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$a)
#> [1] -0.246
median(df$b)
#> [1] -0.287
median(df$c)
#> [1] -0.0567
median(df$d)
#> [1] 0.144
```

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

```
output <- vector("double", ncol(df))  # 1. output
for (i in seq_along(df)) {            # 2. sequence
  output[[i]] <- median(df[[i]])      # 3. body
}
output
#> [1] -0.2458 -0.2873 -0.0567  0.1443
```

Every for loop has three components:

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.

2. 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)
   1:length(y)
   #> [1] 1 0
   ```

   You probably won't create a zero-length vector deliberately, but it's easy to create them accidentally. If you use `1:length(x)` instead of `seq_along(x)`, you're likely to get a confusing error message.

3. 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:

   1. Compute the mean of every column in `mtcars`.
   2. Determine the type of each column in `nycflights13::flights`.
   3. Compute the number of unique values in each column of `iris`.
   4. Generate 10 random normals for each of $\mu = -10$, 0, 10, and 100.

   Think about the output, sequence, and body **before** you start writing the loop.

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

```
out <- ""
for (x in letters) {
  out <- stringr::str_c(out, x)
}

x <- sample(100)
sd <- 0
for (i in seq_along(x)) {
  sd <- sd + (x[i] - mean(x)) ^ 2
}
sd <- sqrt(sd / (length(x) - 1))

x <- runif(100)
out <- vector("numeric", length(x))
out[1] <- x[1]
for (i in 2:length(x)) {
  out[i] <- out[i - 1] + x[i]
}
```

3. Combine your function writing and for loop skills:

    1. Write a for loop that `prints()` the lyrics to the children's song "Alice the camel".

    2. Convert the nursery rhyme "ten in the bed" to a function. Generalise it to any number of people in any sleeping structure.

    3. 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.

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

```
output <- vector("integer", 0)
for (i in seq_along(x)) {
  output <- c(output, lengths(x[[i]]))
}
output
```

How does this affect performance? Design and execute an experiment.

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.

21.3.1 Modifying an existing object

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

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

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

To solve this with a for loop we again think about the three components:

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]])
}
```

Typically you'll be modifying a list or data frame with this sort of loop, so remember to use `[[`, not `[`. You might have spotted that I used `[[` in all my for loops: I think it's better to use `[[` even for atomic vectors because it makes it clear that I want to work with a single element.

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:

```
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 [1:202] 0.912 0.205 2.584 -0.789 0.588 ...
```

But this is not very efficient because in each iteration, R has to copy all the data from the previous iterations. In technical terms you get "quadratic" ($O(n^2)$) behaviour which means that a loop with three times as many elements would take nine (3^2) times as long to run.

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

```
out <- vector("list", length(means))
for (i in seq_along(means)) {
  n <- sample(100, 1)
  out[[i]] <- rnorm(n, means[[i]])
}
str(out)
#> List of 3
#>  $ : num [1:83] 0.367 1.13 -0.941 0.218 1.415 ...
#>  $ : num [1:21] -0.485 -0.425 2.937 1.688 1.324 ...
#>  $ : num [1:40] 2.34 1.59 2.93 3.84 1.3 ...

str(unlist(out))
#>  num [1:144] 0.367 1.13 -0.941 0.218 1.415 ...
```

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

This pattern occurs in other places too:

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 = "")` .

2. 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 a for loop:

```
for (i in seq_along(x)) {
  # body
}

# Equivalent to
i <- 1
while (i <= length(x)) {
  # body
  i <- i + 1
}
```

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

```
flip <- function() sample(c("T", "H"), 1)

flips <- 0
nheads <- 0

while (nheads < 3) {
  if (flip() == "H") {
    nheads <- nheads + 1
  } else {
    nheads <- 0
  }
  flips <- flips + 1
}
flips
#> [1] 3
```

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

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.

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

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

    ```
    show_mean(iris)
    #> Sepal.Length: 5.84
    #> Sepal.Width:  3.06
    #> Petal.Length: 3.76
    #> Petal.Width:  1.20
    ```

(Extra challenge: what function did I use to make sure that the numbers lined up nicely, even though the variable names had different lengths?)

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

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

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]  0.2026 -0.2068  0.1275 -0.0917
```

