

11. Iteration

Operationen auf viele Elemente anwenden mit for-Schleifen und purrr

Dr. Paul Schmidt

Warum Iteration?

Iteration bedeutet, dieselbe Operation wiederholt auf verschiedene Elemente anzuwenden: auf mehrere Spalten eines Dataframes, auf mehrere Dateien in einem Ordner, oder auf mehrere Gruppen in den Daten. Während Kapitel 9 zeigte, wie man wiederholten Code in Funktionen kapselt, zeigt dieses Kapitel, wie man diese Funktionen dann effizient auf viele Elemente anwendet.

R hat eine Besonderheit: Viele Operationen sind bereits **vektorisert**. Wenn man `x * 2` schreibt, multipliziert R automatisch jeden Wert in `x` mit 2 - man braucht keine Schleife. In anderen Sprachen wäre das nicht so selbstverständlich:

```
x <- c(1, 2, 3, 4, 5)
# Vektorisiert - kein explizites Iterieren nötig
x * 2
```

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

```
sqrt(x)
```

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

Aber nicht alles lässt sich so elegant vektorisieren. Wenn man 50 CSV-Dateien einlesen, 20 Plots erstellen, oder ein Modell auf jede Gruppe der Daten fitten möchte, braucht man explizite Iteration. Dafür gibt es zwei Hauptansätze: **for-Schleifen** (imperativ) und **map-Funktionen** (funktional).

💡 Weiterführende Ressourcen

Dieses Kapitel basiert auf dem Kapitel 26: Iteration aus "R for Data Science" (2. Auflage). Für eine ausführlichere Behandlung von purrr empfehlen wir Jenny Bryan's purrr Tutorial und die purrr-Dokumentation.

Implizite Iteration mit across()

Bevor wir zu expliziter Iteration kommen, sollte man wissen: Für viele Spalten-basierte Operationen braucht man gar keine Schleifen oder map-Funktionen. Die `across()`-Funktion aus dplyr erledigt das elegant:

```
# Ohne across() - repetitiv
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
```

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

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

Mit `where()` kann man Spalten nach Typ auswählen:

```
# Mittelwert aller numerischen Spalten
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
```

Und mit dem `.names`-Argument kontrolliert man die Spaltennamen im 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-Änderung in dplyr 1.1.0

Die alte Syntax `across(a:b, mean, na.rm = TRUE)` ist deprecated. Stattdessen eine anonyme Funktion verwenden: `across(a:b, \ (x) mean(x, na.rm = TRUE))`.

💡 Übung: across() mit mehreren Funktionen

Berechne für den `iris`-Datensatz den Mittelwert und die Standardabweichung aller numerischen Spalten, gruppiert nach `Species`. Verwende `across()` mit dem `.names`-Argument.

i Lösung

```
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-Schleifen

Grundsyntax

Eine for-Schleife wiederholt einen Codeblock für jedes Element eines Vektors oder einer Liste:

```
# Einfache for-Schleife
for (i in 1:5) {
  print(glue::glue("Durchlauf {i}"))
}
```

```
Durchlauf 1
Durchlauf 2
Durchlauf 3
Durchlauf 4
Durchlauf 5
```

Die Struktur ist immer gleich: `for (variable in sequenz) { ... }`. In jedem Durchlauf nimmt `variable` den nächsten Wert aus `sequenz` an.

Ergebnisse speichern

Wenn man Ergebnisse aus einer Schleife speichern möchte, **sollte man den Output-Container vorher anlegen**. Das ist wichtig für die Performance:

```
# Gut: Vektor vorher anlegen
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
```

```
# Schlecht: Vektor in der Schleife "wachsen" lassen
results <- c()
for (i in 1:n) {
  results <- c(results, i^2)
}
```

Das zweite Beispiel ist langsam, weil R bei jedem `c()` den gesamten Vektor kopieren muss. Bei großen Datenmengen kann das einen enormen Unterschied machen.

seq_along() statt 1:length()

Man verwendet besser `seq_along()` statt `1:length()`, um Probleme mit leeren Vektoren zu vermeiden:

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

# seq_along() ist sicher
for (i in seq_along(x)) {
  print(x[i])
}
```

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

```
seq_along(y)
```

```
integer(0)
```

```
# 1:length() hat ein Problem bei leeren Vektoren
1:length(y)
```

```
[1] 1 0
```

