The Tidynomicon

Greg Wilson

2019-05-01

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?

See Wilson (2018) for a description of the lesson design process.

Andrzej completed a Master's in library science five years ago and has worked since then for a small consulting company. He learned Python by doing data science courses online, but has no formal training in programming. He just joined team that primarily uses R Markdown; these lessons will show him how to translate his understanding of Python to R.

Padma has been building performance dashboards for a logistics company using Django and D3. The company has just hired some data scientists who use R, and who would like to rebuild some of those dashboards in Shiny. Padma isn't a statistician, but she's comfortable doing linear regression and basic time series analysis on web traffic, and would like to learn enough about R to tidy up the analysts' code and get it into production.

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

Chapter 2

Values and Vectors

2.1 Questions

- How do I print things?
- What are R's basic data types?
- How do I find out what type something is?
- How do I name variables in R?
- How do I create and index lists in R?
- How do ranges in R differ from ranges in Python?
- What special values does R use to represent things that aren't there?

2.2 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.

We will begin our tour of R by looking at what kinds of data we can toy with. To do that, we need to get set up:

1. Create an account on rstudio.cloud, then create a new project and start typing.
2. Alternatively:
    1. Install R. 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.
    2. Install RStudio.
    3. In the RStudio console, run `install.packages("tidyverse")` to install the tidyverse libraries. We will install others as we go along, but we're going to need this soon.

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`.

2.3 How do I say hello?

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

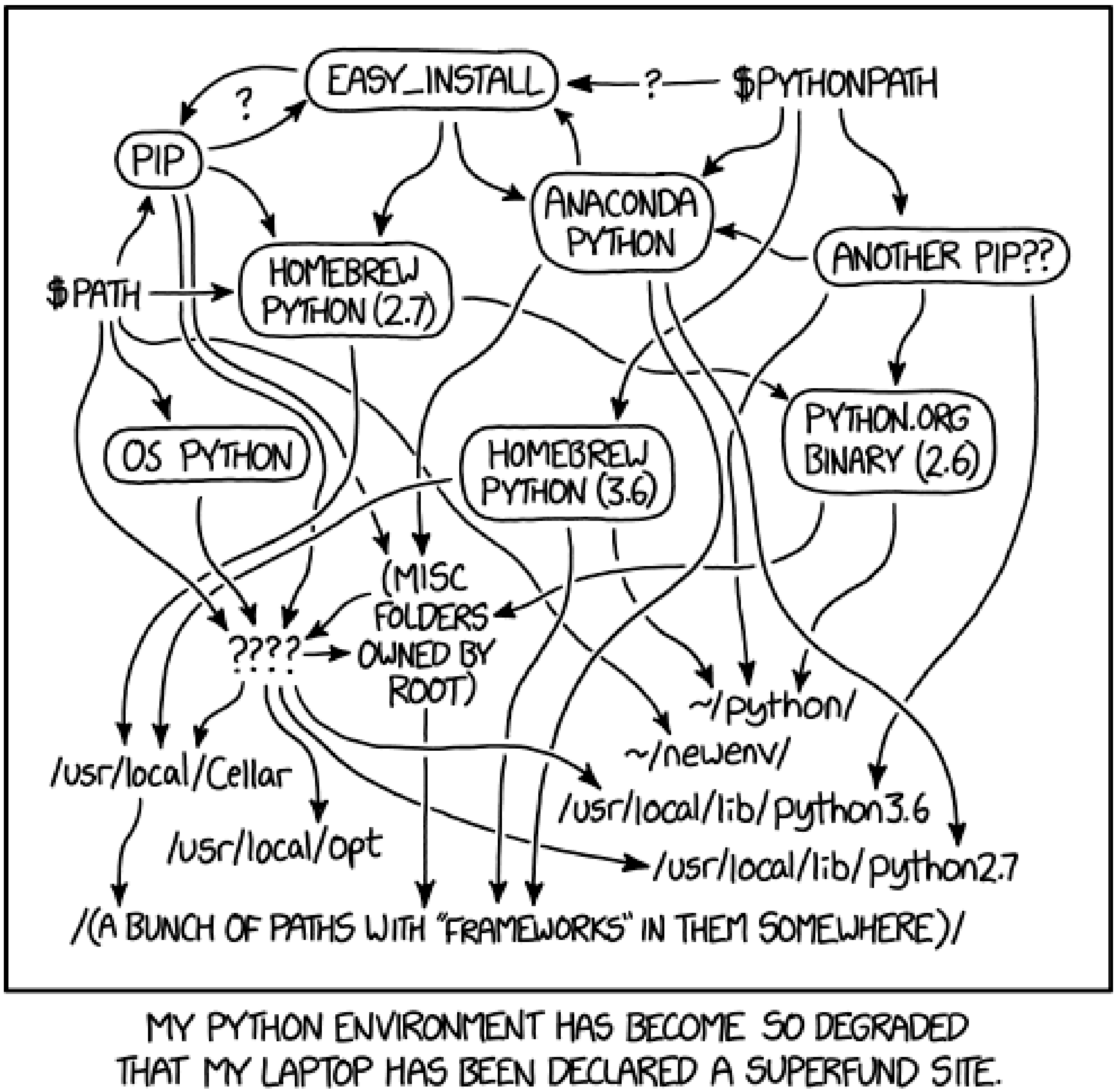


Figure 2.1: XKCD on Python Environments (from https://xkcd.com/1987/)

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

```
Hello, world!
```

We can run the equivalent R in the RStudio Console:

```
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."
```

That the mysterious `[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).

2.4 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 doing in that function's name? Is there an object called `as` with a method called `integer`? The answer is "no": `.` is just another character in R. Like the underscore `_` it is used to make names more readable, but it has no special meaning.

2.5 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 dark 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)
```

Assignment is done using a left-pointing arrow `<-`. (Other forms with their own symbols also exist, but we will not discuss them until Chapter 4.) Like Python, R does not display a value after an assignment statement.

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

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

```
4
```

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

```
TypeError: object of type 'int' has no len()

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 its 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 much 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. Mike Hoye discovered the truth.)

2.6 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]
```

```
IndexError: list index out of range

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:

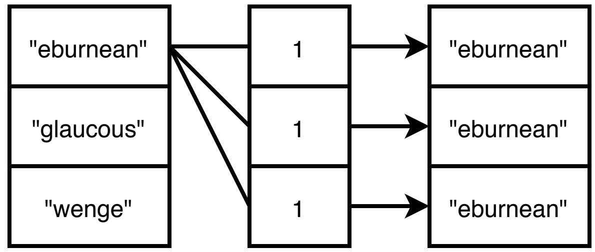


Figure 2.2: Pull Indexing

```
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?

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

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

That didn't work because R interprets the subscript `[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:

We can also select out several elements:

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

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

What we *cannot* do is 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—it appears that `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.

2.7 How do I create new vectors from old?

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

```
doubled = 2 * original
```

R provides the same 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 is straightforward if one of the vectors is a scalar—it is just re-used as many times as necessary—but shouldn't be done if the vectors don't line up nicely:

```
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 (unsurprisingly) 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`:

```
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"
```

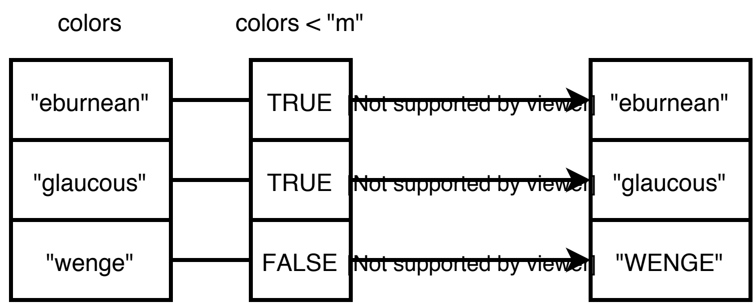


Figure 2.3: Vector Conditionals

2.8 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:

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

```
logical(0)
```

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

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

```
[1] TRUE
```

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

```r
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`:

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

```
[1] FALSE
```

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

```
[1] TRUE
```

2.9 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.

Chapter 3

Indexing

3.1 Questions

- How can I store values of different types in a single data structure?
- How can I index things?
- How can I access values by name in a data structure?
- How can I create a matrix?

3.2 Learning Objectives

- 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.

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.

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

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.

3.4 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"
```

Good: they are vectors. What do we get if we use single square brackets `[`?

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

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

Sure enough, the value itself is a list:

```
thing[1]
```

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

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"
```

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

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

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

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

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

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

> **Flattening**
>
> 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")`.
>
> **Recursive Indexing**
>
> 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.

3.5 How can I access elements by name?

R allows us to name the elements in vectors: 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:

```r
values <- c("m", "f", "u", "f", "f", "m", "m")
lookup <- c(m = "Male", f = "Female", u = "Unstated")
lookup[values]
```

```
         m          f          u          f          f          m 
    "Male"   "Female" "Unstated"   "Female"   "Female"     "Male" 
         m 
    "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"]`:

```r
lookup_list <- list(m = "Male", f = "Female", u = "Unstated")
lookup_list$m
```

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

We will explore this in more detail when we look at the tidyverse in Chapter 5, 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 as an accessor:

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

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

Wherever possible, it's better to choose names that don't require back-quoting, such as `first_field`.

3.6 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. What may surprise you is the order in which the values generated by the expression `1:9` are laid out:

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

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

Under the hood, 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 (because 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
```

3.7 Key Points

- 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.

Chapter 4

Control Flow

4.1 Questions

- How do I write conditionals and loops in R?
- What should I use instead of loops in R?
- How do ranges in R differ from ranges in Python?
- How do I create functions in R?

4.2 Learning Objectives

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

Chapter 2 said that modern R strongly encourages people to write vectorized code. There are times, though, when we need to write loops and conditionals, and we should *always* break our code up into single-purpose functions.

4.3 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. Here, for example, is a snippet of Python that displays the signs of a list of numbers:

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

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

Its direct translation into R is:

```r
values <- c(-1, 0, 1)
for (v in values) {
  if (v < 0) {
    sign <- -1
  }
  else if (v == 0) {
    sign <- 0
  }
  else {
    sign <- 1
  }
  print(paste("The sign of", v, "is", sign))
}
```

```
[1] "The sign of -1 is -1"
[1] "The sign of 0 is 0"
[1] "The sign of 1 is 1"
```

```r
print(paste("final value of v is", v))
```

```
[1] "final value of v is 1"
```

There are a few things to note here:

1. This is *not* how we should write R: everything in this snippet can and should be vectorized.
2. The parentheses in the loop header are required: we cannot simply write `for v in values`.
3. 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.
4. The loop variable `v` persists after the loop is over.
5. `paste` converts its arguments to strings and concatenates them, placing a single space between each unless instructed to do otherwise. `print` then prints the resulting (single) string. The function `cat` (short for "concatenate") can also be used for text output.
6. By calling our temporary variable `sign` we have accidentally overwritten the rather useful built-in R function with that name. Name collisions of this sort are as easy in R as they are in Python.

4.4 How can I express a range of values in R?

By default, R's `for` loop gives us the values in a vector, just as Python's does. If we want to loop over the indices instead, we can use the function `seq_along`:

```r
colors = c("eburnean", "glaucous", "wenge")
for (i in seq_along(colors)) {
  print(paste("The length of color", i, "is", length(colors[i])))
}
```

```
[1] "The length of color 1 is 1"
[1] "The length of color 2 is 1"
[1] "The length of color 3 is 1"
```

This makes no sense at all 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 function `stringr::str_length`:

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

```
[1] "The length of color 1 is 8"
[1] "The length of color 2 is 8"
[1] "The length of color 3 is 5"
```

`seq_along` returns a vector containing a sequence of integers:

```
seq_along(colors)
```

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

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

```
5:10
```

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

Their most common use is as indices to vectors:

```
colors <- c("eburnean", "glaucous", "squamous", "wenge")
colors[1:3]
```

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

We can similarly subtract a range of colors by index:

```
colors[-1:-3]
```

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

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

```
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 as proof some supportive anecdote such as, "Most people do not interpret the expression 'from one to five' to mean 'one, two, three, or four'."

> **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")`.

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

We cannot use vectors 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.

```r
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.

4.6 How do I create and call functions?

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

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

```
[1] 6
```

We define a new function using the `function` keyword. This creates the function, but does not name it; to accomplish that, we must assign the newly-created function to a variable:

```r
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.

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

```
NULL
```

```r
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:

```r
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 8.

4.7 How can I write a function that takes a varying number of 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 in this example we as passing three values, 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, ...) {
  title <- paste("==", title, "==\n")
  items <- paste(..., sep = "\n")
  cat(title)
  cat(items)
  cat("\n")
}
print_with_title("to-do", "Monday", "Tuesday", "Wednesday")
```

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

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

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

```
[1] 16
```

4.8 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(paste(first, "+", second, "+", third))
```

```
}

example("with just first")
```

```
[1] "with just first + second + third"
```

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

```
[1] "with first and second by position + positional + third"
```

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

```
[1] "with first and third by name + second + by name"
```

One caution: when you use a name in a function call, R ignores non-function objects when figuring out what function to call. For example, the call `orange()` in the code below produces 110 because `purple(purple)` is interpreted as "pass the value of the local variable `purple` into the globally-defined function `purple`":

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

```
[1] 110
```

4.9 How can I hide the value that R returns?

If the value returned by a function isn't assigned to something, R will print it out. This isn't always what we want, particularly in library functions, so we can use the function `invisible` to mark a value so that it won't be printed by default (but can still be assigned). This allows us to convert this:

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

```
[1] 20
```

to this:

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

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

4.10 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:

```
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—especially not after what happened to poor Professor Peaslee.

4.11 Key Points

- 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 5

The Tidyverse

5.1 Questions

- How do I install packages in R?
- How do I load packages in R?
- How do a read a CSV file in R?
- How does R store tabular data?
- How does R decide what data types to use for columns in CSV data?
- How can I inspect tabular data that I have loaded or created?
- How can I select sections of tabular data?
- How can I extract vectors from tables?
- How can I calculate basic statistics on tabular data?
- How does R treat missing data when calculating aggregate statistics?
- How can I control how R treats missing data when calculating aggregate statistics?
- What tools does the tidyverse provide for selecting, rearranging, changing, and summarizing tabular data?
- How should I combine tidyverse operations?

5.2 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.
- Name, describe, and use a tidyverse function for choosing rows by value from tabular data.
- Name, describe, and use a tidyverse function for reordering rows of tabular data.
- Name, describe, and use a tidyverse function for selecting columns of tabular data.
- Name, describe, and use a tidyverse function for calculating new columns from existing ones.
- Name, describe, and use a tidyverse function for grouping rows of tabular data.
- Name, describe, and use a tidyverse function for aggregating grouped or ungrouped rows of tabular data.

There is no point in becoming fluent in Enochian if you do not then summon 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.

5.3 How do I read data?

We begin by looking at the file `tidy/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
data = pd.read_csv('tidy/infant_hiv.csv')
print(data)
```

```
     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
9        AGO  2009       NaN   NaN   NaN
10       AGO  2010      0.03  0.04  0.02
11       AGO  2011      0.05  0.07  0.04
12       AGO  2012      0.06  0.08  0.05
13       AGO  2013      0.15  0.20  0.12
```

```
14        AGO  2014      0.10  0.14  0.08
15        AGO  2015      0.06  0.08  0.05
16        AGO  2016      0.01  0.02  0.01
17        AGO  2017      0.01  0.02  0.01
18        AIA  2009       NaN   NaN   NaN
...
```

The equivalent in R is to load the tidyverse collection of libraries 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 this, which we only need to do once per machine:

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

We then load the library once per program:

```
library(tidyverse)
```

```
  Attaching packages                       tidyverse 1.2.1
 ggplot2 3.1.0      purrr   0.3.0
 tibble  2.0.1      dplyr   0.7.8
 tidyr   0.8.2      stringr 1.4.0
 readr   1.1.1      forcats 0.3.0
  Conflicts                         tidyverse_conflicts()
 dplyr::filter() masks stats::filter()
 dplyr::lag()    masks stats::lag()
Warning messages:
1: package ‘tibble’ was built under R version 3.5.2
2: package ‘purrr’ was built under R version 3.5.2
3: package ‘stringr’ was built under R version 3.5.2
```

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

Asking for the tidyverse gives us eight libraries (or packages). One of those, dplyr, defines two functions that mask standard functions in R with the same names. This is deliberate, and if we need the originals, we can get them with their fully-qualified names `stats::filter` and `stats::lag`. (Note that R uses `::` to get functions out of packages rather than Python's `.`.)

