	sal	0.09
734	pb	rad	8.41
734	lake	sal	0.05
734	pb	temp	-21.5
735	pb	rad	7.22
735	-null-	sal	0.06
735	-null-	temp	-26.0
751	pb	rad	4.35
751	pb	temp	-18.5
751	lake	sal	0.1
752	lake	rad	2.19
752	lake	sal	0.09
752	lake	temp	-16.0
752	roe	sal	41.6
837	lake	rad	1.46
837	lake	sal	0.21
837	roe	sal	22.5
844	roe	rad	11.25

Let's get the data about the people into a data frame:

```
library(DBI)
db <- dbConnect(RSQLite::SQLite(), here::here("data", "example.db"))
dbGetQuery(db, "select * from Person;")
```

```
  person_id personal_name family_name
1      dyer       William        Dyer
2        pb         Frank     Pabodie
3      lake      Anderson        Lake
4       roe     Valentina     Roerich
5  danforth         Frank    Danforth
```

That seems simple enough: the database connection is the first argument to `dbGetQuery`, the query itself is the second, and the result is a tibble whose column names correspond to the names of the fields in the database table. What if we want to parameterize our query? Inside the text of the query, we use `:name` as a placeholder for a query parameter, then pass a list of name-value pairs to specify what we actually want:

```
dbGetQuery(db,
           "select * from Measurements where quantity = :desired",
           params = list(desired = "rad"))
```

```
  visit_id person_id quantity reading
1      619      dyer      rad    9.82
2      622      dyer      rad    7.80
3      734        pb      rad    8.41
4      735        pb      rad    7.22
5      751        pb      rad    4.35
6      752      lake      rad    2.19
7      837      lake      rad    1.46
8      844       roe      rad   11.25
```

Do *not* use `glue` or some other kind of string interpolation to construct database queries, as this can leave you open to SQL injection attacks and other forms of digital damnation.

If you expect a large set of results, it's best to page through them:

```
results <- dbSendQuery(db, "select * from Measurements limit 15;")
while (!dbHasCompleted(results)) {
  chunk <- dbFetch(results, n = 3) # artificially low for tutorial purposes
  print(chunk)
}
```

```
  visit_id person_id quantity reading
1      619      dyer      rad    9.82
2      619      dyer      sal    0.13
3      622      dyer      rad    7.80
  visit_id person_id quantity reading
1      622      dyer      sal    0.09
2      734        pb      rad    8.41
3      734      lake      sal    0.05
  visit_id person_id quantity reading
1      734        pb     temp  -21.50
2      735        pb      rad    7.22
3      735      <NA>      sal    0.06
  visit_id person_id quantity reading
1      735      <NA>     temp  -26.00
2      751        pb      rad    4.35
3      751        pb     temp  -18.50
```

```
Warning in result_fetch(res@ptr, n = n): Column `reading`: mixed type,
first seen values of type real, coercing other values of type string
```

```
  visit_id person_id quantity reading
1      751      lake      sal    0.00
2      752      lake      rad    2.19
3      752      lake      sal    0.09
```

```
dbClearResult(results)
```

FIXME: why the error?

9.5.2 How can I populate databases with R?

Data scientists spend most of their time reading data, but someone has to create it. RSQLite makes it easy to map a data frame directly to a database table; to show how it works, we will create an in-memory database:

```
colors <- tribble(
  ~name, ~red, ~green, ~blue,
  'black', 0, 0, 0,
  'yellow', 255, 255, 0,
  'aqua', 0, 255, 255,
  'fuchsia', 255, 0, 0
)
db <- dbConnect(RSQLite::SQLite(), ':memory:')
dbWriteTable(db, "colors", colors)
```

Let's see what the combination of R and SQLite has done with our data and the types thereof:

```
dbGetQuery(db, "select * from colors;")
```

```
     name red green blue
1   black   0     0    0
2  yellow 255   255    0
3    aqua   0   255  255
4 fuchsia 255     0    0
```

Good: the types have been guessed correctly.

But what about dates?

```
appointments <- tribble(
  ~who, ~when,
  'Dyer', '1927-03-01',
  'Peabody', '1927-05-05'
) %>% mutate(when = lubridate::as_date(when))
dbWriteTable(db, "appointments", appointments)
dbGetQuery(db, "select * from appointments;")
```

```
      who   when
1    Dyer -15647
2 Peabody -15582
```

What fresh hell is this? After considerable trial and error, we discover that our dates have been returned to us as the number of days since January 1, 1970:

```
dbExecute(db,
          "insert into appointments values('Testing', :the_date);",
```

```
          params = list(the_date = lubridate::as_date('1971-01-01')))
```

```
[1] 1
```

```
dbGetQuery(db, "select * from appointments where who = 'Testing';")
```

```
      who when
1 Testing  365
```

There is no point screaming: those who might pity you cannot hear, and those who can hear will definitely not pity you.

9.6 Key Points

- The `reticulate` library allows R programs to access data in Python programs and vice versa.
- Use `py.whatever` to access a top-level Python variable from R.
- Use `r.whatever` to access a top-level R definition from Python.
- R is always indexed from 1 (even in Python) and Python is always indexed from 0 (even in R).
- Numbers in R are floating point by default, so use a trailing 'L' to force a value to be an integer.
- A Python script run from an R session believes it is the main script, i.e., `__name__` is `'__main__'` inside the Python script.
- S3 is the most commonly used object-oriented programming system in R.
- Every object can store metadata about itself in attributes, which are set and queried with `attr`.
- The `dim` attribute stores the dimensions of a matrix (which is physically stored as a vector).
- The `class` attribute of an object defines its class or classes (it may have several character entries).
- When `F(X, ...)` is called, and `X` has class `C`, R looks for a function called `F.C` (the `.` is just a naming convention).
- If an object has multiple classes in its `class` attribute, R looks for a corresponding method for each in turn.
- Every user defined class `C` should have functions `new_C` (to create it), `validate_C` (to validate its integrity), and `C` (to create and validate).
- Use the `DBI` package to work with relational databases.
- Use `DBI::dbConnect(...)` with database-specific parameters to connect to a specific database.
- Use `dbGetQuery(connection, "query")` to send an SQL query string to a database and get a data frame of results.
- Parameterize queries using `:name` as a placeholder in the query and `params = list(name = value)` as a third parameter to `dbGetQuery` to specify actual values.
- Use `dbFetch` in a `while` loop to page results.

- Use `dbWriteTable` to write an entire data frame to a table, and `dbExecute` to execute a single insertion statement.
- Dates… why did it have to be dates?

Appendix A

License

This is a human-readable summary of (and not a substitute for) the license. Please see https://creativecommons.org/licenses/by/4.0/legalcode for the full legal text.

This work is licensed under the Creative Commons Attribution 4.0 International license (CC-BY-4.0).

You are free to:

- **Share**—copy and redistribute the material in any medium or format
- **Remix**—remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

- **Attribution**—You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- **No additional restrictions**—You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Notices:

You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation.

No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material.

Appendix B

Code of Conduct

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

B.1 Our Standards

Examples of behavior that contributes to creating a positive environment include:

- using welcoming and inclusive language,
- being respectful of differing viewpoints and experiences,
- gracefully accepting constructive criticism,
- focusing on what is best for the community, and
- showing empathy towards other community members.

Examples of unacceptable behavior by participants include:

- the use of sexualized language or imagery and unwelcome sexual attention or advances,
- trolling, insulting/derogatory comments, and personal or political attacks,
- public or private harassment,
- publishing others' private information, such as a physical or electronic address, without explicit permission, and
- other conduct which could reasonably be considered inappropriate in a professional setting

B.2 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

B.3 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

B.4 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by emailing the project team. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

B.5 Attribution

This Code of Conduct is adapted from the Contributor Covenant version 1.4.