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

```
col_mean <- function(df) {
  output <- vector("double", length(df))
  for (i in seq_along(df)) {
    output[i] <- mean(df[[i]])
  }
  output
}
```

But then you think it'd also be helpful to be able to compute the median, and the standard deviation, so you copy and paste your `col_mean()` function and replace the `mean()` with `median()` and `sd()`:

```
col_median <- function(df) {
  output <- vector("double", length(df))
  for (i in seq_along(df)) {
    output[i] <- median(df[[i]])
  }
  output
}
col_sd <- function(df) {
  output <- vector("double", length(df))
  for (i in seq_along(df)) {
    output[i] <- sd(df[[i]])
  }
  output
}
```

Uh oh! You've copied-and-pasted this code twice, so it's time to think about how to 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's a lot of duplication, and extract it out into an additional argument:

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

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

We can do exactly the same thing with `col_mean()` , `col_median()` and `col_sd()` by adding an argument that supplies the function to apply to each column:

```
col_summary <- function(df, fun) {
  out <- vector("double", length(df))
  for (i in seq_along(df)) {
    out[i] <- fun(df[[i]])
  }
  out
}
col_summary(df, median)
#> [1]  0.237 -0.218  0.254 -0.133
col_summary(df, mean)
#> [1]  0.2026 -0.2068  0.1275 -0.0917
```

The idea of passing a function to another function is 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 functions 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()`:

```
map_dbl(df, mean)
#>       a       b       c       d
#>  0.2026 -0.2068  0.1275 -0.0917
map_dbl(df, median)
#>      a      b      c      d
#>  0.237 -0.218  0.254 -0.133
map_dbl(df, sd)
#>     a     b     c     d
#> 0.796 0.759 1.164 1.062
```

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

```
df %>% map_dbl(mean)
#>       a       b       c       d
#>  0.2026 -0.2068  0.1275 -0.0917
df %>% map_dbl(median)
#>      a      b      c      d
#>  0.237 -0.218  0.254 -0.133
df %>% map_dbl(sd)
#>     a     b     c     d
#> 0.796 0.759 1.164 1.062
```

There are a few differences between `map_*()` and `col_summary()`:

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

```
map_dbl(df, mean, trim = 0.5)
#>      a      b      c      d
#>  0.237 -0.218  0.254 -0.133
```

- The map functions also preserve names:

```
z <- list(x = 1:3, y = 4:5)
map_int(z, length)
#> x y
#> 3 2
```

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:

```
models <- mtcars %>%
  split(.$cyl) %>%
  map(function(df) lm(mpg ~ wt, data = df))
```

The syntax for creating an anonymous function in R is quite verbose so purrr provides a convenient shortcut: a one-sided formula.

```
models <- mtcars %>%
  split(.$cyl) %>%
  map(~lm(mpg ~ wt, data = .))
```

Here I've used `.` as a pronoun: it refers to the current list element (in the same way that `i` referred to the current index in the for loop).

When you're looking at many models, you might want to extract a summary statistic like the R^2. To do that we need to first run `summary()` and then extract the component called `r.squared`. We could do that using the shorthand for anonymous functions:

```
models %>%
  map(summary) %>%
  map_dbl(~.$r.squared)
#>     4     6     8
#> 0.509 0.465 0.423
```

But extracting named components is a common operation, so purrr provides an even shorter shortcut: you can use a string.

```
models %>%
  map(summary) %>%
  map_dbl("r.squared")
#>     4     6     8
#> 0.509 0.465 0.423
```

You can also use an integer to select elements by position:

```
x <- list(list(1, 2, 3), list(4, 5, 6), list(7, 8, 9))
x %>% map_dbl(2)
#> [1] 2 5 8
```

21.5.2 Base R

If you're familiar with the apply family of functions in base R, you might have noticed some similarities with the purrr functions:

- `lapply()` is basically identical to `map()`, except that `map()` is consistent with all the other functions in purrr, and you can use the shortcuts for `.f`.
- Base `sapply()` is a wrapper around `lapply()` that automatically simplifies the output. This is useful for interactive work but is problematic in a function because you never know what sort of output you'll get:

```
x1 <- list(
  c(0.27, 0.37, 0.57, 0.91, 0.20),
  c(0.90, 0.94, 0.66, 0.63, 0.06),
  c(0.21, 0.18, 0.69, 0.38, 0.77)
)
x2 <- list(
  c(0.50, 0.72, 0.99, 0.38, 0.78),
  c(0.93, 0.21, 0.65, 0.13, 0.27),
  c(0.39, 0.01, 0.38, 0.87, 0.34)
)

threshold <- function(x, cutoff = 0.8) x[x > cutoff]
x1 %>% sapply(threshold) %>% str()
#> List of 3
#>  $ : num 0.91
#>  $ : num [1:2] 0.9 0.94
#>  $ : num(0)


x2 %>% sapply(threshold) %>% str()
#>  num [1:3] 0.99 0.93 0.87
```

- `vapply()` is a safe alternative to `sapply()` because you supply an additional argument that defines the type. The only problem with `vapply()` is that it's a lot of typing: `vapply(df, is.numeric, logical(1))` is equivalent to `map_lgl(df, is.numeric)`. One advantage of `vapply()` over purrr's map functions is that it can also produce matrices — the map functions only ever produce vectors.

I focus on purrr functions here because they have more consistent names and arguments, helpful shortcuts, and in the future will provide easy parallelism and progress bars.

21.5.3 Exercises

1. Write code that uses one of the map functions to:
    1. Compute the mean of every column in `mtcars`.
    2. Determine the type of each column in `nycflights13::flights`.
    3. Compute the number of unique values in each column of `iris`.
    4. Generate 10 random normals for each of $\mu = -10$, 0, 10, and 100.
2. How can you create a single vector that for each column in a data frame indicates whether or not it's a factor?
3. What happens when you use the map functions on vectors that aren't lists? What does `map(1:5, runif)` do? Why?
4. What does `map(-2:2, rnorm, n = 5)` do? Why? What does `map_dbl(-2:2, rnorm, n = 5)` do? Why?
5. Rewrite `map(x, function(df) lm(mpg ~ wt, data = df))` to eliminate the anonymous function.

21.6 Dealing with failure

When you use the map functions to repeat many operations, the chances are much higher that one of those operations will fail. When this happens, you'll get an error message, and no output. This is annoying: why does one failure prevent you from accessing all the other successes? How do you ensure that one bad apple doesn't ruin the whole barrel?

In this section you'll learn how to deal this situation with a new function: `safely()`. `safely()` is an adverb: it takes a function (a verb) and returns a modified version. In this case, the modified function will never throw an error. Instead, it always returns a list with two elements:

1. `result` is the original result. If there was an error, this will be `NULL`.

2. `error` is an error object. If the operation was successful, this will be `NULL`.

(You might be familiar with the `try()` function in base R. It's similar, but because it sometimes returns the original result and it sometimes returns an error object it's more difficult to work with.)

Let's illustrate this with a simple example: `log()`:

```
safe_log <- safely(log)
str(safe_log(10))
#> List of 2
#>  $ result: num 2.3
#>  $ error : NULL
str(safe_log("a"))
#> List of 2
#>  $ result: NULL
#>  $ error :List of 2
#>   ..$ message: chr "non-numeric argument to mathematical function"

#>   ..$ call   : language log(x = x, base = base)
#>   ..- attr(*, "class")= chr [1:3] "simpleError" "error" "condition"
```

When the function succeeds, the `result` element contains the result and the `error` element is `NULL`. When the function fails, the `result` element is `NULL` and the `error` element contains an error object.

`safely()` is designed to work with map:

```
x <- list(1, 10, "a")
y <- x %>% map(safely(log))
str(y)
#> List of 3
#>  $ :List of 2
#>   ..$ result: num 0
#>   ..$ error : NULL
#>  $ :List of 2
#>   ..$ result: num 2.3
#>   ..$ error : NULL
#>  $ :List of 2
#>   ..$ result: NULL
#>   ..$ error :List of 2
#>   .. ..$ message: chr "non-numeric argument to mathematical function"
#>   .. ..$ 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 <- y %>% transpose()
str(y)
#> List of 2
#>  $ result:List of 3
#>   ..$ : num 0
#>   ..$ : num 2.3
#>   ..$ : NULL
#>  $ error :List of 3
#>   ..$ : NULL
#>   ..$ : NULL
#>   ..$ :List of 2
#>   .. ..$ message: chr "non-numeric argument to mathematical function"
#>   .. ..$ call   : language log(x = x, base = base)
#>   .. ..- attr(*, "class")= chr [1:3] "simpleError" "error" "condition"
```

