9. Iteration

Applying operations to many elements with for loops and purrr
Dr. Paul Schmidt

Why Iteration?

Iteration means repeatedly applying the same operation to different elements: to multiple columns of a dataframe, to multiple files in a folder, or to multiple groups in your data. While the previous chapter showed how to encapsulate repeated code in functions, this chapter shows how to efficiently apply those functions to many elements.

R has a special feature: many operations are already **vectorized**. When you write `x * 2`, R automatically multiplies every value in `x` by 2 — no loop needed. In other languages this wouldn't be so automatic:

```r
x <- c(1, 2, 3, 4, 5)

# Vectorized - no explicit iteration needed
x * 2
```

```
[1]  2  4  6  8 10
```

```r
sqrt(x)
```

```
[1] 1.000000 1.414214 1.732051 2.000000 2.236068
```

But not everything can be vectorized so elegantly. When you want to read 50 CSV files, create 20 plots, or fit a model to each group of your data, you need explicit iteration. There are two main approaches: **for loops** (imperative) and **map functions** (functional).

Further Resources

This chapter is based on Chapter 26: Iteration from “R for Data Science” (2nd edition). For a more comprehensive treatment of purrr, we recommend Jenny Bryan's purrr Tutorial and the purrr documentation.

Implicit Iteration with across()

Before we get to explicit iteration, you should know: for many column-based operations you don't need loops or map functions at all. The `across()` function from dplyr handles this elegantly:

```r
# Without across() - repetitive
mtcars %>%
  summarize(
    mpg_mean = mean(mpg),
    hp_mean = mean(hp),
    wt_mean = mean(wt)
  )
  mpg_mean  hp_mean wt_mean
1 20.09062 146.6875 3.21725
```

```
# With across() - compact
mtcars %>%
  summarize(across(c(mpg, hp, wt), mean))
```

```
       mpg       hp      wt
1 20.09062 146.6875 3.21725
```

With `where()` you can select columns by type:

```
# Mean of all numeric columns
mtcars %>%
  summarize(across(where(is.numeric), \(x) mean(x, na.rm = TRUE)))
```

```
       mpg    cyl     disp       hp     drat      wt     qsec     vs      am
1 20.09062 6.1875 230.7219 146.6875 3.596563 3.21725 17.84875 0.4375 0.40625
    gear   carb
1 3.6875 2.8125
```

And with the `.names` argument you control the column names in the output:

```
mtcars %>%
  summarize(across(
    c(mpg, hp, wt),
    list(mean = \(x) mean(x, na.rm = TRUE),
         sd = \(x) sd(x, na.rm = TRUE)),
    .names = "{.col}_{.fn}"
  ))
```

```
  mpg_mean   mpg_sd  hp_mean    hp_sd wt_mean     wt_sd
1 20.09062 6.026948 146.6875 68.56287 3.21725 0.9784574
```

! Syntax Change in dplyr 1.1.0

The old syntax `across(a:b, mean, na.rm = TRUE)` is deprecated. Use an anonymous function instead: `across(a:b, \(x) mean(x, na.rm = TRUE))`.

Exercise: across() with Multiple Functions

Calculate the mean and standard deviation of all numeric columns in the `iris` dataset, grouped by `Species`. Use `across()` with the `.names` argument.

i Solution

```
iris %>%
  group_by(Species) %>%
  summarize(across(
    where(is.numeric),
    list(mean = \(x) mean(x), sd = \(x) sd(x)),
    .names = "{.col}_{.fn}"
  ))
```

```
# A tibble: 3 × 9
  Species    Sepal.Length_mean Sepal.Length_sd Sepal.Width_mean Sepal.Width_sd
  <fct>                  <dbl>           <dbl>            <dbl>          <dbl>
1 setosa                  5.01           0.352             3.43          0.379
2 versicolor              5.94           0.516             2.77          0.314
3 virginica               6.59           0.636             2.97          0.322
# ℹ 4 more variables: Petal.Length_mean <dbl>, Petal.Length_sd <dbl>,
#   Petal.Width_mean <dbl>, Petal.Width_sd <dbl>
```

for Loops

Basic Syntax

A for loop repeats a code block for each element of a vector or list:

```
# Simple for loop
for (i in 1:5) {
  print(glue::glue("Iteration {i}"))
}
```

```
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
```

The structure is always the same: `for (variable in sequence) { ... }`. In each iteration, `variable` takes the next value from `sequence`.

