# III. Program

# Chapter 17: Introduction

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

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

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

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

In pipes, you will dive deep into the pipe, %>%, and learn more about how it works, what the alternatives are, and when not to use it.

Copy-and-paste is a powerful tool, but you should avoid doing it more than twice. Repeating yourself in code is dangerous because it can easily lead to errors and inconsistencies. Instead, in functions, you’ll learn how to write functions which let you extract out repeated code so that it can be easily reused.

As you start to write more powerful functions, you’ll need a solid grounding in R’s data structures, provided by vectors. You must master the four common atomic vectors, the three important S3 classes built on top of them, and understand the mysteries of the list and data frame.

Functions extract out repeated code, but you often need to repeat the same actions on different inputs. You need tools for iteration that let you do similar things again and again. These tools include for loops and functional programming, which you’ll learn about in iteration.

## 17.1 For further study

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

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

Hands on Programming with R, by Garrett Grolemund. This is an introduction to R as a programming language and is a great place to start if R is your first programming language. It covers similar material to these chapters, but with a different style and different motivation examples (based in the casino). It’s a useful complement if you find that these four chapters go by too quickly.

Advanced R by Hadley Wickham. This dives into the details of R the programming language. This is a great place to start if you have existing programming experience. It’s also a great next step once you’ve internalised the ideas in these chapters. You can read it online at <http://adv-r.had.co.nz>.

# Chapter 18 Pipes

## 18.1 Introduction

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

### 17.1.1 Prerequisites

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

library(tidyverse)

## Warning: package 'tidyverse' was built under R version 3.5.1

## -- Attaching packages ------------------- tidyverse 1.2.1 --

## √ ggplot2 3.0.0 √ purrr 0.2.5  
## √ tibble 1.4.2 √ dplyr 0.7.6  
## √ tidyr 0.8.1 √ stringr 1.3.1  
## √ readr 1.1.1 √ forcats 0.3.0

## Warning: package 'ggplot2' was built under R version 3.5.1

## Warning: package 'purrr' was built under R version 3.5.1

## Warning: package 'dplyr' was built under R version 3.5.1

## -- Conflicts ---------------------- tidyverse\_conflicts() --  
## x dplyr::filter() masks stats::filter()  
## x dplyr::lag() masks stats::lag()

library(purrr)  
library(magrittr)

##   
## Attaching package: 'magrittr'

## The following object is masked from 'package:purrr':  
##   
## set\_names

## The following object is masked from 'package:tidyr':  
##   
## extract

# install.packages("pryr")  
library(pryr)

##   
## Attaching package: 'pryr'

## The following objects are masked from 'package:purrr':  
##   
## compose, partial

## 18.2 Piping alternatives

The point of the pipe is to help you write code in a way that is easier to read and understand. To see why the pipe is so useful, we’re going to explore a number of ways of writing the same code. Let’s use code to tell a story about a little bunny named Foo Foo: >Little bunny Foo Foo Went hopping through the forest Scooping up the field mice And bopping them on the head

This is a popular Children’s poem that is accompanied by hand actions.We’ll start by defining an object to represent little bunny Foo Foo:

# foo\_foo <- little\_bunny()

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

1. Save each intermediate step as a new object.
2. Overwrite the original object many times.
3. Compose functions.
4. Use the pipe.

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

### 18.2.1 Intermediate steps

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

# foo\_foo\_1 <- hop(foo\_foo,through=forest)  
# foo\_foo\_2 <- scoop(foo\_foo\_1, up = field\_mice)  
# foo\_foo\_3 <- bop(foo\_foo\_2, on = head)

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

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

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

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

diamonds <- ggplot2::diamonds  
diamonds2 <- diamonds %>%   
 dplyr::mutate(price\_per\_carat=price/carat)  
  
pryr::object\_size(diamonds)

## 3.46 MB

pryr::object\_size(diamonds2)

## 3.89 MB

pryr::object\_size(diamonds,diamonds2)

## 3.89 MB

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

* diamonds takes up 3.46 MB,
* diamonds2 takes up 3.89 MB,
* diamonds and diamonds2 together take up 3.89 MB!

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

diamonds$carat[1] <- NA  
pryr::object\_size(diamonds)

## 3.46 MB

pryr::object\_size(diamonds2)

## 3.89 MB

pryr::object\_size(diamonds,diamonds2)

## 4.32 MB

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

### 18.2.2 Overwrite the original

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

# foo\_foo <- hop(foo\_foo, through = forest)  
# foo\_foo <- scoop(foo\_foo, up = field\_mice)  
# foo\_foo <- bop(foo\_foo, on = head)

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

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

### 18.2.3 Function composition

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

# bop(  
# scoop(  
# hop(foo\_foo, through = forest),  
# up = field\_mice  
# ),   
# on = head  
# )

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

### 18.2.4 Use the pipe

Finally, we can use the pipe:

# foo\_foo %>%  
# hop(through = forest) %>%  
# scoop(up = field\_mice) %>%  
# bop(on = head)

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

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

# my\_pipe <- function(.) {  
# . <- hop(., through = forest)  
# . <- scoop(., up = field\_mice)  
# bop(., on = head)  
# }  
# my\_pipe(foo\_foo)

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

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

assign("x",10)  
x

## [1] 10

"x" %>% assign(100)  
x

## [1] 10

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

env <- environment()  
"x" %>% assign(100,envir=env)  
x

## [1] 100

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

1. Functions that use lazy evaluation. In R, function arguments are only computed when the function uses them, not prior to calling the function. The pipe computes each element in turn, so you can’t rely on this behaviour.

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

# tryCatch(stop("!"),error=function(e)"An error")  
#   
# stop("!") %>%   
# tryCatch(error = function(e) "An error")  
# # Error in eval(lhs, parent, parent) : !

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

## 18.3 When not to use the pipe

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

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

## 18.4 Other tools from magrittr

All packages in the tidyverse automatically make %>% available for you, so you don’t normally load magrittr explicitly. However, there are some other useful tools inside magrittr that you might want to try out:

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

To work around this problem, you can use the “tee” pipe. %T>% works like %>% except that it returns the left-hand side instead of the right-hand side. It’s called “tee” because it’s like a literal T-shaped pipe.

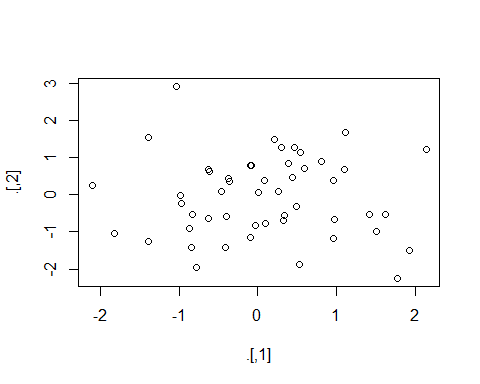
## 18.4 Other tools from magrittr

All packages in the tidyverse automatically make %>% available for you, so you don’t normally load magrittr explicitly. However, there are some other useful tools inside magrittr that you might want to try out:

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

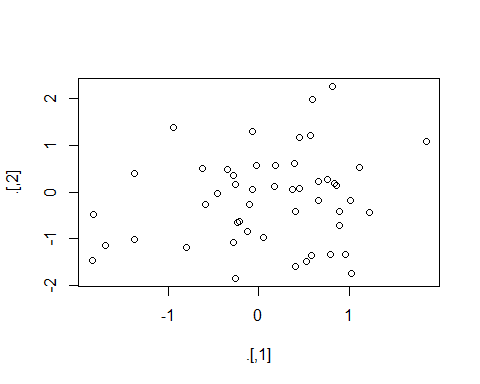
To work around this problem, you can use the “tee” pipe. %T>% works like %>% except that it returns the left-hand side instead of the right-hand side. It’s called “tee” because it’s like a literal T-shaped pipe.

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



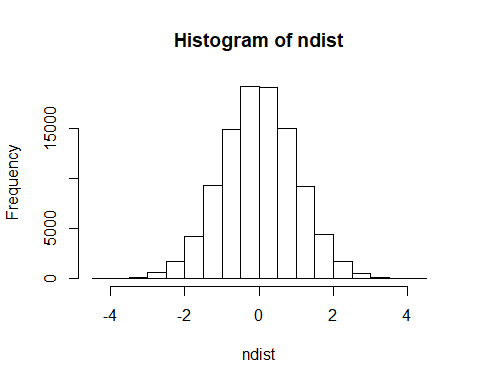
## NULL

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



## NULL

ndist <- rnorm(100000)  
hist(ndist)



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

mtcars %$%  
 cor(disp, mpg)

## [1] -0.8475514

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

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

with

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

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

# Chapter 19 Functions

## 19.1 Introduction

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

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

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

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

## 19.1.1 Prerequisites

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

## 19.2 When should you write a function?

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

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

## # A tibble: 10 x 4  
## a b c d  
## <dbl> <dbl> <dbl> <dbl>  
## 1 -0.930 0.506 -0.235 -0.287   
## 2 0.461 -0.348 -1.75 -0.791   
## 3 1.04 2.10 -0.651 0.835   
## 4 2.55 1.25 1.65 -0.236   
## 5 -0.0253 -2.51 0.628 -0.978   
## 6 0.851 -0.271 -0.159 -0.948   
## 7 -0.163 1.02 0.951 -0.335   
## 8 -0.458 -0.0271 1.05 -0.0824  
## 9 2.37 1.19 0.368 -0.554   
## 10 -0.318 -0.170 -1.13 1.76

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

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

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

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

## [1] 0.0000000 0.4001581 0.5657082 1.0000000 0.2602358 0.5122901 0.2207014  
## [8] 0.1357276 0.9493551 0.1760415

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

x <- df$a  
(x - min(x, na.rm = T)) / (max(x, na.rm = T)-min(x, na.rm = T))

## [1] 0.0000000 0.4001581 0.5657082 1.0000000 0.2602358 0.5122901 0.2207014  
## [8] 0.1357276 0.9493551 0.1760415

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

rng <- range(x, na.rm = T)  
(x-rng[1])/(rng[2]-rng[1])

## [1] 0.0000000 0.4001581 0.5657082 1.0000000 0.2602358 0.5122901 0.2207014  
## [8] 0.1357276 0.9493551 0.1760415

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

rescale01 <- function(x){  
 rng <- range(x, na.rm = T)  
 (x-rng[1])/(rng[2]-rng[1])  
}  
rescale01(c(0,5,10))

## [1] 0.0 0.5 1.0

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

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

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

rescale01(c(-10,0,10))

## [1] 0.0 0.5 1.0

rescale01(c(1,2,3,NA,5))

## [1] 0.00 0.25 0.50 NA 1.00

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

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

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

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

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

x <- c(1:10,Inf)  
rescale01(x)

## [1] 0 0 0 0 0 0 0 0 0 0 NaN

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

rescale01 <- function(x){  
 rng <- range(x,na.rm=T,finite=T)  
 (x-rng[1])/(rng[2]-rng[1])  
}  
rescale01(x)

## [1] 0.0000000 0.1111111 0.2222222 0.3333333 0.4444444 0.5555556 0.6666667  
## [8] 0.7777778 0.8888889 1.0000000 Inf

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

### 19.2.1 Practice

1. Why is TRUE not a parameter to rescale01()? What would happen if x contained a single missing value, and na.rm was FALSE?

rescale01 <- function(x){  
 rng <- range(x,na.rm=F)  
 (x-rng[1])/(rng[2]-rng[1])  
}  
rescale01(c(-1,0,5,20,NA))

## [1] NA NA NA NA NA

Everything is NA! We should set an argument to control the TRUE or FALSE of range.