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

```
data <- read_csv('tidy/infant_hiv.csv')
```

```
Parsed with column specification:
cols(
  country = col_character(),
  year = col_integer(),
  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.

```
data
```

```
# A tibble: 1,728 x 5
   country  year estimate    hi    lo
   <chr>   <int>    <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.

5.4 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(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
```

```
print(data.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 (not methods):

```
head(data)
```

```
# A tibble: 6 x 5
  country  year estimate    hi    lo
  <chr>   <int>    <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(data)
```

```
# A tibble: 6 x 5
  country  year estimate    hi    lo
  <chr>   <int>    <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(data)
```

```
# A tibble: 6 x 5
  country  year estimate    hi    lo
  <chr>   <int>    <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. (Notice also that R starts numbering from 1.) What about overall information?

```
print(data.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(data)
```

```
   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.

5.5 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. Columns in R tibbles are, not coincidentally, the same.

```
print(data['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 `data.estimate`, i.e., with an attribute name rather than a string subscript. The same tricks work in R:

```
data['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 `data$estimate` provides all the data:

```r
data$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:

```python
print(data.estimate[11])
```

```
0.05
```

```r
data$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.

```python
print(len(data.estimate[11]))
```

```
TypeError: object of type 'numpy.float64' has no len()

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

```r
length(data$estimate[12])
```

```
[1] 1
```

And yes, ranges work:

```python
print(data.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
```

```
data$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 neither library prevents us from selecting a range of data that spans logical groups such as 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(data.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(data.iloc[:, 0][0])
```

```
AFG
```

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

```
data[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:

```
data[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:

```
data[[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:

```
data[[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"):

```
data[[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.

5.6 How do I calculate basic statistics?

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

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

```
1728
```

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

```
0.3870192307692308
```

This translates almost directly to R:

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

```
[1] 1728
```

```
mean(estimates)
```

```
[1] NA
```

The void is always there, waiting for us… Let's fix this in R first:

```
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:

```
print(estimates.min())
```

```
0.0
```

```
print(estimates.max())
```

```
0.95
```

```
print(estimates.std())
```

```
0.3034511074214113
```

Here are the equivalent computations in R:

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

```
[1] 0
```

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

```
[1] 0.95
```

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

```
[1] 0.3034511
```

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((data.hi.isnull() != data.lo.isnull()).any())
```

```
False
```

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

```
[1] FALSE
```

5.7 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:

```
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`:

```
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:

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

```
[1] 52
```

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

```
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.

5.8 How do I write tidy code?

The five 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
- `summarize`: combine many values to create a single new value

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

```
filter(data, lo > 0.5)
```

```
# A tibble: 183 x 5
   country  year estimate    hi    lo
   <chr>   <int>    <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 returns the entire table because the string `"lo"` is greater than the number 0.5 everywhere.

But wait: how is it that `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 of arguments. 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. This is much tidier than `filter(data, data$lo > 0.5)` or `filter(data, "lo > 0.5")`, and is *not* some kind of eldritch wizardry. Many languages rely on lazy evaluation, and when used circumspectly, it allows us to produce code that is easier to read.

But we can do even better by using the pipe operator `%>%`, which is about to become your new best friend:

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

```
# A tibble: 183 x 5
```

```
   country  year estimate    hi    lo
   <chr>   <int>    <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(data, (estimate != 0.95) & (lo > 0.5) & (hi <= (lo + 0.1)))
```

```
# A tibble: 1 x 5
  country  year estimate    hi    lo
  <chr>   <int>    <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 tidyverse equivalent is:

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

```
# A tibble: 1 x 5
  country  year estimate    hi    lo
  <chr>   <int>    <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%:

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

```
# A tibble: 55 x 5
   country  year estimate    hi    lo
   <chr>   <int>    <dbl> <dbl> <dbl>
 1 ARG      2016     0.67  0.77  0.61
 2 ARG      2017     0.66  0.77  0.6
 3 CHL      2011     0.64  0.72  0.56
 4 CHL      2013     0.67  0.77  0.59
 5 CHL      2014     0.77  0.87  0.67
 6 CHL      2015     0.92  0.95  0.79
 7 CHL      2016     0.7   0.79  0.62
 8 CHL      2017     0.85  0.95  0.76
 9 CPV      2014     0.94  0.95  0.76
10 CPV      2016     0.94  0.95  0.76
# ... with 45 more rows
```

And then order by `lo` in descending order, breaking the line the way the tidyverse style guide recommends:

```
data %>%
  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>   <int>    <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:

```
data %>%
  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
   <int> <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:

```
data %>%
  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
   <int> <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. Since we want to keep `hi` and `lo`, we decide to use `mutate`:

```
data %>%
  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
   <int> <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()`:

```
data %>%
  filter(estimate != 0.95) %>%
  filter(lo > 0.5) %>%
  filter(hi <= (lo + 0.2)) %>%
  mutate(difference = hi - lo) %>%
```

```
  group_by(year) %>%
  summarize(n(), mean(year))
```

```
# A tibble: 9 x 3
   year `n()` `mean(year)`
  <int> <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
```

Let's do that again with more meaningful names for the final table's columns:

```
data %>%
  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
  <int> <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
```

(We could also add a call to `rename`, but for small tables like this, setting column names on the fly is perfectly comprehensible.)

Now, how might we do this with Pandas? On 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('tidy/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'}})
```

```
/Users/gvwilson/anaconda3/lib/python3.6/site-packages/pandas/core/groupby/groupby.py:4658: FutureWarning
  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.

5.9 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 6

Cleaning Up Data

6.1 Questions

- How do I read tabular data into a program?
- How do I control the way missing values are handled while I'm reading data?
- What functions should I use to tidy up messy data?
- How can I combine partial tables into a single large table?

6.2 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.

Here is a sample of data from `raw/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%,37%,31%,61%,68%
AGO,Angola,-,-,-,3%,4%,2%,5%,7%,4%,6%,8%,5%,15%,20%,12%,10%,14%,8%,6%,8%,5%,1%,2%,1%,1%,2%,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%,52%,39%,43%,51%
ZWE,Zimbabwe,-,-,-,12%,15%,10%,23%,28%,20%,38%,47%,33%,57%,70%,49%,54%,67%,47%,59%,73%,51%,71%,88%,62%,
,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
,,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%,34%,24%,26%,31%,22
,Eastern and Southern Africa,23%,29%,20%,44%,57%,37%,48%,62%,40%,54%,69%,46%,51%,65%,43%,62%,80%,53%,62
,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%,41%,50%,66%,41%
,Global,17%,23%,13%,33%,45%,27%,36%,49%,29%,41%,55%,34%,40%,53%,32%,48%,64%,39%,49%,64%,40%,44%,59%,36%
```

```
,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
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 `tidy/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
```

To bring this data to a state of grace will take some trial and effort, which we shall do in stages.

6.3 How do I inspect the raw data?

We will begin by reading the data into a tibble:

```r
raw <- read_csv("raw/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()
)

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

```r
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 <chr>
```

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("raw/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()
)

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

```
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 <chr>
```

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):

```
raw <- read_csv("raw/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()
)

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 <chr>
```

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).

```r
raw <- read_csv("raw/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()
)
```

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

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

```
Error in filter_impl(.data, quo): Evaluation error: 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("raw/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()
)
```

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

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

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

```
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 virological
[22] "Note: Data are not available if country did not submit data to Global AIDS Monitoring or if estima
[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/sites/def
[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 manually count the number of rows we are interested in (or rather, open the file with an editor or spreadsheet program, scroll down, and check the line number), and then 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.

```r
num_rows <- 192
raw <- read_csv("raw/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()
)
```

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

```r
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 a faint whimpering sound. We promise we will revisit the problem of slicing data without counting rows manually so as to reduce the frequency with which that sound is heard.

And notice also that we are slicing, *then* extracting the column containing the countries. We did, in a temporary version of this script, peel off the countries, slice those, and then wonder why our main data 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("raw/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()
)

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
```

Oh dear. It appears that some function `str_replace` is calling is expecting an atomic vector, not 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)
```

```
Warning: `is.tibble()` is deprecated, use `is_tibble()`.
This warning is displayed once per session.
```

```
[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("raw/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()
)
```

```
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("raw/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()
```

```
)
```

```
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("raw/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()
)
```

```
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:

```
raw <- read_csv("raw/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()
)

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")
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("raw/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()
)

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] ""   ""   ""   ""
```

Empty strings. Why'd it have to be empty strings? More importantly, where are they coming from? Let's backtrack by displaying the `hi` column of each of our intermediate variables…

…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, in which we check *both* the head and the tail:

```
raw <- read_csv("raw/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()
)
```

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

```
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)
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>
```

```
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.

6.4 How do I tidy the data?

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 all in his trade, did most of the actual work but was given only crumbs of credit).

Our starting point is this:

1. Each row in `percents` corresponds positionally to an ISO3 code in `countries`.
2. Each group of three consecutive columns in `percents` has the estimate, high, and low values for a single year.
3. The years themselves are not stored in `percents`, but we know from inspection that they start at 2009 and run without interruption to 2017.

Our strategy is to make a list of temporary tables:

1. Take three columns at a time from `percents` to create a temporary tibble.
2. Join `countries` to it.
3. Create a column holding the year in each row and join that as well.

and then join those temporary tables row-wise to create our final tidy table. (We might, through clever use of `scatter` and `gather`, be able to do this without a loop, but at this point on our journey, a loop is probably simpler.) Here is the addition to our script:

```
first_year <- 2009
last_year <- 2017
num_years <- (last_year - first_year) + 1
chunks <- vector("list", num_years)
for (year in 1:num_years) {
  end <- year + 2
  temp <- select(percents, year:end)
  names(temp) <- c("estimate", "hi", "lo")
  temp$country <- countries
  temp$year <- rep((first_year + year) - 1, num_rows)
  temp <- select(temp, country, year, everything())
  chunks[[year]] <- temp
}
chunks
```

```
[[1]]
# A tibble: 192 x 5
   country  year estimate    hi    lo
   <chr>   <dbl>    <dbl> <dbl> <dbl>
 1 AFG      2009       NA    NA    NA
 2 ALB      2009       NA    NA    NA
 3 DZA      2009       NA    NA    NA
 4 AGO      2009       NA    NA    NA
 5 AIA      2009       NA    NA    NA
 6 ATG      2009       NA    NA    NA
 7 ARG      2009       NA    NA    NA
 8 ARM      2009       NA    NA    NA
 9 AUS      2009       NA    NA    NA
10 AUT      2009       NA    NA    NA
# ... with 182 more rows

[[2]]
# A tibble: 192 x 5
   country  year estimate    hi    lo
   <chr>   <dbl>    <dbl> <dbl> <dbl>
...
```

We start by giving names to our years; if or when we decide to use this script for other data files, we should extract the years from the data itself. We then use `vector` to create the storage we are going to need to hold our temporary tables. We could grow the list one item at a time, but allocating storage in advance is more efficient and serves as a check on our logic: if our loop doesn't run for the right number of iterations, we will either overflow our list or have empty entries, either of which should draw our attention.

Within the loop we figure out the bounds on the next three-column stripe, select that, and then give those three columns meaningful names. This ensures that when we join all the sub-tables together, the columns of the result will also be sensibly named. Attaching the ISO3 country codes is as easy as assigning to `temp$country`, and replicating the year for each row is easily done using the `rep` function. We then reorder the columns to put country and year first (the call to `everything` inside `select` selects all columns that aren't specifically selected), and then we assign the temporary table to the appropriate slot in `chunks` using `[[..]]`.

As its name suggests, `bind_rows` takes a list of tables and concatenates their rows in order. Since we have taken care to give all of those tables the same column names, no subsequent renaming is necessary. We do, however, use `arrange` to order entries by country and year.

Now comes the payoff for all that hard work:

```
tidy <- bind_rows(chunks)
tidy <- arrange(tidy, country, year)
tidy
```

```
# A tibble: 1,728 x 5
   country  year estimate    hi    lo
   <chr>   <dbl>    <dbl> <dbl> <dbl>
 1 ""       2009       NA    NA    NA
 2 ""       2010       NA    NA    NA
 3 ""       2011       NA    NA    NA
 4 ""       2012       NA    NA    NA
 5 ""       2013       NA    NA    NA
 6 ""       2014       NA    NA    NA
 7 ""       2015       NA    NA    NA
 8 ""       2016       NA    NA    NA
```

```
 9 ""       2017       NA     NA     NA
10 AFG      2009       NA     NA     NA
# ... with 1,718 more rows
```

What fresh hell is this? Why do some rows have empty strings where country codes should be and NAs for the three percentages? Is our indexing off? Have we somehow created one extra row for each year with nonsense values?

No. It is not our tools that have failed us, or our reason, but our data. ("These parts are not fresh, Igor—I must have *fresh* parts to work with!") Let us do this:

```
raw <- read_csv("raw/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()
)
```

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

```
missing <- raw %>%
  filter(is.na(Countries) | (Countries == "") | is.na(ISO3) | (ISO3 == "")) %>%
  select(Countries, ISO3)
missing
```

```
# A tibble: 21 x 2
   Countries                       ISO3
   <chr>                           <chr>
 1 Kosovo                          ""
 2 ""                              ""
 3 ""                              ""
 4 ""                              ""
 5 Eastern and Southern Africa     ""
 6 Eastern Europe and Central Asia ""
 7 Latin America and the Caribbean ""
 8 Middle East and North Africa    ""
 9 North America                   ""
10 South Asia                      ""
# ... with 11 more rows
```

The lack of ISO3 country code for the region names doesn't bother us, but Kosovo is definitely a problem. According to Wikipedia, UNK is used for Kosovo residents whose travel documents were issued by the United Nations, so we will fill that in with an ugly hack immediately after loading the data:

```
raw <- read_csv("raw/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()
)
```

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

```
raw$ISO3[raw$Countries == "Kosovo"] <- "UNK"
missing <- raw %>%
  filter(is.na(Countries) | (Countries == "") | is.na(ISO3) | (ISO3 == "")) %>%
  select(Countries, ISO3)
missing
```

```
# A tibble: 20 x 2
   Countries                 ISO3
   <chr>                     <chr>
 1 ""                        ""
 2 ""                        ""
 3 ""                        ""
 4 Eastern and Southern Af~ ""
 5 Eastern Europe and Cent~ ""
 6 Latin America and the C~ ""
 7 Middle East and North A~ ""
 8 North America             ""
 9 South Asia                ""
10 West and Central Africa   ""
11 Western Europe            ""
12 Europe and Central Asia   ""
13 Sub-Saharan Africa        ""
14 Global                    ""
15 ""                        ""
16 ""                        Indicator definition: Percentage of infants bo~
17 ""                        Note: Data are not available if country did no~
18 ""                        Data source: Global AIDS Monitoring 2018 and U~
19 ""                        For more information on this indicator, please~
20 ""                        For more information on the data, visit data.u~
```

All right. Let's add that hack to our script, then save the result to a file. The whole thing is now 38 lines long:

```
# Constants.
raw_filename <- "raw/infant_hiv.csv"
tidy_filename <- "tidy/infant_hiv.csv"
num_rows <- 192
first_year <- 2009
last_year <- 2017
```

```
# Get and clean percentages.
raw <- read_csv(raw_filename, 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()
)

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

```
raw$ISO3[raw$Countries == "Kosovo"] <- "UNK"
sliced <- slice(raw, 1:num_rows)
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)

# Separate three-column chunks and add countries and years.
num_years <- (last_year - first_year) + 1
chunks <- vector("list", num_years)
for (year in 1:num_years) {
  end <- year + 2
  temp <- select(percents, year:end)
  names(temp) <- c("estimate", "hi", "lo")
  temp$country <- countries
  temp$year <- rep((first_year + year) - 1, num_rows)
  temp <- select(temp, country, year, everything())
  chunks[[year]] <- temp
}

# Combine chunks and order by country and year.
tidy <- bind_rows(chunks)
tidy <- arrange(tidy, country, year)

# Save.
write_csv(tidy, tidy_filename)
```

"It's alive!", we exclaim, but we can do better. Let's start by using a pipeline for the code that extracts and formats the percentages:

```
# Constants...