Storing Results

When you want to store results from a loop, **you should pre-allocate the output container**. This is important for performance:

```
# Good: Pre-allocate vector
n <- 10
results <- vector("double", n)

for (i in 1:n) {
  results[i] <- i^2
}

results
```

```
 [1]   1   4   9  16  25  36  49  64  81 100
```

```
# Bad: "Growing" the vector in the loop
results <- c()
```

```r
for (i in 1:n) {
  results <- c(results, i^2)
}
```

The second example is slow because R has to copy the entire vector with each `c()`. With large datasets this can make an enormous difference.

seq_along() Instead of 1:length()

It's better to use `seq_along()` instead of `1:length()` to avoid problems with empty vectors:

```r
x <- c("a", "b", "c")
y <- character(0)

# seq_along() is safe
for (i in seq_along(x)) {
  print(x[i])
}
```

```
[1] "a"
[1] "b"
[1] "c"
```

```r
seq_along(y)
```

```
integer(0)
```

```r
# 1:length() has a problem with empty vectors
1:length(y)
```

```
[1] 1 0
```

When for Loops Are Useful

for loops are especially useful when:

- The iteration has side effects (writing files, displaying plots)
- Each iteration depends on the result of the previous one
- The logic is very complex and you need maximum control

```r
# Iteration with dependency: Cumulative sum
x <- c(3, 1, 4, 1, 5)
cumsum_manual <- vector("double", length(x))
cumsum_manual[1] <- x[1]

for (i in 2:length(x)) {
  cumsum_manual[i] <- cumsum_manual[i - 1] + x[i]
}

cumsum_manual
```

```
[1]  3  4  8  9 14
```

```r
cumsum(x)
```

```
[1]  3  4  8  9 14
```

Exercise: Column Means with for Loop

Calculate the means of the first four columns of `mtcars` using a for loop. Store the results in a pre-allocated vector.

Solution

```
# Pre-allocate vector
means <- vector("double", 4)
names(means) <- names(mtcars)[1:4]

for (i in 1:4) {
  means[i] <- mean(mtcars[[i]])
}

means
```

```
      mpg       cyl      disp        hp 
 20.09062   6.18750 230.72188 146.68750 
```

The map Family from purrr

The Basic Principle

The `map()` function from the purrr package is the functional alternative to the for loop. The principle: you provide a list (or vector) and a function — `map()` applies the function to each element and returns a list.

```
# Apply a function to each element
numbers <- list(1:3, 4:6, 7:9)

map(numbers, mean)
```

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

[[2]]
[1] 5

[[3]]
[1] 8
```

The advantage over for loops: the code is more compact and expresses more clearly *what* happens (apply function to all elements), not *how* it happens (loop variable, index, etc.).

Type-Safe Variants

`map()` always returns a list. But often you know what type to expect. The variants `map_dbl()`, `map_chr()`, `map_lgl()`, and `map_int()` return vectors of the corresponding type — and throw an error if the result doesn't match:

```
# map() returns a list
map(numbers, mean)
```

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

[[2]]
[1] 5

[[3]]
[1] 8
```

```r
# map_dbl() returns a numeric vector
map_dbl(numbers, mean)
```

```
[1] 2 5 8
```

```r
# map_chr() returns a character vector
map_chr(numbers, \(x) glue::glue("Mean: {mean(x)}"))
```

```
[1] "Mean: 2" "Mean: 5" "Mean: 8"
```

```r
# Error when type doesn't match
map_chr(numbers, mean)
```

```
Warning: Automatic coercion from double to character was deprecated in purrr 1.0.0.
ℹ Please use an explicit call to `as.character()` within `map_chr()` instead.
```

```
[1] "2.000000" "5.000000" "8.000000"
```

Specifying Functions

There are several ways to specify the function to apply:

```r
# 1. Named function
map_dbl(numbers, mean)
```

```
[1] 2 5 8
```

```r
# 2. Anonymous function (modern syntax)
map_dbl(numbers, \(x) mean(x, na.rm = TRUE))
```

```
[1] 2 5 8
```

```r
# 3. Anonymous function (classic syntax)
map_dbl(numbers, function(x) mean(x, na.rm = TRUE))
```

```
[1] 2 5 8
```

```r
# 4. purrr formula (legacy, but still common)
map_dbl(numbers, ~ mean(.x, na.rm = TRUE))
```

```
[1] 2 5 8
```

The modern `\(x)` syntax (since R 4.1) is clearest. But you'll often see the formula syntax with `~` and `.x` in older code.