1. In the second variant of rescale01(), infinite values are left unchanged. Rewrite rescale01() so that -Inf is mapped to 0, and Inf is mapped to 1.

rescale01 <- function(x){  
 rng <- range(x,na.rm=T,finite=T)  
 (x-rng[1])/(rng[2]-rng[1])  
 x[x==Inf]<-1  
 x[x==-Inf] <- 0  
 x  
}  
  
rescale01(c(Inf,-Inf,0:5,NA))

## [1] 1 0 0 1 2 3 4 5 NA

1. Practice turning the following code snippets into functions. Think about what each function does. What would you call it? How many arguments does it need? Can you rewrite it to be more expressive or less duplicative?

mean(is.na(x))

## [1] 0

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

## [1] 0 0 0 0 0 0 0 0 0 0 NaN

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

## [1] NaN

prop\_miss <- function(x){  
 mean(is.na(x))  
}  
  
my\_mean <- function(x){  
 x/sum(x,na.rm=T)  
}  
  
my\_var <- function(x){  
 sd(x,na.rm=T)/mean(x,na.rm=T)  
}

1. Follow <http://nicercode.github.io/intro/writing-functions.html> to write your own functions to compute the variance and skew of a numeric vector.
2. Write both\_na(), a function that takes two vectors of the same length and returns the number of positions that have an NA in both vectors.

both\_na <- function(x,y){  
 stopifnot(length(x)==length(y))  
 which(is.na(x) & is.na(y))  
}  
both\_na(c(1,2,NA,2,NA),c(1,2,3,4,NA))

## [1] 5

1. What do the following functions do? Why are they useful even though they are so short?

is\_directory <- function(x) file.info(x)$isdir  
is\_readable <- function(x) file.access(x, 4) == 0

## 19.3 Functions are for humans and computers

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

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

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

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

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

# Good  
# input\_select()  
# input\_checkbox()  
# input\_text()

A good example of this design is the stringr package: if you don’t remember exactly which function you need, you can type str\_ and jog your memory.

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

## 19.4 Conditional execution

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

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

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

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

has\_name <- function(x){  
 nms <- names(x)  
 if(is.null(nms)){  
 rep(FALSE,length(x))  
 }else {  
 !is.na(nms) & nms !=""  
 }  
}

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

### 19.4.1 Conditions

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

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

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

identical(0L,0)

## [1] FALSE

x <- sqrt(2)^2  
x==2

## [1] FALSE

x-2

## [1] 4.440892e-16

Instead use dplyr::near() for comparisons, as described in comparisons.

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

### 19.4.2 Multiple conditions

You can chain multiple if statement together:

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

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

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

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

### 19.4.3 Code style

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

An opening curly brace should never go on its own line and should always be followed by a new line. A closing curly brace should always go on its own line, unless it’s followed by else. Always indent the code inside curly braces.

# Good  
# if (y < 0 && debug) {  
# message("Y is negative")  
# }  
#   
# if (y == 0) {  
# log(x)  
# } else {  
# y ^ x  
# }  
#   
# # Bad  
# if (y < 0 && debug)  
# message("Y is negative")  
#   
# if (y == 0) {  
# log(x)  
# }   
# else {  
# y ^ x  
# }

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

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

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

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

## 19.5 Function arguments

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

* In log(), the data is x, and the detail is the base of the logarithm.
* In mean(), the data is x, and the details are how much data to trim from the ends (trim) and how to handle missing values (na.rm).
* In t.test(), the data are x and y, and the details of the test are alternative, mu, paired, var.equal, and conf.level.
* In str\_c() you can supply any number of strings to …, and the details of the concatenation are controlled by sep and collapse.

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

# Compute confidence interval around mean using normal approximation  
mean\_ci <- function(x, conf = 0.95) {  
 se <- sd(x) / sqrt(length(x))  
 alpha <- 1 - conf  
 mean(x) + se \* qnorm(c(alpha / 2, 1 - alpha / 2))  
}  
  
x <- runif(100)  
mean\_ci(x)

## [1] 0.4415317 0.5604717

#> [1] 0.498 0.610  
mean\_ci(x, conf = 0.99)

## [1] 0.4228449 0.5791586

#> [1] 0.480 0.628

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

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

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

## [1] 5.5

# Bad  
mean(x = 1:10, , FALSE)

## [1] 5.5

mean(, TRUE, x = c(1:10, NA))

## [1] 5.5

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

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

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

### 19.5.1 Choosing names

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

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

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

### 19.5.2 Cheking values

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

wt\_mean <- function(x, w) {  
 sum(x \* w) / sum(w)  
}  
wt\_var <- function(x, w) {  
 mu <- wt\_mean(x, w)  
 sum(w \* (x - mu) ^ 2) / sum(w)  
}  
wt\_sd <- function(x, w) {  
 sqrt(wt\_var(x, w))  
}

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

wt\_mean(1:6, 1:3)

## [1] 7.666667

#> [1] 7.67

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

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

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

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

wt\_mean <- function(x, w, na.rm = FALSE) {  
 if (!is.logical(na.rm)) {  
 stop("`na.rm` must be logical")  
 }  
 if (length(na.rm) != 1) {  
 stop("`na.rm` must be length 1")  
 }  
 if (length(x) != length(w)) {  
 stop("`x` and `w` must be the same length", call. = FALSE)  
 }  
   
 if (na.rm) {  
 miss <- is.na(x) | is.na(w)  
 x <- x[!miss]  
 w <- w[!miss]  
 }  
 sum(w \* x) / sum(w)  
}

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

wt\_mean <- function(x, w, na.rm = FALSE) {  
 stopifnot(is.logical(na.rm), length(na.rm) == 1)  
 stopifnot(length(x) == length(w))  
   
 if (na.rm) {  
 miss <- is.na(x) | is.na(w)  
 x <- x[!miss]  
 w <- w[!miss]  
 }  
 sum(w \* x) / sum(w)  
}  
# wt\_mean(1:6, 6:1, na.rm = "foo")  
#> Error in wt\_mean(1:6, 6:1, na.rm = "foo"): is.logical(na.rm) is not TRUE

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

### 19.5.3 Dot-dot-dot(…)

Many functions in R take an arbitrary number of inputs:

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

## [1] 55

#> [1] 55  
stringr::str\_c("a", "b", "c", "d", "e", "f")

## [1] "abcdef"

#> [1] "abcdef"

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

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

commas <- function(...) stringr::str\_c(..., collapse = ", ")  
commas(letters[1:10])

## [1] "a, b, c, d, e, f, g, h, i, j"

#> [1] "a, b, c, d, e, f, g, h, i, j"  
  
rule <- function(..., pad = "-") {  
 title <- paste0(...)  
 width <- getOption("width") - nchar(title) - 5  
 cat(title, " ", stringr::str\_dup(pad, width), "\n", sep = "")  
}  
rule("Important output")

## Important output ------------------------------------------------------

#> Important output ------------------------------------------------------

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

x <- c(1,2)  
sum(x,na.rm=T)

## [1] 3

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

### 19.5.4 Lazy evaluation

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

### 19.5.5 Exercises

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

## 19.6 Return values

Figuring outwhat your function should is usually straightfoward: it’s why you created the function in the first place! There are two things you should consider when returning a value: 1. Does returning early make your function easier to read? 2. Can you make your function pipeable?

### 19.6.1 Explicit return statements

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

complicated\_function <- function(x,y,z){  
 if (lenth(x)==0 || length(y)==0){  
 return(0)  
 }  
 # completated code is here  
}

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

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

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

f <- function() {  
 if (!x) {  
 return(something\_short)  
 }  
  
 # Do   
 # something  
 # that  
 # takes  
 # many  
 # lines  
 # to  
 # express  
}

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

### 19.6.2 Writing pipeable functions

If you want to write your own pipeable functions, it’s important to think about the return value. Knowing the return value’s object type will mean that your pipeline will “just work”. For example, with dplyr and tidyr the object type is the data frame.

There are two basic types of pipeable functions: transformations and side-effects. With transformations, an object is passed to the function’s first argument and a modified object is returned. With side-effects, the passed object is not transformed. Instead, the function performs an action on the object, like drawing a plot or saving a file. Side-effects functions should “invisibly” return the first argument, so that while they’re not printed they can still be used in a pipeline. For example, this simple function prints the number of missing values in a data frame:

show\_missing <- function(df){  
 n <- sum(is.na(df))  
 cat("Missing values:",n,"\n",sep="")  
   
 invisible(df)  
}

If we call it interatively, the invisible() means that the input df does not get printed out:

show\_missing(mtcars)

## Missing values:0

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

x <- show\_missing(mtcars)

## Missing values:0

class(x)

## [1] "data.frame"

dim(x)

## [1] 32 11

And we can still use it in a pipe:

library(magrittr)  
library(tidyverse)  
  
mtcars %>%   
 show\_missing() %>%   
 mutate(mpg=ifelse(mpg<20,NA,mpg)) %>%   
 show\_missing()

## Missing values:0  
## Missing values:18

## 19.7 Environment

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

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

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

y <- 100  
f(10)

## [1] 110

y <- 1000  
f(10)

## [1] 1010

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

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

`+` <- function(x, y) {  
 if (runif(1) < 0.1) {  
 sum(x, y)  
 } else {  
 sum(x, y) \* 1.1  
 }  
}  
table(replicate(1000, 1 + 2))

##   
## 3 3.3   
## 129 871

#>   
#> 3 3.3   
#> 100 900  
rm(`+`)

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

# Chapter 20: Vectors

## 20.1 Introduction

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

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

### 20.1.1 PRerequisites

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

library(tidyverse)

## 20.2 Vector basics

There are two types of vectors: 1. Atomic vectors, of which there are six types: logical, integer, double, character, complex, and raw. Integer and double vectors are collectively known as numeric vectors. 2. Lists, which are sometimes called recursive vectors because lists can contain other lists.