Appendix C

Citation

Please cite this work as:

> Greg Wilson: *The Tidynomicon: A Brief Introduction to R for Python Programmers.* https://github.com/gvwilson/tidynomicon, 2018.

Appendix D

Contributing

Contributions of all kinds are welcome, from errata and minor improvements to entirely new sections and chapters: please email us or submit an issue or pull request to our GitHub repository. Everyone whose work is incorporated will be acknowledged; please note that all contributors are required to abide by our Code of Conduct (s:conduct).

The Jekyll template used in this tutorial can support multiple languages. All English content should go in the `_en` directory. (Please note that we use Simplified English rather than Traditional English, i.e., American rather than British spelling and grammar.) We encourage translations; if you would like to take this on, please email us.

If you wish to report errata or suggest improvements to wording, please include the chapter name in the first line of the body of your report (e.g., `Testing Data Analysis`).

Appendix E

Practice Problems

You need more practice with the functions in Chapter 3. To begin, open a fresh file and begin by loading the tidyverse and the here package used to construct paths:

```
library(tidyverse)
library(here)
```

Next, use `here::here` to construct a path to a file and `readr::read_csv` to read that file:

```
path = here::here("data", "person.csv")
person <- readr::read_csv(path)
```

```
Parsed with column specification:
cols(
  person_id = col_character(),
  personal_name = col_character(),
  family_name = col_character()
)
```

We don't need to write out fully-qualified names—`here` and `read_csv` will do—but we will use them to make it easier to see what comes from where.

Next, have a look at the tibble `person`, which contains some basic information about a group of foolhardy scientists who ventured into the Antarctic in the 1920s and 1930s in search of things best left undisturbed:

```
person
```

```
# A tibble: 5 x 3
  person_id personal_name family_name
  <chr>     <chr>         <chr>
1 dyer      William       Dyer
```

```
2 pb        Frank       Pabodie
3 lake      Anderson    Lake
4 roe       Valentina   Roerich
5 danforth  Frank       Danforth
```

How many rows and columns does this tibble contain?

```
nrow(person)
```

```
[1] 5
```

```
ncol(person)
```

```
[1] 3
```

(These names don't have a package prefix because they are built in.) Let's show that information in a slightly nicer way using `glue` to insert values into a string and `print` to display the result:

```
print(glue::glue("person has {nrow(person)} rows and {ncol(person)} columns"))
```

```
person has 5 rows and 3 columns
```

If we want to display several values, we can use the function `paste` to combine the elements of a vector. `colnames` gives us the names of a tibble's columns, and `paste`'s `collapse` argument tells the function to use a single space to separate concatenated values:

```
print(glue::glue("person columns are {paste(colnames(person), collapse = ' ')}"))
```

```
person columns are person_id personal_name family_name
```

Time for some data manipulation. Let's get everyone's family and personal names:

```
dplyr::select(person, family_name, personal_name)
```

```
# A tibble: 5 x 2
  family_name personal_name
  <chr>       <chr>
1 Dyer        William
2 Pabodie     Frank
3 Lake        Anderson
4 Roerich     Valentina
5 Danforth    Frank
```

and then filter that list to keep only those that come in the first half of the alphabet:

```
dplyr::select(person, family_name, personal_name) %>%
  dplyr::filter(family_name < "N")
```

```
# A tibble: 3 x 2
  family_name personal_name
  <chr>       <chr>
1 Dyer        William
2 Lake        Anderson
3 Danforth    Frank
```

It would be more consistent to rewrite this as:

```
person %>%
  dplyr::select(family_name, personal_name) %>%
  dplyr::filter(family_name < "N")
```

```
# A tibble: 3 x 2
  family_name personal_name
  <chr>       <chr>
1 Dyer        William
2 Lake        Anderson
3 Danforth    Frank
```

It's easy to add a column that records the lengths of family names:

```
person %>%
  dplyr::mutate(name_length = stringr::str_length(family_name))
```

```
# A tibble: 5 x 4
  person_id personal_name family_name name_length
  <chr>     <chr>         <chr>             <int>
1 dyer      William       Dyer                  4
2 pb        Frank         Pabodie               7
3 lake      Anderson      Lake                  4
4 roe       Valentina     Roerich               7
5 danforth  Frank         Danforth              8
```

and then arrange in descending order:

```
person %>%
  dplyr::mutate(name_length = stringr::str_length(family_name)) %>%
  dplyr::arrange(dplyr::desc(name_length))
```

```
# A tibble: 5 x 4
  person_id personal_name family_name name_length
  <chr>     <chr>         <chr>             <int>
1 danforth  Frank         Danforth              8
2 pb        Frank         Pabodie               7
3 roe       Valentina     Roerich               7
4 dyer      William       Dyer                  4
5 lake      Anderson      Lake                  4
```

E.1 Do I need even more practice?

Yes. Yes, you do. Let's load a slightly larger dataset:

```
measurements <- readr::read_csv(here::here("data", "measurements.csv"))
```

```
Parsed with column specification:
cols(
  visit_id = col_double(),
  person_id = col_character(),
  quantity = col_character(),
  reading = col_double()
)
```

```
measurements
```

```
# A tibble: 21 x 4
   visit_id person_id quantity reading
      <dbl> <chr>     <chr>      <dbl>
 1      619 dyer      rad         9.82
 2      619 dyer      sal         0.13
 3      622 dyer      rad         7.8
 4      622 dyer      sal         0.09
 5      734 pb        rad         8.41
 6      734 lake      sal         0.05
 7      734 pb        temp      -21.5
 8      735 pb        rad         7.22
 9      735 <NA>      sal         0.06
10      735 <NA>      temp      -26
# ... with 11 more rows
```

If we want an overview of our data's properties, we can use the aptly-named `summarize` function:

```
dplyr::summarize(measurements)
```

```
# A tibble: 1 x 0
```

Removing records with missing readings is straightforward:

```
measurements %>%
  dplyr::filter(!is.na(reading))
```

```
# A tibble: 20 x 4
   visit_id person_id quantity reading
      <dbl> <chr>     <chr>      <dbl>
 1      619 dyer      rad         9.82
 2      619 dyer      sal         0.13
 3      622 dyer      rad         7.8
 4      622 dyer      sal         0.09
```

```
 5      734 pb        rad         8.41
 6      734 lake      sal         0.05
 7      734 pb        temp      -21.5
 8      735 pb        rad         7.22
 9      735 <NA>      sal         0.06
10      735 <NA>      temp      -26
11      751 pb        rad         4.35
12      751 pb        temp      -18.5
13      752 lake      rad         2.19
14      752 lake      sal         0.09
15      752 lake      temp      -16
16      752 roe       sal        41.6
17      837 lake      rad         1.46
18      837 lake      sal         0.21
19      837 roe       sal        22.5
20      844 roe       rad        11.2
```

Removing rows that contain *any* NAs is equally easy, though it may be statistically unsound:

We can now group our data by the quantity measured and count the number of each—the column is named `n` automatically:

```
cleaned %>%
  dplyr::group_by(quantity) %>%
  dplyr::count()
```

```
# A tibble: 3 x 2
# Groups:   quantity [3]
  quantity     n
  <chr>    <int>
1 rad          8
2 sal          7
3 temp         3
```

How are the readings of each type distributed?

```
cleaned %>%
  dplyr::group_by(quantity) %>%
  dplyr::summarize(low = min(reading),
                   mid = mean(reading),
                   high = max(reading))
```

```
# A tibble: 3 x 4
  quantity    low    mid  high
  <chr>     <dbl>  <dbl> <dbl>
1 rad        1.46   6.56  11.2
2 sal        0.05   9.24  41.6
```

```
3 temp     -21.5  -18.7  -16
```