Extraction by Name or Position

A particularly practical feature: you can pass `map()` a string or number to extract elements:

```r
# List with named elements
people <- list(
  list(name = "Anna", age = 25),
  list(name = "Bob", age = 30),
  list(name = "Clara", age = 28)
)

# Extract by name
map_chr(people, "name")
```

```
[1] "Anna"  "Bob"   "Clara"
```

```r
# Extract by position
map_int(people, 2)
```

```
[1] 25 30 28
```

Exercise: Applying map_dbl()

Given a list of vectors. Calculate the range (maximum minus minimum) for each vector using `map_dbl()`.

```r
data <- list(
  a = c(1, 5, 3),
  b = c(10, 20, 15, 25),
  c = c(-5, 0, 5)
)
```

Solution

```r
map_dbl(data, \(x) max(x) - min(x))
```

```
 a  b  c 
 4 15 10 
```

```r
# Or with range()
map_dbl(data, \(x) diff(range(x)))
```

```
 a  b  c 
 4 15 10 
```

map2 and pmap: Multiple Inputs

Sometimes you need to iterate over multiple lists in parallel. `map2()` takes two lists, `pmap()` takes any number:

```r
# Two lists in parallel
x <- list(1, 2, 3)
y <- list(10, 20, 30)

map2_dbl(x, y, \(a, b) a + b)
```

```
[1] 11 22 33
```

```r
# Multiple lists with pmap()
params <- list(
  n = c(10, 20, 30),
  mean = c(0, 5, 10),
  sd = c(1, 2, 3)
)

set.seed(42)
pmap(params, \(n, mean, sd) rnorm(n, mean, sd)) %>%
  map_dbl(mean)
```

```
[1] 0.5472968 4.6584637 9.6342745
```

imap: With Index or Names

`imap()` is shorthand for `map2(x, names(x), ...)` — useful when you need both the value and the index/name:

```
x <- c(a = 10, b = 20, c = 30)

imap_chr(x, \(value, name) glue::glue("{name}: {value}"))
```

```
      a       b       c 
"a: 10" "b: 20" "c: 30" 
```

Exercise: Robust Division with map2()

Write a function `safe_divide()` that returns `NA` for division by zero (instead of `Inf`). Then apply it with `map2_dbl()` to two vectors.

```
numerator <- c(10, 20, 30, 40)
denominator <- c(2, 0, 5, 0)

# Desired result: c(5, NA, 6, NA)
```

Solution

```
safe_divide <- function(x, y) {
  if (y == 0) return(NA_real_)
  x / y
}

numerator <- c(10, 20, 30, 40)
denominator <- c(2, 0, 5, 0)

map2_dbl(numerator, denominator, safe_divide)
```

```
[1]  5 NA  6 NA
```

```
# Alternative with possibly()
map2_dbl(numerator, denominator, possibly(\(x, y) x / y, otherwise = NA_real_))
```

```
[1]   5 Inf   6 Inf
```

walk: Iteration for Side Effects

When you're not interested in the return value but in side effects (writing files, displaying plots), use `walk()` instead of `map()`. It invisibly returns the input, making it ideal for pipe chains:

```r
# Save multiple plots
plots <- list(
  ggplot(mtcars, aes(mpg)) + geom_histogram(),
  ggplot(mtcars, aes(hp)) + geom_histogram(),
  ggplot(mtcars, aes(wt)) + geom_histogram()
)

filenames <- c("mpg.png", "hp.png", "wt.png")

walk2(plots, filenames, \(plot, file) {
  ggsave(file, plot, width = 6, height = 4)
})
```

`walk()` exists in the same variants as `map()`: `walk2()`, `pwalk()`, `iwalk()`.