Wann for-Schleifen sinnvoll sind

for-Schleifen sind besonders nützlich, wenn:

- Die Iteration Seiteneffekte hat (Dateien schreiben, Plots anzeigen)
- Jede Iteration vom Ergebnis der vorherigen abhängt
- Die Logik sehr komplex ist und man maximale Kontrolle braucht

```
# Iteration mit Abhängigkeit: Kumulative Summe
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
```

```
cumsum(x)
```

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

💡 Übung: Spalten-Mittelwerte mit for-Schleife

Berechne die Mittelwerte der ersten vier Spalten von `mtcars` mit einer for-Schleife. Speichere die Ergebnisse in einem vorab angelegten Vektor.

i Lösung

```
# Vektor vorher anlegen
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

Die map-Familie aus purrr

Das Grundprinzip

Die `map()`-Funktion aus dem purrr-Paket ist die funktionale Alternative zur for-Schleife. Das Prinzip: Man gibt eine Liste (oder einen Vektor) und eine Funktion - `map()` wendet die Funktion auf jedes Element an und gibt eine Liste zurück.

```
# Eine Funktion auf jedes Element anwenden
zahlen <- list(1:3, 4:6, 7:9)

map(zahlen, mean)
```

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

[[2]]
[1] 5

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

Der Vorteil gegenüber for-Schleifen: Der Code ist kompakter und drückt klarer aus, *was* passiert (Funktion auf alle Elemente anwenden), nicht *wie* es passiert (Schleifenvariable, Index, etc.).

Typsichere Varianten

`map()` gibt immer eine Liste zurück. Oft weiß man aber, welchen Typ man erwartet. Die Varianten `map_dbl()`, `map_chr()`, `map_lgl()` und `map_int()` geben Vektoren des entsprechenden Typs zurück - und werfen einen Fehler, wenn das Ergebnis nicht passt:

```
# map() gibt Liste zurück
map(zahlen, mean)
```

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

[[2]]
[1] 5

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

```
# map_dbl() gibt numerischen Vektor zurück
map_dbl(zahlen, mean)
```

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

```
# map_chr() gibt Character-Vektor zurück
map_chr(zahlen, \(x) glue::glue("Mittelwert: {mean(x)}"))
```

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

```
# Fehler, wenn Typ nicht passt
map_chr(zahlen, mean)
```

```
Error in `map_chr()`:
! In index: 1.
Caused by error:
! Can't coerce from a number to a string.
```

Funktionen spezifizieren

Es gibt mehrere Wege, die anzuwendende Funktion zu spezifizieren:

```
# 1. Benannte Funktion
map_dbl(zahlen, mean)
```

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

```
# 2. Anonyme Funktion (moderne Syntax)
map_dbl(zahlen, \(x) mean(x, na.rm = TRUE))
```

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

```
# 3. Anonyme Funktion (klassische Syntax)
map_dbl(zahlen, function(x) mean(x, na.rm = TRUE))
```

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

```
# 4. purrr-Formel (legacy, aber noch verbreitet)
map_dbl(zahlen, ~ mean(.x, na.rm = TRUE))
```

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

Die moderne `\(x)`-Syntax (seit R 4.1) ist am klarsten. Die Formel-Syntax mit `~` und `.x` wird man aber in älterem Code oft sehen.

Extraktion per Name oder Position

Ein besonders praktisches Feature: Man kann `map()` einen String oder eine Zahl übergeben, um Elemente zu extrahieren:

```
# Liste mit benannten Elementen
personen <- list(
  list(name = "Anna", alter = 25),
  list(name = "Bob", alter = 30),
  list(name = "Clara", alter = 28)
)

# Per Name extrahieren
map_chr(personen, "name")
```

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

```
# Per Position extrahieren
map_int(personen, 2)
```

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

💡 Übung: map_dbl() anwenden

Gegeben ist eine Liste von Vektoren. Berechne für jeden Vektor die Spannweite (Maximum minus Minimum) mit `map_dbl()`.

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

i Lösung

```
map_dbl(daten, \(x) max(x) - min(x))
```

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

```
# Oder mit range()
map_dbl(daten, \(x) diff(range(x)))
```

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

map2 und pmap: Mehrere Inputs

Manchmal muss man über mehrere Listen parallel iterieren. `map2()` nimmt zwei Listen, `pmap()` nimmt beliebig viele:

```
# Zwei Listen parallel
x <- list(1, 2, 3)
y <- list(10, 20, 30)

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

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

```
# Mehrere Listen mit 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: Mit Index oder Namen

`imap()` ist eine Kurzform für `map2(x, names(x), ...)` - nützlich, wenn man sowohl den Wert als auch den Index/Namen braucht:

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

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

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

💡 Übung: Robuste Division mit map2()

Schreibe eine Funktion `safe_divide()`, die bei Division durch 0 `NA` zurückgibt (statt `Inf`). Wende sie dann mit `map2_dbl()` auf zwei Vektoren an.