# Get and clean percentages.
```

```
raw <- read_csv(raw_filename, 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()
)
```

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

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

# Separate three-column chunks and add countries and years...

# Combine chunks and order by country and year...

# Check...
```

The two changes are:

1. We use a `%>%` pipe for the various transformations involved in creating percentages.
2. We write the result to `temp.csv` so that we can compare it to the file created by our previous script. We should always do this sort of comparison when refactoring code in ways that isn't meant to change output; if the file is small enough to store in version control, we could overwrite it and use `git diff` or something similar to check whether it has changed. However, we would then have to trust ourselves to be careful enough not to accidentally commit changes, and frankly, we are no longer sure how trustworthy we are…

After checking that this has not changed the output, we pipeline the computation in the loop:

```
# Constans...

# Get and clean percentages...

# Separate three-column chunks and add countries and years.
num_years <- (last_year - first_year) + 1
chunks <- vector("list", num_years)
for (year in 1:num_years) {
  chunks[[year]] <- select(percents, year:(year + 2)) %>%
    rename(estimate = 1, hi = 2, lo = 3) %>%
    mutate(country = countries,
```

```
          year = rep((first_year + year) - 1, num_rows)) %>%
    select(country, year, everything())
}

# Combine chunks and order by country and year.
tidy <- bind_rows(chunks) %>%
  arrange(country, year)
```

We have introduced a call to `rename` here to give the columns of each sub-table the right names, and used `mutate` instead of assigning to named columns one by one. The lack of intermediate variables may make the code harder to debug using print statements, but certainly makes this incantation easier to read aloud.

So we run it and inspect the output and it's the same as what we had and we're about to commit to version control when we decide to double check against the original data and guess what? The values for Argentina are wrong. In fact, the values for most countries and years are wrong: only the ones in the first three columns are right. The problem, it turns out, is that our loop index `year` is going up in ones, while each year's data is three columns wide. Here's the final, *final*, ***final*** version:

```
library(tidyverse)

# Constants.
raw_filename <- "raw/infant_hiv.csv"
tidy_filename <- "tidy/infant_hiv.csv"
first_year <- 2009
last_year <- 2017
num_rows <- 192

# Get and clean percentages.
raw <- read_csv(raw_filename, 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()
)

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

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

```r
# Separate three-column chunks and add countries and years.
num_years <- (last_year - first_year) + 1
chunks <- vector("list", num_years)
for (year in 1:num_years) {
  start = 3 * (year - 1) + 1
  chunks[[year]] <- select(percents, start:(start + 2)) %>%
    rename(estimate = 1, hi = 2, lo = 3) %>%
    mutate(country = countries,
           year = rep((first_year + year) - 1, num_rows)) %>%
    select(country, year, everything())
}

# Combine chunks and order by country and year.
tidy <- bind_rows(chunks) %>%
  arrange(country, year)

# Check.
write_csv(tidy, tidy_filename)
```

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. If people are going to use our cleaned-up data in their analyses, we need a better way to develop and check our scripts.

6.5 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.

Chapter 7

Non-Standard Evaluation

7.1 Questions

- When and how does R evaluate code?
- How can we take advantage of this?

7.2 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.

The biggest difference between R and Python is not where R starts counting, but its use of lazy evaluation. Nothing truly makes sense in R until we understand how this works.

7.3 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 just been assigned `"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 a lot more going on here than first meets the eye. 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"
```

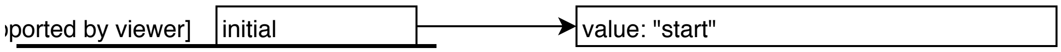


Figure 7.1: Python Step 1

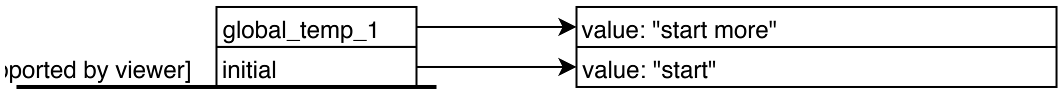


Figure 7.2: Python Step 2

```
    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)
```

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:

Step 5: Python creates a new stack frame to hold the call to `ones_func`:

Step 6: Python creates a temporary 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 its 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 modern programming languages do.

7.4 How does R evaluate the same kind of thing?

In contrast, R uses lazy evaluation. Here's an R program that's roughly equivalent to the Python shown above:

```
ones_func <- function(ones_arg) {
  paste(ones_arg, "ones")
```

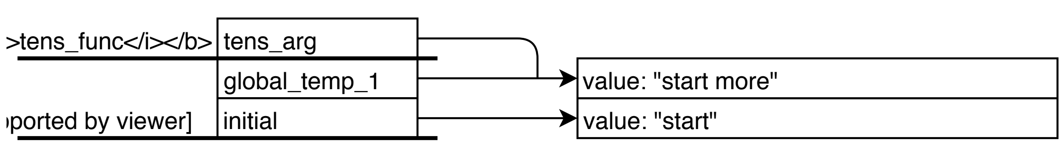


Figure 7.3: Python Step 3

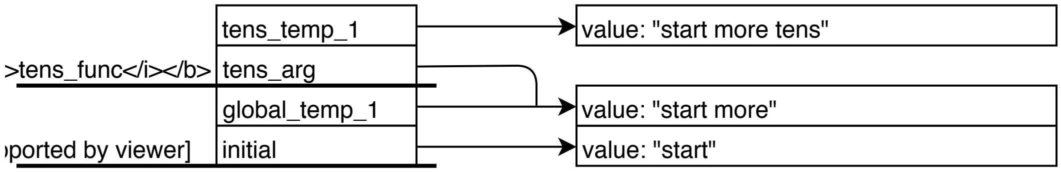


Figure 7.4: Python Step 4

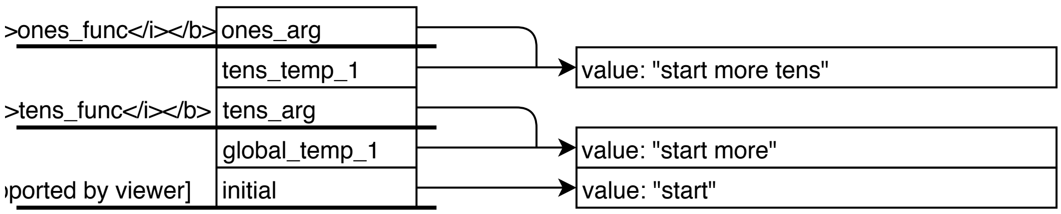


Figure 7.5: Python Step 5

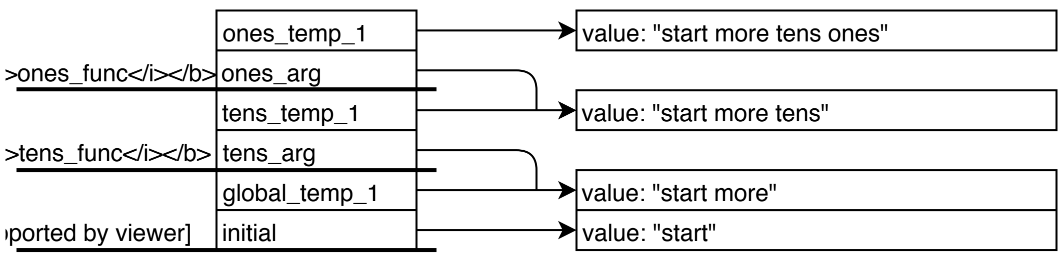


Figure 7.6: Python Step 6

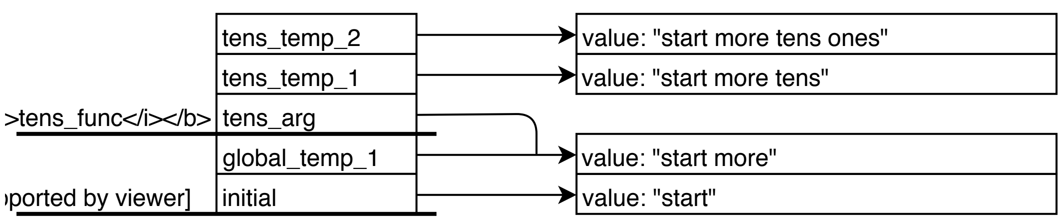


Figure 7.7: Python Step 7

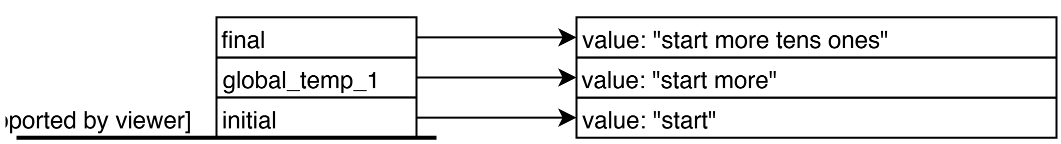


Figure 7.8: Python Step 8

```
}

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 `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'
> ```

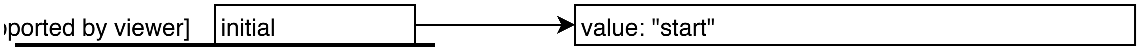


Figure 7.9: R Step 1

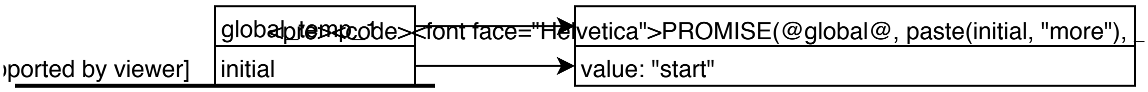


Figure 7.10: R Step 2

```
op <- '+'
combined <- paste(left, op, right)
tree <- parse(text = combined)
```

or by calling functions. The function-based approach is safer and more flexible, so once we introduce the way R handles regular function calls, we'll dive into that.

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`:

Crucially, the promise in `tens_func` remembers that it was created in the global environment: it's eventually going to 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:

Step 10: `tens_func` returns:

We got the same answer, 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

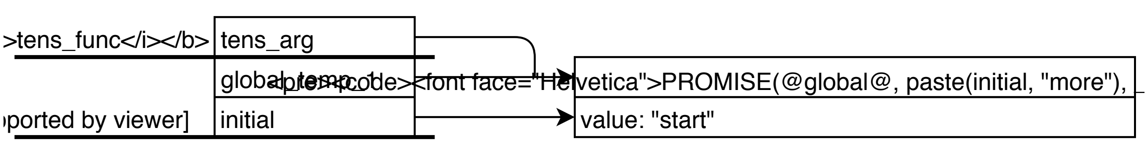


Figure 7.11: R Step 3

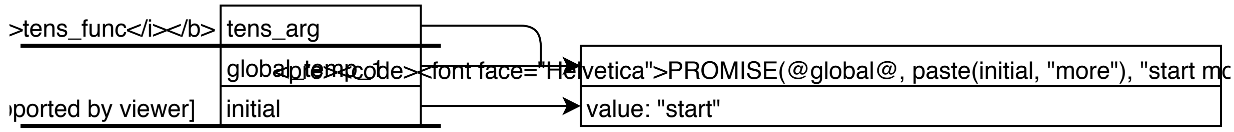


Figure 7.12: R Step 4

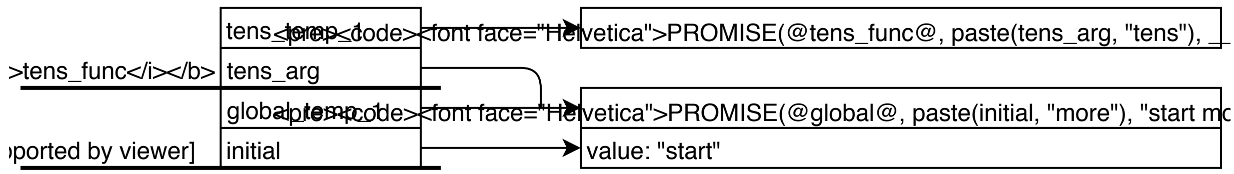


Figure 7.13: R Step 5

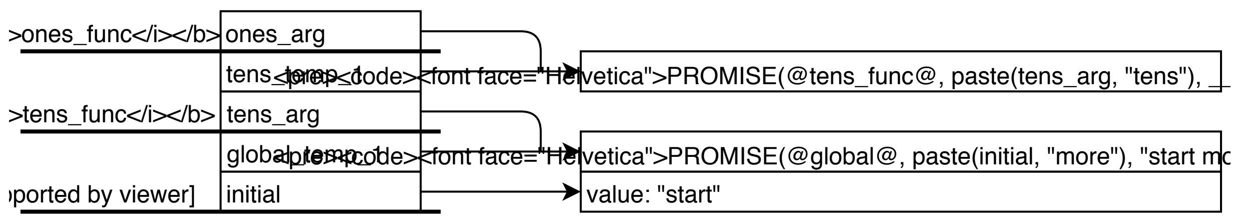


Figure 7.14: R Step 6

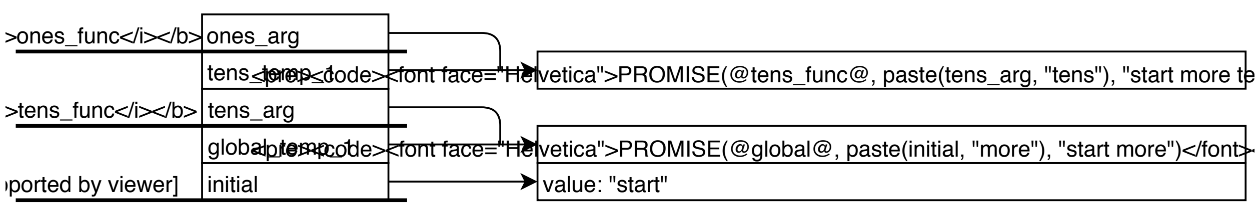


Figure 7.15: R Step 7

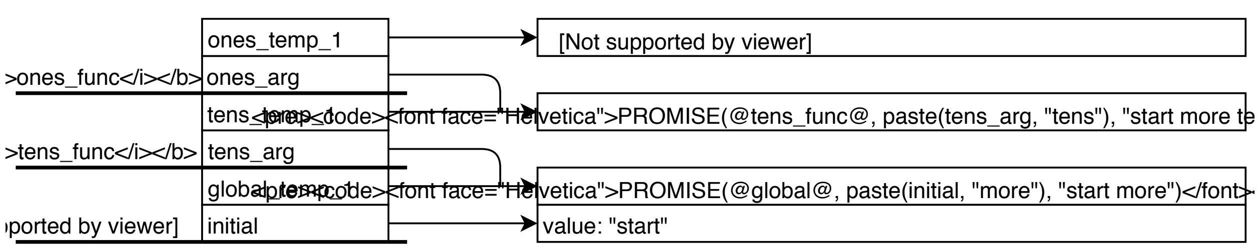


Figure 7.16: R Step 8

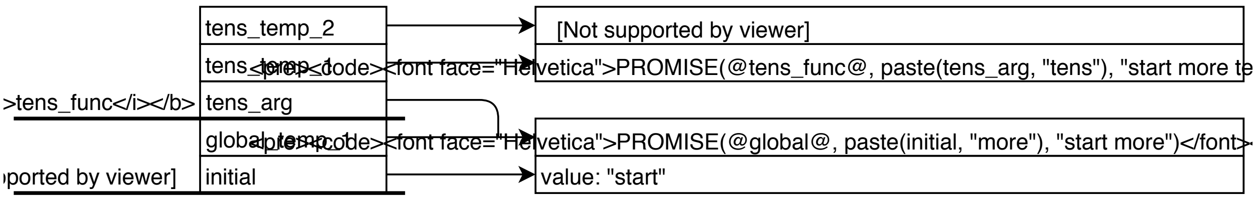


Figure 7.17: R Step 9

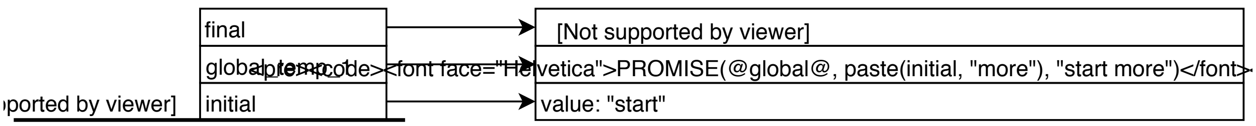


Figure 7.18: R Step 10

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, but that's a lot of typing even for me.)

7.5 Why is lazy evaluation useful?

R's lazy evaluation seems pointless if it always produces the same answer as Python's eager evaluation, but as you may already have guessed, it doesn't have to. To see how powerful lazy evaluation can be, let's create an expression of our own:

```
my_expr <- expr(a)
```

Displaying the value of `my_expr` isn't very exciting:

```
my_expr
```

```
a
```

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)
```

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

We haven't created a variable called `my_expr`, 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?

```
a <- "this is a"
eval(my_expr)
```

```
[1] "this is a"
```

More usefully, what if we create something that has a value for `a`:

```
my_data <- tribble(
  ~a, ~b,
  1,  10,
  2,  20
)
my_data
```

```
# A tibble: 2 x 2
      a     b
  <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, my_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, in the tibble `my_data`. Since that tibble has a column called `a`, `eval(my_expr, my_data)` gives us that column.

This may not seem life-changing yet, but being able to pass expressions around and evaluate them in various contexts allows us to seem very clever indeed. For example, let's create another expression:

```
add_a_b <- expr(a + b)
typeof(add_a_b)
```

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

The type of `add_a_b` is `language` rather than `symbol` because it contains more than just a symbol, but it's still an expression, so we can evaluate it in the context of our data frame:

```
eval(add_a_b, my_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
}
```

It takes a tibble and some logical expressions, evaluates each expression in turn, and returns a vector of results:

```
run_many_checks(my_data, expr(0 < a), expr(a < b))
```

```
[[1]]
```

```
[1] TRUE TRUE

[[2]]
[1] TRUE TRUE
```

We can take it one step further and simply report whether 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(my_data, expr(0 < a), expr(a < b))
```

```
[1] TRUE
```

Just to make sure it's actually working, we'll try something that ought to fail:

```
run_all_checks(my_data, expr(b < 0))
```

```
[1] FALSE
```

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?

```
check_all(my_data, 0 < a, a < b)
```

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(a + b))
> eval(conditions[1], my_data)
> ```
>
> ```
> [[1]]
> a + b
> ```
>
> What I did wrong was use `[` instead of `[[`, which meant that `conditions[1]` was not an expression—it wa a list containing a single expression:
>
> ```
> conditions[1]
> ```
>
> ```
> [[1]]
> a + b
> ```
>
> 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.