The chief difference between atomic vectors and lists is that atomic vectors are homogeneous, while lists can be heterogeneous. There’s one other related object: NULL. NULL is often used to represent the absence of a vector (as opposed to NA which is used to represent the absence of a value in a vector). NULL typically behaves like a vector of length 0. Figure 20.1 summarises the interrelationships.

Every vector has two key properties: 1. Its type, which you can determine with typeof().

typeof(letters)

## [1] "character"

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

## [1] "integer"

#> [1] "integer"

1. Its length, which you can determine with length()

x <- list("a","b",1:10)  
length(x)

## [1] 3

x

## [[1]]  
## [1] "a"  
##   
## [[2]]  
## [1] "b"  
##   
## [[3]]  
## [1] 1 2 3 4 5 6 7 8 9 10

Vectors can also contain arbitrary additional metadata in the form of attributes. These attributes are used to create augmented vectors which build on additional behaviour. There are three important types of augmented vector: - Factors are built on top of integer vectors. - Dates and date-times are built on top of numeric vectors. - Data frames and tibbles are built on top of lists.

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

## 20.3 Important types of atomic vector

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

### 20.3.1 Logical

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

1:10 %% 3 ==0

## [1] FALSE FALSE TRUE FALSE FALSE TRUE FALSE FALSE TRUE FALSE

c(T,T,F,NA)

## [1] TRUE TRUE FALSE NA

### 20.3.2 Numeric

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

typeof(1)

## [1] "double"

typeof(1L)

## [1] "integer"

1.5

## [1] 1.5

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

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

x <- sqrt(2)^2  
x

## [1] 2

x-2

## [1] 4.440892e-16

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

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

c(-1,0,1)%/% 0

## [1] -Inf NaN Inf

# [1] -Inf NaN Inf

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

### 20.3.3 Character

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

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

x <- "This is a reasonably long string."  
pryr::object\_size(x)

## 152 B

y <- rep(x,1000)  
pryr::object\_size(y)

## 8.14 kB

y doesn’t take up 1,000x as much memory as x, because each element of y is just a pointer to that same string. A pointer is 8 bytes, so 1000 pointers to a 136 B string is 8 \* 1000 + 136 = 8.13 kB.

### 20.3.4 Missing values

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

NA # logical

## [1] NA

#> [1] NA  
NA\_integer\_ # integer

## [1] NA

#> [1] NA  
NA\_real\_ # double

## [1] NA

#> [1] NA  
NA\_character\_ # character

## [1] NA

#> [1] NA

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

### 20.3.5 Exercises

**1. Describe the difference between is.finite(x) and !is.infinite(x).**  
There is no difference! is.finite() tests whether a value is **not** -Inf or Inf and is.infinite tests whether it is -Inf or Inf. Logically, ! negates the previous.

However… is.infinite NA and NaN to be non infinite! See the answer to ex 19.3.1 here.

**2. Read the source code for dplyr::near() (Hint: to see the source code, drop the ()). How does it work?**  
It calculates absolute difference up to a certain tolerance level. If the difference is 0.2 it calculates the difference and compares it to the tolerance level.

**3. A logical vector can take 3 possible values. How many possible values can an integer vector take? How many possible values can a double take? Use google to do some research.** Taken from: <https://jrnold.github.io/r4ds-exercise-solutions/vectors.html>

**4. Brainstorm at least four functions that allow you to convert a double to an integer. How do they differ? Be precise.**

x <- seq(-10,10,by=0.5)  
as.integer(x) # turns to a literal integer

## [1] -10 -9 -9 -8 -8 -7 -7 -6 -6 -5 -5 -4 -4 -3 -3 -2 -2  
## [18] -1 -1 0 0 0 1 1 2 2 3 3 4 4 5 5 6 6  
## [35] 7 7 8 8 9 9 10

floor(x) # rounds down

## [1] -10 -10 -9 -9 -8 -8 -7 -7 -6 -6 -5 -5 -4 -4 -3 -3 -2  
## [18] -2 -1 -1 0 0 1 1 2 2 3 3 4 4 5 5 6 6  
## [35] 7 7 8 8 9 9 10

`%/%`(x,1) # extracts the integer part of x

## [1] -10 -10 -9 -9 -8 -8 -7 -7 -6 -6 -5 -5 -4 -4 -3 -3 -2  
## [18] -2 -1 -1 0 0 1 1 2 2 3 3 4 4 5 5 6 6  
## [35] 7 7 8 8 9 9 10

**5. What functions from the readr package allow you to turn a string into logical, integer, and double vector?**

library(tidyverse)  
x <- c("TRUE", "FALSE")  
parse\_logical(x)

## [1] TRUE FALSE

x <- c("1", "2")  
parse\_integer(x)

## [1] 1 2

x <- c("1.1", "1.2")  
parse\_double(x)

## [1] 1.1 1.2

## 20.4 Using atomic vectors

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

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

### 20.4.1 Coercion

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

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

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

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

x <- sample(20,100,replace=T)  
y <- x > 10  
sum(y) # how many are greater than 10?

## [1] 53

mean(y) # what proportion are greater than 10?

## [1] 0.53

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

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

## NULL

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

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

typeof(c(TRUE,1L))

## [1] "integer"

typeof(c(1L,1.5))

## [1] "double"

typeof(c(1.5,"a"))

## [1] "character"

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

### 20.4.2 Test functions

Sometimes you want to do different things based on the type of vector. One option is to use typeof(). Another is to use a test function which returns a TRUE or FALSE. Base R provides many functions line is.vector() and is.atomic(), but they often returns surprising results. Instead, it’s safer to use the is\_\* functions provided by purrr, which are summarised in the table below.

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

### 20.4.3 Scalars and recicling rules

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

This is generally most useful when you are mixing vectors and “scalars”. I put scalars in quotes because R doesn’t actually have scalars: instead, a single number is a vector of length 1. Because there are no scalars, most built-in functions are vectorised, meaning that they will operate on a vector of numbers. That’s why, for example, this code works:

sample(10)+100

## [1] 107 106 102 108 101 104 105 109 110 103

runif(10)>0.5

## [1] TRUE FALSE FALSE FALSE TRUE FALSE TRUE FALSE FALSE FALSE

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

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

1:10 +1:2

## [1] 2 4 4 6 6 8 8 10 10 12

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

1:10+1:3

## Warning in 1:10 + 1:3: 長いオブジェクトの長さが短いオブジェクトの長さの倍数  
## になっていません

## [1] 2 4 6 5 7 9 8 10 12 11

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

library(tidyverse)  
  
# tibble(  
# x=1:4,  
# y=1:2  
# )  
  
tibble(  
 x=1:4,  
 y=rep(1:2,each=2)  
)

## # A tibble: 4 x 2  
## x y  
## <int> <int>  
## 1 1 1  
## 2 2 1  
## 3 3 2  
## 4 4 2

#### 20.4.4 Naming vectors

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

c(x=1,y=2,z=4)

## x y z   
## 1 2 4

Or after the fact with purr::set\_names()

set\_names(1:3,c("a","b","c"))

## a b c   
## 1 2 3

Named vectors are most useful for subsetting, described next.

### 20.4.5 Subsetting

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

1. A numeric vector containing only integers. The integers must either be all positive, all negative, or zero. Subsetting with positive integers keeps the elements at those positions:

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

## [1] "three" "two" "five"

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

x[c(1,1,5,5,5,2)]

## [1] "one" "one" "five" "five" "five" "two"

Negative values drop the elements at the specified positions:

x[c(-1,-3,-5)]

## [1] "two" "four"

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

# x[c(1,-1)]

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

x[0]

## character(0)

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

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

library(tidyverse)  
x <- c(10,3,NA,5,8,1)  
  
# tibble test  
x <- as.tibble(x,ncol=1)  
names(x)="v1"  
is.na(x)

## v1  
## [1,] FALSE  
## [2,] FALSE  
## [3,] TRUE  
## [4,] FALSE  
## [5,] FALSE  
## [6,] FALSE

x %>% filter(v1 == NA)

## # A tibble: 0 x 1  
## # ... with 1 variable: v1 <dbl>

# all non-missing values of x  
x <- c(10,3,NA,5,8,1)  
x[!is.na(x)]

## [1] 10 3 5 8 1

# all even (or missing) values of x  
x[x %% 2==0]

## [1] 10 NA 8

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

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

## xyz def   
## 5 2

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

1. The simplest type of subsetting is nothing, x[], which returns the complete x. This is not useful for subsetting vectors, but it is useful when subsetting matrices (and other high dimensional) structures because it lets you select the rows or all the columns, by leaving that index blank. For example, if x is 2d, x[1, ] selects the first row and all the columns, and x[, -1] selects all rows and all columns except the first.

To learn more about the applications of subsetting, reading the “Subsetting” chapter of Advanced R: <http://adv-r.had.co.nz/Subsetting.html#applications>.

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

### 20.4.6 Exercises

**1. What does mean(is.na(x)) tell you about a vector x? What about sum(!is.finite(x))?**

mean(is.na(x)) gives the proportion or percentage of missing values in that vector sum(!is.finite(x)) gives the total number of infinite objects in x

**2. Carefully read the documentation of is.vector(). What does it actually test for? Why does is.atomic() not agree with the definition of atomic vectors above?**

s.atomic() tests for objects that area “logical”, “integer”, “numeric” (synonym “double”), “complex”, “character” and “raw”. is.vector() on the other hand tests for any of the atomic modes but must have no attributes other than names.

A factor, for example, shows how they contradict.

x <- factor()  
is.vector(x)

## [1] FALSE

is.atomic(x)

## [1] TRUE

**3. Compare and contrast setNames() with purrr::set\_names().**

The do the same thing but purrr::set\_names but with stricter argument checking and cool argument to rename variable based on a function.

**4. Create functions that take a vector as input and returns:**

* The last value. Should you use [ or [[?

[[ because it want the specific values. x might have other attributes such as names and it is asking for one value.

returner <- function(x){  
 x[[length(x)]]  
}  
returner(1:10)

## [1] 10

* The elements at even numbered positions.

even\_num <- function(x){  
 x[seq(2,length(x),by=2)]  
}  
even\_num(1:10)

## [1] 2 4 6 8 10

* Every element except the last value.

except\_last <- function(x){  
 x[-length(x)]  
}  
except\_last(1:10)

## [1] 1 2 3 4 5 6 7 8 9

* Only even numbers (and no missing values).

even\_numbers <- function(x){  
 x[!is.na(x) & x%%2==0]  
}  
even\_numbers(sample(c(NA,1:100),99))

## [1] 52 38 78 44 34 2 28 98 70 40 50 72 56 66 82 36 12  
## [18] 80 10 20 18 84 94 46 92 30 16 86 96 22 54 76 90 62  
## [35] 64 74 32 42 48 100 26 24 60 14 8 88 4 6 58

**5. Why is x[-which(x > 0)] not the same as x[x <= 0]?**

x[-which(x > 0)] subset the values of x which ar at 0 or below x[x <= 0] subset the values of x which ar at 0 or below.