After inspection, we realize that most of the salinity measurements lie between 0 and 1, but a handful range up to 100. During a brief interval of lucidity, the librarian who collected the battered notebooks from which the data was transcribed informs us that one of the explorers recorded percentages rather than actual values. We therefore decide to normalize all salinity measurements greater than 1.0 using `ifelse` (a two-branch analog of `case_when`):

```
cleaned <- cleaned %>%
  dplyr::mutate(reading = ifelse(quantity == 'sal' & reading > 1.0,
                                 reading/100,
                                 reading))
cleaned
```

```
# A tibble: 18 x 4
   visit_id person_id quantity reading
      <dbl> <chr>     <chr>      <dbl>
 1      619 dyer      rad         9.82
 2      619 dyer      sal         0.13
 3      622 dyer      rad         7.8
 4      622 dyer      sal         0.09
 5      734 pb        rad         8.41
 6      734 lake      sal         0.05
 7      734 pb        temp      -21.5
 8      735 pb        rad         7.22
 9      751 pb        rad         4.35
10      751 pb        temp      -18.5
11      752 lake      rad         2.19
12      752 lake      sal         0.09
13      752 lake      temp      -16
14      752 roe       sal         0.416
15      837 lake      rad         1.46
16      837 lake      sal         0.21
17      837 roe       sal         0.225
18      844 roe       rad        11.2
```

To answer our next set of questions, we need data about when each site was visited. Let's read `visited.csv` and discard entries that are missing the visit date:

```
visited <- readr::read_csv(here::here("data", "visited.csv")) %>%
  dplyr::filter(!is.na(visit_date))
```

```
Parsed with column specification:
cols(
  visit_id = col_double(),
  site_id = col_character(),
```

```
  visit_date = col_date(format = "")
)
```

```
visited
```

```
# A tibble: 7 x 3
  visit_id site_id visit_date
     <dbl> <chr>   <date>
1      619 DR-1    1927-02-08
2      622 DR-1    1927-02-10
3      734 DR-3    1930-01-07
4      735 DR-3    1930-01-12
5      751 DR-3    1930-02-26
6      837 MSK-4   1932-01-14
7      844 DR-1    1932-03-22
```

and then combine that table with our cleaned measurement data. We will use an inner join that matches records on the visit ID; dplyr also provides other kinds of joins should we need them.

```
combined <- visited %>%
  dplyr::inner_join(cleaned,
                    by = c("visit_id" = "visit_id"))
```

We can now find the date of the highest radiation reading at each site:

```
combined %>%
  dplyr::filter(quantity == "rad") %>%
  dplyr::group_by(site_id) %>%
  dplyr::mutate(max_rad = max(reading)) %>%
  dplyr::filter(reading == max_rad)
```

```
# A tibble: 3 x 7
# Groups:   site_id [3]
  visit_id site_id visit_date person_id quantity reading max_rad
     <dbl> <chr>   <date>     <chr>     <chr>      <dbl>   <dbl>
1      734 DR-3    1930-01-07 pb        rad         8.41    8.41
2      837 MSK-4   1932-01-14 lake      rad         1.46    1.46
3      844 DR-1    1932-03-22 roe       rad        11.2    11.2
```

or:

```
combined %>%
  dplyr::filter(quantity == "rad") %>%
  dplyr::group_by(site_id) %>%
  dplyr::top_n(1, reading) %>%
  dplyr::select(site_id, visit_date, reading)
```

```
# A tibble: 3 x 3
# Groups:   site_id [3]
```

```
  site_id visit_date reading
  <chr>   <date>       <dbl>
1 DR-3    1930-01-07    8.41
2 MSK-4   1932-01-14    1.46
3 DR-1    1932-03-22   11.2
```

The function `dplyr::lag` shifts the values in a column. We can use it to calculate the difference in radiation at each site between visits:

```
combined %>%
  dplyr::filter(quantity == "rad") %>%
  dplyr::group_by(site_id) %>%
  dplyr::mutate(delta_rad = reading - dplyr::lag(reading)) %>%
  dplyr::arrange(site_id, visit_date)
```

```
# A tibble: 7 x 7
# Groups:   site_id [3]
  visit_id site_id visit_date person_id quantity reading delta_rad
     <dbl> <chr>   <date>     <chr>     <chr>      <dbl>     <dbl>
1      619 DR-1    1927-02-08 dyer      rad         9.82     NA
2      622 DR-1    1927-02-10 dyer      rad         7.8      -2.02
3      844 DR-1    1932-03-22 roe       rad        11.2       3.45
4      734 DR-3    1930-01-07 pb        rad         8.41     NA
5      735 DR-3    1930-01-12 pb        rad         7.22     -1.19
6      751 DR-3    1930-02-26 pb        rad         4.35     -2.87
7      837 MSK-4   1932-01-14 lake      rad         1.46     NA
```

Going one step further, we can create a list of sites at which radiation increased between any two visits:

```
combined %>%
  dplyr::filter(quantity == "rad") %>%
  dplyr::group_by(site_id) %>%
  dplyr::mutate(delta_rad = reading - dplyr::lag(reading)) %>%
  dplyr::filter(!is.na(delta_rad)) %>%
  dplyr::summarize(any_increase = any(delta_rad > 0)) %>%
  dplyr::filter(any_increase)
```

```
# A tibble: 1 x 2
  site_id any_increase
  <chr>   <lgl>
1 DR-1    TRUE
```

E.2 Please may I create some charts?

Certainly. We will use data on the mass and home range area (HRA) of various species from:

Tamburello N, Côté IM, Dulvy NK (2015) Data from: Energy and the scaling of animal space use. Dryad Digital Repository. https://doi.org/10.5061/dryad.q5j65

```
hra <- readr::read_csv(here::here("data", "home-range-database.csv"))
```

```
Parsed with column specification:
cols(
  .default = col_character(),
  mean.mass.g = col_double(),
  log10.mass = col_double(),
  mean.hra.m2 = col_double(),
  log10.hra = col_double(),
  preymass = col_double(),
  log10.preymass = col_double(),
  PPMR = col_double()
)
```

```
See spec(...) for full column specifications.
```

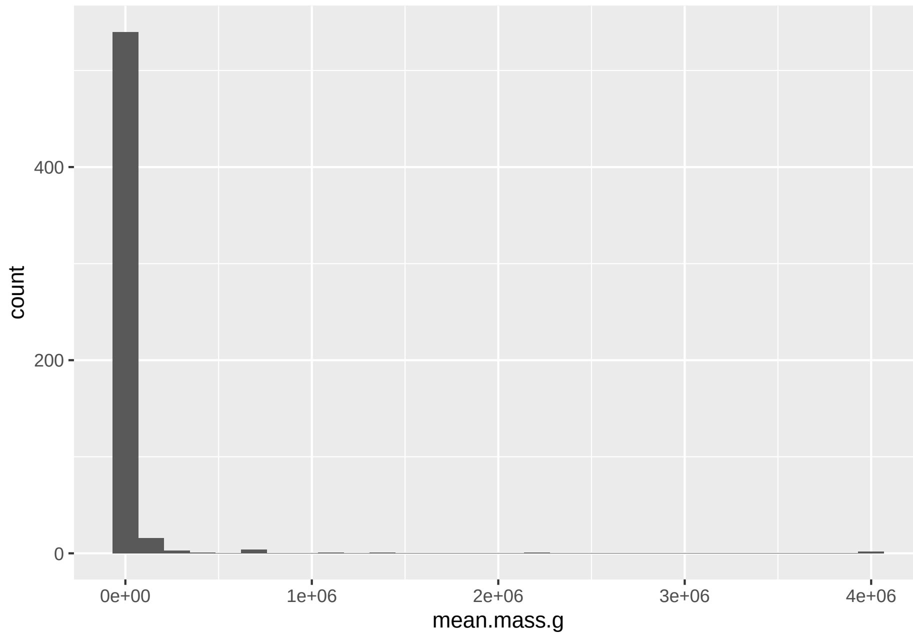
```
head(hra)
```

```
# A tibble: 6 x 24
  taxon common.name class order family genus species primarymethod N
  <chr> <chr>       <chr> <chr> <chr>  <chr> <chr>   <chr>         <chr>
1 lake~ american e~ acti~ angu~ angui~ angu~ rostra~ telemetry     16
2 rive~ blacktail ~ acti~ cypr~ catos~ moxo~ poecil~ mark-recaptu~ <NA>
3 rive~ central st~ acti~ cypr~ cypri~ camp~ anomal~ mark-recaptu~ 20
4 rive~ rosyside d~ acti~ cypr~ cypri~ clin~ fundul~ mark-recaptu~ 26
5 rive~ longnose d~ acti~ cypr~ cypri~ rhin~ catara~ mark-recaptu~ 17
6 rive~ muskellunge acti~ esoc~ esoci~ esox  masqui~ telemetry     5
# ... with 15 more variables: mean.mass.g <dbl>, log10.mass <dbl>,
#   alternative.mass.reference <chr>, mean.hra.m2 <dbl>, log10.hra <dbl>,
#   hra.reference <chr>, realm <chr>, thermoregulation <chr>,
#   locomotion <chr>, trophic.guild <chr>, dimension <chr>,
#   preymass <dbl>, log10.preymass <dbl>, PPMR <dbl>,
#   prey.size.reference <chr>
```