7.6 What is tidy evaluation?

Our goal is to write something that looks like it belongs in the tidyverse. We'll start by creating a tibble to play with:

```
both_hands <- tribble(
  ~left, ~right,
  1,     10,
  2,     20
)
both_hands
```

```
# A tibble: 2 x 2
   left right
  <dbl> <dbl>
1     1    10
2     2    20
```

We want to be able to write this:

```
check_all(both_hands, 0 < left, left < right)
```

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(both_hands, left != right)
```

```
Error in eval(test, data): object 'left' not found
```

This makes sense: by the time we get to the `eval` call, `test` refers to a promise that represents the value of `left != right` 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(both_hands, left != right)
```

```
[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 should use a function called `enquo` which returns an object called a quosure that contains only an unevaluated expression and an environment. Let's try using that:

```
check_using_enquo <- function(data, test) {
  q_test <- enquo(test)
  eval(q_test, data)
}
check_using_enquo(both_hands, left != right)
```

```
<quosure>
expr: ^left != right
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
```

All right: we're ready to write `check_all`. As a reminder, our test data looks like this:

```
both_hands
```

```
# A tibble: 2 x 2
   left right
  <dbl> <dbl>
1     1    10
2     2    20
```

Our first attempt (which only handles a single test) is a deliberate failure:

```
check_without_quoting_test <- function(data, test) {
  data %>% transmute(result = test) %>% pull(result) %>% all()
}
check_without_quoting_test(both_hands, left < right)
```

```
Error in mutate_impl(.data, dots): object 'left' not found
```

Good: we expected that to fail because we're not enquoting the test. (If this *had* worked, it would have told us that we still don't understand what we're doing.) Let's modify it 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(both_hands, left < right)
```

```
Error in mutate_impl(.data, dots): 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(both_hands, left < right)
```

```
[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(both_hands, 0 < left, left < right)
```

```
[1] TRUE
```

And just to make sure that it fails when it's supposed to:

```
check_all(both_hands, left > right)
```

```
[1] FALSE
```

Backing up a bit, `!!` works because there are two broad categories 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 `both_hands$left`, the `$` function is being passed `both_hands` and the quoted expression `left`. This is why we can't use variables as field names with `$`:

```
the_string_left <- "left"
both_hands$the_string_left
```

```
Warning: Unknown or uninitialised column: 'the_string_left'.
```

```
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:

```
both_hands[the_string_left]     # single square brackets
```

```
# A tibble: 2 x 1
   left
  <dbl>
1     1
2     2
```

or a naked vector:

```
both_hands[[the_string_left]]   # double square brackets
```

```
[1] 1 2
```

7.7 What have we learned?

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 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.

If you would like to know more, or check that what you now think you understand is accurate, this tutorial by Ian Lyttle is a good next step.

7.8 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 8

Handling Errors

8.1 Questions

- How does R signal errors?
- How can I handle errors myself?

8.2 Learning Objectives

- Name and describe the three levels of error handling in R.
- Handle an otherwise-fatal error in a function call in R.

Cautious programmers plan for the unexpected. In Python, this is done 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.) We can signal conditions of these three 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. Note also that there are only a few situations in which a warning is appropriate: if something has truly gone wrong, we should stop, and if it hasn't, 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`, 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)
```

```
result is NA 0.01 0.1 1
```

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
```

If we are *sure* that we wish to incur the risk of silent failure, we can suppress error messages from `try`:

```
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, for it will one day leave you lost and gibbering in an incomprehensible 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) cat("error object is", as.character(cnd))
)
```

```
error object is Error in doTryCatch(return(expr), name, parentenv, handler): our message
```

(We need to convert the error object `cnd` to character for printing because it is a list of two elements, the message and the call, but `cat` only handles character data.) We can now run a function that would otherwise blow up:

```
tryCatch(
  attemptWithoutTry(1, "two"),
  error = function(cnd) cat("error object is", as.character(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* for details, or this tutorial by Omayma Said.

8.3 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.

Chapter 9

Object-Oriented Programming

9.1 Questions

- How can I do object-oriented programming in R?
- How do I specify an object's class?
- How do I provide methods for a class?
- How should I create objects of a class I have defined?

9.2 Learning Objectives

- 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.

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 experienced ones to create horrors.

R has not one, not two, but at least three different frameworks for object-oriented programming. By far the most widely used is known as S3 (because it was first introduced with Version 3 of S, the language from which R is derived). Unlike the approaches used in Java, Python, and similarly pedestrian languages, S3 does not require users to define classes. Instead, they add attributes to data, then write specialized version 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 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), 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 finally 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
```

9.4 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'll create two coordinate vectors:

```r
first <- c(0.5, 0.7)
class(first) <- "two_d"
print(first)
```

```
[1] 0.5 0.7
attr(,"class")
[1] "two_d"
```

```r
second <- c(1.3, 3.1)
class(second) <- "two_d"
print(second)
```

```
[1] 1.3 3.1
attr(,"class")
[1] "two_d"
```

Separately, let's define the behavior of `toString` for such objects:

```r
toString.two_d <- function(obj){
  paste0("<", obj[1], ", ", obj[2], ">")
}
toString(first)
```

```
[1] "<0.5, 0.7>"
```

```r
toString(second)
```

```
[1] "<1.3, 3.1>"
```

S3's protocol is simple: given a function F and an object whose class is C, it 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`:

```r
library(sloop)
s3_dispatch(toString(first))
```

```
=> toString.two_d
 * toString.default
```

Compare this with calling `toString` on a plain old character vector:

```r
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 the staggering complexity that can result from doing so. Instead, we should always write three functions that work together for a class like `prospects`:

- 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 is the helper that provides a user-friendly interface to construction of our 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. Purely for illustrative purposes, 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>"
```

9.5 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 explore this, 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.

```
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:

```
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.

9.6 Key Points

- 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).

Chapter 10

Intellectual Debt

10.1 Questions

- What grievous sin can I most easily avoid when using R?
- How can I pipeline functions when the incoming data doesn't belong in the first parameter's position?
- Why does assigning to elements of data structures sometimes appear not to change them?
- How does R handle errors, and how can I handle them myself?

10.2 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.

We have accumulated some intellectual debt in the previous lessons, and we should clear some of before we go on to new topics.

10.3 Why shouldn't I use `setwd`?

Because reasons.

But…

No. Use the here package.

10.4 How do I write formulas?

One feature of R that doesn't have an exact parallel in Python is the formula operator `~` (tilde). Its original (and still most common) purpose is to provide a convenient syntax for expressing the formulas used in fitting linear regression models. The basic format of these formulas is `response ~ predictor`, where `response` and `predictor` depend on the variables in the program. For example, `Y ~ X` means, "`Y` is modeled as a function of `X`", so `lm(Y ~ X)` means "fit a linear model that regresses `Y` on `X`".

What makes `~` work is lazy evaluation: what actually gets passed to `lm` in the example above is a formula object that stores the expression representing the left side of the `~`, the expression representing the right side, and the environment in which they are to be evaluated. This means that we can write something like:

```
fit <- lm(Z ~ X + Y)
```

to mean "fit `Z` to both `X` and `Y`", or:

```
fit <- lm(Z ~ . - X, data = D)
```

to mean "fit `Z` to all the variables in the data frame `D` *except* the variable `X`." (Here, we use the shorthand `.` to mean "the data being manipulated".)

But `~` can also be used as a unary operator, because its true effect is to delay computation. For example, we can use it in the function `tribble` to give names to columns as we create a tibble on the fly:

```
temp <- tribble(
  ~left, ~right,
  1,     10,
  2,     20
)
temp
```

```
# A tibble: 2 x 2
   left right
  <dbl> <dbl>
1     1    10
2     2    20
```

Used cautiously and with restraint, lazy evaluation allows us to accomplish marvels. Used unwisely—well, there's no reason for us to dwell on that, particularly not after what happened to poor Higgins…

10.5 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). What R and other languages therefore do is store each string once and associate 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. (Just to keep us on our toes, R allows factors to be either ordered or unordered.)

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 fit into any of the categories on offer.
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,
```

```r
  "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:

```r
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 the `person` column into factors, since names are actually character strings, but the `flavor` column is a good candidate:

```r
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:

```r
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"))
```

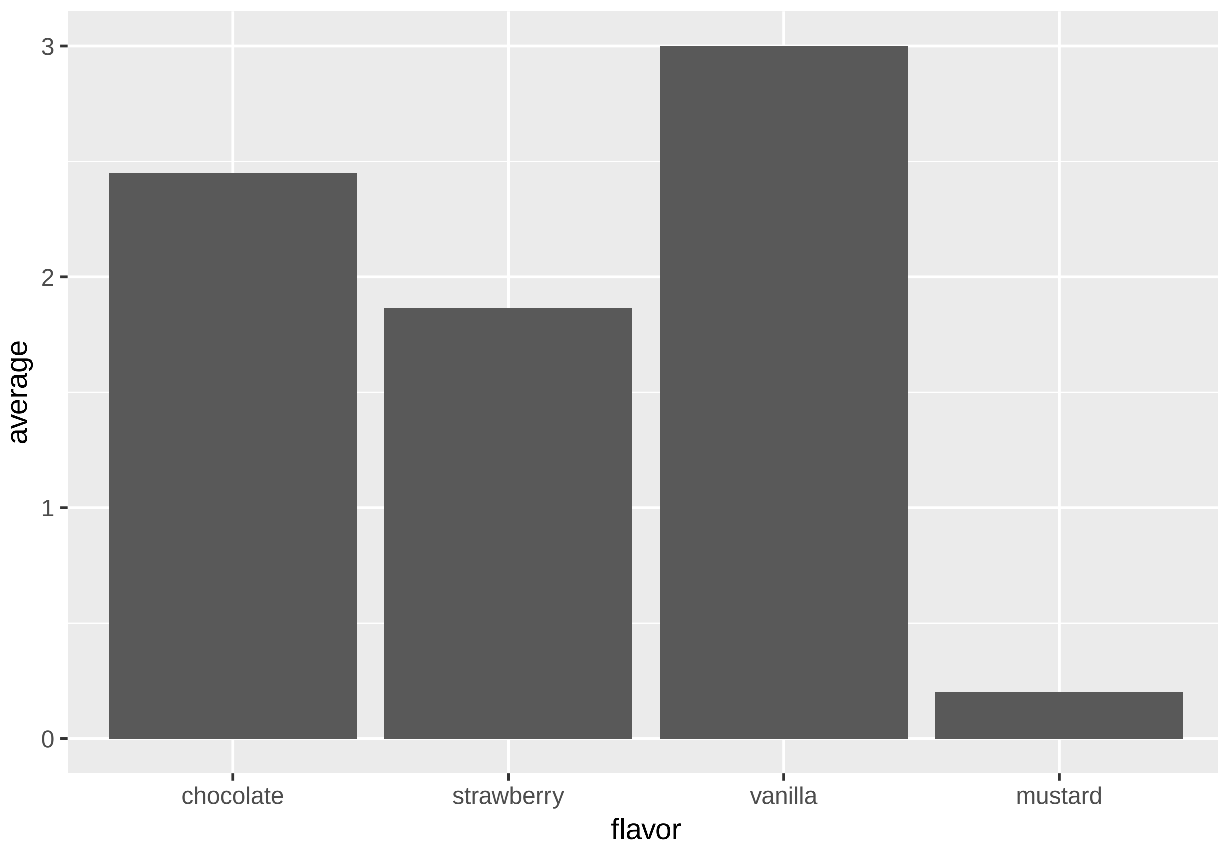
This changes the order in which they are displayed after grouping:

```
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() +
  geom_col(mapping = aes(x = flavor, y = average))
```



To learn more about how factors work and how to use them when analyzing categorical data, please see this paper by McNamara and Horton.

10.6 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 use `.x` and `.y`, and for more arguments, we can use `..1`, `..2`, and so on:

```
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, it's easy to 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
```

10.7 How does R give the appearance of immutable data?

Another feature of R that can surprise the unwary is copy-on-modify, which means that if two or more variables refer to the same data and that data is updated via one variable, R automatically makes a copy so that the other variable's value doesn't change. Here's a simple example:

```
first <- c("red", "green", "blue")
second <- first
cat("before modification, first is", first, "and second is", second, "\n")
```

```
before modification, first is red green blue and second is red green blue
```

```
first[[1]] <- "sulphurous"
cat("after modification, first is", first, "and second is", second, "\n")
```

```
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
cat("after modification\n")
```

```
after modification
```

```
first
```

```
# A tibble: 2 x 2
   left right
  <dbl> <dbl>
1   999   202
2   303   404
```

```
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 pryr library:

```
library(pryr)
```

```
Attaching package: 'pryr'

The following objects are masked from 'package:sloop':

    ftype, is_s3_generic, is_s3_method, otype

The following object is masked from 'package:rlang':

    bytes
```

```
The following objects are masked from 'package:purrr':

    compose, partial
```

```r
first <- tribble(
  ~left, ~right,
  101,   202,
  303,   404
)
tracemem(first)
```

```
[1] "<0x7f8c3e1da408>"
```

```r
first$left[[1]] <- 999
```

```
tracemem[0x7f8c3e1da408 -> 0x7f8c3c763088]: eval eval withVisible withCallingHandlers handle timing_fn
tracemem[0x7f8c3c763088 -> 0x7f8c3c763908]: eval eval withVisible withCallingHandlers handle timing_fn
tracemem[0x7f8c3c763908 -> 0x7f8c3c764188]: $<-.data.frame $<- eval eval withVisible withCallingHandlers
tracemem[0x7f8c3c764188 -> 0x7f8c3c73d708]: $<-.data.frame $<- eval eval withVisible withCallingHandlers
```

```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 `address`:

```r
left <- first$left # alias
cat("left column is initially at", address(left), "\n")
```

```
left column is initially at 0x7f8c3c7634c8
```

```r
first$left[[2]] <- 888
cat("after modification, the original column is still at", address(left), "\n")
```

```
after modification, the original column is still at 0x7f8c3c7634c8
```

```r
temp <- first$left # another alias
cat("but the first column of the tibble is at", address(temp), "\n")
```

```
but the first column of the tibble is at 0x7f8c3bbd9008
```

(We need to use aliases because `address(first$left)` doesn't work: the argument 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.)

10.8 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.

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:

```r
order(c("g", "c", "t", "a"))
```

```
[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.

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.