They are the same.

However, which will ignore NA’s and leave them as is and <= will turn any value that cannot be comparable to NA like NaN!

x <- c(-5:5, Inf, -Inf, NaN, NA)  
  
x[-which(x > 0)]

## [1] -5 -4 -3 -2 -1 0 -Inf NaN NA

x[x <= 0]

## [1] -5 -4 -3 -2 -1 0 -Inf NA NA

**6. What happens when you subset with a positive integer that’s bigger than the length of the vector? What happens when you subset with a name that doesn’t exist?**

x <- 1:10  
x[11]

## [1] NA

# Returns an NA  
x <- c("a" = 1)  
x["b"]

## <NA>   
## NA

# Both the name and element is NA.

## 20.5 Recursive vectors (lists)

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

x <- list(1,2,3)  
x

## [[1]]  
## [1] 1  
##   
## [[2]]  
## [1] 2  
##   
## [[3]]  
## [1] 3

A very useful tool for working with lists is str() because it focuses on the structure, not the contents.

str(x)

## List of 3  
## $ : num 1  
## $ : num 2  
## $ : num 3

x\_named <- list(a=1,b=2,c=3)  
str(x\_named)

## List of 3  
## $ a: num 1  
## $ b: num 2  
## $ c: num 3

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

y <- list("a",1L,1.5,T)  
str(y)

## List of 4  
## $ : chr "a"  
## $ : int 1  
## $ : num 1.5  
## $ : logi TRUE

List can even contain other lists!

z <- list(list(1,2),list(3,4))  
str(z)

## List of 2  
## $ :List of 2  
## ..$ : num 1  
## ..$ : num 2  
## $ :List of 2  
## ..$ : num 3  
## ..$ : num 4

### 20.5.1 Visualizing lists

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

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

## [[1]]  
## [1] 1 2  
##   
## [[2]]  
## [1] 3 4

x2

## [[1]]  
## [[1]][[1]]  
## [1] 1  
##   
## [[1]][[2]]  
## [1] 2  
##   
##   
## [[2]]  
## [[2]][[1]]  
## [1] 3  
##   
## [[2]][[2]]  
## [1] 4

x3

## [[1]]  
## [1] 1  
##   
## [[2]]  
## [[2]][[1]]  
## [1] 2  
##   
## [[2]][[2]]  
## [[2]][[2]][[1]]  
## [1] 3

There are three principles:

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

### 20.5.2 Subsetting

There are three ways ti subset a list, which I’ll illustrate with a list named a:

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

* [ extracts a sub-list. The result will always be a list.

str(a)

## List of 4  
## $ a: int [1:3] 1 2 3  
## $ b: chr "a string"  
## $ c: num 3.14  
## $ d:List of 2  
## ..$ : num -1  
## ..$ : num -5

str(a[1:2])

## List of 2  
## $ a: int [1:3] 1 2 3  
## $ b: chr "a string"

str(a[4])

## List of 1  
## $ d:List of 2  
## ..$ : num -1  
## ..$ : num -5

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

* [[ extracts a single component from a list. It removes a level of hierarchy from the list.

str(a[[1]])

## int [1:3] 1 2 3

str(a[[4]])

## List of 2  
## $ : num -1  
## $ : num -5

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

a$a

## [1] 1 2 3

a[["a"]]

## [1] 1 2 3

The distinction between [ and [[ is really important for lists, because [[ drills down into the list while [ returns a new, smaller list. Compare the code and output above with the visual representation in Figure 20.2.

### 20.5.3 Lists of condiments

The difference between [ and [[ is very important, but it’s easy to get confused. To help you remeber, let me show you an unusual peppr shaker.

### 20.5.4 Exercies

1. Draw the following lists as nested sets:

# list(a, b, list(c, d), list(e, f))  
# list(list(list(list(list(list(a))))))

1. What happens if you subset a tibble as if you’re subsetting a list? What are the key differences between a list and a tibble?

x <- as\_tibble(mtcars)  
x[1] # gives back a tibble (same as lists)

## # A tibble: 32 x 1  
## mpg  
## <dbl>  
## 1 21   
## 2 21   
## 3 22.8  
## 4 21.4  
## 5 18.7  
## 6 18.1  
## 7 14.3  
## 8 24.4  
## 9 22.8  
## 10 19.2  
## # ... with 22 more rows

x[[1]] # gives back a vector (same as lists)

## [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2  
## [15] 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4  
## [29] 15.8 19.7 15.0 21.4

# In fact, data frames and tibbles are lists!!

## 20.6 Attributes

Any vector can contain arbitrary additional metadata through its attributes. You can think of attributes as named list of vectors that can be attached to any object. You can get and set individual attribute values with attr() or see them all at once with attributes().

x <- 1:10  
attr(x,"greeting")

## NULL

attr(x,"greeting") <- "Hi!"  
attr(x,"farewell") <- "Bye!"  
attributes(x)

## $greeting  
## [1] "Hi!"  
##   
## $farewell  
## [1] "Bye!"

There are three very important attributes that are used to inplement fundamental parts of R: 1. **Names** are used to name the elements of a vector. 2. **Dimensions** (dims, for short) make a vector behave like a matrix or array. 3. **Class** is used to implement the S3 object oriented system.

You’ve seen names above, and we won’t cover dimensions because we don’t use matrices in this book. It remains to describe the class, which controls how generic functions work. Generic functions are key to object oriented programming in R, because they make functions behave differently for different classes of input. A detailed discussion of object oriented programming is beyond the scope of this book, but you can read more about it in Advanced R at <http://adv-r.had.co.nz/OO-essentials.html#s3>.

Here is what a typical generic function looks like:

as.Date

## function (x, ...)   
## UseMethod("as.Date")  
## <bytecode: 0x000000000ad45228>  
## <environment: namespace:base>

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

methods("as.Date")

## [1] as.Date.character as.Date.default as.Date.factor as.Date.numeric   
## [5] as.Date.POSIXct as.Date.POSIXlt   
## see '?methods' for accessing help and source code

For example, if x is a character vector, as.Date() will call as.Date.character(); if it’s a factor, it’ll call as.Date.factor().

You can see the specific implementation of a method with getS3method()

getS3method("as.Date","default")

## function (x, ...)   
## {  
## if (inherits(x, "Date"))   
## x  
## else if (is.logical(x) && all(is.na(x)))   
## .Date(as.numeric(x))  
## else stop(gettextf("do not know how to convert '%s' to class %s",   
## deparse(substitute(x)), dQuote("Date")), domain = NA)  
## }  
## <bytecode: 0x000000000b06cbb0>  
## <environment: namespace:base>

getS3method("as.Date","numeric")

## function (x, origin, ...)   
## {  
## if (missing(origin))   
## stop("'origin' must be supplied")  
## as.Date(origin, ...) + x  
## }  
## <bytecode: 0x000000000b06eb78>  
## <environment: namespace:base>

The most important S3 generic is print(): it controls how the object is printed when you type its name at the console. Other important generics are the subsetting functions [, [[, and $.

## 20.7 Augumented vectors

Atomic vectors and lists are the building blocks for other important vector types like factors and dates. I call these **augmented vectors**, because they are vectors with additional **attributes**, including class. Because augmented vectors have a class, they behave differently to the atomic vector on which they are built. In this book, we make use of four important augmented vectors: - Factors - Dates - Date-times - Tibbles

These are described below.

### 20.7.1 Factors

Factors are designed to represent categorical data that can take a fixed set of possible values. Factors are built on top of integers, and have a levels attribute:

x <- factor(c("ab","cd","ab"),levels=c("ab","cd","ed"))  
typeof(x)

## [1] "integer"

attributes(x)

## $levels  
## [1] "ab" "cd" "ed"  
##   
## $class  
## [1] "factor"

### 20.7.2 Dates and date-times

Dates in R are numeric vectors that represent the number of days since 1 January 1970.

x <- as.Date("1971-01-01")  
unclass(x)

## [1] 365

typeof(x)

## [1] "double"

attributes(x)

## $class  
## [1] "Date"

Date-times are numeric vectors with class POSIXct that represent the number of seconds since 1 January 1970. (In case you were wondering, “POSIXct” stands for “Portable Operating System Interface”, calendar time.)

x <- lubridate::ymd\_hm("1970-01-01 01:00")  
unclass(x)

## [1] 3600  
## attr(,"tzone")  
## [1] "UTC"

typeof(x)

## [1] "double"

attributes(x)

## $class  
## [1] "POSIXct" "POSIXt"   
##   
## $tzone  
## [1] "UTC"

The tzone attribute is optional. It controls how the time is printed, not what absolute time it refers to.

attr(x,"tzone") <- "US/Pacific"  
x

## [1] "1969-12-31 17:00:00 PST"

attr(x,"tzone") <- "US/Eastern"  
x

## [1] "1969-12-31 20:00:00 EST"

There is another type of date-times called POSIXIt. There are built on top of named lists:

y <- as.POSIXlt(x)  
typeof(y)

## [1] "list"

#> [1] "list"  
attributes(y)

## $names  
## [1] "sec" "min" "hour" "mday" "mon" "year" "wday"   
## [8] "yday" "isdst" "zone" "gmtoff"  
##   
## $class  
## [1] "POSIXlt" "POSIXt"   
##   
## $tzone  
## [1] "US/Eastern" "EST" "EDT"

POSIXlts are rare inside the tidyverse. They do crop up in base R, because they are needed to extract specific components of a date, like the year or month. Since lubridate provides helpers for you to do this instead, you don’t need them. POSIXct’s are always easier to work with, so if you find you have a POSIXlt, you should always convert it to a regular data time lubridate::as\_date\_time().

### 20.7.3 Tibbles

Tibbles are augmented lists: they have class “tbl\_df” + “tbl” + “data.frame”, and names (column) and row.names attributes:

tb <- tibble::tibble(x = 1:5, y = 5:1)  
typeof(tb)

## [1] "list"

#> [1] "list"  
attributes(tb)

## $names  
## [1] "x" "y"  
##   
## $row.names  
## [1] 1 2 3 4 5  
##   
## $class  
## [1] "tbl\_df" "tbl" "data.frame"

#> $names  
#> [1] "x" "y"  
#>   
#> $row.names  
#> [1] 1 2 3 4 5  
#>   
#> $class  
#> [1] "tbl\_df" "tbl" "data.frame"

The difference between a tibble and a list is that all the elements of a data frame must be vectors with the same length. All functions that work with tibbles enforce this constraint.