A few keystrokes show us how the masses of these animals are distributed:

```
ggplot2::ggplot(hra) +
  ggplot2::geom_histogram(mapping = aes(x = mean.mass.g))
```

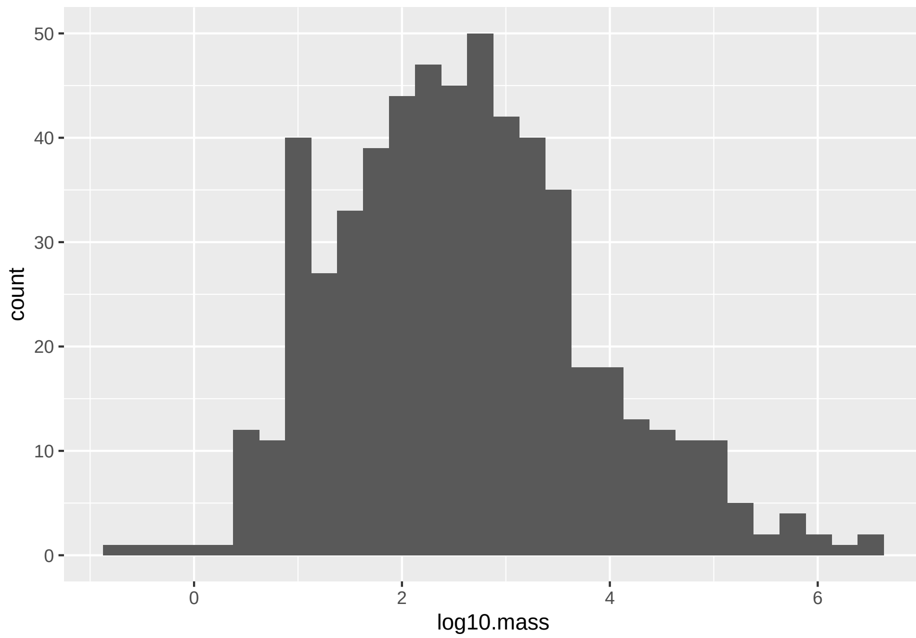
```
`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



The distribution becomes much clearer if we plot the logarithms of the masses, which are helpfully precalculated in `log10.mass`:

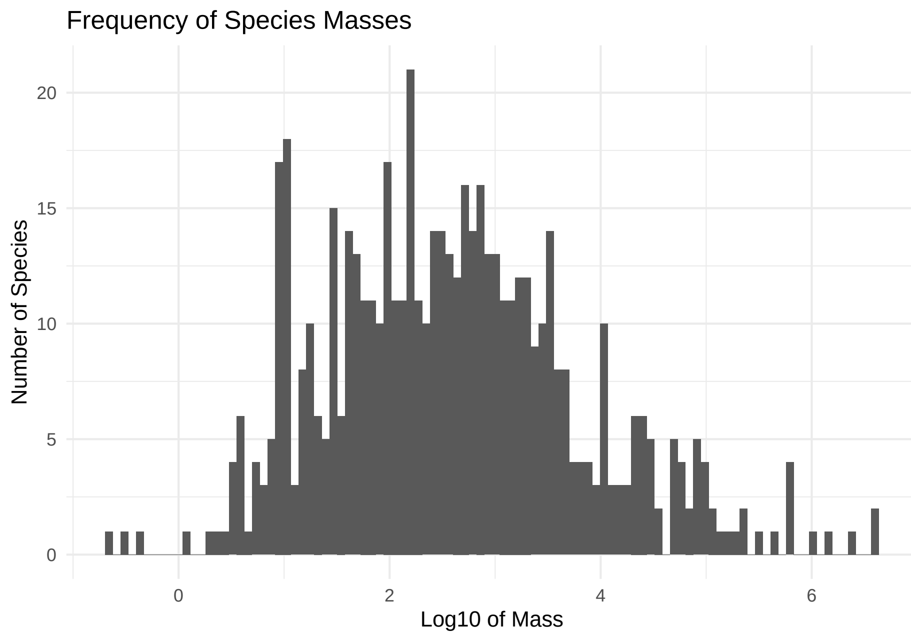
```
ggplot2::ggplot(hra) +
  ggplot2::geom_histogram(mapping = aes(x = log10.mass))
```

```
`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



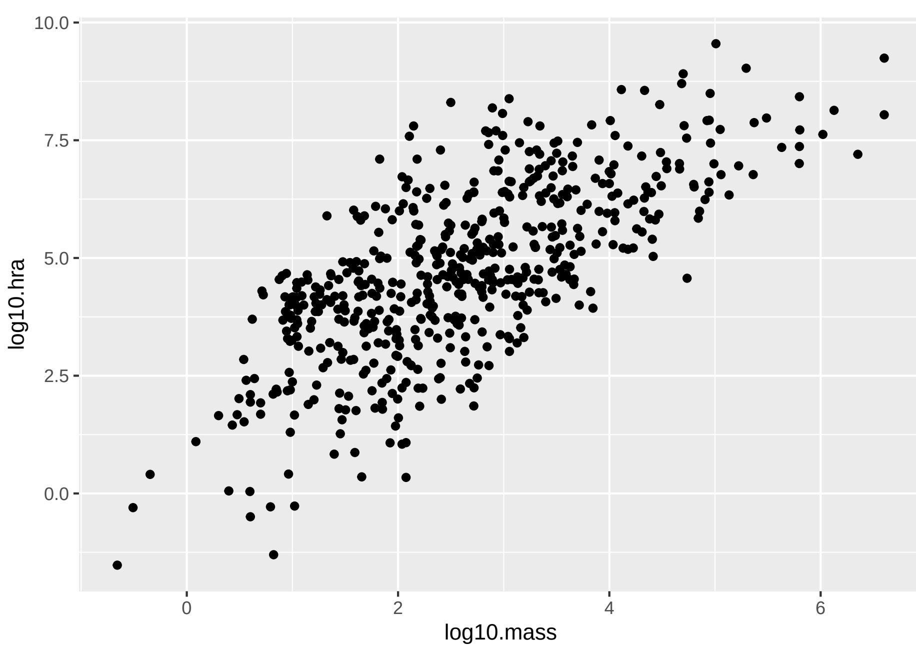
Let's tidy that up a bit:

```
ggplot2::ggplot(hra) +
  ggplot2::geom_histogram(mapping = aes(x = log10.mass), bins = 100) +
  ggplot2::ggtitle("Frequency of Species Masses") +
  ggplot2::xlab("Log10 of Mass") +
  ggplot2::ylab("Number of Species") +
  ggplot2::theme_minimal()
```