```
zaehler <- c(10, 20, 30, 40)
nenner <- c(2, 0, 5, 0)

# Gewünschtes Ergebnis: c(5, NA, 6, NA)
```

i Lösung

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

zaehler <- c(10, 20, 30, 40)
nenner <- c(2, 0, 5, 0)

map2_dbl(zaehler, nenner, safe_divide)
```

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

```
# Alternative mit possibly()
map2_dbl(zaehler, nenner, possibly(\(x, y) x / y, otherwise = NA_real_))
```

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

walk: Iteration fuer Seiteneffekte

Wenn man nicht am Rückgabewert interessiert ist, sondern an Seiteneffekten (Dateien schreiben, Plots anzeigen), verwendet man `walk()` statt `map()`. Es gibt unsichtbar den Input zurück, was es ideal für Pipe-Ketten macht:

```
# Mehrere Plots speichern
plots <- list(
  ggplot(mtcars, aes(mpg)) + geom_histogram(),
  ggplot(mtcars, aes(hp)) + geom_histogram(),
  ggplot(mtcars, aes(wt)) + geom_histogram()
)

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

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

`walk()` existiert in denselben Varianten wie `map()`: `walk2()`, `pwalk()`, `iwalk()`.

Robuste Iteration: Fehler abfangen

Das Problem

Bei Iteration über viele Elemente kann ein einzelner Fehler die gesamte Operation abbrechen:

```
# Ein Element verursacht Fehler
inputs <- list(1, "a", 3)

map_dbl(inputs, log)
```

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

Element 2 ist keine Zahl, und die ganze Operation schlägt fehl. Bei 1000 Dateien wäre das ärgerlich - man will wissen, welche Dateien Probleme hatten, aber trotzdem die anderen verarbeiten.

safely(): Fehler als Daten

`safely()` ist ein "Wrapper" (Adverb), der eine Funktion so modifiziert, dass sie nie abbricht. Stattdessen gibt sie eine Liste mit `$result` und `$error` zurück:

```
safe_log <- safely(log)

safe_log(10)
```

```
$result
[1] 2.302585

$error
NULL
```

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

```
$result
NULL

$error
<simpleError in .f(...): Nicht-numerisches Argument für mathematische Funktion>
```

Kombiniert mit `map()` :

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

```
Warning in .f(...): NaNs wurden erzeugt
```

```
results
```

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

[[1]]$error
NULL

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

[[2]]$error
<simpleError in .f(...): Nicht-numerisches Argument für mathematische Funktion>

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

[[3]]$error
NULL

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

[[4]]$error
NULL
```

Mit `transpose()` kann man die Ergebnisse umstrukturieren:

```
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 .f(...): Nicht-numerisches Argument für mathematische Funktion>

[[3]]
NULL

[[4]]
NULL
```

possibly(): Fehler mit Default ersetzen

Oft reicht ein einfacherer Ansatz: Fehler durch einen Default-Wert ersetzen. Dafür gibt es

```
possibly() :
```

```
# Fehler werden zu NA
map_dbl(inputs, possibly(log, otherwise = NA_real_))
```

```
Warning in .f(...): NaNs wurden erzeugt
```

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

Das ist besonders praktisch mit `map_dbl()`, da man direkt einen Vektor bekommt statt einer verschachtelten Liste.

