8. Kontrollfluss und Listen

Entscheidungen im Code treffen und mit flexiblen Datenstrukturen arbeiten

Dr. Paul Schmidt

Um alle in diesem Kapitel verwendeten Pakete zu installieren und zu laden, führt man folgenden Code aus:

```
for (pkg in c("tidyverse")) {
  if (!require(pkg, character.only = TRUE)) install.packages(pkg)
}

library(tidyverse)
```

Einleitung

Bis zu diesem Punkt war der R-Code in diesem Workshop weitgehend linear: eine Zeile wird nach der anderen ausgeführt, von oben nach unten. Das funktioniert gut für einfache Datenaufbereitung, aber reale Analyseaufgaben erfordern oft Entscheidungen. Soll der Code Pfad A oder Pfad B nehmen? Soll eine Beobachtung als “hoch” oder “niedrig” eingestuft werden? Soll eine Warnung ausgegeben werden, wenn etwas Unerwartetes passiert?

Kontrollfluss ist der Oberbegriff für Sprachkonstrukte, die es dem Code ermöglichen, Entscheidungen zu treffen und zwischen Alternativen zu wählen. Das grundlegendste davon ist das `if`-Statement. Im Alltag denkt jeder Analyst bereits in Kontrollfluss: “Wenn der p-Wert unter 0.05 liegt, berichte das Ergebnis als signifikant; andernfalls berichte es als nicht signifikant.” Genau diese Logik in R-Code zu übersetzen, ist das Thema dieses Kapitels.

Dieses Kapitel führt außerdem **Listen** ein — R’s flexibelste Datenstruktur. Während Vektoren erfordern, dass alle Elemente denselben Typ haben, können Listen alles enthalten: Zahlen, Strings, Dataframes, sogar andere Listen. Das Verständnis von Listen ist essenziell, weil viele R-Funktionen ihre Ergebnisse als Listen zurückgeben (z.B. `lm()` und `t.test()`) und weil Listen das Rückgrat der Iteration mit `purrr::map()` in Kapitel 11 bilden. Iterativer Kontrollfluss (`for`- und `while`-Schleifen) wird hier nicht behandelt — er hat ein eigenes Kapitel (Kapitel 11).

Zusammen bilden Kontrollfluss und Listen die Grundlage für das Schreiben eigener Funktionen (Kapitel 9) und die Iteration über Daten (Kapitel 11).

if und else

Einfaches if

Die einfachste Form des Kontrollflusses ist ein einzelnes `if`-Statement. Es prüft eine Bedingung und führt, wenn die Bedingung `TRUE` ist, den Code innerhalb der geschweiften Klammern aus:

```
temp_mean <- mean(airquality$Temp)
temp_mean
```

```
[1] 77.88235
```

```r
if (temp_mean > 70) {
  message("The average temperature is above 70 degrees Fahrenheit.")
}
```

```
The average temperature is above 70 degrees Fahrenheit.
```

Wenn die Bedingung `FALSE` ist, passiert nichts — der Code innerhalb der Klammern wird einfach übersprungen.

if / else

Häufiger möchte man eine Sache tun, wenn die Bedingung wahr ist, und etwas anderes, wenn sie falsch ist. Hier kommt `else` ins Spiel:

```r
ozone_mean <- mean(airquality$Ozone, na.rm = TRUE)

if (ozone_mean > 50) {
  message(glue::glue("Mean ozone ({round(ozone_mean, 1)} ppb) is elevated."))
} else {
  message(glue::glue("Mean ozone ({round(ozone_mean, 1)} ppb) is within normal
range."))
}
```

```
Mean ozone (42.1 ppb) is within normal range.
```

Zu beachten ist, dass das Schlüsselwort `else` in der **gleichen Zeile** wie die schließende Klammer `}` des `if`-Blocks stehen muss. Steht es in der nächsten Zeile, entsteht ein Syntaxfehler, weil R denkt, das `if`-Statement sei bereits beendet.

if / else if / else

Wenn es mehr als zwei mögliche Ergebnisse gibt, können zusätzliche Bedingungen mit `else if` verkettet werden:

```r
wind_mean <- mean(airquality$Wind)

if (wind_mean > 12) {
  category <- "Windy"
} else if (wind_mean > 8) {
  category <- "Moderate"
} else {
  category <- "Calm"
}

message(glue::glue("Average wind: {round(wind_mean, 1)} mph => {category}"))
```

```
Average wind: 10 mph => Moderate
```

R wertet die Bedingungen von oben nach unten aus und betritt den ersten Zweig, dessen Bedingung `TRUE` ist. Sobald ein Zweig betreten wird, werden alle verbleibenden Zweige übersprungen.

Skalare Bedingungen: && und ||

Ein wichtiges Detail ist, dass `if` einen **einzelnen** logischen Wert erwartet — `TRUE` oder `FALSE`, Länge 1. Die Übergabe eines Vektors mit einer Länge größer als 1 erzeugt eine Warnung und verwendet nur das erste Element, was fast nie das beabsichtigte Verhalten ist.