Functions and columns: There's a function called `n`. It's not the same thing as a column called `n`.

```
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
```

10.9 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 11

Projects

11.1 Questions

- How do I create a package in R?
- What can go in an R package?
- How are R packages distributed?
- What data formats can be used in an R package?
- How should I document an R package?

11.2 Learning Objectives

- 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.

Mistakes were made in the previous tutorial. 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. Luckily for us, R provides tools for this purpose not unlike those available in Python. In order to use them, we must first venture into the greater realm of packaging in R.

11.3 What's in an R package?

Unlike Python, with its confusing plethora of packaging tools, R has one way to do it. Before converting our project into a package, we will explore what a package should contain.

- The text file `DESCRIPTION` (with no suffix) holds most of the package's metadata, including a description of what it, 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 extension, 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.
- Just as `.gitignore` tells Git what files in a project to ignore, `.Rbuildignore` tells RStudio which files it doesn't need to worry about when building a package from source.
- All of the R source for our package must go in a directory called `R`; sub-directories are not allowed.
- As you would expect from its name, the `data` directory contains all the data in our package. In order for it to be loadable as part of the package, the data must be saved in `.rda` format. We can use R's function `save` to do this (and use `load` in our code to restore it), but a better choice is to load the `usethis` library and call `usethis::use_data(object, overwrite = TRUE)`.
- 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 comments in our source code and use a tool called roxygen2 to extract them and format them as required.
- 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 later.

In order to understand the rest of what follows, it's important to understand 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":

```
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? The safest way is to use fully-qualified names such as `stringr::str_replace` every time we call a function defined somewhere outside our package. We will explore other options below.

And while we're hedging the statements we have already made:

1. 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.
2. We should always put `LazyData: TRUE` in `DESCRIPTION` so that datasets are only loaded on demand.

11.4 How do I create a package?

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.

The first step is to run RStudio's project creation wizard. We will create `unicefdata` directly under our home directory, make it a Git repository, and turn on Packrat (a package manager for R):

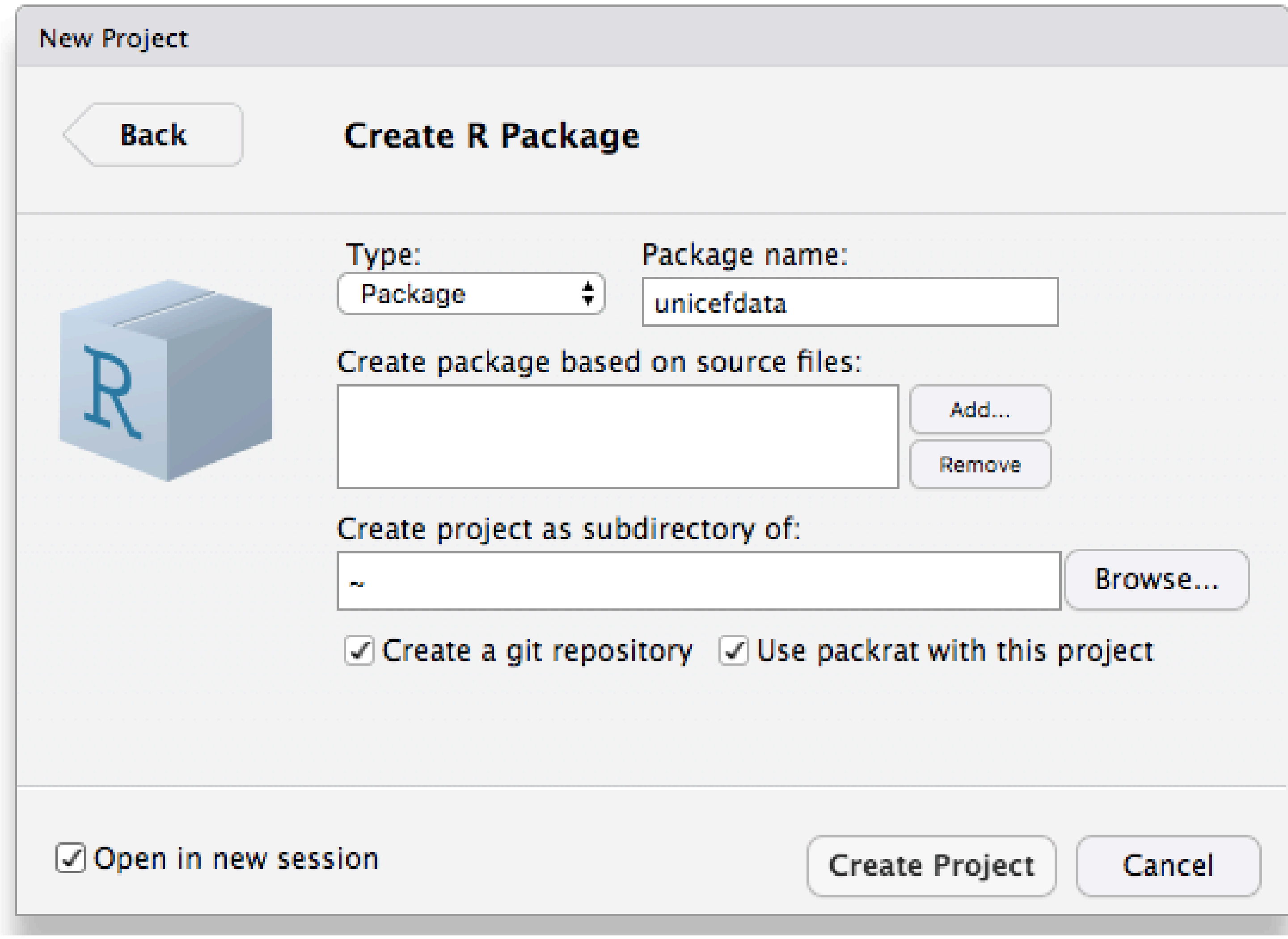


Figure 11.1: RStudio Project Creation Wizard

Before doing our first commit to version control, we remove `R/hello.R` and `man/hello.Rd` (which the project wizard helpfully provides as starting points), and add `README.md`, `LICENSE.md`, `CONDUCT.md`, and `CITATION.md` to describe the project as a whole, its license, the contributor code of conduct, and how we want the project cited. These files are nothing to do with R per se (and as we'll see below, R isn't entirely happy having them here with these names), but every project should have all four of these *somewhere*.

After committing all of this to version control, we copy the data tidying script we wrote previous into `R/tidy_datasets.R`. For reference, this is what the file looks like at this point:

```
library(tidyverse)

# Constants.
raw_filename <- "inst/extdata/infant_hiv.csv"
tidy_filename <- "/tmp/infant_hiv.csv"
first_year <- 2009
last_year <- 2017
num_rows <- 192

# Get and clean percentages.
raw <- read_csv(raw_filename, skip = 2, na = c("-"))
raw$ISO3[raw$Countries == "Kosovo"] <- "UNK"
sliced <- slice(raw, 1:num_rows)
countries <- sliced$ISO3
percents <- sliced %>%
  select(-ISO3, -Countries) %>%
  map_dfr(str_replace, pattern = ">?(\\d+)%", replacement = "\\1") %>%
  map_dfr(function(col) as.numeric(col) / 100)

# Separate three-column chunks and add countries and years.
num_years <- (last_year - first_year) + 1
chunks <- vector("list", num_years)
for (year in 1:num_years) {
  start = 3 * (year - 1) + 1
  chunks[[year]] <- select(percents, start:(start + 2)) %>%
    rename(estimate = 1, hi = 2, lo = 3) %>%
    mutate(country = countries,
           year = rep((first_year + year) - 1, num_rows)) %>%
    select(country, year, everything())
}

# Combine chunks and order by country and year.
tidy <- bind_rows(chunks) %>%
  arrange(country, year)

# Check.
write_csv(tidy, tidy_filename)
```

We're going to need to wrap this up as a function so that these commands aren't executed while the library loads, and we should probably also allow the user to specify the locations of the input and output files—we'll come back and do all of this later.

First, though, let's edit `DESCRIPTION` and:

1. change the `Title`, `Author`, `Maintainer`, and `Description`;
2. change the `License` to `MIT` (see Choose a License for other options); and
3. go to the `Build` tab in RStudio and run 'Check' to see if our package meets CRAN's standards.

A note on this: CRAN is the Comprehensive R Archive Network. Like the Python Package Index, it is the place to go to find the packages you need. CRAN's rules are famously strict, which ensures that packages run for everyone, but which also makes package development a little more onerous than it might be.

For example, when we run `Check`, we get this:

```
* checking top-level files ... NOTE
Non-standard files/directories found at top level:
  'CITATION.md' 'CONDUCT.md' 'LICENSE.md'
...
Found the following CITATION file in a non-standard place:
  CITATION.md
Most likely 'inst/CITATION' should be used instead.
```

After a bit of searching online, we rearrange these files as follows:

1. `LICENSE.md` becomes `LICENSE` with no extension (but still in the root directory).
2. The `DESCRIPTION` entry for the license is updated to `License: MIT + file LICENSE` (spelled exactly that way).
3. We add lines to `.Rbuildignore` to tell R to ignore `CITATION.md` and `CONDUCT.md`. We could instead move `CITATION.md` to `inst/CITATION` so that it will be copied into the root of the installation directory on users' machines, but a lot of people expect to find the citation description in the root directory of the original project. We could also duplicate the file, and once the package is mature enough to deploy, that might be the best answer.

These changes fix the warnings about non-standard files in non-standard places, but we are far from done—we also have this to deal with:

```
checking for missing documentation entries ... WARNING
Undocumented code objects:
  'tidy_infant_hiv'
All user-level objects in a package should have documentation entries.
```

Fair enough—we want people to know what our package is for and how to use it, so 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 wrap our existing code with this:

```r
#' Tidy up the infant HIV data set.
#'
#' @param src_path path to source file
#'
#' @return a tibble of tidy data
#'
#' @export

tidy_infant_hiv <- function(src_path) {
  # ...all the code from before...
  # Return the final tidy dataset.
  tidy
}
```

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; here, we are saying that:

- the function has a single parameter called `src_path`;
- 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()` to regenerate documentation: it isn't done automatically. When we do this, we get good news and bad news:

```
Updating unicefdata documentation
Loading unicefdata
First time using roxygen2. Upgrading automatically...
Updating roxygen version in /Users/gvwilson/unicefdata/DESCRIPTION
Warning: The existing 'NAMESPACE' file was not generated by roxygen2, and will not be overwritten.
Writing tidy_infant_hiv.Rd
```

Ah—the tutorials did warn us about this. We need to delete `NAMESPACE` and re-run `devtools:document()` in order to create this file:

```
# Generated by roxygen2: do not edit by hand

export(tidy_infant_hiv)
```

The comment at the start tells roxygen2 it can overwrite the file, and reminds us that we shouldn't edit it by hand. The `export(tidy_infant_hiv)` directive is what we really want: as you might guess, it tells the package builder which function to make visible.

After doing this, we go into "Build…More…Configure build tools" and check "Generate documentation with Roxygen". Running the build again gives us:

```
tidy_infant_hiv: no visible global function definition for 'read_csv'
tidy_infant_hiv: no visible global function definition for 'slice'
tidy_infant_hiv: no visible global function definition for '%>%'
tidy_infant_hiv: no visible global function definition for 'select'
...several more...
tidy_infant_hiv: no visible global function definition for 'bind_rows'
tidy_infant_hiv: no visible global function definition for 'arrange'
Undefined global functions or variables:
  %>% Countries ISO3 arrange bind_rows country everything map_dfr
  mutate read_csv rename select slice str_replace year
```

This is the package loading problem mentioned earlier: since our compiled-and-distributable package will only contain the bytecodes for its own functions, direct calls to functions from other libraries won't work after the package is installed. We will demonstrate two ways to fix this. First, we can add this to the roxygen2 comment block for our function:

```
#' @import dplyr
#' @importFrom magrittr %>%
```

which tells the package builder that we want all of the functions in these two packages available. Second (and more properly) we can change various call to use their package prefix, e.g.:

```
  percents <- sliced %>%
    select(ISO3, Countries) %>%
    purrr::map_dfr(stringr::str_replace, pattern = ">?(\\d+)%", replacement = "\\1") %>%
    purrr::map_dfr(function(col) as.numeric(col) / 100)
```

This changes the error to one that is slightly more confusing:

```
* checking package dependencies ... ERROR
Namespace dependencies not required: 'dplyr' 'magrittr'

See section 'The DESCRIPTION file' in the 'Writing R Extensions'
manual.
```

```
* DONE
Status: 1 ERROR
checking package dependencies ... ERROR
Namespace dependencies not required: 'dplyr' 'magrittr'
```

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)
```

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.

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 `tidy_infant_hiv("inst/extdata/infant_hiv.csv")`. This produces:

```
Error in read_csv(src_path, skip = 2, na = c("-")) :
  could not find function "read_csv"
```

After calling upon the names of Those Who Shall Not Be Named and making a fresh cup of tea, we re-read our code and realize that we forgot to rename `read_csv` to `readr::read_csv`. Fixing this doesn't fix the problem with column names, though; to do that, we add this to the roxygen2 comment block:

```
#' @importFrom rlang .data
```

and then modify the calls that use naked column names to be:

```
    select(-.data$ISO3, -.data$Countries) %>%
    ...
    select(.data$country, .data$year, everything())
    ...
    arrange(.data$country, .data$year)
```

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.

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 load and clean the data, storing the cleaned tibble in a variable called `infant_hiv`, then load the `usethis` package and called `usethis::use_data(infant_hiv)` to store the tibble in `data/infant_hiv.rda`. (We could just call `save` with the appropriate parameters, but `usethis` is a useful set of tools for creating and managing R projects, and in retrospect, we should have started using it earlier.) Note: 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.

Second, we create a file called `R/infant_hiv.R` to hold documentation about the dataset and put this in it:

```
#' 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{year}{Year reported}
#'   \item{estimate}{Best estimate of rate (may be NA)}
#'   \item{hi}{High end of estimate range (may be NA)}
#'   \item{lo}{Low end of estimate range (may be NA)}
#' }
"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.

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 and maternal mortality.
#'
#' @author Greg Wilson, \email{greg.wilson@rstudio.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?)

11.5 Key Points

- 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 12

Testing

12.1 Questions

- How can I test R code?
- How can I test an R package?

12.2 Learning Objectives

- Create unit tests in R.
- Create unit tests for an R package.

In keeping with common programming practice, we have left testing until the end.

12.3 What should I know about testing in general?

The standard testing library for R is testthat. Like Python's unittest library, it is a member of the xUnit family of 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:

```r
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
```

```r
test_that("Zero equals itself", {expect_equal(0, 0)})
```

As is conventional with xUnit-style 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:

```r
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 They 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 of some sort for `test_that` to run. We can run `expect_equal` on its own:

```r
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:

```r
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
```

Note that a block of code is *not* the same thing as an anonymous function, which is why running this block of code does nothing:

```r
test_that("Using an anonymous function", function(){
  print("In our anonymous function")
  expect_equal(0, 1)
})
```

But running blocks of tests by hand is a bad practice no matter what is in them. What we should do instead is put related tests in files, 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/test_example.R`:

```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", {
```

```r
  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:

```r
test_dir("tests/testthat")
```

```
v |  OK F W S | Context

/ |   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 |   0 1 1   | Finding empty rows
---------------------------------------------------------------------------
test_find_empty_a.R:5: warning: (unknown)
cannot open file '../scripts/find_empty_02.R': No such file or directory

test_find_empty_a.R:5: error: (unknown)
cannot open the connection
1: source("../scripts/find_empty_02.R") at tests/testthat/test_find_empty_a.R:5
2: file(filename, "r", encoding = encoding)
---------------------------------------------------------------------------

/ |   0       | Finding empty rows
x |   0 1 1   | Finding empty rows
---------------------------------------------------------------------------
test_find_empty_b.R:5: warning: (unknown)
cannot open file '../scripts/find_empty_02.R': No such file or directory

test_find_empty_b.R:5: error: (unknown)
cannot open the connection
1: source("../scripts/find_empty_02.R") at tests/testthat/test_find_empty_b.R:5
2: file(filename, "r", encoding = encoding)
---------------------------------------------------------------------------

/ |   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 ================================================================
OK:       5
Failed:   4
Warnings: 3
Skipped:  0
```

A bit of 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 results of 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/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 is likely to ensue if we do not.) Now let's run all of our tests:

```r
test_dir("tests/testthat")
```

```
v |  OK F W S | Context

/ |   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 |   0 1 1   | Finding empty rows
---------------------------------------------------------------------------
test_find_empty_a.R:5: warning: (unknown)
cannot open file '../scripts/find_empty_02.R': No such file or directory
```

```
test_find_empty_a.R:5: error: (unknown)
cannot open the connection
1: source("../scripts/find_empty_02.R") at tests/testthat/test_find_empty_a.R:5
2: file(filename, "r", encoding = encoding)
---------------------------------------------------------------------------

/ |   0       | Finding empty rows
x |   0 1 1   | Finding empty rows
---------------------------------------------------------------------------
test_find_empty_b.R:5: warning: (unknown)
cannot open file '../scripts/find_empty_02.R': No such file or directory

test_find_empty_b.R:5: error: (unknown)
cannot open the connection
1: source("../scripts/find_empty_02.R") at tests/testthat/test_find_empty_b.R:5
2: file(filename, "r", encoding = encoding)
---------------------------------------------------------------------------

/ |   0       | Testing properties of tibbles
v |   1       | Testing properties of tibbles

== Results ================================================================
OK:       5
Failed:   4
Warnings: 2
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, : No matc
```

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.