Fehler inspizieren

Nach der Iteration möchte man oft wissen, welche Elemente fehlgeschlagen sind:

```
# Welche hatten Fehler?
results <- map(inputs, safe_log)
```

```
Warning in .f(...): NaNs wurden erzeugt
```

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

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

```
# Die fehlerhaften Inputs
inputs[fehlgeschlagen]
```

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

```
# Nur die erfolgreichen Ergebnisse
erfolgreich <- map(results, "result") %>%
  compact() %>%
  map_dbl(identity)

erfolgreich
```

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

💡 Übung: Fehler identifizieren

Gegeben ist eine Liste von Dateipfaden, von denen einige nicht existieren. Verwende `safely()`, um alle vorhandenen Dateien einzulesen und herauszufinden, welche Dateien nicht gefunden wurden.

```
# Testdaten vorbereiten
temp_dir <- tempdir()

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

dateipfade <- c(
  file.path(temp_dir, "test_1.csv"),
  "nicht_vorhanden.csv",
  file.path(temp_dir, "test_2.csv"),
  "auch_nicht_da.csv"
)
```

i Lösung

```
safe_read <- safely(read_csv)

ergebnisse <- dateipfade %>%
  set_names() %>%
  map(\(f) safe_read(f, show_col_types = FALSE))

# Welche haben funktioniert?
erfolg <- map_lgl(ergebnisse, \(x) is.null(x$error))

cat("Erfolgreich gelesen:\n")
```

Erfolgreich gelesen:

```
names(ergebnisse)[erfolg]
```

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

```
cat("\nNicht gefunden:\n")
```

Nicht gefunden:

```
names(ergebnisse)[!erfolg]
```

```
[1] "nicht_vorhanden.csv" "auch_nicht_da.csv"
```

```
# Nur die erfolgreichen Daten kombinieren
daten <- ergebnisse[erfolg] %>%
  map("result") %>%
  list_rbind(names_to = "quelle")

daten
```

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

Praktische Anwendungen

Batch-Import: Mehrere Dateien einlesen

Ein häufiger Anwendungsfall: Man hat einen Ordner voller CSV-Dateien und möchte alle einlesen und kombinieren.

```
# Alle CSV-Dateien im Ordner finden
dateien <- list.files("data/", pattern = "\\..csv$", full.names = TRUE)

# Alle einlesen und zu einem Dataframe kombinieren
alle_daten <- dateien %>%
  map(\(f) read_csv(f, show_col_types = FALSE)) %>%
  list_rbind()
```

```
# Mit Dateinamen als Spalte
alle_daten <- dateien %>%
  set_names() %>%
  map(\(f) read_csv(f, show_col_types = FALSE)) %>%
  list_rbind(names_to = "quelle")
```

Der Trick mit `set_names()` ohne Argument macht die Dateipfade zu Namen der Liste, die dann in die `quelle`-Spalte übernommen werden.

Batch-Export: Mehrere Dateien schreiben

Das Gegenstück: Daten aufteilen und in separate Dateien schreiben.

```
# Daten nach Gruppe aufteilen
mtcars_split <- mtcars %>%
  group_by(cyl) %>%
  group_split()

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

# Alle Dateien schreiben
walk2(mtcars_split, dateinamen, \(daten, datei) {
  write_csv(daten, datei)
})
```

💡 Übung: Batch-Import simulieren

Erstelle erst drei temporäre CSV-Dateien, lies sie dann mit `map()` ein und kombiniere sie zu einem Dataframe.

```
# Temporäre Dateien erstellen
batch_dir <- tempdir()

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

i Lösung

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

alle_daten <- dateien %>%
  set_names() %>%
  map(\(f) read_csv(f, show_col_types = FALSE)) %>%
  list_rbind(names_to = "quelle")

alle_daten
```

```
# A tibble: 15 × 4
  quelle id wert gruppe
  <chr> <dbl> <dbl> <dbl>
1 "C:\\Users\\PAULSC~1\\AppData\\Local\\Temp\\RtmpAPMRaq/..." 1 0.336 1
2 "C:\\Users\\PAULSC~1\\AppData\\Local\\Temp\\RtmpAPMRaq/..." 2 1.04 1
3 "C:\\Users\\PAULSC~1\\AppData\\Local\\Temp\\RtmpAPMRaq/..." 3 0.921 1
4 "C:\\Users\\PAULSC~1\\AppData\\Local\\Temp\\RtmpAPMRaq/..." 4 0.721 1
5 "C:\\Users\\PAULSC~1\\AppData\\Local\\Temp\\RtmpAPMRaq/..." 5 -1.04 1
6 "C:\\Users\\PAULSC~1\\AppData\\Local\\Temp\\RtmpAPMRaq/..." 1 -0.0902 2
7 "C:\\Users\\PAULSC~1\\AppData\\Local\\Temp\\RtmpAPMRaq/..." 2 0.624 2
8 "C:\\Users\\PAULSC~1\\AppData\\Local\\Temp\\RtmpAPMRaq/..." 3 -0.954 2
9 "C:\\Users\\PAULSC~1\\AppData\\Local\\Temp\\RtmpAPMRaq/..." 4 -0.543 2
10 "C:\\Users\\PAULSC~1\\AppData\\Local\\Temp\\RtmpAPMRaq/..." 5 0.581 2
11 "C:\\Users\\PAULSC~1\\AppData\\Local\\Temp\\RtmpAPMRaq/..." 1 0.768 3
12 "C:\\Users\\PAULSC~1\\AppData\\Local\\Temp\\RtmpAPMRaq/..." 2 0.464 3
13 "C:\\Users\\PAULSC~1\\AppData\\Local\\Temp\\RtmpAPMRaq/..." 3 -0.886 3
14 "C:\\Users\\PAULSC~1\\AppData\\Local\\Temp\\RtmpAPMRaq/..." 4 -1.10 3
15 "C:\\Users\\PAULSC~1\\AppData\\Local\\Temp\\RtmpAPMRaq/..." 5 1.51 3
```