Traditional data.frames have a very similar structure:

df <- data.frame(x = 1:5, y = 5:1)  
typeof(df)

## [1] "list"

#> [1] "list"  
attributes(df)

## $names  
## [1] "x" "y"  
##   
## $class  
## [1] "data.frame"  
##   
## $row.names  
## [1] 1 2 3 4 5

#> $names  
#> [1] "x" "y"  
#>   
#> $class  
#> [1] "data.frame"  
#>   
#> $row.names  
#> [1] 1 2 3 4 5

The main difference is the class. The class of tibble includes “data.frame” which means tibbles inherit the regular data frame behaviour by default.

### 20.7.4 Exercises

1. What does hms::hms(3600) return? How does it print? What primitive type is the augmented vector built on top of? What attributes does it use?

x <- hms::hms(3600)  
  
class(x) # augmented classes

## [1] "hms" "difftime"

typeof(x)# a double, in fact, it is 3600 seconds if we use as.numeric

## [1] "double"

1. Try and make a tibble that has columns with different lengths. What happens?

# The length of all columns must be the same or 1 (scalar), otherwise throws an error.  
# tibble::tibble(a=1:2,b=1:3)

1. Based on the definition above, is it ok to have a list as a column of a tibble? Yes, as long as it has the same length as other columns. But beware, because to ‘open’ that column all slots need to have the same class.

# Chapter 21: Iteration

## 21.1 Introduction

In functions, we talked about how important it is to reduce duplication in your code by creating functions instead of copying-and-pasting. Reducing code duplication has three main benefits:

1. It’s easier to see the intent of your code, because your eyes are drawn to what’s different, not what stays the same.
2. It’s easier to respond to changes in requirements. As your needs change, you only need to make changes in one place, rather than remembering to change every place that you copied-and-pasted the code.
3. You’re likely to have fewer bugs because each line of code is used in more places.

One tool for reducing duplication is functions, which reduce duplication by identifying repeated patterns of code and extract them out into independent pieces that can be easily reused and updated. Another tool for reducing duplication is iteration, which helps you when you need to do the same thing to multiple inputs: repeating the same operation on different columns, or on different datasets. In this chapter you’ll learn about two important iteration paradigms: imperative programming and functional programming. On the imperative side you have tools like for loops and while loops, which are a great place to start because they make iteration very explicit, so it’s obvious what’s happening. However, for loops are quite verbose, and require quite a bit of bookkeeping code that is duplicated for every for loop. Functional programming (FP) offers tools to extract out this duplicated code, so each common for loop pattern gets its own function. Once you master the vocabulary of FP, you can solve many common iteration problems with less code, more ease, and fewer errors.

### 21.1.1 Prerequisites

Once you’ve mastered the for loops provided by base R, you’ll learn some of the powerful programming tools provided by purrr, one of the tidyverse core packages.

library(tidyverse)

## 21.2 For loops

Imagine we have this simple tibble:

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

We want to compute the median of each column. You could do with copy-and-paste.

median(df$a)

## [1] 0.134828

median(df$b)

## [1] 0.553326

median(df$c)

## [1] -0.190996

median(df$d)

## [1] -0.2542251

But that breaks our rule of thumb: never copy and paste more than twice. Instead, we could use a for loop:

df

## # A tibble: 10 x 4  
## a b c d  
## <dbl> <dbl> <dbl> <dbl>  
## 1 0.897 0.845 0.490 -0.376  
## 2 2.38 -0.389 -0.131 1.85   
## 3 1.11 0.203 0.270 0.678  
## 4 -0.497 -0.660 -0.251 -0.432  
## 5 0.362 0.951 0.577 -0.132  
## 6 -1.26 0.899 -1.18 0.378  
## 7 -1.67 1.58 -0.286 0.667  
## 8 -1.47 -0.983 -0.972 -0.427  
## 9 0.643 0.262 1.50 -1.35   
## 10 -0.0920 1.19 -0.679 -1.03

output <- vector("double",ncol(df))  
for (i in seq\_along(df)){  
 output[[i]] <- median(df[[i]])  
}  
output <- tibble(output)

Every for loop has three components 1. The output: output <- vector("double", length(x)). Before you start the loop, you must always allocate sufficient space for the output. This is very important for efficiency: if you grow the for loop at each iteration using c() (for example), your for loop will be very slow.

A general way of creating an empty vector of given length is the vector() function. It has two arguments: the type of the vector (“logical”, “integer”, “double”, “character”, etc) and the length of the vector.

1. The sequence: i in seq\_along(df). This determines what to loop over: each run of the for loop will assign i to a different value from seq\_along(df). It’s useful to think of i as a pronoun, like “it”.

You might not have seen seq\_along() before. It’s a safe version of the familiar 1:length(l), with an important difference: if you have a zero-length vector, seq\_along() does the right thing:

y <- vector("double", 0)  
seq\_along(y)

## integer(0)

#> integer(0)  
1:length(y)

## [1] 1 0

#> [1] 1 0

You probably won’t create a zero-length vector deliberately, but it’s easy to create them accidentally. If you use1:length(x) instead of seq\_along(x), you’re likely to get a confusing error message.

1. The body: output[[i]] <- median(df[[i]]). This is the code that does the work. It’s run repeatedly, each time with a different value for i. The first iteration will run output[[1]] <- median(df[[1]]), the second will run output[[2]] <- median(df[[2]]), and so on.

That’s all there is to the for loop! Now is a good time to practice creating some basic (and not so basic) for loops using the exercises below. Then we’ll move on some variations of the for loop that help you solve other problems that will crop up in practice.

### 21.2.1 Exercises

1. Write for loops to:

* Compute the mean of every column in mtcars.
* Determine the type of each column in nycflights13::flights.
* Compute the number of unique values in each column of iris.
* Generate 10 random normals for each of . Think about the output, sequence, and body before you start writing the loop.

looper <- function(the\_type,object,fun){  
 empty\_vec=vector(the\_type,length(object))  
   
 for(every\_column in seq\_along(object)){  
 empty\_vec[every\_column] <- fun(object[[every\_column]])  
 }  
 empty\_vec  
}  
  
# mtcars case  
looper("numeric",mtcars,mean)

## [1] 20.090625 24.750000 230.721875 146.687500 3.596563 3.217250  
## [7] 17.848750 0.437500 0.406250 3.687500 2.812500