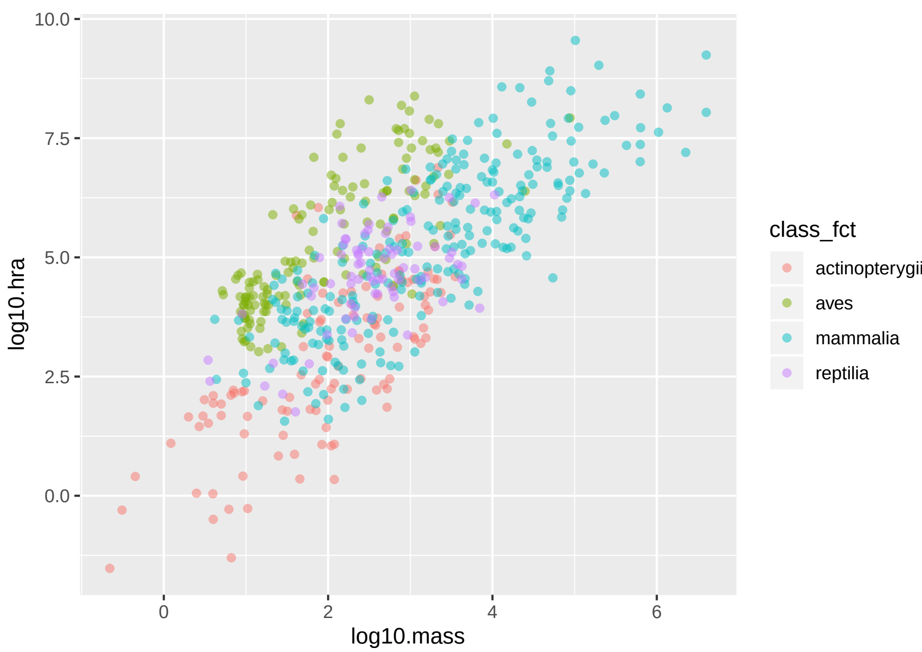
How are mass and home range area related?

```
ggplot2::ggplot(hra) +
  ggplot2::geom_point(mapping = aes(x = log10.mass, y = log10.hra))
```



Does the relationship depend on the class of animal? (Here, we use the word "class" in the biological sense: the class "aves" is birds.)

```
hra %>%
  dplyr::mutate(class_fct = as.factor(class)) %>%
  ggplot2::ggplot(mapping = aes(x = log10.mass, y = log10.hra, color = class_fct)) +
  ggplot2::geom_point(alpha = 0.5)
```

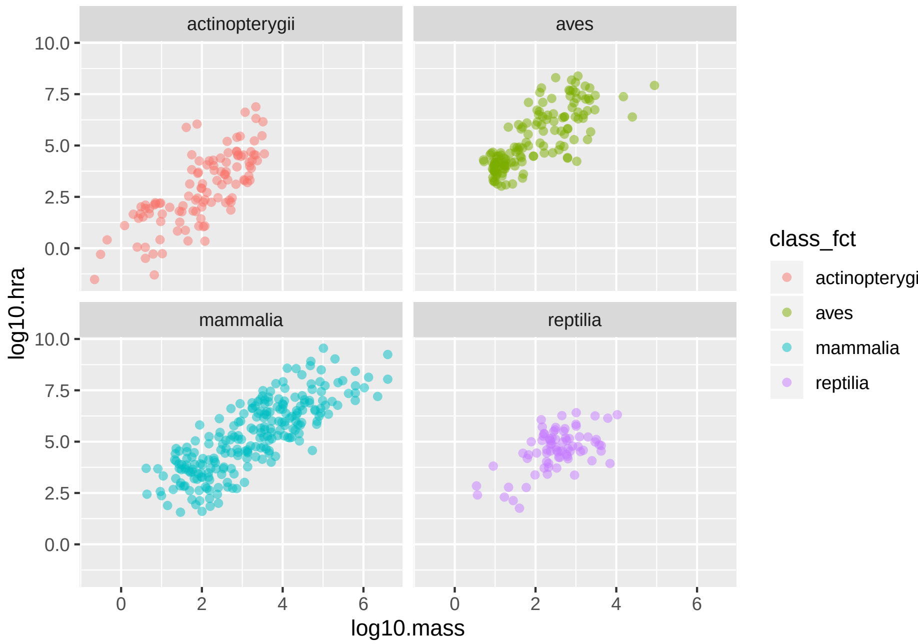


*What's a Factor?

The code above creates a new column `class_fct` by converting the text values in `class` to a factor. Other languages call this an enumeration: we will discuss factors in more detail in Chapter 7.

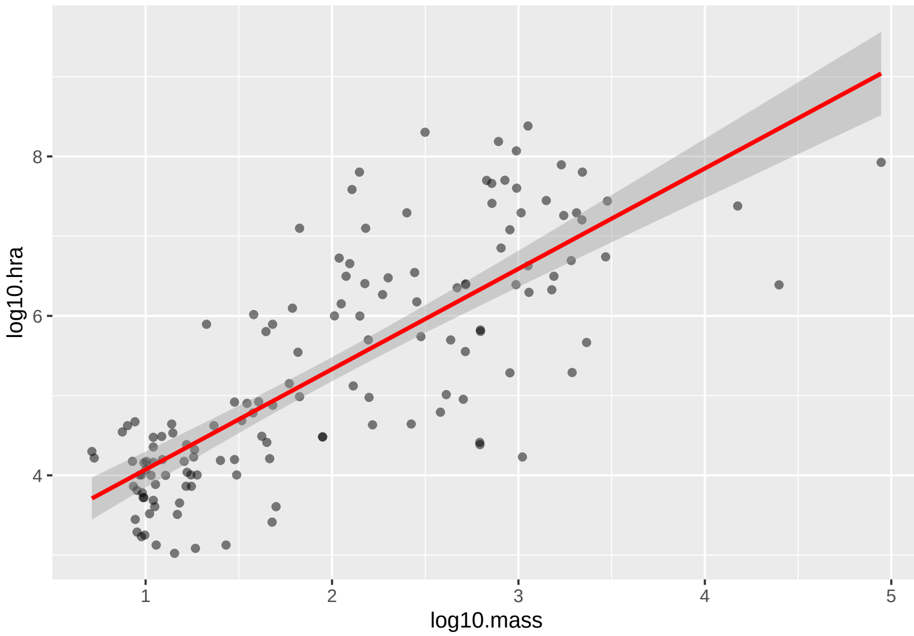
Our chart may be clearer if we display the facets separately:

```
hra %>%
  dplyr::mutate(class_fct = as.factor(class)) %>%
  ggplot2::ggplot(mapping = aes(x = log10.mass, y = log10.hra, color = class_fct)) +
  ggplot2::geom_point(alpha = 0.5) +
  ggplot2::facet_wrap(~class_fct)
```