Modelle auf Gruppen fitten

Mit `nest()` kann man Dataframes verschachteln und dann Modelle pro Gruppe fitten:

```
# Daten verschachteln
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]>
```

```
# Modell pro Gruppe fitten
mtcars_models <- mtcars_nested %>%
  mutate(
    model = map(data, \(df) lm(mpg ~ wt, data = df)),
    tidied = map(model, broom::tidy)
  )

# Ergebnisse extrahieren
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

Mehrere Plots erstellen und speichern

Ein vollständiges Beispiel, das `nest()`, `map()` und `walk()` kombiniert:

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

# Alle Plots speichern
walk2(plot_data$plot, plot_data$filename, \(p, f) {
  ggsave(f, p, width = 6, height = 4)
})
```

💡 Übung: Summary-Statistiken pro Gruppe

Verwende `nest()` und `map()`, um für jeden Wert von `cyl` im `mtcars`-Datensatz

Mittelwert und Standardabweichung von `mpg` zu berechnen. Das Ergebnis sollte ein übersichtlicher Dataframe sein.

i Lösung

```
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 mit Listen als Spalten

Die vorherigen Beispiele haben bereits `nest()` verwendet, um "List-Columns" zu erstellen - Spalten, die Listen statt atomarer Vektoren enthalten. Das ist ein mächtiges Konzept, das hier kurz vorgestellt wird.

```
# nest() erstellt eine 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]>
```

```
# Die data-Spalte enthält 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
```

Mit `unnest()` kann man List-Columns wieder "auspacken":

```
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 sind besonders nützlich in Kombination mit `map()` innerhalb von `mutate()`. Sie ermöglichen es, komplexe Workflows (wie das Fitten vieler Modelle) in einem übersichtlichen, tabellarischen Format zu organisieren.

i Weiterführend

List-Columns und fortgeschrittene Anwendungen von `nest()` / `unnest()` sind ein eigenes großes Thema. Für mehr Details empfehlen wir das Kapitel 23: Hierarchical Data und das Kapitel 25: Many Models (aus der 1. Auflage von R4DS).

for vs. map: Entscheidungshilfe

Wann sollte man for-Schleifen verwenden, wann map-Funktionen? Hier eine Orientierung:

for-Schleifen sind oft besser, wenn:

- Die Logik komplex ist und man maximale Kontrolle braucht
- Jede Iteration vom Ergebnis der vorherigen abhängt
- Man gerade erst programmieren lernt und die explizite Schreibweise hilft

map-Funktionen sind oft besser, wenn:

- Man dieselbe Operation auf viele Elemente anwendet (der Standardfall)
- Man den Code in einer Pipe-Kette verwenden möchte
- Man Typsicherheit will (`map_dbl`, `map_chr`, etc.)
- Man die funktionale, deklarative Schreibweise bevorzugt

Der wichtigste Ratschlag: **Das verwenden, was man versteht.** Beide Ansätze sind legitim. for-Schleifen sind nicht “schlecht” oder “langsam” (dieses Vorurteil ist veraltet). map-Funktionen sind nicht “besser”, nur anders. Mit der Zeit entwickelt man ein Gefühl dafür, welcher Ansatz in welcher Situation natürlicher passt.

```
# Dasselbe Ergebnis, unterschiedliche Stile

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

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

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

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

Bibliography