Robust Iteration: Catching Errors

The Problem

When iterating over many elements, a single error can abort the entire operation:

```r
# One element causes an error
inputs <- list(1, "a", 3)

map_dbl(inputs, log)
```

```
Error in `map_dbl()`:
ℹ In index: 2.
Caused by error:
! Nicht-numerisches Argument für mathematische Funktion
```

Element 2 is not a number, and the whole operation fails. With 1000 files this would be annoying — you want to know which files had problems while still processing the others.

safely(): Errors as Data

`safely()` is a “wrapper” (adverb) that modifies a function so it never aborts. Instead, it returns a list with `$result` and `$error`:

```r
safe_log <- safely(log)

safe_log(10)
```

```
$result
[1] 2.302585

$error
NULL
```

```r
safe_log("a")
```

```
$result
NULL

$error
<simpleError in .Primitive("log")(x, base): Nicht-numerisches Argument für
mathematische Funktion>
```

Combined with `map()`:

```
inputs <- list(1, "a", 3, -1)
results <- map(inputs, safe_log)
```

```
Warning in .Primitive("log")(x, base): NaNs wurden erzeugt
```

```
results
```

```
[[1]]
[[1]]$result
[1] 0

[[1]]$error
NULL


[[2]]
[[2]]$result
NULL

[[2]]$error
<simpleError in .Primitive("log")(x, base): Nicht-numerisches Argument für
mathematische Funktion>


[[3]]
[[3]]$result
[1] 1.098612

[[3]]$error
NULL


[[4]]
[[4]]$result
[1] NaN

[[4]]$error
NULL
```

With `transpose()` you can restructure the results:

```
results_t <- results %>% transpose()

results_t$result
```

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

[[2]]
NULL

[[3]]
[1] 1.098612

[[4]]
[1] NaN
```

```
results_t$error
```

```
[[1]]
NULL

[[2]]
<simpleError in .Primitive("log")(x, base): Nicht-numerisches Argument für
mathematische Funktion>
```

```
[[3]]
NULL

[[4]]
NULL
```

possibly(): Replace Errors with Default

Often a simpler approach suffices: replace errors with a default value. For this there's `possibly()`:

```
# Errors become NA
map_dbl(inputs, possibly(log, otherwise = NA_real_))
```

```
Warning in .Primitive("log")(x, base): NaNs wurden erzeugt
```

```
[1] 0.000000       NA 1.098612      NaN
```

This is especially practical with `map_dbl()`, since you get a vector directly instead of a nested list.

Inspecting Errors

After iteration you often want to know which elements failed:

```
# Which had errors?
results <- map(inputs, safe_log)
```

```
Warning in .Primitive("log")(x, base): NaNs wurden erzeugt
```

```
failed <- map_lgl(results, \(x) !is.null(x$error))
failed
```

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

```
# The failed inputs
inputs[failed]
```

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

```
# Only the successful results
successful <- map(results, "result") %>%
  compact() %>%
  map_dbl(identity)

successful
```

```
[1] 0.000000 1.098612      NaN
```

Exercise: Identifying Errors

Given a list of file paths, some of which don’t exist. Use `safely()` to read all existing files and find out which files were not found.

```r
# Prepare test data
temp_dir <- tempdir()

for (i in 1:2) {
  tibble(id = 1:3, value = rnorm(3)) %>%
    write_csv(file.path(temp_dir, glue::glue("test_{i}.csv")))
}

file_paths <- c(
  file.path(temp_dir, "test_1.csv"),
  "not_found.csv",
  file.path(temp_dir, "test_2.csv"),
  "also_missing.csv"
)
```

i Solution

```
safe_read <- safely(read_csv)

results <- file_paths %>%
  set_names() %>%
  map(\(f) safe_read(f, show_col_types = FALSE))

# Which succeeded?
success <- map_lgl(results, \(x) is.null(x$error))

cat("Successfully read:\n")
```

```
Successfully read:
```

```
names(results)[success]
```

```
[1] "C:\\Users\\BIOMAT~1\\AppData\\Local\\Temp\\RtmpqKrC2J/test_1.csv"
[2] "C:\\Users\\BIOMAT~1\\AppData\\Local\\Temp\\RtmpqKrC2J/test_2.csv"
```