12.4 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` that defines a single function `find_empty_rows` that identifies 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`, from which 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. This one's 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 (as I am at the time of writing) new to R 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 RMarkdown 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 have strongly-typed variants as well.)
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 |   0 1 1   | Finding empty rows
------------------------------------------------------------------------------
test_find_empty_a.R:5: warning: (unknown)
cannot open file '../scripts/find_empty_02.R': No such file or directory

test_find_empty_a.R:5: error: (unknown)
cannot open the connection
1: source("../scripts/find_empty_02.R") at tests/testthat/test_find_empty_a.R:5
2: file(filename, "r", encoding = encoding)
------------------------------------------------------------------------------

== Results =====================================================================
OK:       0
Failed:   1
Warnings: 1
Skipped:  0

Don't worry, you'll get it.
```

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")
```

Ah: we are being given an integer vector of zero length rather than `NULL`. Let's have a closer look at the properties of this strange beast:

```r
print("is integer(0) equal to NULL")
```

```
[1] "is integer(0) equal to NULL"
```

```r
is.null(integer(0))
```

```
[1] FALSE
```

```r
print("any(logical(0))")
```

```
[1] "any(logical(0))"
```

```r
any(logical(0))
```

```
[1] FALSE
```

```r
print("all(logical(0))")
```

```
[1] "all(logical(0))"
```

```r
all(logical(0))
```

```
[1] 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. But `all` of an empty vector being `TRUE` is…unexpected. 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 behavior 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
x |   0 1 1   | Finding empty rows
---------------------------------------------------------------------------
test_find_empty_b.R:5: warning: (unknown)
cannot open file '../scripts/find_empty_02.R': No such file or directory

test_find_empty_b.R:5: error: (unknown)
cannot open the connection
1: source("../scripts/find_empty_02.R") at tests/testthat/test_find_empty_b.R:5
2: file(filename, "r", encoding = encoding)
---------------------------------------------------------------------------

== Results ================================================================
OK:       0
Failed:   1
Warnings: 1
Skipped:  0
```

12.5 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,age 35-49
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 for MI
,,,,,,,,,,,,,,
Note:,"Database include reanalyzed data from DHS and MICS, using a reference period of two years before
,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.

12.5.1 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:

```
read_csv("raw/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_character(),
  X14 = col_character(),
  X15 = col_character(),
  X16 = col_character()
)

[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:

```
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`:

```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)
})
```

```
Error: Test failed: 'The right row is found when there are header rows'
* `result` not equal to 2.
Lengths differ: 0 is not 1
```

```r
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)
})
```

```
Error: Test failed: 'The right row is found when there are header rows and blank lines'
* `result` not equal to 3.
Lengths differ: 0 is not 1
```

```r
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)
})
```

```
Error: Test failed: 'The right row is found when there are no header rows to discard'
* `result` not equal to 0.
Lengths differ: 0 is not 1
```

```r
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")
})
```

```
Error: Test failed: 'No row is found when 'iso3' isn't present'
* `determine_skip_rows("a1,a2\nb1,b1\n")` did not throw an error.
```

```r
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")
})
```

```
Error: Test failed: 'No row is found when 'iso3' is in the wrong place'
* `determine_skip_rows("stuff,iso3\n")` did not throw an error.
```

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
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 is the roxygen2 header and other modifications needed to make it package-worthy:

```
#' Determine how many rows to skip at the start of a raw maternal data set.
#' This works by finding the first row with `iso3` in the first column.
#'
#' @param src_path path to source file
#'
#' @return the number of rows to skip (or halt if marker 'iso3' not found)
#'
#' @importFrom magrittr %>%
#' @importFrom rlang .data

determine_skip_rows <- function(src_path) {
  data <- readr::read_csv(src_path)
  if (names(data)[1] == "iso3") {
    return(0)
  }
  result <- data %>%
    dplyr::select(check = 1) %>%
    dplyr::mutate(id = dplyr::row_number(.data)) %>%
    dplyr::filter(.data$check == "iso3") %>%
    dplyr::select(.data$id) %>%
    first()
  if (length(result) == 0) {
    stop("No start row found in", src_path)
  }
  result
}
```

Note that this roxygen2 comment block *doesn't* include an `@export` directive, since this function is only going to be used within our project.

The code to find the first and last row of interest looks very similar:

```
determine_first_and_last_row <- function(data) {
  result <- data %>%
    dplyr::mutate(rownum = dplyr::row_number()) %>%
    dplyr::filter(.data$iso3 %in% c("AFG", "ZWE")) %>%
    dplyr::filter(dplyr::row_number() %in% c(1, n())) %>%
    dplyr::select(.data$rownum) %>%
    dplyr::pull(.data$rownum)
  if (length(result) != 2) {
    stop("First or last row missing")
  }
  result
}
```

as does the code to select the region of interest and reformat the numbers as fractions:

```
subsection_maternal_data <- function(raw_data, first_and_last) {
  raw_data %>%
    slice(1:first_and_last[[2, 1]]) %>%
    select(-.data$`Country/areas`, -starts_with("X")) %>%
    purrr::map_dfr(function(x) ifelse(stringr::str_detect(x, "-"), NA, x)) %>%
    dplyr::mutate_at(vars(-c(.data$iso3, .data$Source)), as.numeric) %>%
    dplyr::mutate_at(vars(-c(.data$iso3, .data$year, .data$Source, .data$`Source year`)),
                     function(x) x / 100) %>%
    dplyr::rename(
      total = .data$Total,
      age_15_17 = .data$`age 15-17`,
      age_18_19 = .data$`age 18-19`,
      age_less_than_20 = .data$`age less than 20`,
      age_more_than_20 = .data$`age more than 20`,
      age_20_34 = .data$`age 20-34`,
      age_35_49 = .data$`age 35-49`,
      source = .data$Source,
      source_year = .data$`Source year`
    )
}
```

(We have renamed the columns whose names included spaces and dashes so that they'll be easier for other people to use.) We can now stitch everything together—we omit the roxygen2 header blocks for clarity:

```
tidy_maternal_data <- function(src_path) {
  skip_rows <- determine_skip_rows(src_path)
  data <- readr::read_csv(src_path, skip = skip_rows)
  first_and_last <- determine_first_and_last_row(data)
  subsection_maternal_data(data, first_and_last)
}

regenerate_all_datasets <- function() {
  infant_hiv <- tidy_infant_hiv("raw/infant_hiv.csv")
  at_health_facilities <- tidy_maternal_data("raw/at_health_facilities.csv")
  c_sections <- tidy_maternal_data("raw/c_sections.csv")
  skilled_attendant_at_birth <- tidy_maternal_data("raw/skilled_attendant_at_birth.csv")
  usethis::use_data(
    infant_hiv,
    at_health_facilities,
    c_sections,
    skilled_attendant_at_birth,
    overwrite = TRUE
  )
}
```

12.6 Key Points

- 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 13

Web Applications

13.1 Questions

- How can I build a user interface in R?

13.2 Learning Objectives

- Describe the three essential parts of a Shiny application.
- Explain how element names are used to connect interface elements to server actions.
- Describe the structure of a simple Shiny application and how to run it.
- Explain what reactive variables are and how they differ from normal variables.
- Use functions to create and style HTML elements.
- Explain how to avoid circular updates in interfaces.

Sooner or later, almost every application needs a graphical interface so that users can load data files, control parameters, and view results. While the desktop still has a role to play, the best place to build an interface these days is on the web, and the best toolkit for doing that in R is Shiny. This lesson will walk through the construction of a simple application and along the way introduce you to reactive programming. To follow along, please install Shiny using:

```
install.packages("shiny")
```

```
library(shiny)
```

13.3 How do I set up a simple application?

Every Shiny app has three things:

1. A user interface object that shows things to user.
2. A server function, which is the back end that provides data.
3. A call to `shinyApp` that binds the two together.

These can all live in the same file, but most developers prefer to put the UI and server code in separate files.

To start, we will reproduce the first example from the Shiny tutorials. The first step is to create a directory called `faithful_app`: we *must* do this because every Shiny application needs to be in its own directory. Inside that directory, create a file called `app.R`. The function call `runApp(directory_name)` automatically looks in the named directory for a file with that name.

Inside `app.R`, we can create the skeleton of the application:

```
library(shiny)

ui <- # ...user interface...

server <- # ...server...

shinyApp(ui = ui, server = server)
```

Our next tasks are to fill in the user interface and server.

13.4 How do I create a user interface?

Our interface is a fluid page, i.e., it resizes as needed based on the size of the browser. It uses a single sidebar layout with two elements, `sidebarPanel` and `mainPanel`. The sidebar contains a `sliderInput` object that (as you'd expect from the name) creates a slider; we must specify `label`, `min`, `max`, and `value` to set it up. We must also specify a value called `inputId`; this gives it a name that we can use to refer to it in the server.

Our interface also contains a `mainPanel` object that in turn contains a single `plotOutput` whose `outputId` attribute is used to refer to it. The whole thing looks like this:

```
ui <- fluidPage(
  titlePanel("Hello Shiny!"),
  sidebarLayout(
    sidebarPanel(
      sliderInput(inputId = "bins",
                  label = "Number of bins:",
                  min = 1,
                  max = 50,
                  value = 30)),
    mainPanel(plotOutput(outputId = "distPlot"))
  )
)
```

13.5 How do I create a server?

In any interactive application, something has to react to changes in controls and update displays. Shiny watches for the former and takes care of the latter automatically, but we have to tell it what to watch, what to update, and how to make those updates. We do this by creating a function called `server` that Shiny calls when it needs to:

```
server <- function(input, output) {
  output$distPlot <- renderPlot({
    x    <- faithful$waiting
    bins <- seq(min(x), max(x), length.out = input$bins + 1)
    hist(x, breaks = bins, col = "#75AADB", border = "white",
         xlab = "Waiting time to next eruption (in mins)",
         main = "Histogram of waiting times")
    })
}
```

When there is a change in one of the input elements, Shiny notices and calls our function, giving it inputs (i.e., our controls) and outputs (our displays). For example, `input$bins` matches the `bins` ID for the slider, so the value of `input$bins` will be the value of the slider. Similarly, `output$distPlot` matches the `distPlot` ID of the plot, so we can use Shiny's `renderPlot` function to tell it what to plot. (We can't use ggplot2 calls directly, but the terminology is very similar.) In this case: the `x` axis is waiting times from the `faithful` data, `bins` is the bin labels (we use `input$bins` to get the value), and `hist` is the histogram we want plotted.

13.6 How do I run my application?

We can now run `app.R` from the command line or use:

```
runApp("faithful_app")
```

from inside RStudio. Once the application is running, we can narrow the window to see things automatically resize (the "fluid" part of the interface).

13.7 How can I improve my user interface?

Let's trying building a tool for exploring the UNICEF data we tidied up in earlier lessons. To start, we `mkdir unicef/skeleton` and create `app.R`:

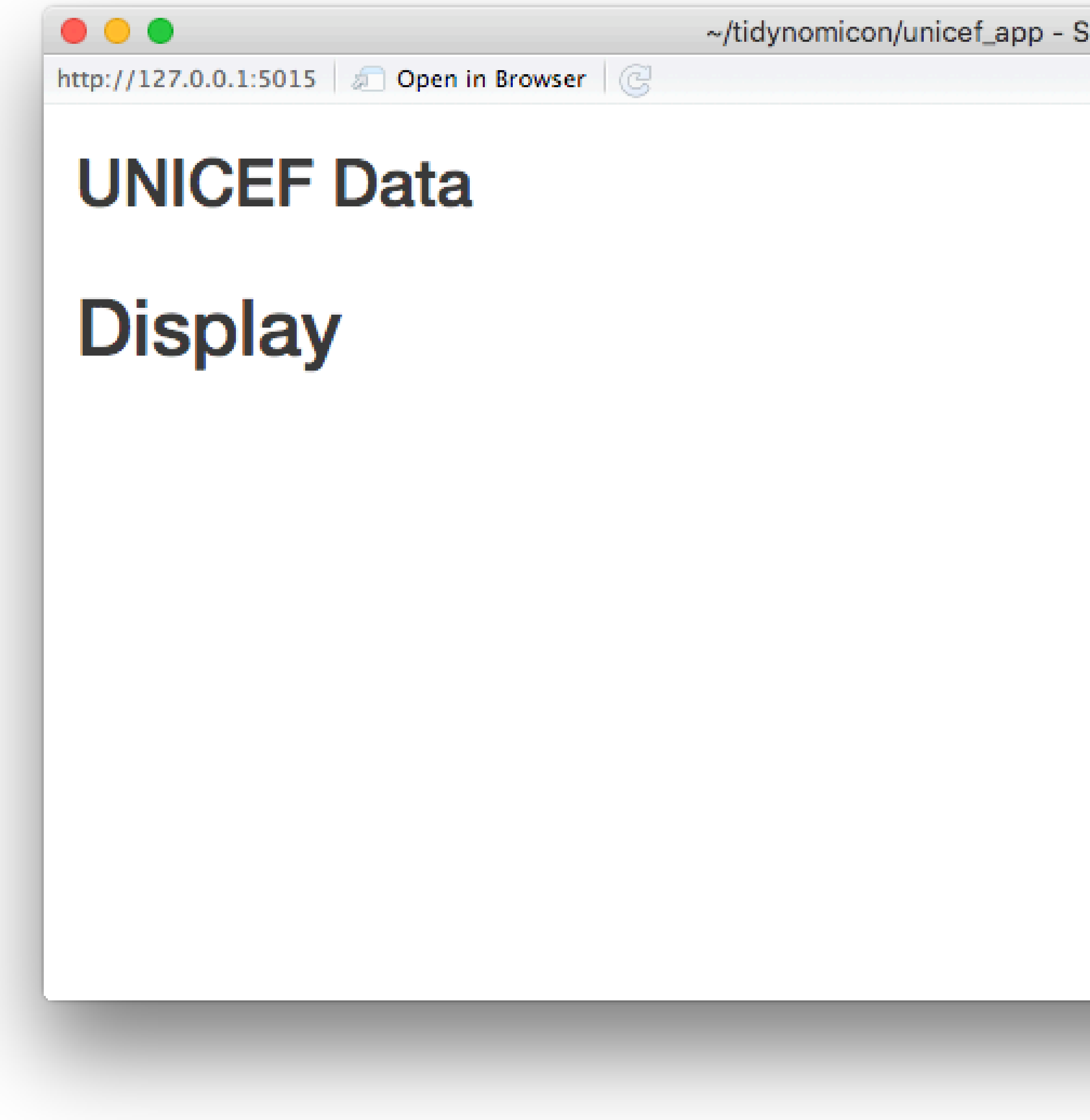
```
library(shiny)

ui <- fluidPage(
  titlePanel("UNICEF Data"),
  sidebarLayout(
    position = "right",
    sidebarPanel(
      img(src = "logo.png", width = 200),
      h2("Controls")
    ),
    mainPanel(h1("Display"))
  )
)

server <- function(input, output){
  # Empty for now.
}

shinyApp(ui = ui, server = server)
```

```
knitr::include_graphics("figures/shiny/unicef-skeleton.png")
```



As the screenshot shows, this positions the controls on the right. We use `h1`, `h2`, and similarly-named functions to create HTML elements and `img` to display a logo. The server is empty for now; when we run it, everything looks good except the image, because it turns out that static images must be in the application's `www` folder, i.e., in `unicef/skeleton/www`.