# number of unique values iris  
iris

## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
## 1 5.1 3.5 1.4 0.2 setosa  
## 2 4.9 3.0 1.4 0.2 setosa  
## 3 4.7 3.2 1.3 0.2 setosa  
## 4 4.6 3.1 1.5 0.2 setosa  
## 5 5.0 3.6 1.4 0.2 setosa  
## 6 5.4 3.9 1.7 0.4 setosa  
## 7 4.6 3.4 1.4 0.3 setosa  
## 8 5.0 3.4 1.5 0.2 setosa  
## 9 4.4 2.9 1.4 0.2 setosa  
## 10 4.9 3.1 1.5 0.1 setosa  
## 11 5.4 3.7 1.5 0.2 setosa  
## 12 4.8 3.4 1.6 0.2 setosa  
## 13 4.8 3.0 1.4 0.1 setosa  
## 14 4.3 3.0 1.1 0.1 setosa  
## 15 5.8 4.0 1.2 0.2 setosa  
## 16 5.7 4.4 1.5 0.4 setosa  
## 17 5.4 3.9 1.3 0.4 setosa  
## 18 5.1 3.5 1.4 0.3 setosa  
## 19 5.7 3.8 1.7 0.3 setosa  
## 20 5.1 3.8 1.5 0.3 setosa  
## 21 5.4 3.4 1.7 0.2 setosa  
## 22 5.1 3.7 1.5 0.4 setosa  
## 23 4.6 3.6 1.0 0.2 setosa  
## 24 5.1 3.3 1.7 0.5 setosa  
## 25 4.8 3.4 1.9 0.2 setosa  
## 26 5.0 3.0 1.6 0.2 setosa  
## 27 5.0 3.4 1.6 0.4 setosa  
## 28 5.2 3.5 1.5 0.2 setosa  
## 29 5.2 3.4 1.4 0.2 setosa  
## 30 4.7 3.2 1.6 0.2 setosa  
## 31 4.8 3.1 1.6 0.2 setosa  
## 32 5.4 3.4 1.5 0.4 setosa  
## 33 5.2 4.1 1.5 0.1 setosa  
## 34 5.5 4.2 1.4 0.2 setosa  
## 35 4.9 3.1 1.5 0.2 setosa  
## 36 5.0 3.2 1.2 0.2 setosa  
## 37 5.5 3.5 1.3 0.2 setosa  
## 38 4.9 3.6 1.4 0.1 setosa  
## 39 4.4 3.0 1.3 0.2 setosa  
## 40 5.1 3.4 1.5 0.2 setosa  
## 41 5.0 3.5 1.3 0.3 setosa  
## 42 4.5 2.3 1.3 0.3 setosa  
## 43 4.4 3.2 1.3 0.2 setosa  
## 44 5.0 3.5 1.6 0.6 setosa  
## 45 5.1 3.8 1.9 0.4 setosa  
## 46 4.8 3.0 1.4 0.3 setosa  
## 47 5.1 3.8 1.6 0.2 setosa  
## 48 4.6 3.2 1.4 0.2 setosa  
## 49 5.3 3.7 1.5 0.2 setosa  
## 50 5.0 3.3 1.4 0.2 setosa  
## 51 7.0 3.2 4.7 1.4 versicolor  
## 52 6.4 3.2 4.5 1.5 versicolor  
## 53 6.9 3.1 4.9 1.5 versicolor  
## 54 5.5 2.3 4.0 1.3 versicolor  
## 55 6.5 2.8 4.6 1.5 versicolor  
## 56 5.7 2.8 4.5 1.3 versicolor  
## 57 6.3 3.3 4.7 1.6 versicolor  
## 58 4.9 2.4 3.3 1.0 versicolor  
## 59 6.6 2.9 4.6 1.3 versicolor  
## 60 5.2 2.7 3.9 1.4 versicolor  
## 61 5.0 2.0 3.5 1.0 versicolor  
## 62 5.9 3.0 4.2 1.5 versicolor  
## 63 6.0 2.2 4.0 1.0 versicolor  
## 64 6.1 2.9 4.7 1.4 versicolor  
## 65 5.6 2.9 3.6 1.3 versicolor  
## 66 6.7 3.1 4.4 1.4 versicolor  
## 67 5.6 3.0 4.5 1.5 versicolor  
## 68 5.8 2.7 4.1 1.0 versicolor  
## 69 6.2 2.2 4.5 1.5 versicolor  
## 70 5.6 2.5 3.9 1.1 versicolor  
## 71 5.9 3.2 4.8 1.8 versicolor  
## 72 6.1 2.8 4.0 1.3 versicolor  
## 73 6.3 2.5 4.9 1.5 versicolor  
## 74 6.1 2.8 4.7 1.2 versicolor  
## 75 6.4 2.9 4.3 1.3 versicolor  
## 76 6.6 3.0 4.4 1.4 versicolor  
## 77 6.8 2.8 4.8 1.4 versicolor  
## 78 6.7 3.0 5.0 1.7 versicolor  
## 79 6.0 2.9 4.5 1.5 versicolor  
## 80 5.7 2.6 3.5 1.0 versicolor  
## 81 5.5 2.4 3.8 1.1 versicolor  
## 82 5.5 2.4 3.7 1.0 versicolor  
## 83 5.8 2.7 3.9 1.2 versicolor  
## 84 6.0 2.7 5.1 1.6 versicolor  
## 85 5.4 3.0 4.5 1.5 versicolor  
## 86 6.0 3.4 4.5 1.6 versicolor  
## 87 6.7 3.1 4.7 1.5 versicolor  
## 88 6.3 2.3 4.4 1.3 versicolor  
## 89 5.6 3.0 4.1 1.3 versicolor  
## 90 5.5 2.5 4.0 1.3 versicolor  
## 91 5.5 2.6 4.4 1.2 versicolor  
## 92 6.1 3.0 4.6 1.4 versicolor  
## 93 5.8 2.6 4.0 1.2 versicolor  
## 94 5.0 2.3 3.3 1.0 versicolor  
## 95 5.6 2.7 4.2 1.3 versicolor  
## 96 5.7 3.0 4.2 1.2 versicolor  
## 97 5.7 2.9 4.2 1.3 versicolor  
## 98 6.2 2.9 4.3 1.3 versicolor  
## 99 5.1 2.5 3.0 1.1 versicolor  
## 100 5.7 2.8 4.1 1.3 versicolor  
## 101 6.3 3.3 6.0 2.5 virginica  
## 102 5.8 2.7 5.1 1.9 virginica  
## 103 7.1 3.0 5.9 2.1 virginica  
## 104 6.3 2.9 5.6 1.8 virginica  
## 105 6.5 3.0 5.8 2.2 virginica  
## 106 7.6 3.0 6.6 2.1 virginica  
## 107 4.9 2.5 4.5 1.7 virginica  
## 108 7.3 2.9 6.3 1.8 virginica  
## 109 6.7 2.5 5.8 1.8 virginica  
## 110 7.2 3.6 6.1 2.5 virginica  
## 111 6.5 3.2 5.1 2.0 virginica  
## 112 6.4 2.7 5.3 1.9 virginica  
## 113 6.8 3.0 5.5 2.1 virginica  
## 114 5.7 2.5 5.0 2.0 virginica  
## 115 5.8 2.8 5.1 2.4 virginica  
## 116 6.4 3.2 5.3 2.3 virginica  
## 117 6.5 3.0 5.5 1.8 virginica  
## 118 7.7 3.8 6.7 2.2 virginica  
## 119 7.7 2.6 6.9 2.3 virginica  
## 120 6.0 2.2 5.0 1.5 virginica  
## 121 6.9 3.2 5.7 2.3 virginica  
## 122 5.6 2.8 4.9 2.0 virginica  
## 123 7.7 2.8 6.7 2.0 virginica  
## 124 6.3 2.7 4.9 1.8 virginica  
## 125 6.7 3.3 5.7 2.1 virginica  
## 126 7.2 3.2 6.0 1.8 virginica  
## 127 6.2 2.8 4.8 1.8 virginica  
## 128 6.1 3.0 4.9 1.8 virginica  
## 129 6.4 2.8 5.6 2.1 virginica  
## 130 7.2 3.0 5.8 1.6 virginica  
## 131 7.4 2.8 6.1 1.9 virginica  
## 132 7.9 3.8 6.4 2.0 virginica  
## 133 6.4 2.8 5.6 2.2 virginica  
## 134 6.3 2.8 5.1 1.5 virginica  
## 135 6.1 2.6 5.6 1.4 virginica  
## 136 7.7 3.0 6.1 2.3 virginica  
## 137 6.3 3.4 5.6 2.4 virginica  
## 138 6.4 3.1 5.5 1.8 virginica  
## 139 6.0 3.0 4.8 1.8 virginica  
## 140 6.9 3.1 5.4 2.1 virginica  
## 141 6.7 3.1 5.6 2.4 virginica  
## 142 6.9 3.1 5.1 2.3 virginica  
## 143 5.8 2.7 5.1 1.9 virginica  
## 144 6.8 3.2 5.9 2.3 virginica  
## 145 6.7 3.3 5.7 2.5 virginica  
## 146 6.7 3.0 5.2 2.3 virginica  
## 147 6.3 2.5 5.0 1.9 virginica  
## 148 6.5 3.0 5.2 2.0 virginica  
## 149 6.2 3.4 5.4 2.3 virginica  
## 150 5.9 3.0 5.1 1.8 virginica

looper("numeric",iris,function(x) sum(table(unique(x))))

## [1] 35 23 43 22 3

# generate 10 random normals for each of u =-10,0,10 and 100  
ten\_draws <- function(x){rnorm(10,mean=x)}  
# map()  
map(c(-10,0,10,100),ten\_draws)

## [[1]]  
## [1] -10.334859 -10.159373 -8.851752 -9.633061 -9.394355 -9.586417  
## [7] -10.211471 -9.153369 -9.925674 -7.876738  
##   
## [[2]]  
## [1] -2.0663632 0.6954685 0.6562400 -1.2656215 -0.6930066 1.0162898  
## [7] 0.3864444 0.7954032 0.8522951 -0.1884486  
##   
## [[3]]  
## [1] 8.034422 10.622450 9.934833 8.774340 8.735873 9.605158 9.373052  
## [8] 9.789924 12.191097 9.563512  
##   
## [[4]]  
## [1] 101.95141 99.57913 98.66989 99.20373 98.43091 99.49576 100.56717  
## [8] 99.78259 99.24376 99.06273

1. Eliminate the for loop in each of the following examples by taking advantage of an existing function that works with vectors:

# out <- ""  
# for (x in letters)`{}

1. Combine your function writing and for loop skills:

* Write a for loop that prints() the lyrics to the children’s song “Alice the camel”.
* Convert the nursery rhyme “ten in the bed” to a function. Generalise it to any number of people in any sleeping structure.
* Convert the song “99 bottles of beer on the wall” to a function. Generalise to any number of any vessel containing any liquid on any surface.

1. It’s common to see for loops that don’t preallocate the output and instead increase the length of a vector at each step:

## 21.3 For loop variations

Once you have the basic for loop under your belt, there are some variations that you should be aware of. These variations are important regardless of how you do iteration, so don’t forget about them once you’ve mastered the FP techniques you’ll learn about in the next section.

There are four variations on the basic theme of the for loop:

1. Modifying an existing object, instead of creating a new object.
2. Looping over names or values, instead of indices.
3. Handling outputs of unknown length.
4. Handling sequences of unknown length.

### 21.3.1v Modifying an existing object

Sometimes, you want to use a for loop to modify an existing object. For example, remember our challenges from functions. We wanted to rescale every column in a data frame:

library(tidyverse)  
  
df <- tibble(  
 a=rnorm(10),  
 b=rnorm(10),  
 c=rnorm(10),  
 d=rnorm(10)  
)  
  
rescale01 <- function(x){  
 rng <- range(x,na.rm=T)  
 (x-rng[1])/(rng[2]-rng[1])  
}  
  
df$a <- rescale01(df$a)  
df$b <- rescale01(df$b)  
df$c <- rescale01(df$c)  
df$d <- rescale01(df$d)  
  
df

## # A tibble: 10 x 4  
## a b c d  
## <dbl> <dbl> <dbl> <dbl>  
## 1 0.730 0.820 0.676 0.205   
## 2 0.471 0.158 0 0.135   
## 3 0.642 0.431 0.235 0.494   
## 4 0 0.380 0.250 1   
## 5 0.0607 0.206 0.663 0.488   
## 6 0.0401 1 0.411 0.566   
## 7 0.425 0.995 0.609 0.0809  
## 8 0.823 0.116 0.113 0.327   
## 9 0.791 0 0.00586 0   
## 10 1 0.753 1 0.493

To sove this with a for loop, we again think about the three components: 1. Output: we already have the output — it’s the same as the input! 2. Sequence: we can think about a data frame as a list of columns, so we can iterate over each column with seq\_along(df). 3. Body: apply rescale01().

This gives us:

for ( i in seq\_along(df)){  
 df[[i]] <- rescale01(df[[i]])  
}

### 21.3.2 Looping patterns

There are three basic ways to loop over a vector. So far I’ve shown you the most general: looping over the numeric indices with for (i in seq\_along(xs)), and extracting the value with x[[i]]. There are two other forms:

1. Loop over the elements: for (x in xs). This is most useful if you only care about side-effects, like plotting or saving a file, because it’s difficult to save the output efficiently.
2. Loop over the names: for (nm in names(xs)). This gives you name, which you can use to access the value with x[[nm]]. This is useful if you want to use the name in a plot title or a file name. If you’re creating named output, make sure to name the results vector like so:

x

## 01:00:00

results <- vector("list",length(x))  
names(results) <- names(x)

Iteration over the numeric indices is the most general form, because given the position you can extract both the name and the value:

for(i in seq\_along(x)){  
 name <- names(x)[[i]]  
 value <- x[[i]]  
}

### 21.3.3 Unknown output length

Sometimes you might not know how long the output will be. For example, imagine you want to simulate some random vectors of random lengths. You might be tempted to solve this problem by progressively growing the vector:

means <- c(0,1,2)  
  
output <- double()  
for (i in seq\_along(means)){  
 n <- sample(100,1)  
 output <- c(output,rnorm(n,means[[i]]))  
}  
str(output)