```
cat("\nNot found:\n")
```

```

Not found:
```

```
names(results)[!success]
```

```
[1] "not_found.csv"    "also_missing.csv"
```

```
# Combine only successful data
data <- results[success] %>%
  map("result") %>%
  list_rbind(names_to = "source")

data
```

```
# A tibble: 6 × 3
  source                                                            id  value
  <chr>                                                          <dbl>  <dbl>
1 "C:\\Users\\BIOMAT~1\\AppData\\Local\\Temp\\RtmpqKrC2J/test_1.cs…     1 -0.367
2 "C:\\Users\\BIOMAT~1\\AppData\\Local\\Temp\\RtmpqKrC2J/test_1.cs…     2  0.185
3 "C:\\Users\\BIOMAT~1\\AppData\\Local\\Temp\\RtmpqKrC2J/test_1.cs…     3  0.582
4 "C:\\Users\\BIOMAT~1\\AppData\\Local\\Temp\\RtmpqKrC2J/test_2.cs…     1  1.40
5 "C:\\Users\\BIOMAT~1\\AppData\\Local\\Temp\\RtmpqKrC2J/test_2.cs…     2 -0.727
6 "C:\\Users\\BIOMAT~1\\AppData\\Local\\Temp\\RtmpqKrC2J/test_2.cs…     3  1.30
```

Practical Applications

Batch Import: Reading Multiple Files

A common use case: you have a folder full of CSV files and want to read and combine them all.

```
# Find all CSV files in folder
files <- list.files("data/", pattern = "\\.csv$", full.names = TRUE)

# Read all and combine into one dataframe
all_data <- files %>%
  map(\(f) read_csv(f, show_col_types = FALSE)) %>%
  list_rbind()
```

```r
# With filename as column
all_data <- files %>%
  set_names() %>%
  map(\(f) read_csv(f, show_col_types = FALSE)) %>%
  list_rbind(names_to = "source")
```

The trick with `set_names()` without an argument makes the file paths the names of the list, which then get transferred to the `source` column.

Batch Export: Writing Multiple Files

The counterpart: split data and write to separate files.

```r
# Split data by group
mtcars_split <- mtcars %>%
  group_by(cyl) %>%
  group_split()

# Generate filenames
filenames <- mtcars %>%
  distinct(cyl) %>%
  pull(cyl) %>%
  map_chr(\(x) glue::glue("output/mtcars_cyl{x}.csv"))

# Write all files
walk2(mtcars_split, filenames, \(data, file) {
  write_csv(data, file)
})
```

Exercise: Simulating Batch Import

First create three temporary CSV files, then read them with `map()` and combine them into a dataframe.

```r
# Create temporary files
batch_dir <- tempdir()

for (i in 1:3) {
  tibble(
    id = 1:5,
    value = rnorm(5),
    group = i
  ) %>%
    write_csv(file.path(batch_dir, glue::glue("batch_{i}.csv")))
}
```

i Solution

```
files <- list.files(batch_dir, pattern = "batch_.*\\.csv$", full.names = TRUE)

all_data <- files %>%
  set_names() %>%
  map(\(f) read_csv(f, show_col_types = FALSE)) %>%
  list_rbind(names_to = "source")