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

```
is_ok <- y$error %>% map_lgl(is_null)
x[!is_ok]
#> [[1]]
#> [1] "a"
y$result[is_ok] %>% flatten_dbl()
#> [1] 0.0 2.3
```

Purrr provides two other useful adverbs:

- Like `safely()`, `possibly()` always succeeds. It's simpler than `safely()`, because you give it a default value to return when there is an error.

  ```
  x <- list(1, 10, "a")
  x %>% map_dbl(possibly(log, NA_real_))
  #> [1] 0.0 2.3  NA
  ```

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

  ```
  x <- list(1, -1)
  x %>% map(quietly(log)) %>% str()
  #> List of 2
  #>  $ :List of 4
  #>   ..$ result  : num 0
  #>   ..$ output  : chr ""
  #>   ..$ warnings: chr(0)
  #>   ..$ messages: chr(0)
  #>  $ :List of 4
  #>   ..$ result  : num NaN
  #>   ..$ output  : chr ""
  #>   ..$ warnings: chr "NaNs produced"
  #>   ..$ messages: chr(0)
  ```

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] 5.45 5.5 5.78 6.51 3.18
#>  $ : num [1:5] 10.79 9.03 10.89 10.76 10.65
#>  $ : num [1:5] -3.54 -3.08 -5.01 -3.51 -2.9
```

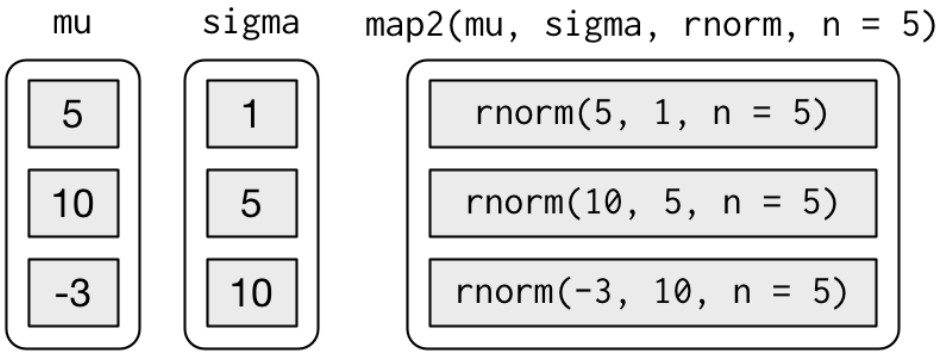
What if you also want to vary the standard deviation? One way to do that would be to iterate over the indices and index into vectors of means and sds:

```
sigma <- list(1, 5, 10)
seq_along(mu) %>%
  map(~rnorm(5, mu[[.]], sigma[[.]])) %>%
  str()
#> List of 3
#>  $ : num [1:5] 4.94 2.57 4.37 4.12 5.29
#>  $ : num [1:5] 11.72 5.32 11.46 10.24 12.22
#>  $ : num [1:5] 3.68 -6.12 22.24 -7.2 10.37
```

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

```
map2(mu, sigma, rnorm, n = 5) %>% str()
#> List of 3
#>  $ : num [1:5] 4.78 5.59 4.93 4.3 4.47
#>  $ : num [1:5] 10.85 10.57 6.02 8.82 15.93
#>  $ : num [1:5] -1.12 7.39 -7.5 -10.09 -2.7
```

`map2()` generates this series of function calls:



Note that the arguments that vary for each call come *before* the function; arguments that are the same for every call come *after*.

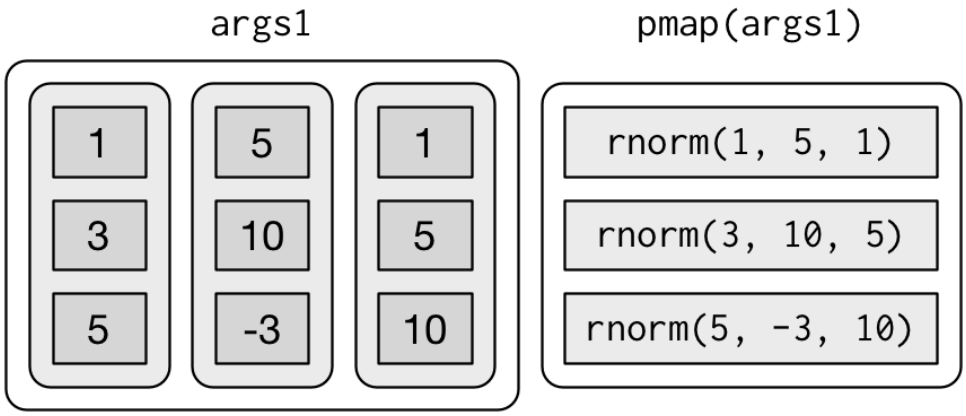
Like `map()`, `map2()` is just a wrapper around a for loop:

```
map2 <- function(x, y, f, ...) {
  out <- vector("list", length(x))
  for (i in seq_along(x)) {
    out[[i]] <- f(x[[i]], y[[i]], ...)
  }
  out
}
```

You could also imagine `map3()`, `map4()`, `map5()`, `map6()` etc, but that would get tedious quickly. Instead, purrr provides `pmap()` which takes a list of arguments. You might use that if you wanted to vary the mean, standard deviation, and number of samples:

```
n <- list(1, 3, 5)
args1 <- list(n, mu, sigma)
args1 %>%
  pmap(rnorm) %>%
  str()
#> List of 3
#>  $ : num 4.55
#>  $ : num [1:3] 13.4 18.8 13.2
#>  $ : num [1:5] 0.685 10.801 -11.671 21.363 -2.562
```

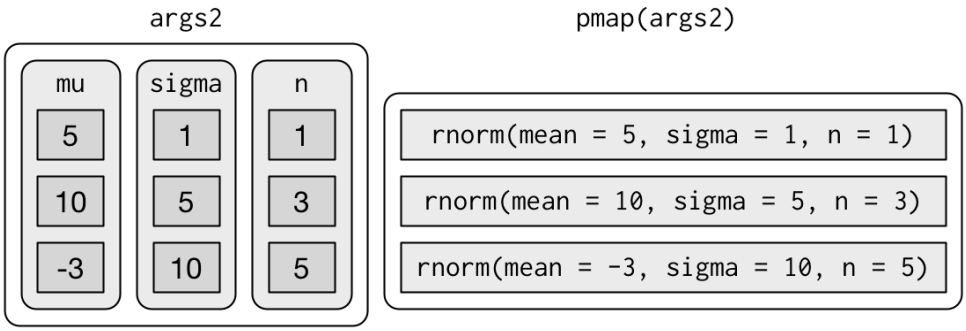
That looks like:



If you don't name the elements of list, `pmap()` will use positional matching when calling the function. That's a little fragile, and makes the code harder to read, so it's better to name the arguments:

```
args2 <- list(mean = mu, sd = sigma, n = n)
args2 %>%
  pmap(rnorm) %>%
  str()
```

That generates longer, but safer, calls:



Since the arguments are all the same length, it makes sense to store them in a data frame:

```
params <- tribble(
  ~mean, ~sd, ~n,
    5,     1,  1,
   10,     5,  3,
   -3,    10,  5
)
params %>%
  pmap(rnorm)
#> [[1]]
#> [1] 4.68
#>
#> [[2]]
#> [1] 23.44 12.85  7.28
#>
#> [[3]]
#> [1]  -5.34 -17.66   0.92   6.06   9.02
```

As soon as your code gets complicated, I think a data frame is a good approach because it ensures that each column has a name and is the same length as all the other columns.

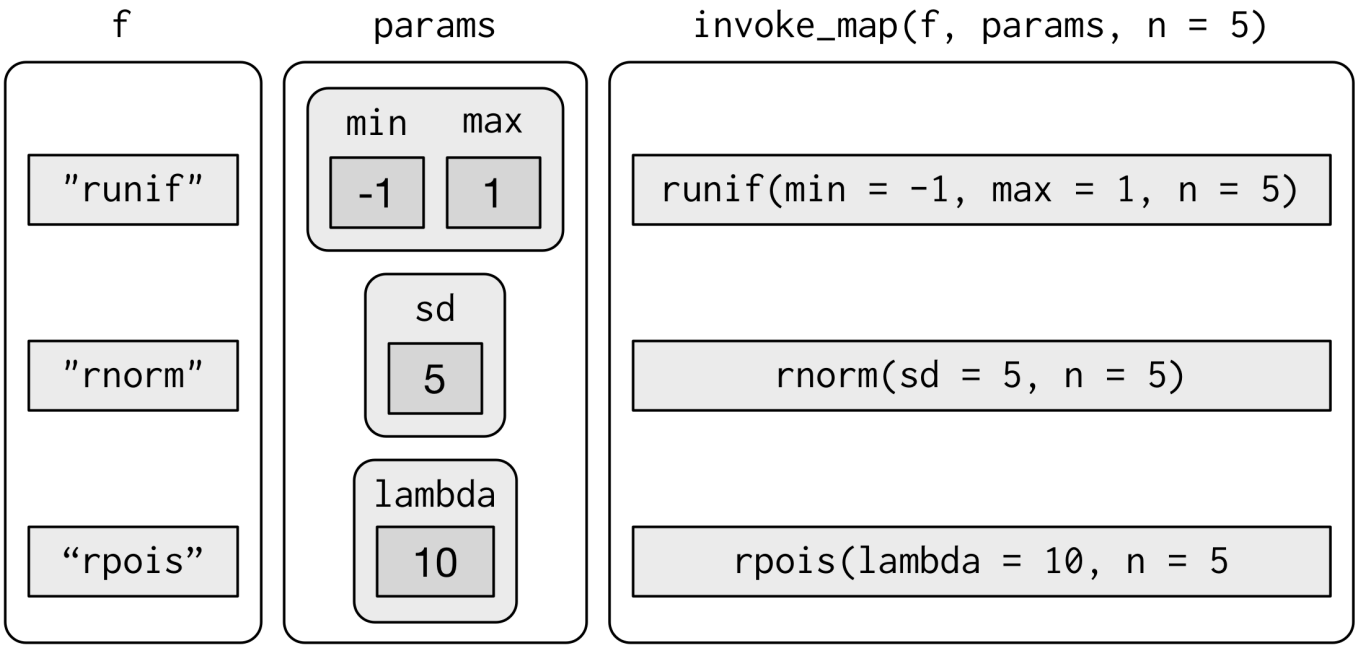
21.7.1 Invoking different functions

There's one more step up in complexity - as well as varying the arguments to the function you might also vary the function itself:

```
f <- c("runif", "rnorm", "rpois")
param <- list(
  list(min = -1, max = 1),
  list(sd = 5),
  list(lambda = 10)
)
```

To handle this case, you can use `invoke_map()`:

```
invoke_map(f, param, n = 5) %>% str()
#> List of 3
#>  $ : num [1:5] 0.762 0.36 -0.714 0.531 0.254
#>  $ : num [1:5] 3.07 -3.09 1.1 5.64 9.07
#>  $ : int [1:5] 9 14 8 9 7
```



The first argument is a list of functions or character vector of function names. The second argument is a list of lists giving the arguments that vary for each function. The subsequent arguments are passed on to every function.

And again, you can use `tribble()` to make creating these matching pairs a little easier:

```
sim <- tribble(
  ~f,      ~params,
  "runif", list(min = -1, max = 1),
  "rnorm", list(sd = 5),
  "rpois", list(lambda = 10)
)
sim %>%
  mutate(sim = invoke_map(f, params, n = 10))
```

21.8 Walk

Walk is an alternative to map that you use when you want to call a function for its side effects, rather than for its return value. You typically do this because you want to render output to the screen or save files to disk - the important thing is the action, not the return value. Here's a very simple example:

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

x %>%
  walk(print)
#> [1] 1
#> [1] "a"
#> [1] 3
```

`walk()` is generally not that useful compared to `walk2()` or `pwalk()`. For example, if you had a list of plots and a vector of file names, you could use `pwalk()` to save each file to the corresponding location on disk:

```
library(ggplot2)
plots <- mtcars %>%
  split(.$cyl) %>%
  map(~ggplot(., aes(mpg, wt)) + geom_point())
paths <- stringr::str_c(names(plots), ".pdf")

pwalk(list(paths, plots), ggsave, path = tempdir())
```

`walk()`, `walk2()` and `pwalk()` all invisibly return `.x`, the first argument. This makes them suitable for use in the middle of pipelines.