## num [1:172] -0.3372 -1.7341 -0.9576 -0.0699 -0.2042 ...

output

## [1] -0.337187979 -1.734141857 -0.957588588 -0.069885708 -0.204170028  
## [6] 1.231831799 0.974446309 -0.305177465 0.865077212 -0.474569875  
## [11] 1.112522847 1.121926566 -0.125700783 -0.523917598 0.810120199  
## [16] 0.517937248 -0.086935756 -0.900015857 -0.934097629 -0.491180936  
## [21] -0.113667990 0.332297841 -0.440033268 -1.098086019 0.718394022  
## [26] -0.454555543 -0.351858642 -0.541535742 -0.275779842 -1.826986108  
## [31] -1.035114428 0.670410653 -1.453995544 -1.451207339 1.847944735  
## [36] 2.004804543 -0.654225837 0.998987648 1.199702225 -1.722081380  
## [41] -0.440239903 -1.190758563 -0.205733288 -1.322726891 0.279205126  
## [46] 0.772581184 -0.450571361 -1.039170890 0.404835818 0.202301345  
## [51] 0.511922337 -2.259071799 -1.346964506 -1.055976628 -0.656139142  
## [56] 0.437703907 0.830508143 1.492371050 -0.615392082 -0.186590252  
## [61] -0.112913474 -0.007838672 0.434558107 -0.113060692 -0.388832212  
## [66] -0.042411057 1.945446512 -0.217458942 -0.393720522 1.032492314  
## [71] 1.594048751 1.348273070 -1.329924431 2.925420162 3.021901050  
## [76] 0.551696544 1.083306763 1.086145151 1.775176698 1.819447265  
## [81] 2.472692149 1.430841621 2.643465535 1.845252667 1.901652783  
## [86] 2.976173915 2.248573925 2.831486756 2.091142926 4.032257219  
## [91] 2.617082826 3.691307385 1.417869210 2.970664525 1.957865030  
## [96] 1.872191510 0.455617177 -1.011324842 1.072353060 1.905274239  
## [101] 1.877434372 2.991562503 2.405382912 3.939577539 2.287914136  
## [106] 0.957732318 1.618527755 2.534938184 1.400471492 1.566051704  
## [111] 1.726061372 2.766964420 2.523524815 2.256566781 0.219664917  
## [116] 0.227698846 1.335758808 2.184004876 1.235232373 1.241262368  
## [121] 3.565694820 1.246727236 3.166452689 1.095283664 1.256678895  
## [126] 3.293218471 2.317843873 1.165995190 1.065640801 1.121960767  
## [131] 2.723156482 2.629722606 1.761851207 0.500694088 2.202332535  
## [136] 3.718310713 3.541067145 3.681880242 2.127621800 2.056356100  
## [141] -0.687278983 -1.580249891 1.040336169 0.566678153 1.716010681  
## [146] 2.607746726 2.744631672 1.733282655 3.002644637 0.990587811  
## [151] 1.593888572 3.137098032 2.497594615 1.290764703 2.656105880  
## [156] 3.284183530 1.779868749 1.699368890 1.302881974 3.511200838  
## [161] 1.541822165 1.798550667 1.879561211 2.806104024 2.475841707  
## [166] 2.945212376 -0.745634609 2.177886430 2.672887578 1.068372670  
## [171] 2.453581026 1.216851161

But this is not very efficient because in each iteration, R has to copy all the data from the previous iterations. In technical terms, you get “quadratic” behaviour which means that a loop with three times as many elements would take nine times as long to run.

A better solution to save the results in a list, and them combine into a single vector after the loop is done:

out <- vector("list",length(means))  
for (i in seq\_along(means)){  
 n <- sample(100,1)  
 out[[i]] <- rnorm(n,means[[i]])  
}  
str(out)

## List of 3  
## $ : num [1:29] 2.316 0.355 -1.498 1.085 1.703 ...  
## $ : num [1:70] 1.072 1.324 1.215 -0.453 1.101 ...  
## $ : num [1:97] 1.2303 2.8086 2.5762 -0.0309 2.6657 ...

str(unlist(out))

## num [1:196] 2.316 0.355 -1.498 1.085 1.703 ...

Here I’ve used unlist() to flatten a list of vectors into a single vector. A stricter option is to use purrr::flatten\_dbl() — it will throw an error if the input isn’t a list of doubles.

This pattern occurs in other places too: 1. You might be generating a long string. Instead of paste()ing together each iteration with the previous, save the output in a character vector and then combine that vector into a single string with paste(output, collapse = "").

1. You might be generating a big data frame. Instead of sequentially rbind()ing in each iteration, save the output in a list, then use dplyr::bind\_rows(output) to combine the output into a single data frame.

Watch out for this pattern. Whenever you see it, switch to a more complex result object, and then combine in one step at the end.

### 21.3.4 Unknown sequence length

Sometimes you don’t even know how long the input sequence should run for. This is common when doing simulations. For example, you might want to loop until you get three heads in a row. You can’t do that sort of iteration with the for loop. Instead, you can use a while loop. A while loop is simpler than for loop because it only has two components, a condition and a body:

# while (condition){  
# #body  
# }

A while loop is also more general than a for loop, because you can rewrite any for loop as a while loop, but you can’t rewrite every while loop as for loop:

# for (i in seq\_along(x)){  
# #body  
# }  
  
# Equivalent to   
# i <- 1  
# while(i <= length(x)){  
# #body  
# i <- i+1  
# }

Herhow we could use a while loop to find how many tries it takes to get three heads in a row:

flip <- function() sample(c("T", "H"), 1)  
  
flips <- 0  
nheads <- 0  
  
while (nheads < 3) {  
 if (flip() == "H") {  
 nheads <- nheads + 1  
 } else {  
 nheads <- 0  
 }  
 flips <- flips + 1  
}  
flips

## [1] 14

#> [1] 3

I mention while loops only briefly, because I hardly ever use them. They’re most often used for simulation, which is outside the scope of this book. However, it is good to know they exist so that you’re prepared for problems where the number of iterations is not known in advance.

### 21.3.5 Exercises

1. Imagine you have a directory full of CSV files that you want to read in. You have their paths in a vector, files <- dir("data/", pattern = "\\.csv$", full.names = TRUE), and now want to read each one with read\_csv(). Write the for loop that will load them into a single data frame.

all\_csv <- c("one.csv","two.csv")  
all\_dfs <- vector("list",length(all\_csv))  
  
# for (i in all.csv){  
# all\_dfs[[i]] <- read.csv(all\_csv[[i]])  
# }  
# bind\_rows(all\_dfs)

1. What happens if you use for (nm in names(x)) and x has no names? What if only some of the elements are named? What if the names are not unique?

no\_names <- 1:5  
some\_names <- c("one" = 1, 2, "three" = 3)  
repeated\_names <- c("one" = 1, "one" = 2, "three" = 3)  
  
for (nm in names(no\_names)) print(identity(nm)) # nothing happens!  
for (nm in names(some\_names)) print(identity(nm)) # the empty name get's filled with a ""

## [1] "one"  
## [1] ""  
## [1] "three"

for (nm in names(repeated\_names)) print(identity(nm)) # everything get's printed out

## [1] "one"  
## [1] "one"  
## [1] "three"

1. Write a function that prints the mean of each numeric column in a data frame, along with its name. For example, show\_mean(iris) would print:

library(tidyverse)  
# show\_mean(iris)  
#> Sepal.Length: 5.84  
#> Sepal.Width: 3.06  
#> Petal.Length: 3.76  
#> Petal.Width: 1.20  
  
show\_means <- function(x) {  
   
 the\_class <- vector("logical", length(x))  
  
 for (i in seq\_along(x)) the\_class[[i]] <- is.numeric(x[[i]])  
   
 x <- x[the\_class]  
   
 for (i in seq\_along(x)) {  
 cat(paste0(names(x)[i], ": ", round(mean(x[[i]]), 2)), fill = TRUE)  
 }  
}  
  
show\_means(iris)

## Sepal.Length: 5.84  
## Sepal.Width: 3.06  
## Petal.Length: 3.76  
## Petal.Width: 1.2

show\_means(mtcars)

## mpg: 20.09  
## cyl: 24.75  
## disp: 230.72  
## hp: 146.69  
## drat: 3.6  
## wt: 3.22  
## qsec: 17.85  
## vs: 0.44  
## am: 0.41  
## gear: 3.69  
## carb: 2.81

1. What does this code do? How does it work?

trans <- list(   
 disp = function(x) x \* 0.0163871,  
 am = function(x) {  
 factor(x, labels = c("auto", "manual"))  
 }  
)  
for (var in names(trans)) {  
 mtcars[[var]] <- trans[[var]](mtcars[[var]])  
}

It converts disp and am by multiplying and then into a factor respectively. This is simply iterating over a list with functions, and applying in that same order to both variables.

## 21.4 For loops vs. functionals

For loops are not as important in R as they are in other languages because R is a functional programming language. This means that it’s possible to wrap up for loops in a function, and call that function instead of using the for loop directly.

To see why this is important, consider (again) this simple data frame:

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

Imagine you want to compute the mean of every column. You could do that with a for loop:

output <- vector("double",length(df))  
for (i in seq\_along(df)){  
 output[[i]] <- mean(df[[i]])  
}  
output

## [1] 0.02276673 0.09865831 0.61447062 -0.07774047

You realize that you’re going to want to compute the means of every column pretty frequently,so you extract it out into a function:

col\_mean <- function(df){  
 output <- vector("double",length(df))  
 for (i in seq\_along(df)){  
 output[i] <- mean(df[[i]])  
 }  
 output  
}

But then you think it would also be helpful to be able to compute the median, and the standard deviation, so you copy and paste your col\_mean() function and replace themean() with median() and sd():

col\_median <- function(df){  
 output <- vector("double",hh(df))  
 for (i in seq\_along(df)){  
 output[i] <- median(df[[i]])  
 }  
 output  
}  
  
col\_sd <- function(df){  
 output <- vector("double",length(df))  
 for (i in seq\_along(df)){  
 output[i] <- sd(df[[i]])  
 }  
 output  
}  
  
df

## # A tibble: 10 x 4  
## a b c d  
## <dbl> <dbl> <dbl> <dbl>  
## 1 0.0798 0.360 0.601 -1.14   
## 2 0.987 0.952 1.07 0.208  
## 3 1.18 -0.504 -0.176 2.69   
## 4 -0.497 0.865 -0.291 -0.337  
## 5 -0.209 0.384 1.94 -1.46   
## 6 1.30 -1.96 0.257 0.550  
## 7 0.457 -0.246 0.677 0.382  
## 8 -1.37 -0.865 0.443 -1.29   
## 9 -1.77 0.815 0.920 -1.15   
## 10 0.0712 1.18 0.710 0.765

# col\_median(df)

Uh oh! You’ve copied-and-pasted this code twice, so it’s time to think about how to generalise it. Notice that most of this code is for-loop boilerplate and it’s hard to see the one thing (mean(), median(), sd()) that is different between the functions.

What would you do if you saw a set of functions like this:

f1 <- function(x) abs(x-mean(x))^1  
f2 <- function(x) abs(x-mean(x))^2  
f3 <- function(x) abs(x-mean(x))^3

Hopefully, you’d notice that there is a lot of duplication, and extract it out into an additional argument:

f <- function(x,i) abs(x-mean(x))^i

You have reduced the chance of bugs (because you know have 1/3 less code), and made it easy to generalize to new situations.