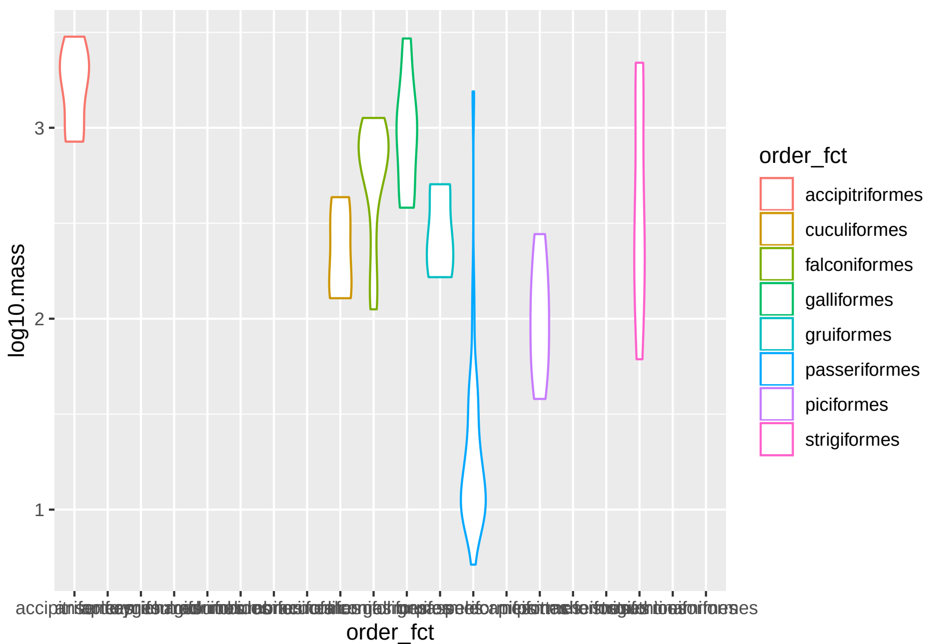
If we want to look at the mass-area relationship more closely for birds, we can construct a regression line:

```
hra %>%
  dplyr::filter(class == "aves") %>%
  ggplot2::ggplot(mapping = aes(x = log10.mass, y = log10.hra)) +
  ggplot2::geom_point(alpha = 0.5) +
  ggplot2::geom_smooth(method = lm, color = 'red')
```



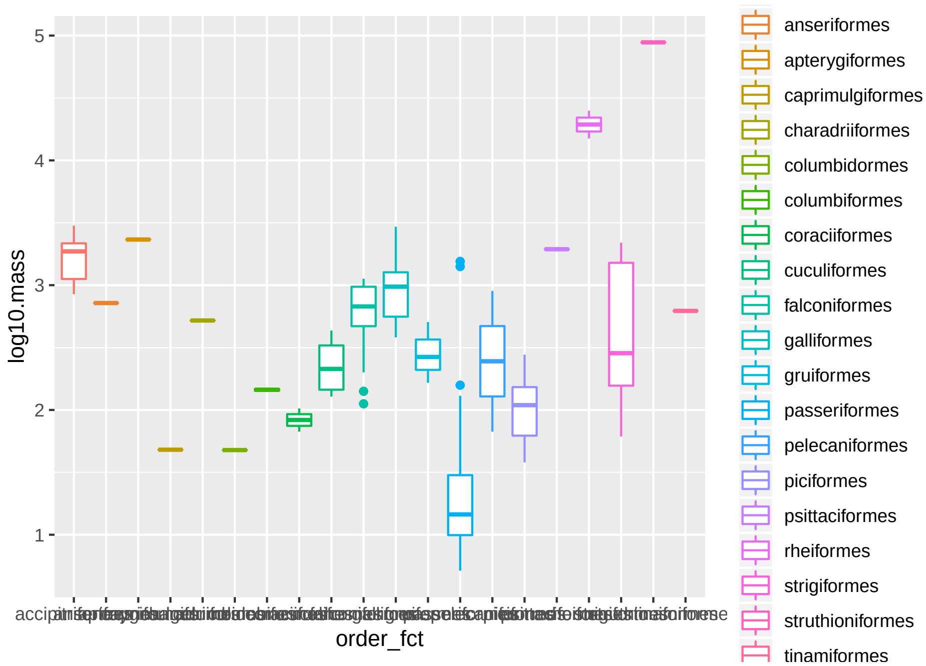
Drilling down even further, we can create a violin plot of mass by order for the birds (where "order" is the biological division below "class"):

```
hra %>%
  dplyr::filter(class == "aves") %>%
  dplyr::mutate(order_fct = as.factor(order)) %>%
  ggplot2::ggplot(mapping = aes(x = order_fct, y = log10.mass, color = order_fct)) +
  ggplot2::geom_violin()
```



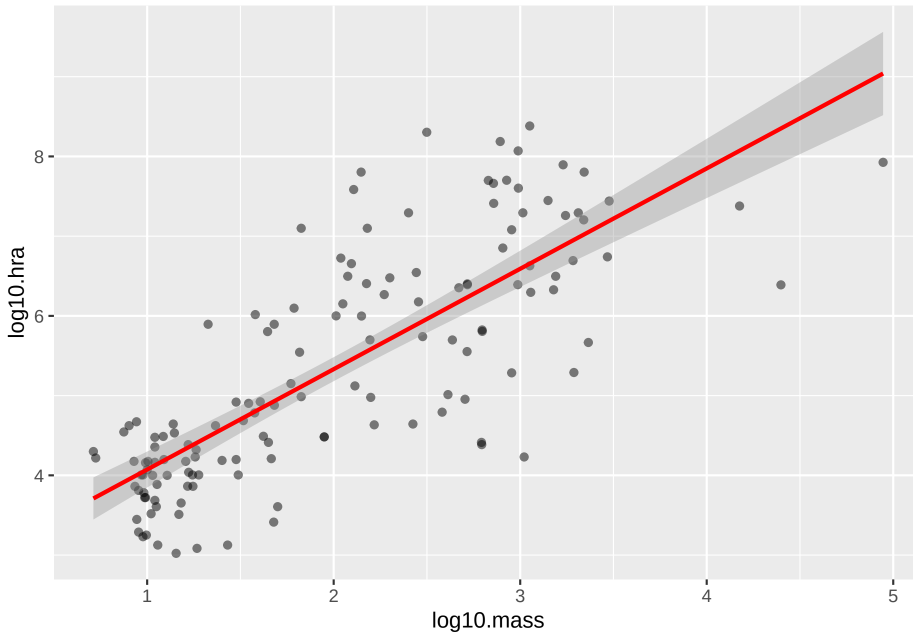
Changing just one line gives us a box plot instead:

```
hra %>%
  dplyr::filter(class == "aves") %>%
  dplyr::mutate(order_fct = as.factor(order)) %>%
  ggplot2::ggplot(mapping = aes(x = order_fct, y = log10.mass, color = order_fct)) +
  ggplot2::geom_boxplot()
```



And if we want to save our chart to a file, that's just one more call as well:

```
hra %>%
  dplyr::filter(class == "aves") %>%
  ggplot2::ggplot(mapping = aes(x = log10.mass, y = log10.hra)) +
  ggplot2::geom_point(alpha = 0.5) +
  ggplot2::geom_smooth(method = lm, color = 'red')
```




```
ggsave("/tmp/birds.png")
```

```
Saving 6.5 x 4.5 in image
```

Appendix F

Glossary

Absolute row number The sequential index of a row in a table, regardless of what sections of the table is being displayed.

Aggregation To combine many values into one, e.g., by summing a set of numbers or concatenating a set of strings.

Alias To have two (or more) references to the same physical data.

Anonymous function A function that has not been assigned a name. Anonymous functions are usually quite short, and are usually defined where they are used, e.g., as callbacks.

Attribute A name-value pair associated with an object, used to store metadata about the object such as an array's dimensions.

Catch (exception) To accept responsibility for handling an error or other unexpected event. R prefers "handling a condition" to "catching an exception".

Condition An error or other unexpected event that disrupts the normal flow of control. See also handle.

Constructor (class) A function that creates an object of a particular class. In the S3 object system, constructors are a convention rather than a requirement.

Copy-on-modify The practice of creating a new copy of aliased data whenever there is an attempt to modify it so that each reference will believe theirs is the only one.

Double square brackets An index enclosed in `[[...]]`, used to return a single value of the underlying type. See also single square brackets.

Eager evaluation Evaluating an expression as soon as it is formed.

Empty vector A vector that contains no elements. Empty vectors have a type such as logical or character, and are *not* the same as null.

Environment A structure that stores a set of variable names and the values they refer to.

Error The most severe type of built-in condition in R.

Evaluating function A function that takes arguments as values. Most functions are evaluating functions.