21.9 Other patterns of for loops

Purrr provides a number of other functions that abstract over other types of for loops. You'll use them less frequently than the map functions, but they're useful to know about. The goal here is to briefly illustrate each function, so hopefully it will come to mind if you see a similar problem in the future. Then you can go look up the documentation for more details.

21.9.1 Predicate functions

A number of functions work with **predicate** functions that return either a single `TRUE` or `FALSE`.

`keep()` and `discard()` keep elements of the input where the predicate is `TRUE` or `FALSE` respectively:

```
iris %>%
  keep(is.factor) %>%
  str()
#> 'data.frame':    150 obs. of  1 variable:
#>  $ Species: Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...

iris %>%
  discard(is.factor) %>%
  str()
#> 'data.frame':    150 obs. of  4 variables:
#>  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
#>  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
#>  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
#>  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
```

`some()` and `every()` determine if the predicate is true for any or for all of the elements.

```
x <- list(1:5, letters, list(10))

x %>%
  some(is_character)
#> [1] TRUE

x %>%
  every(is_vector)
#> [1] TRUE
```

`detect()` finds the first element where the predicate is true; `detect_index()` returns its position.

```
x <- sample(10)
x
#>  [1]  8  7  5  6  9  2 10  1  3  4

x %>%
  detect(~ . > 5)
#> [1] 8

x %>%
  detect_index(~ . > 5)
#> [1] 1
```

`head_while()` and `tail_while()` take elements from the start or end of a vector while a predicate is true:

```
x %>%
  head_while(~ . > 5)
#> [1] 8 7

x %>%
  tail_while(~ . > 5)
#> integer(0)
```

21.9.2 Reduce and accumulate

Sometimes you have a complex list that you want to reduce to a simple list by repeatedly applying a function that reduces a pair to a singleton. This is useful if you want to apply a two-table dplyr verb to multiple tables. For example, you might have a list of data frames, and you want to reduce to a single data frame by joining the elements together:

```
dfs <- list(
  age = tibble(name = "John", age = 30),
  sex = tibble(name = c("John", "Mary"), sex = c("M", "F")),
  trt = tibble(name = "Mary", treatment = "A")
)

dfs %>% reduce(full_join)
#> Joining, by = "name"
#> Joining, by = "name"
#> # A tibble: 2 x 4
#>   name    age sex   treatment
#>   <chr> <dbl> <chr> <chr>
#> 1 John     30 M     <NA>
#> 2 Mary     NA F     A
```

Or maybe you have a list of vectors, and want to find the intersection:

```
vs <- list(
  c(1, 3, 5, 6, 10),
  c(1, 2, 3, 7, 8, 10),
  c(1, 2, 3, 4, 8, 9, 10)
)

vs %>% reduce(intersect)
#> [1]  1  3 10
```

The reduce function takes a “binary” function (i.e. a function with two primary inputs), and applies it repeatedly to a list until there is only a single element left.

Accumulate is similar but it keeps all the interim results. You could use it to implement a cumulative sum:

```
x <- sample(10)
x
#>  [1]  6  9  8  5  2  4  7  1 10  3
x %>% accumulate(`+`)
#>  [1]  6 15 23 28 30 34 41 42 52 55
```

21.9.3 Exercises

1. Implement your own version of `every()` using a for loop. Compare it with `purrr::every()`. What does purrr's version do that your version doesn't?

2. Create an enhanced `col_sum()` that applies a summary function to every numeric column in a data frame.

3. A possible base R equivalent of `col_sum()` is:

```
col_sum3 <- function(df, f) {
  is_num <- sapply(df, is.numeric)
  df_num <- df[, is_num]

  sapply(df_num, f)
}
```

   But it has a number of bugs as illustrated with the following inputs:

```
df <- tibble(
  x = 1:3,
  y = 3:1,
  z = c("a", "b", "c")
)
# OK
col_sum3(df, mean)
# Has problems: don't always return numeric vector
col_sum3(df[1:2], mean)

col_sum3(df[1], mean)
col_sum3(df[0], mean)
```

   What causes the bugs?