It's now time to add interactive control elements, better known as widgets. Shiny provides:

- buttons
- checkboxes
- radio buttons
- pulldown selectors (for cases when checkboxes or radio buttons would take up too much space)
- date inputs and date ranges
- filenames
- sliders (like the one seen before)
- free-form text input

That's a lot of options; since we don't have a user to consult, we need to decide what we're going to visualize. One obvious choice is to allow people to choose a data file, and then select a data range based on the years in that file and see a line plot of the average estimate by year. This makes our interface:

```
ui <- fluidPage(
  titlePanel("UNICEF Data"),
  sidebarLayout(
    position = "right",
    sidebarPanel(
      img(src = "logo.png", width = 200),
      fileInput("datafile", p("data file")),
      dateRangeInput("years", p("years"), format = "yyyy")
    ),
    mainPanel(h1("Display"))
  )
)
```

```
knitr::include_graphics("figures/shiny/unicef-prototype.png")
```

Let's show the chosen filename in the output display:

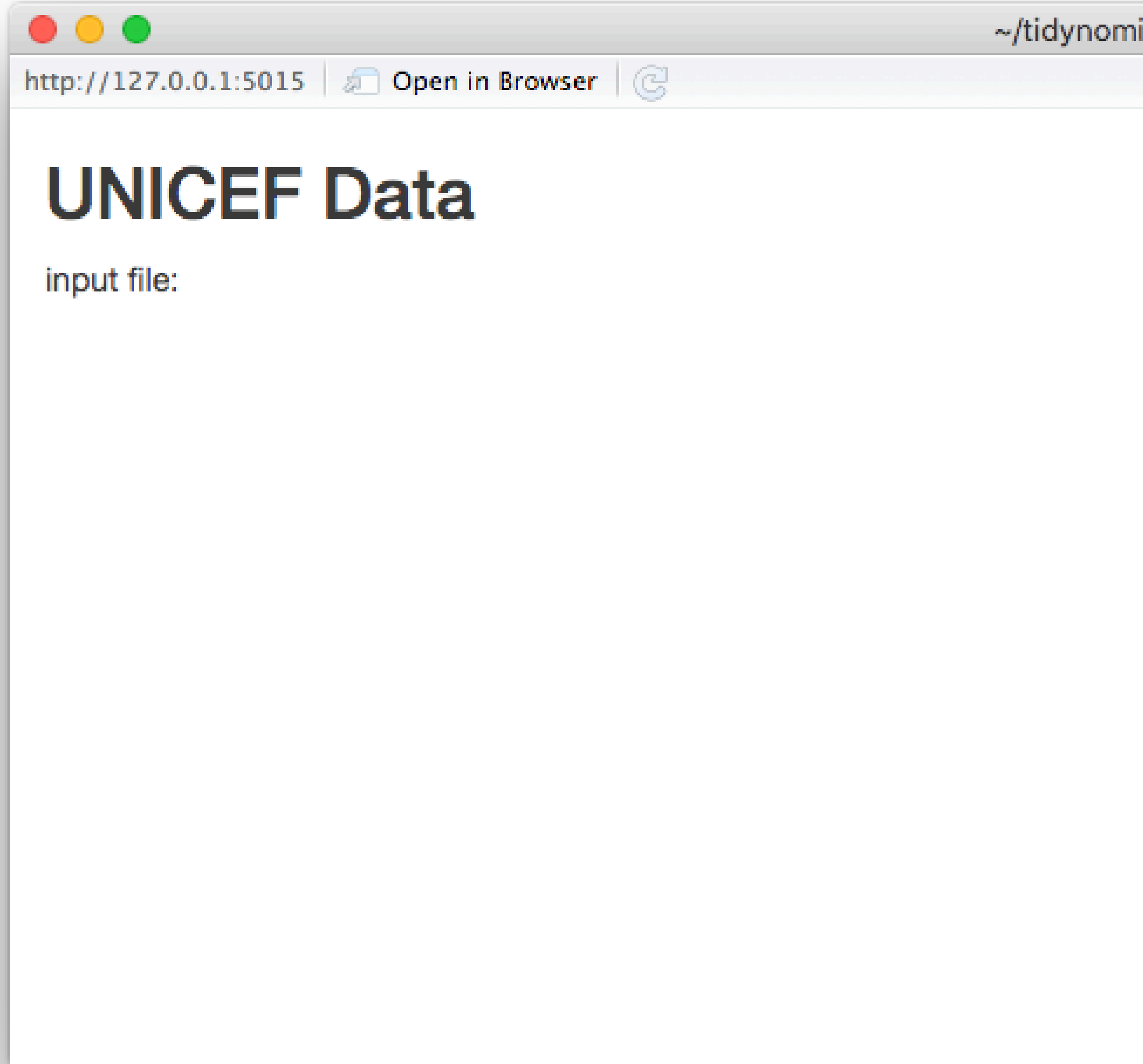
```
ui <- fluidPage(
  titlePanel("UNICEF Data"),
  sidebarLayout(
    # ...as before...
    mainPanel(
      textOutput("filename")
```

```
    )
  )
)

server <- function(input, output){
  output$filename <- renderText({
    paste("input file:", input$datafile)
  })
}
```

The initial display looks good:

```
knitr::include_graphics("figures/shiny/unicef-filename-wrong-before.png")
```



but when we fill in the filename, something is clearly wrong:

```
knitr::include_graphics("figures/shiny/unicef-filename-wrong-after.png")
```

A quick browse of the documentation reveals that `input` is a named list-like object of everything set up in the interface. `input$datafile` picks out one element, but it turns out that's a data frame: what we actually want is `input$datafile$datapath`:

```
server <- function(input, output){
  output$filename <- renderText({
```

```
    paste("input file:", input$datafile$datapath)
  })
}
```

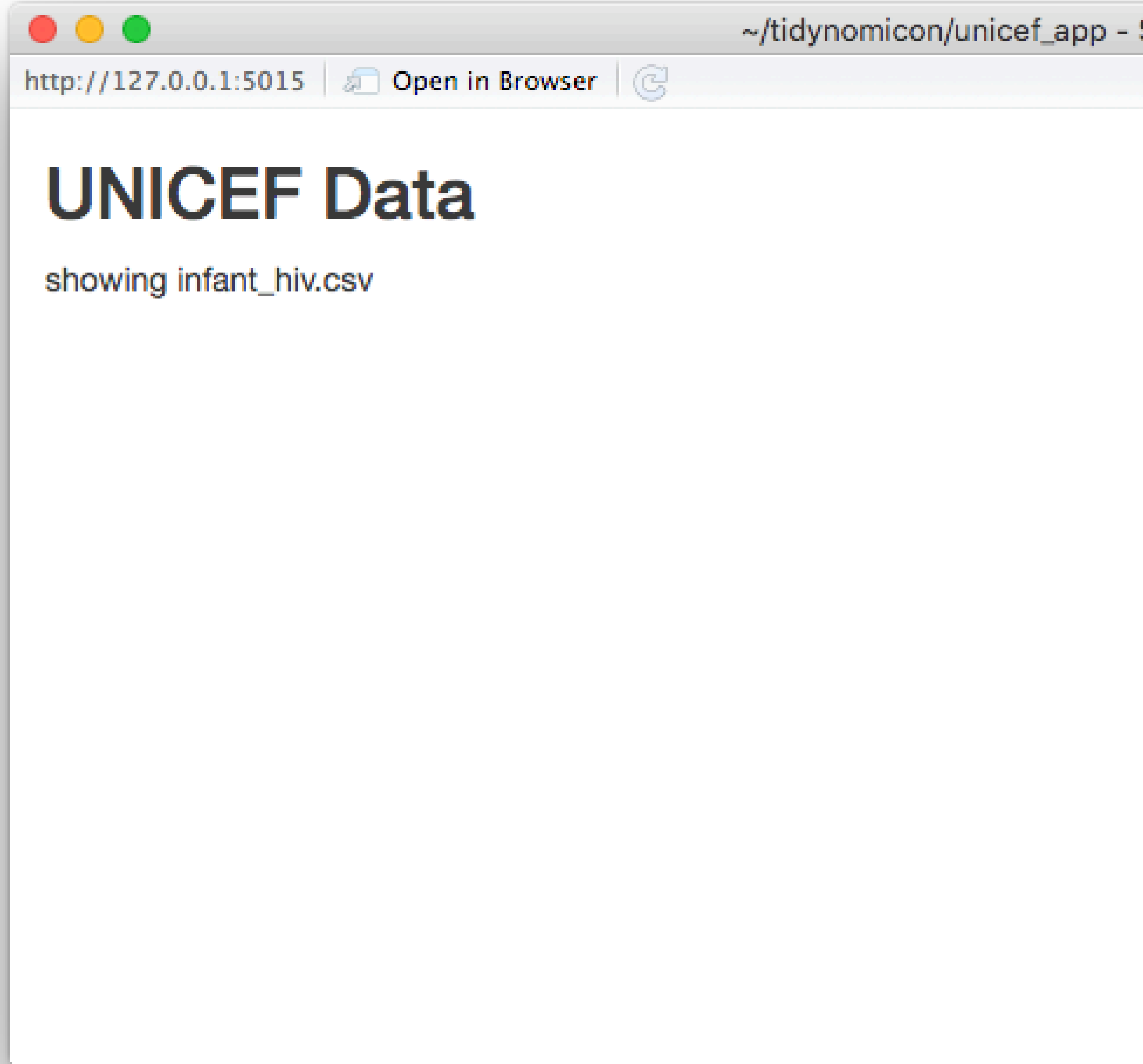
Some more experimentation reveals that we should display `name`, but use `datapath` when reading data. Let's fill in the server a bit:

```
server <- function(input, output){
  currentData <- NULL
  output$filename <- renderText({
    currentName <- input$datafile$name
    currentPath <- input$datafile$datapath
    if (is.null(currentName)) {
      currentData <- NULL
      text <- "no filename set"
    } else {
      currentData <- read_csv(currentPath)
      text <- paste("showing", currentName)
    }
    text
  })
}
```

This short server shows that there are three places we can put variables:

1. At the top of the script outside functions, they are run once when the app launches.
2. Inside `server`, they are run once for each user.
3. Inside a handler like `renderText`, they are run once on each change.

```
knitr::include_graphics("figures/shiny/unicef-filename-right.png")
```



13.8 How can I display the data in a file?

Now comes the hard part: updating the chart when the file changes. The trick is to use a reactive variable, which is actually a function that changes value whenever something it depends on changes. That "something" is usually another reactive, like the ones provided by Shiny.

In the code below, `currentData` is created by calling `reactive` with a block of code that produces the variable's value. It uses `input$datafile`, so it will automatically be triggered whenever `input$datafile` changes. Other things can depend on it in the same way, which allows us to get rid of `currentData`: `output$filename` uses `currentData()`, so it is automatically called when the reactive variable's value changes.

```
server <- function(input, output){
  currentData <- reactive({
    currentPath <- input$datafile$datapath
    if (is.null(currentPath)) {
      result <- NULL
    } else {
      result <- read_csv(currentPath)
    }
    result
  })

  output$filename <- renderText({
    currentName <- input$datafile$name
    if (is.null(currentName)) {
      text <- "no filename set"
    } else {
      text <- paste("showing", currentName)
    }
    text
  })

  output$chart <- renderPlot({
    data <- currentData()
    if (is.null(data)) {
      message("no data")
      chart <- NULL
    } else {
      message("we have data, creating chart")
      chart <- data %>%
        group_by(year) %>%
        summarize(average = mean(estimate, na.rm = TRUE)) %>%
        ggplot() +
        geom_line(mapping = aes(x = year, y = average))
    }
    chart
  })
}
```

13.9 How can I break circular dependencies?

Now comes the *other* hard part: handling changes to the date range. We want the chart to display data for the selected range of years and have the minimum and maximum possible year set by the data. That means we have to change something in the user interface from the server; to do that, we add a third parameter `session` to the `server` function. This variable holds the backward connection from the server to the UI.

Inside our server, we get the current years from `input$years` and use `updateDateRangeInput` to push a change from the output function to the input controls. (This is the part that needs `session`.)

```
server <- function(input, output, session){
  # ...other code as before...
```

```
  output$chart <- renderPlot({
    years <- input$years
    message('years', years)
    data <- currentData()
    if (is.null(data)) {
      chart <- NULL
    } else {
      minYear <- as.character(min(data$year))
      maxYear <- as.character(max(data$year))
      updateDateRangeInput(session, "years", min = minYear, max = maxYear,
                           start = minYear, end = maxYear)
      chart <- data %>%
        group_by(year) %>%
        summarize(average = mean(estimate, na.rm = TRUE)) %>%
        ggplot() +
        geom_line(mapping = aes(x = year, y = average)) +
        ggtitle(paste("Years", minYear, "-", maxYear))
    }
    chart
  })
}
```

When we run this, it displays the current date twice on startup before a file is selected because that's the default for the date input. Once dates are entered, though, it goes into an infinite loop because ghe chart depends on the dates, but we're changing the dates inside the plot update.

Let's try again. We will just read `years` inside the chart update and display it:

```
  output$chart <- renderPlot({
    years <- input$years
    message('years', years)
    data <- currentData()
    if (is.null(data)) {
      chart <- NULL
    } else {
      minYear <- as.character(min(data$year))
      maxYear <- as.character(max(data$year))
      chart <- data %>%
        group_by(year) %>%
        summarize(average = mean(estimate, na.rm = TRUE)) %>%
        ggplot() +
        geom_line(mapping = aes(x = year, y = average)) +
        ggtitle(paste("Years", minYear, "-", maxYear))
    }
    chart
  })
```

Whoops: the message appears every time a character is typed in one of the date controls, i.e., deleting the start year and typing 2, 0, 1, 8 produces `0002`, `0020`, and `0201` before producing a usable year. That's clearly not what we want, so we'll try a third approach and only show the year selector when there's data. While we're doing this, we'll change the year selector to a double-ended slider, because seeing the day and month is misleading. The revised UI code looks like this:

```
ui <- fluidPage(
  titlePanel("UNICEF Data"),
```

```
  sidebarLayout(
    position = "right",
    sidebarPanel(
      img(src = "logo.png", width = 200),
      div(
        id = "datafileInput",
        fileInput("datafile", p("data file"))
      )
    ),
    mainPanel(
      p(textOutput("filename")),
      plotOutput("chart")
    )
  )
)
```

Here, we have wrapped the file selector in a `div` so that we have a named element after which to insert our date range selector, but *haven't* included the date range selector (yet).

The outline of the corresponding server is:

```
server <- function(input, output){

  currentData <- reactive({
    # ...provide currentData...
  })

  selectedData <- reactive({
    # ...provide selectedData...
  })

  observeEvent(input$datafile, {
    # ...insert year selector when datafile changes...
  })

  output$chart <- renderPlot({
    # ...update chart when selectedData changes...
  })

  output$filename <- renderText({
    # ...update displayed filename when selected file changes...
  })
}
```

The zero'th change is getting rid of the `session` variable: we don't need it any longer because we're not modifying the interface from the server. The first change is to create a reactive variable for the selected data. We need this because the chart depends on the selected data, while the range of years we can select depends on the current (actual) data. As a rule, everywhere we might need to "see" data, we should create a reactive variable.

The function `observeEvent` allows us to create event handlers that aren't directly attached to display objects; we need one so that we can create the year display. Once that's set up, `currentData` is straightforward: if the filename changes, load that CSV file:

```
  currentData <- reactive({
    read_csv(input$datafile$datapath)
```

```
})
```

`selectedData` is also straightforward: if `currentData` changes, filter by year range:

```
selectedData <- reactive({
  req(input$years)
  currentData() %>%
    filter(between(year, input$years[1], input$years[2]))
})
```

This function uses `currentData()` so that Shiny knows it depends on changes to the current data. But how do we know we *have* a year range? The answer is that `req(input$years)` means "make sure this thing exists before going any further". Once we have the years, we can filter as required.