Evaluation The process of taking a complex expression such as `1+2*3/4` and turning it into a single irreducible value.

Exception An object containing information about an error, or the condition that led to the error. R prefers "handling a condition" to "catching an exception".

Filter To choose a set of records according to the values they contain.

Fully qualified name An unambiguous name of the form package::thing.

Functional programming A style of programming in which functions transform data rather than modifying it. Functional programming relies heavily on higher-order functions.

Generic function A collection of functions with similar purpose, each operating on a different class of data.

Global environment The environment that holds top-level definitions in R, e.g., those written directly in the interpreter.

Group To divide data into subsets according to some criteria while leaving records in a single structure.

Handle (a condition) To accept responsibility for handling an error or other unexpected event. R prefers "handling a condition" to "catching an exception".

Helper (class) In S3, a function that constructs and validates an instance of a class.

Heterogeneous Potentially containing data of different types. Most vectors in R are homogeneous, but lists can be heterogeneous.

Higher-order function A function that takes one or more other functions as parameters. Higher-order functions such as `map` are commonly used in functional programming.

Homogeneous Containing data of only a single type. Most vectors in R are homogeneous.

Hubris Excessive pride or self-confidence. See also unit test (lack of).

ISO3 country code A three-letter code defined by ISO 3166-1 that identifies a specific country, dependent territory, or other geopolitical entity.

Lazy evaluation Delaying evaluation of an expression until the value is actually needed (or at least until after the point where it is first encountered).

List A vector that can contain values of many different types.

List comprehension An expression that generates a new list from an existing one via an implicit loop.

Logical indexing To index a vector or other structure with a vector of Booleans, keeping only the values that correspond to true values.

Message The least severe type of built-in condition in R.

Method An implementation of a generic function that handles objects of a specific class.

NA A special value used to represent data that is Not Available.

Name collision A situation in which the same name has been used in two different packages which are then used together, leading to ambiguity.

Named list FIXME

Negative selection To specify the elements of a vector or other data structure that *aren't* desired by negating their indices.

Null A special value used to represent a missing object. `NULL` is not the same as `NA`, and neither is the same as an empty vector.

Package A collection of code, data, and documentation that can be distributed and re-used.

Pipe operator The `%>%` used to make the output of one function the input of the next.

Prefix operator An operator that comes before the single value it operates on, such as the `-` in `-(a*b)`.

Promise A data structure used to record an unevaluated expression for lazy evaluation.

Pull indexing Vectorized indexing in which the value at location *i* in the index vector specifies which element of the source vector is being pulled into that location in the result vector, i.e., `result[i] = source[index[i]]`. See also push indexing.

Push indexing Vectorized indexing in which the value at location *i* in the index vector specifies an element of the result vector that gets the corresponding element of the source vector, i.e., `result[index[i]] = source[i]`. Push indexing can easily produce gaps and collisions. See also pull indexing.

Quosure A data structure containing an unevaluated expression and its environment.

Quoting function A function that is passed expressions rather than the values of those expressions.

Raise (exception) A way of indicating that something has gone wrong in a program, or that some other unexpected event has occurred. R prefers "signalling a condition" to "raising an exception".

Range expression An expression of the form low:high that is transformed a sequence of consecutive integers.

Reactive programming A style of programming in which actions are triggered by external events.

Reactive variable A variable whose value is automatically updated when some other value or values change.

Recycle To re-use values from a shorter vector in order to generate a sequence of the same length as a longer one.

Regular expression A pattern for matching text. Regular expressions are themselves written as text, which makes them as cryptic as they are powerful.

Relative row number The index of a row in a displayed portion of a table, which may or may not be the same as the absolut row number within the table.

Repository The place where a version control system stores a project's files and the metadata used to record their history.

S3 A framework for object-oriented programming in R.

Scalar A single value of a particular type, such as 1 or “a”. Scalars don’t really exist in R; values that appear to be scalars are actually vectors of unit length.

Select To choose entire columns from a table by name or location.

Setup (testing) Code that is automatically run once before each unit test.

Signal (a condition) A way of indicating that something has gone wrong in a program, or that some other unexpected event has occurred. R prefers “signalling a condition” to “raising an exception”.

Single square brackets An index enclosed in `[...]`, used to select a structure from another structure. See also double square brackets.

Storage allocation Setting aside a block of memory for future use.

Teardown (testing) Code that is automatically run once after each unit test.

Test fixture The data structures, files, or other artefacts on which a unit test operates.

Test runner A software tool that finds and runs unit tests.

Tibble A modern replacement for R’s data frame, which stores tabular data in columns and rows, defined and used in the tidyverse.

Tidyverse A collection of R packages for operating on tabular data in consistent ways.

Unit test A function that tests one aspect or property of a piece of software.

Validator (class) A function that checks the consistency of an S3 object.

Variable arguments In a function, the ability to take any number of arguments. R uses `...` to capture the “extra” arguments.

Vector A sequence of values, usually of homogeneous type. Vectors are *the* fundamental data structure in R; scalars are actually vectors of unit length.

Vectorize To write code so that operations are performed on entire vectors, rather than element-by-element within loops.

Warning A built-in condition in R of middling severity.

Widget An interactive control element in an user interface.

Appendix G

Key Points

G.1 Simple Beginnings

- Use `print(expression)` to print the value of a single expression.
- Variable names may include letters, digits, `.`, and `_`, but `.` should be avoided, as it sometimes has special meaning.
- R's atomic data types include logical, integer, double (also called numeric), and character.
- R stores collections in homogeneous vectors of atomic types, or in heterogeneous lists.
- 'Scalars' in R are actually vectors of length 1.
- Vectors and lists are created using the function `c(...)`.
- Vector indices from 1 to length(vector) select single elements.
- Negative indices to vectors deselect elements from the result.
- The index 0 on its own selects no elements, creating a vector or list of length 0.
- The expression `low:high` creates the vector of integers from `low` to `high` inclusive.
- Subscripting a vector with a vector of numbers selects the elements at those locations (possibly with repeats).
- Subscripting a vector with a vector of logicals selects elements where the indexing vector is `TRUE`.
- Values from short vectors (such as 'scalars') are repeated to match the lengths of longer vectors.
- The special value `NA` represents missing values, and (almost all) operations involving `NA` produce `NA`.
- The special values `NULL` represents a nonexistent vector, which is not the same as a vector of length 0.
- A list is a heterogeneous vector capable of storing values of any type (including other lists).

- Indexing with `[` returns a structure of the same type as the structure being indexed (e.g., returns a list when applied to a list).
- Indexing with `[[` strips away one level of structure (i.e., returns the indicated element without any wrapping).
- Use `list('name' = value, ...)` to name the elements of a list.
- Use either `L['name']` or `L$name` to access elements by name.
- Use back-quotes around the name with `$` notation if the name is not a legal R variable name.
- Use `matrix(values, nrow = N)` to create a matrix with `N` rows containing the given values.
- Use `m[i, j]` to get the value at the i'th row and j'th column of a matrix.
- Use `m[i,]` to get a vector containing the values in the i'th row of a matrix.
- Use `m[,j]` to get a vector containing the values in the j'th column of a matrix.
- Use `for (loop_variable in collection){ ...body... }` to create a loop.
- Use `if (expression) { ...body... } else if (expression) { ...body... } else { ...body... }` to create conditionals.
- Expression conditions must have length 1; use `any(...)` and `all(...)` to collapse logical vectors to single values.
- Use `function(...arguments...) { ...body... }` to create a function.
- Use variable <- function(…arguments…) { …body… }‘ to create a function and give it a name.
- The body of a function can be a single expression or a block in curly braces.
- The last expression evaluated in a function is returned as its result.
- Use `return(expression)` to return a result early from a function.