all_data
```

```
# A tibble: 15 × 4
   source                                                      id   value group
   <chr>                                                    <dbl>   <dbl> <dbl>
 1 "C:\\Users\\BIOMAT~1\\AppData\\Local\\Temp\\RtmpqKrC2J/b…     1  0.336      1
 2 "C:\\Users\\BIOMAT~1\\AppData\\Local\\Temp\\RtmpqKrC2J/b…     2  1.04       1
 3 "C:\\Users\\BIOMAT~1\\AppData\\Local\\Temp\\RtmpqKrC2J/b…     3  0.921      1
 4 "C:\\Users\\BIOMAT~1\\AppData\\Local\\Temp\\RtmpqKrC2J/b…     4  0.721      1
 5 "C:\\Users\\BIOMAT~1\\AppData\\Local\\Temp\\RtmpqKrC2J/b…     5 -1.04       1
 6 "C:\\Users\\BIOMAT~1\\AppData\\Local\\Temp\\RtmpqKrC2J/b…     1 -0.0902     2
 7 "C:\\Users\\BIOMAT~1\\AppData\\Local\\Temp\\RtmpqKrC2J/b…     2  0.624      2
 8 "C:\\Users\\BIOMAT~1\\AppData\\Local\\Temp\\RtmpqKrC2J/b…     3 -0.954      2
 9 "C:\\Users\\BIOMAT~1\\AppData\\Local\\Temp\\RtmpqKrC2J/b…     4 -0.543      2
10 "C:\\Users\\BIOMAT~1\\AppData\\Local\\Temp\\RtmpqKrC2J/b…     5  0.581      2
11 "C:\\Users\\BIOMAT~1\\AppData\\Local\\Temp\\RtmpqKrC2J/b…     1  0.768      3
12 "C:\\Users\\BIOMAT~1\\AppData\\Local\\Temp\\RtmpqKrC2J/b…     2  0.464      3
13 "C:\\Users\\BIOMAT~1\\AppData\\Local\\Temp\\RtmpqKrC2J/b…     3 -0.886      3
14 "C:\\Users\\BIOMAT~1\\AppData\\Local\\Temp\\RtmpqKrC2J/b…     4 -1.10       3
15 "C:\\Users\\BIOMAT~1\\AppData\\Local\\Temp\\RtmpqKrC2J/b…     5  1.51       3
```

Fitting Models to Groups

With `nest()` you can nest dataframes and then fit models per group:

```
# Nest data
mtcars_nested <- mtcars %>%
  group_by(cyl) %>%
  nest()

mtcars_nested
```

```
# A tibble: 3 × 2
# Groups:   cyl [3]
    cyl data
  <dbl> <list>
1     6 <tibble [7 × 10]>
2     4 <tibble [11 × 10]>
3     8 <tibble [14 × 10]>
```

```
# Fit model per group
mtcars_models <- mtcars_nested %>%
  mutate(
    model = map(data, \(df) lm(mpg ~ wt, data = df)),
    tidied = map(model, broom::tidy)
  )

# Extract results
mtcars_models %>%
  select(cyl, tidied) %>%
  unnest(tidied)
```

```
# A tibble: 6 × 6
# Groups:   cyl [3]
```

```
    cyl term         estimate std.error statistic    p.value
  <dbl> <chr>           <dbl>     <dbl>     <dbl>      <dbl>
1     6 (Intercept)     28.4      4.18       6.79 0.00105
2     6 wt              -2.78     1.33      -2.08 0.0918
3     4 (Intercept)     39.6      4.35       9.10 0.00000777
4     4 wt              -5.65     1.85      -3.05 0.0137
5     8 (Intercept)     23.9      3.01       7.94 0.00000405
6     8 wt              -2.19     0.739     -2.97 0.0118
```

Creating and Saving Multiple Plots

A complete example combining nest(), map(), and walk():

```
# Prepare data
plot_data <- mtcars %>%
  group_by(cyl) %>%
  nest() %>%
  mutate(
    plot = map2(data, cyl, \(df, cyl_val) {
      ggplot(df, aes(x = wt, y = mpg)) +
        geom_point() +
        geom_smooth(method = "lm", se = FALSE) +
        labs(title = glue::glue("{cyl_val} Cylinders: MPG vs. Weight"))
    }),
    filename = glue::glue("plots/scatter_cyl{cyl}.png")
  )

# Save all plots
walk2(plot_data$plot, plot_data$filename, \(p, f) {
  ggsave(f, p, width = 6, height = 4)
})
```

Exercise: Summary Statistics per Group

Use `nest()` and `map()` to calculate the mean and standard deviation of `mpg` for each value of `cyl` in the mtcars dataset. The result should be a tidy dataframe.

Solution

```
mtcars %>%
  group_by(cyl) %>%
  nest() %>%
  mutate(
    mean_mpg = map_dbl(data, \(df) mean(df$mpg)),
    sd_mpg = map_dbl(data, \(df) sd(df$mpg))
  ) %>%
  select(cyl, mean_mpg, sd_mpg)
```

```
# A tibble: 3 × 3
# Groups:   cyl [3]
    cyl mean_mpg sd_mpg
  <dbl>    <dbl>  <dbl>
1     6     19.7   1.45
2     4     26.7   4.51
3     8     15.1   2.56
```

List-Columns: Dataframes with Lists as Columns

The previous examples already used `nest()` to create “list-columns” — columns that contain lists instead of atomic vectors. This is a powerful concept that we’ll briefly introduce here.