We can do exactly the same thing with col\_mean(), col\_median() and col\_sd() by adding an argument that supplies the function to apply to each column:

col\_summary <- function(df, fun) {  
 out <- vector("double", length(df))  
 for (i in seq\_along(df)) {  
 out[i] <- fun(df[[i]])  
 }  
 out  
}  
col\_summary(df, median)

## [1] 0.07549438 0.37191041 0.63915492 -0.06447181

#> [1] 0.237 -0.218 0.254 -0.133  
col\_summary(df, mean)

## [1] 0.02276673 0.09865831 0.61447062 -0.07774047

#> [1] 0.2026 -0.2068 0.1275 -0.0917

The idea of passing a function to another function is extremely powerful idea, and it’s one of the behaviours that makes R a functional programming language. It might take you a while to wrap your head around the idea, but it’s worth the investment. In the rest of the chapter, you’ll learn about and use the purrr package, which provides functions that eliminate the need for many common for loops. The apply family of functions in base R (apply(), lapply(), tapply(), etc) solve a similar problem, but purrr is more consistent and thus is easier to learn.

The goal of using purrr function instead of for loops is to allow you break common list manipulation challenges into independent pieces:

1. How can you solve the problem for a single element of the list? Once you’ve solved that problem, purrr takes care of generalising your solution to every element in the list.
2. If you’re solving a complex problem, how can you break it down into bite-sized pieces that allow you to advance one small step towards a solution? With purrr, you get lots of small pieces that you can compose together with the pipe.

This structure makes it easier to solve new problems. It also makes it easier to understand your solutions to old problems when you re-read your old code.

### 21.4.1 Exercises

1. Read the documentation for apply(). In the 2d case, what two for loops does it generalise?
2. Adapt col\_summary() so that it only applies to numeric columns You might want to start with an is\_numeric() function that returns a logical vector that has a TRUE corresponding to each numeric column.

## 21.5 The map functions

The pattern of looping over a vector, doing something to each element and saving the results is so common that the purrr package provides a family of functions to do it for you. There is one function for each type of output:

* map() makes a list.
* map\_lgl() makes a logical vector.
* map\_int() makes an integer vector.
* map\_dbl() makes a double vector.
* map\_chr() makes a character vector.

Each function takes a vector as input, applies a function to each piece, and then returns a new vector that’s the same length(and has the same names) as the input. The type of the vector is determined by the suffix to the map function.

Once you master these functions, you’ll find it takes much less time to solve iteration problems. But you should never feel bad about using a for loop instead of a map function. The map functions are a step up a tower of abstraction, and it can take a long time to get your head around how they work. The important thing is that you solve the problem that you’re working on, not write the most concise and elegant code (although that’s definitely something you want to strive towards!).

Some people will tell you to avoid for loops because they are slow. They’re wrong! (Well at least they’re rather out of date, as for loops haven’t been slow for many years). The chief benefits of using functions like map() is not speed, but clarity: they make your code easier to write and to read.

We can use these functions to perform the same computations as the last for loop. Those summary functions returned doubles, so we need to use map\_dbl():

library(purrr)  
head(df)

## # A tibble: 6 x 4  
## a b c d  
## <dbl> <dbl> <dbl> <dbl>  
## 1 0.0798 0.360 0.601 -1.14   
## 2 0.987 0.952 1.07 0.208  
## 3 1.18 -0.504 -0.176 2.69   
## 4 -0.497 0.865 -0.291 -0.337  
## 5 -0.209 0.384 1.94 -1.46   
## 6 1.30 -1.96 0.257 0.550

# Reference - for loop()  
output <- vector("double",length(df))  
for (i in seq\_along(df)){  
 output[[i]] <- mean(df[[i]])  
}  
output

## [1] 0.02276673 0.09865831 0.61447062 -0.07774047

map\_dbl(df,mean)

## a b c d   
## 0.02276673 0.09865831 0.61447062 -0.07774047

map\_dbl(df,median)

## a b c d   
## 0.07549438 0.37191041 0.63915492 -0.06447181

map\_dbl(df,sd)

## a b c d   
## 1.0312944 0.9883998 0.6377394 1.2825290

Compared to using a for loop, focus is on the operation being performed (i.e. mean(), median(), sd()), not the bookkeeping required to loop over every element and store the output. This is even more apparent if we use the pipe:

df %>% map\_dbl(mean)

## a b c d   
## 0.02276673 0.09865831 0.61447062 -0.07774047

df %>% map\_dbl(median)

## a b c d   
## 0.07549438 0.37191041 0.63915492 -0.06447181

df %>% map\_dbl(sd)

## a b c d   
## 1.0312944 0.9883998 0.6377394 1.2825290

There are a few differences between map\_\*() and col\_summary():

* All purrr functions are implemented in C. This makes them a little faster at the expense of readability.
* The second argument, .f, the function to apply, can be a formula, a character vector, or an integer vector. You’ll learn about those handy shortcuts in the next section.
* map\_\*() uses … ([dot dot dot]) to pass along additional arguments to .f each time it’s called:

map\_dbl(df,mean,trim=0.5)

## a b c d   
## 0.07549438 0.37191041 0.63915492 -0.06447181

* The map functions also preserve names:

z <- list(x=1:3,y=4:5)  
z

## $x  
## [1] 1 2 3  
##   
## $y  
## [1] 4 5

map\_int(z,length)

## x y   
## 3 2

### 21.5.1 Shortcuts

There are a few shortcuts that you can use with .f in order to save a little typing. Imagine you want to fit a linear model to each group in a dataset. The following toy example splits the up the mtcars dataset in to three pieces (one for each value of cylinder) and fits the same linear model to each piece:

safe\_log <- safely(log)  
str(safe\_log(10))

## List of 2  
## $ result: num 2.3  
## $ error : NULL

str(safe\_log("a"))

## List of 2  
## $ result: NULL  
## $ error :List of 2  
## ..$ message: chr " 数学関数に数値でない引数が渡されました "  
## ..$ call : language log(x = x, base = base)  
## ..- attr(\*, "class")= chr [1:3] "simpleError" "error" "condition"

When the functions suceeds, the result element contains the result and the error element is NULL. When the function fails, the result element is NULL and the error element contains an error object.

safely() is designed to work with map:

x <- list(1,10,"a")  
y <- x %>% map(safely(log))  
str(y)

## List of 3  
## $ :List of 2  
## ..$ result: num 0  
## ..$ error : NULL  
## $ :List of 2  
## ..$ result: num 2.3  
## ..$ error : NULL  
## $ :List of 2  
## ..$ result: NULL  
## ..$ error :List of 2  
## .. ..$ message: chr " 数学関数に数値でない引数が渡されました "  
## .. ..$ call : language log(x = x, base = base)  
## .. ..- attr(\*, "class")= chr [1:3] "simpleError" "error" "condition"

This would be easier to work with if we had two lists: one of all the errors and one of all the output. That’s easy to get with purrr::transpose():

y <- x %>% transpose()  
str(y)

## List of 1  
## $ :List of 3  
## ..$ : num 1  
## ..$ : num 10  
## ..$ : chr "a"

It’s up to you how to deal with the errors, but typically you’ll either look at the values of x where y is an error, or work with the values of y that are ok:

is\_ok <- y$error %>% map\_lgl(is\_null)  
x[!is\_ok]

## list()

#> [[1]]  
#> [1] "a"  
# y$result[is\_ok] %>% flatten\_dbl()  
#> [1] 0.0 2.3

Purrr provides two other useful adverbs: 1. Like safely(), possibly() always succeeds. It’s simpler than safely(), because you give it a default value to return when there is an error.

x <- list(1,10,"a")  
x %>% map\_dbl(possibly(log,NA\_real\_))

## [1] 0.000000 2.302585 NA

1. quietly() performs a similar role to safely(), but instead of capturing errors, it captures printed output, messages, and warnings:

x <- list(1,-1)  
x %>% map(quietly(log)) %>% str()

## List of 2  
## $ :List of 4  
## ..$ result : num 0  
## ..$ output : chr ""  
## ..$ warnings: chr(0)   
## ..$ messages: chr(0)   
## $ :List of 4  
## ..$ result : num NaN  
## ..$ output : chr ""  
## ..$ warnings: chr " 計算結果が NaN になりました "  
## ..$ messages: chr(0)

## 21.7 Mapping over multiple arguments

So far we’ve mapped along a single input. But often you have multiple related inputs that you need iterate along in parallel. That’s the job of the map2() and pmap() functions. For example, imagine you want to simulate some random normals with different means. You know how to do that with map():

mu <- list(5,10,-3)  
mu %>%   
 map(rnorm,n=5) %>%   
 str()

## List of 3  
## $ : num [1:5] 5.35 4.31 6.4 5.76 5.6  
## $ : num [1:5] 9.91 10.43 10.83 10.85 9.42  
## $ : num [1:5] -1.346 -4.921 -0.453 -2.709 -3.402

What if you also want to vary the standard deviation? One way to do that would be to iterate over the indices and index into vectors of means and sds:

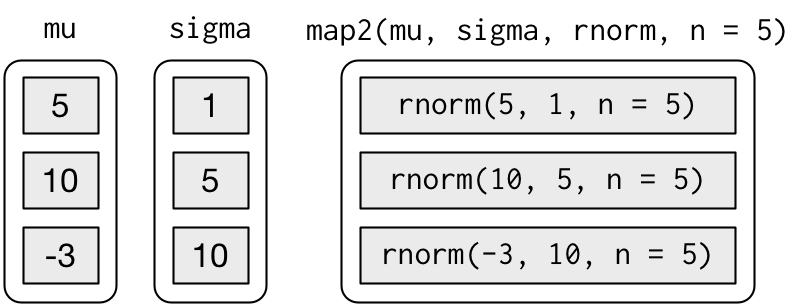
sigma <- list(1,5,10)  
seq\_along(mu) %>%   
 map(~rnorm(5,mu[[.]],sigma[[.]])) %>%   
 str()

## List of 3  
## $ : num [1:5] 4.84 5.55 6.28 4.51 3.37  
## $ : num [1:5] 2.35 8.05 5.49 7.24 5.55  
## $ : num [1:5] 17.12 -4.46 -17.79 -22.65 -10.62

But that obfuscates the intent of the code. Instead we could use map2() which iterates over two vectors in parallel:

map2(mu,sigma,rnorm,n=5) %>% str()

## List of 3  
## $ : num [1:5] 4.14 4.38 3.7 4.39 5.67  
## $ : num [1:5] 5.87 19.17 12.12 11.71 9.33  
## $ : num [1:5] 9.53 5.03 -12.62 -13.45 15.28



map2 overview