G.2 The Tidyverse

- `install.packages('name')` installs packages.
- `library(name)` (without quoting the name) loads a package.
- `library(tidyverse)` loads the entire collection of tidyverse libraries at once.
- `read_csv(filename)` reads CSV files that use the string ‘NA’ to represent missing values.
- `read_csv` infers each column's data types based on the first thousand values it reads.
- A tibble is the tidyverse's version of a data frame, which represents tabular data.
- `head(tibble)` and `tail(tibble)` inspect the first and last few rows of a tibble.
- `summary(tibble)` displays a summary of a tibble's structure and values.
- `tibble$column` selects a column from a tibble, returning a vector as a result.

- `tibble['column']` selects a column from a tibble, returning a tibble as a result.
- `tibble[,c]` selects column `c` from a tibble, returning a tibble as a result.
- `tibble[r,]` selects row `r` from a tibble, returning a tibble as a result.
- Use ranges and logical vectors as indices to select multiple rows/columns or specific rows/columns from a tibble.
- `tibble[[c]]` selects column `c` from a tibble, returning a vector as a result.
- `min(...)`, `mean(...)`, `max(...)`, and `std(...)` calculates the minimum, mean, maximum, and standard deviation of data.
- These aggregate functions include `NA`s in their calculations, and so will produce `NA` if the input data contains any.
- Use `func(data, na.rm = TRUE)` to remove `NA`s from data before calculations are done (but make sure this is statistically justified).
- `filter(tibble, condition)` selects rows from a tibble that pass a logical test on their values.
- `arrange(tibble, column)` or `arrange(desc(column))` arrange rows according to values in a column (the latter in descending order).
- `select(tibble, column, column, ...)` selects columns from a tibble.
- `select(tibble, -column)` selects *out* a column from a tibble.
- `mutate(tibble, name = expression, name = expression, ...)` adds new columns to a tibble using values from existing columns.
- `group_by(tibble, column, column, ...)` groups rows that have the same values in the specified columns.
- `summarize(tibble, name = expression, name = expression)` aggregates tibble values (by groups if the rows have been grouped).
- `tibble %>% function(arguments)` performs the same operation as `function(tibble, arguments)`.
- Use `%>%` to create pipelines in which the left side of each `%>%` becomes the first argument of the next stage.

G.3 Creating Packages

- Develop data-cleaning scripts one step at a time, checking intermediate results carefully.
- Use `read_csv` to read CSV-formatted tabular data into a tibble.
- Use the `skip` and `na` parameters of `read_csv` to skip rows and interpret certain values as `NA`.
- Use `str_replace` to replace portions of strings that match patterns with new strings.
- Use `is.numeric` to test if a value is a number and `as.numeric` to convert it to a number.
- Use `map` to apply a function to every element of a vector in turn.
- Use `map_dfc` and `map_dfr` to map functions across the columns and rows of a tibble.
- Pre-allocate storage in a list for each result from a loop and fill it in rather

than repeatedly extending the list.
- An R package can contain code, data, and documentation.
- R code is distributed as compiled bytecode in packages, not as source.
- R packages are almost always distributed through CRAN, the Comprehensive R Archive Network.
- Most of a project's metadata goes in a file called `DESCRIPTION`.
- Metadata related to imports and exports goes in a file called `NAMESPACE`.
- Add patterns to a file called `.Rbuildignore` to ignore files or directories when building a project.
- All source code for a package must go in the `R` sub-directory.
- `library` calls in a package's source code will *not* be executed as the package is loaded after distribution.
- Data can be included in a package by putting it in the `data` sub-directory.
- Data must be in `.rda` format in order to be loaded as part of a package.
- Data in other formats can be put in the `inst/extdata` directory, and will be installed when the package is installed.
- Add comments starting with `#'` to an R file to document functions.
- Use roxygen2 to extract these comments to create manual pages in the `man` directory.
- Use `@export` directives in roxygen2 comment blocks to make functions visible outside a package.
- Add required libraries to the `Imports` section of the `DESCRIPTION` file to indicate that your package depends on them.
- Use `package::function` to access externally-defined functions inside a package.
- Alternatively, add `@import` directives to roxygen2 comment blocks to make external functions available inside the package.
- Import `.data` from `rlang` and use `.data$column` to refer to columns instead of using bare column names.
- Create a file called R/package.R and document `NULL` to document the package as a whole.
- Create a file called R/dataset.R and document the string 'dataset' to document a dataset.

G.4 Non-Standard Evaluation

- R uses lazy evaluation: expressions are evaluated when their values are needed, not before.
- Use `expr` to create an expression without evaluating it.
- Use `eval` to evaluate an expression in the context of some data.
- Use `enquo` to create a quosure containing an unevaluated expression and its environment.
- Use `quo_get_expr` to get the expression out of a quosure.
- Use `!!` to splice the expression in a quosure into a function call.

G.5 Intellectual Debt

- Don't use `setwd`.
- The formula operator `~` delays evaluation of its operand or operands.
- `~` was created to allow users to pass formulas into functions, but is used more generally to delay evaluation.
- Some tidyverse functions define `.` to be the whole data, `.x` and `.y` to be the first and second arguments, and `..N` to be the N'th argument.
- These convenience parameters are primarily used when the data being passed to a pipelined function needs to go somewhere other than in the first parameter's slot.
- 'Copy-on-modify' means that data is aliased until something attempts to modify it, at which point it duplicated, so that data always appears to be unchanged.

G.6 Testing and Error Handling

- Operations signal conditions in R when errors occur.
- The three built-in levels of conditions are messages, warnings, and errors.
- Programs can signal these themselves using the functions `message`, `warning`, and `stop`.
- Operations can be placed in a call to the function `try` to suppress errors, but this is a bad idea.
- Operations can be placed in a call to the function `tryCatch` to handle errors.
- Use testthat to write unit tests for R.
- Put unit tests for an R package in the `tests/testthat` directory.
- Put tests in files called test_group.R and call them test_something.
- Use `test_dir` to run tests from a particular that match a pattern.
- Write tests for data transformation steps as well as library functions.

G.7 Advanced Topics

- The `reticulate` library allows R programs to access data in Python programs and vice versa.
- Use `py.whatever` to access a top-level Python variable from R.
- Use `r.whatever` to access a top-level R definition from Python.
- R is always indexed from 1 (even in Python) and Python is always indexed from 0 (even in R).
- Numbers in R are floating point by default, so use a trailing 'L' to force a value to be an integer.
- A Python script run from an R session believes it is the main script, i.e., `__name__` is `'__main__'` inside the Python script.
- S3 is the most commonly used object-oriented programming system in R.

- Every object can store metadata about itself in attributes, which are set and queried with `attr`.
- The `dim` attribute stores the dimensions of a matrix (which is physically stored as a vector).
- The `class` attribute of an object defines its class or classes (it may have several character entries).
- When `F(X, ...)` is called, and `X` has class `C`, R looks for a function called `F.C` (the `.` is just a naming convention).
- If an object has multiple classes in its `class` attribute, R looks for a corresponding method for each in turn.
- Every user defined class `C` should have functions `new_C` (to create it), `validate_C` (to validate its integrity), and `C` (to create and validate).
- Use the `DBI` package to work with relational databases.
- Use `DBI::dbConnect(...)` with database-specific parameters to connect to a specific database.
- Use `dbGetQuery(connection, "query")` to send an SQL query string to a database and get a data frame of results.
- Parameterize queries using `:name` as a placeholder in the query and `params = list(name = value)` as a third parameter to `dbGetQuery` to specify actual values.
- Use `dbFetch` in a `while` loop to page results.
- Use `dbWriteTable` to write an entire data frame to a table, and `dbExecute` to execute a single insertion statement.
- Dates… why did it have to be dates?

Bibliography

Wickham, H. (2019). *Advanced R*. Chapman and Hall/CRC, 2nd edition. A deep dive into the details of R.

Wilkinson, L. (2005). *The Grammar of Graphics*. Springer. Presents a systematic foundation for constructing informational graphics.