```
# nest() creates a list-column
nested <- mtcars %>%
  group_by(cyl) %>%
  nest()

nested
```

```
# A tibble: 3 × 2
# Groups:   cyl [3]
    cyl data
  <dbl> <list>
1     6 <tibble [7 × 10]>
2     4 <tibble [11 × 10]>
3     8 <tibble [14 × 10]>
```

```
# The data column contains dataframes
nested$data[[1]]
```

```
# A tibble: 7 × 10
    mpg  disp    hp  drat    wt  qsec    vs    am  gear  carb
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1  21    160    110  3.9   2.62  16.5     0     1     4     4
2  21    160    110  3.9   2.88  17.0     0     1     4     4
3  21.4  258    110  3.08  3.22  19.4     1     0     3     1
4  18.1  225    105  2.76  3.46  20.2     1     0     3     1
5  19.2  168.   123  3.92  3.44  18.3     1     0     4     4
6  17.8  168.   123  3.92  3.44  18.9     1     0     4     4
7  19.7  145    175  3.62  2.77  15.5     0     1     5     6
```

With `unnest()` you can “unpack” list-columns:

```
nested %>%
  unnest(data)
```

```
# A tibble: 32 × 11
# Groups:   cyl [3]
     cyl   mpg  disp    hp  drat    wt  qsec    vs    am  gear  carb
   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1     6  21    160    110  3.9   2.62  16.5     0     1     4     4
 2     6  21    160    110  3.9   2.88  17.0     0     1     4     4
 3     6  21.4  258    110  3.08  3.22  19.4     1     0     3     1
 4     6  18.1  225    105  2.76  3.46  20.2     1     0     3     1
 5     6  19.2  168.   123  3.92  3.44  18.3     1     0     4     4
 6     6  17.8  168.   123  3.92  3.44  18.9     1     0     4     4
 7     6  19.7  145    175  3.62  2.77  15.5     0     1     5     6
 8     4  22.8  108     93  3.85  2.32  18.6     1     1     4     1
 9     4  24.4  147.    62  3.69  3.19  20       1     0     4     2
10     4  22.8  141.    95  3.92  3.15  22.9     1     0     4     2
# ℹ 22 more rows
```

List-columns are especially useful in combination with `map()` inside `mutate()`. They allow organizing complex workflows (like fitting many models) in a clear, tabular format.

i Further Reading

List-columns and advanced applications of `nest()` / `unnest()` are a large topic on their own. For more details we recommend Chapter 23: Hierarchical Data and Chapter 25: Many Models (from the 1st edition of R4DS).

for vs. map: Decision Guide

When should you use for loops, when map functions? Here's some guidance:

for loops are often better when:

- The logic is complex and you need maximum control
- Each iteration depends on the result of the previous one
- You're just learning to program and the explicit notation helps

map functions are often better when:

- You're applying the same operation to many elements (the standard case)
- You want to use the code in a pipe chain
- You want type safety (map_dbl, map_chr, etc.)
- You prefer the functional, declarative style

The most important advice: **use what you understand.** Both approaches are legitimate. for loops are not "bad" or "slow" (this prejudice is outdated). map functions are not "better", just different. Over time you'll develop a feel for which approach fits more naturally in which situation.

```r
# Same result, different styles

# for loop
results_for <- vector("double", 3)
for (i in 1:3) {
  results_for[i] <- mean(mtcars[[i]])
}
results_for
```

```
[1]  20.09062   6.18750 230.72188
```

```r
# map
results_map <- map_dbl(mtcars[1:3], mean)
results_map
```

```
      mpg       cyl      disp 
 20.09062   6.18750 230.72188 
```

Bibliography