Now for the clever bit: we will create a slider *after* loading a data file. More specifically, we will use `observeEvent(input$datafile, {...})` to indicate that this action depends on changes to the filename, then get the current data, grab the year range, create a `sliderInput`, and use `insertUI` to add it after the `div` we created:

```
observeEvent(input$datafile, {
  current <- currentData()
  lowYear <- min(current$year)
  highYear <- max(current$year)
  insertUI(
    selector = "#datafileInput",
    where = "afterEnd",
    ui = sliderInput("years", "years",
                     min = lowYear,
                     max = highYear,
                     value = c(lowYear, highYear),
                     sep = "")
  )
})
```

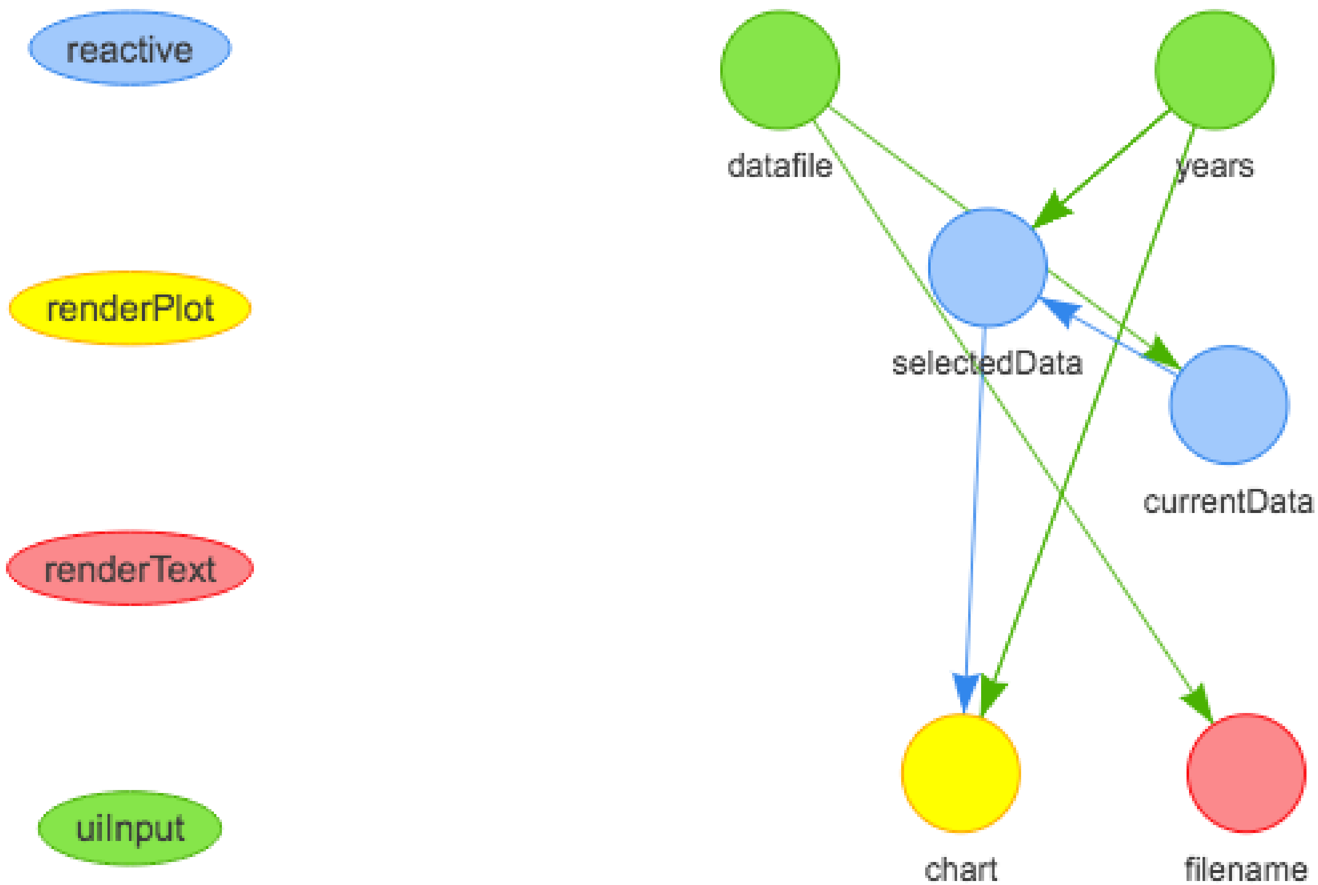
Creating the chart and displaying the filename is done as before, though we have switched to `ifelse` for the filename's value to be idiomatic. Note that the chart depends on `selectedData()` and *not* the raw data:

```
output$chart <- renderPlot({
  selectedData() %>%
    group_by(year) %>%
    summarize(average = mean(estimate, na.rm = TRUE)) %>%
      ggplot() +
      geom_line(mapping = aes(x = year, y = average)) +
      labs(title = paste("Years", input$years[1], "to", input$years[2]))
})

output$filename <- renderText({
  currentName <- input$datafile$name
  ifelse(is.null(currentName), "no filename set", paste("showing", currentName))
})
```

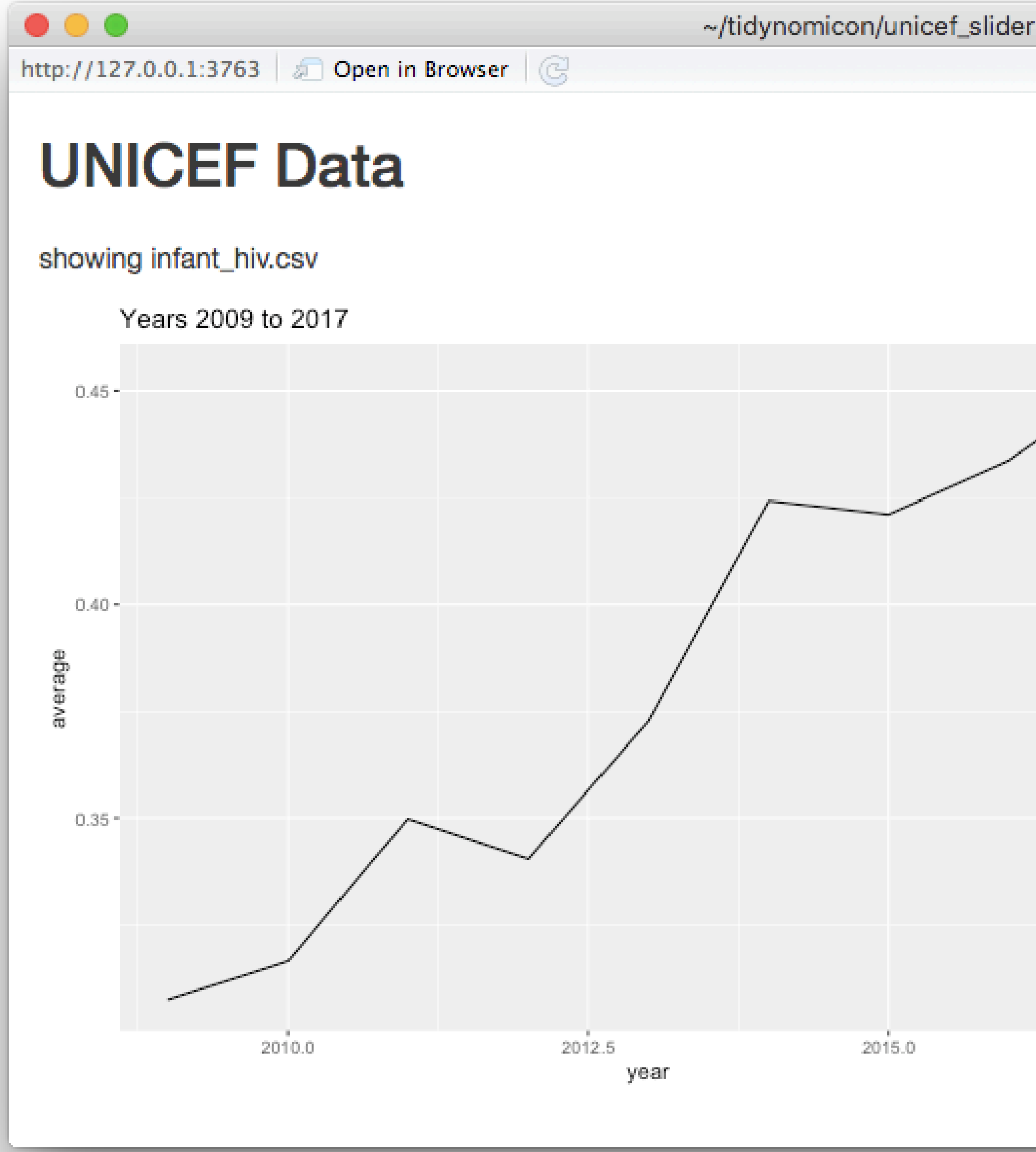
Here are the dependencies we have created:

```r
knitr::include_graphics("figures/shiny/unicef-slider-dependencies.png")
```



It works! Except that we're adding a slider every time we open a file, so if we open the same file twice, we get two sliders with identical ranges:

```r
knitr::include_graphics("figures/shiny/unicef-slider.png")
```


~/tidynomicon/unicef_slider
http://127.0.0.1:3763
Open in Browser
UNICEF Data
showing infant_hiv.csv
Years 2009 to 2017
0.45
0.40
0.35
average
2010.0
2012.5
2015.0
year

13.10 How can I control how the user interface is rendered?

Our last refinement uses `uiOutput` and `renderUI` to (re-)create the slider at just the right moment. The UI looks familiar, except there's a `uiOutput` placeholder where the slider is to go—its name "slider" will be used in the server:

```
ui <- fluidPage(
  titlePanel("UNICEF Data"),
  sidebarLayout(
    position = "right",
    sidebarPanel(
      img(src = "logo.png", width = 200),
      fileInput("datafile", p("data file")),
      uiOutput("slider")
    ),
    mainPanel(
      p(textOutput("filename")),
      plotOutput("chart")
    )
  )
)
```

`uiOutput` is always used in conjunction with `renderUI` in the server, so let's look at the server:

```
server <- function(input, output){

  currentData <- reactive({
    # ...get the data...
  })

  output$slider <- renderUI({
    # ...create a widget to allow year selection...
  })

  selectedData <- reactive({
    # ...select data using values from the year selector...
  })

  output$chart <- renderPlot({
    # ...draw the chart...
  })

  output$filename <- renderText({
    # ...display the filename...
  })
}
```

What does `currentData` look like?

```
  currentData <- reactive({
    req(input$datafile)
    read_csv(input$datafile$datapath)
  })
```

We use `req(...)` to tell Shiny that there's no point proceeding unless `input$datafile` actually has a value, because we can't load data if we have a `NULL` filename.

Once we have data, we create a slider or overwrite the existing slider if there already is one; this prevents the problem of multiple sliders. Note that the slider's ID is `"years"` and that its range is set based on data, so we avoid the problem of having to create a slider when we don't know what its range should be:

```
output$slider <- renderUI({
  current <- currentData()
  lowYear <- min(current$year)
  highYear <- max(current$year)

  sliderInput("years", "years",
                    min = lowYear,
                    max = highYear,
                    value = c(lowYear, highYear),
                    sep = "")
})
```

Once we have a slider, we select the data; this depends on the years from the slider, so we make that explicit using `req`:

```
selectedData <- reactive({
  req(input$years)

  currentData() %>%
    filter(between(year, input$years[1], input$years[2]))
})
```

Displaying the chart and the filename are exactly as we've seen before: the chart depends on `selectedData` and the filename display depends on `input$datafile$name`.

So how and why does this all work?

1. When the UI is initially created:
    - There is no data file, so `req(input$datafile)` in the definition of `currentData` halts.
    - Without `currentData`, the `renderUI` call used to create the slider doesn't proceed.
    - So the UI doesn't get a slider and doesn't try to display data it doesn't have.
2. When a filename is selected for the first time:
    - `input$datafile` gets a value.
    - So we load data *and* we can display the filename.
    - `currentData` now has a meaningful value.
    - So we can create a slider *and* initialize its limits to the min and max years from the actual data.
    - So `selectedData` can now be constructed (all of the things it depends on exist).
    - So we can draw the chart.
3. When a new file is selected:
    - `input$datafile` gets a value.
    - So we load data and display the filenmae.
    - `currentData` is then re-created.
    - So we replace the slider with a new one whose bounds are set by the new data.
    - And then construct `selectedData` and draw the chart.

This isn't the only way to build our interface, but the alternatives use more advanced functions like `freeze` and are harder to debug. The way to get where we want to is to break everything down into single actions, each of which is probably smaller than we might first expect. It takes a bit of practice, but once you're used to it, you'll be able to build some powerful tools with just a page or two of code.

13.11 Key Points

- Every Shiny application has a user interface, a server, and a call to `shinyApp` that connects them.

- Every Shiny application must be in its own directory.
- Images and other static assets must be in that directory's `www` sub-directory.
- The `inputId` and `outputId` attributes of UI elements are used to refer to them from the server.
- Use `input$name` and `output$name` in the server to refer to UI elements.
- Code placed at the top of the script outside functions is run once when the app launches.
- Code placed inside `server` is run once for each user.
- Code placed inside a handler is run once on each change.
- A reactive variable is a function whose value changes automatically whenever anything it depends on changes.
- Use `reactive({...})` to create a reactive variable explicitly.
- The server can change UI elements via the `session` variable.
- Use `uiOutput` and `renderUI` to (re-)create UI elements as needed in order to break circular dependencies.

Chapter 14

Using Python

14.1 Questions

- How can I use Python and R together?

14.2 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.

As the previous lessons have shown, you can do a lot with R, but sometimes you might feel a cold, serpentine tug on your soul pulling you back to Python. You can put Python code in RMarkdown 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
"/Users/gvwilson/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.

14.3 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('tidy/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 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.

14.4 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] 0.384182
```

(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('tidy/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.

14.5 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.

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

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.

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.

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.

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.

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 F

Learning Objectives

F.1 Values and Vectors

- 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.

F.2 Indexing

- 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.

F.3 Control Flow

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

F.4 The Tidyverse

- 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.
- Name, describe, and use a tidyverse function for choosing rows by value from tabular data.
- Name, describe, and use a tidyverse function for reordering rows of tabular data.
- Name, describe, and use a tidyverse function for selecting columns of tabular data.
- Name, describe, and use a tidyverse function for calculating new columns from existing ones.
- Name, describe, and use a tidyverse function for grouping rows of tabular data.
- Name, describe, and use a tidyverse function for aggregating grouped or ungrouped rows of tabular data.

F.5 Cleaning Up Data

- 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.

F.6 Non-Standard Evaluation

- 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.

F.7 Handling Errors

- Name and describe the three levels of error handling in R.
- Handle an otherwise-fatal error in a function call in R.

F.8 Object-Oriented Programming

- 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.

F.9 Intellectual Debt

- 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.

F.10 Projects

- 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.

F.11 Testing

- Create unit tests in R.
- Create unit tests for an R package.

F.12 Web Applications with Shiny

- Describe the three essential parts of a Shiny application.
- Explain how element names are used to connect interface elements to server actions.
- Describe the structure of a simple Shiny application and how to run it.
- Explain what reactive variables are and how they differ from normal variables.
- Use functions to create and style HTML elements.
- Explain how to avoid circular updates in interfaces.

F.13 Reticulate

- 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.

Appendix G

Key Points

G.1 Values and Vectors

- 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.

G.2 Indexing

- 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.

G.3 Control Flow

- 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.4 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.5 Cleaning Up Data

- 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.

G.6 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.7 Handling Errors

- 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.

G.8 Object-Oriented Programming

- 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).

G.9 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.10 Projects

- 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.11 Testing

- 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.12 Web Applications with Shiny

- Every Shiny application has a user interface, a server, and a call to `shinyApp` that connects them.
- Every Shiny application must be in its own directory.
- Images and other static assets must be in that directory's `www` sub-directory.
- The `inputId` and `outputId` attributes of UI elements are used to refer to them from the server.
- Use `input$name` and `output$name` in the server to refer to UI elements.
- Code placed at the top of the script outside functions is run once when the app launches.
- Code placed inside `server` is run once for each user.
- Code placed inside a handler is run once on each change.
- A reactive variable is a function whose value changes automatically whenever anything it depends on changes.
- Use `reactive({...})` to create a reactive variable explicitly.
- The server can change UI elements via the `session` variable.

- Use `uiOutput` and `renderUI` to (re-)create UI elements as needed in order to break circular dependencies.

G.13 Reticulate

- 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.

Bibliography

Wilson, G. (2018). *Teaching Tech Together*. Lulu.com. How to create and deliver lessons that work and build a teaching community around them.