Beim Kombinieren von Bedingungen innerhalb von `if` verwendet man die **Short-Circuit**-Operatoren `&&` (und) und `||` (oder). Diese werten von links nach rechts aus und stoppen, sobald das Ergebnis feststeht. Ihre vektorisierten Gegenstücke `&` und `|` sind für elementweise Operationen auf Vektoren gedacht und sollten nicht innerhalb von `if` verwendet werden:

```
x <- 15

# Correct: scalar operators for if
if (x > 10 && x < 20) {
  message("x is between 10 and 20")
}
```

```
x is between 10 and 20
```

```
# & is vectorized - works element-wise on vectors
c(TRUE, FALSE, TRUE) & c(TRUE, TRUE, FALSE)
```

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

Die Faustregel ist einfach: `&&` / `||` innerhalb von `if()` verwenden und `&` / `|` in vektorisierten Operationen wie `filter()` oder `ifelse()`.

ifelse() und case_when()

Vektorisierte Entscheidungen mit ifelse()

Das `if` / `else`-Konstrukt verarbeitet eine einzelne Bedingung. Aber was, wenn man jede Zeile eines Dataframes klassifizieren muss? Dafür bietet R `ifelse()`, das vektorisiert ist — es prüft jedes Element eines Vektors und gibt einen Wert für den `TRUE`-Fall oder den `FALSE`-Fall zurück:

```
aq <- airquality %>%
  mutate(temp_class = ifelse(Temp >= 80, "Hot", "Not hot"))

aq %>%
  count(temp_class)
```

```
  temp_class  n
1        Hot 73
2    Not hot 80
```

`ifelse()` funktioniert, wird aber umständlich, wenn es mehr als zwei Kategorien gibt, weil die Aufrufe verschachtelt werden müssen:

```
aq <- airquality %>%
  mutate(
    temp_class = ifelse(Temp >= 85, "Hot",
                        ifelse(Temp >= 70, "Warm", "Cold"))
  )

aq %>%
  count(temp_class)
```

```
  temp_class  n
1       Cold 32
2        Hot 39
3       Warm 82
```

Übersichtlichere Multi-Kategorie-Logik mit case_when()

Die Funktion `dplyr::case_when()` bietet eine deutlich lesbarere Syntax für mehrere Bedingungen. Jede Zeile enthält eine Bedingung auf der linken Seite von `~` und den zugehörigen Wert auf der rechten:

```
aq <- airquality %>%
  mutate(
    temp_class = case_when(
      Temp >= 85 ~ "Hot",
      Temp >= 70 ~ "Warm",
      .default = "Cold"
    )
  )

aq %>%
  count(temp_class)
```

```
  temp_class  n
1       Cold 32
2        Hot 39
3       Warm 82
```

Genau wie bei `if` / `else if`-Ketten wertet `case_when()` die Bedingungen von oben nach unten aus und weist den Wert der ersten zutreffenden Bedingung zu. Das Argument `.default = "Cold"` dient als “else”-Fall, der alles auffängt, was keiner früheren Bedingung entsprochen hat.

> **Hinweis**
>
> In älterem Code findet man häufig `TRUE ~ "Cold"` statt `.default = "Cold"` als Auffangklausel. Beide Varianten funktionieren, aber `.default` ist seit dplyr 1.1.0 der empfohlene Ansatz, da er expliziter ist und subtile Probleme bei `NA`-Bedingungen vermeidet.

Wann welches verwenden

Die drei Ansätze dienen unterschiedlichen Zwecken:

- **`if` / `else`**: Für Kontrollfluss-Entscheidungen, die die Programmlogik beeinflussen (z.B. Auswahl der durchzuführenden Analyse). Arbeitet mit einer einzelnen skalaren Bedingung.
- **`ifelse()`**: Für einfache vektorisierte Zwei-Kategorie-Operationen. Kann auch außerhalb des tidyverse verwendet werden.
- **`case_when()`**: Für vektorisierte Multi-Kategorie-Operationen innerhalb von `mutate()`. Übersichtlicher und sicherer als verschachteltes `ifelse()`.

Übung: Luftqualität kategorisieren

Verwende den `airquality`-Datensatz:

1. Erstelle eine Spalte `temp_category` mit drei Stufen: “Cold” (unter 70), “Warm” (70 bis 84) und “Hot” (85 und höher) mit `case_when()`.
2. Erstelle eine Spalte `wind_class` mit zwei Stufen: “Breezy” (Wind >= 10) und “Calm” (Wind < 10) mit `case_when()`.
3. Zähle die Kombinationen von `temp_category` und `wind_class`.

Lösung

```
aq_classified <- airquality %>%
  mutate(
    temp_category = case_when(
      Temp >= 85 ~ "Hot",
      Temp >= 70 ~ "Warm",
      .default = "Cold"
    ),
    wind_class = case_when(
      Wind >= 10 ~ "Breezy",
      .default = "Calm"
    )
  )

aq_classified %>%
  count(temp_category, wind_class)
```

```
  temp_category wind_class  n
1          Cold     Breezy 22
2          Cold       Calm 10
3           Hot     Breezy  9
4           Hot       Calm 30
5          Warm     Breezy 41
6          Warm       Calm 41
```

switch()

Wenn eine Entscheidung davon abhängt, einen einzelnen Wert mit mehreren diskreten Optionen abzugleichen, bietet `switch()` eine kompakte Alternative zu langen `if` / `else if` -Ketten. Es nimmt einen Ausdruck (typischerweise einen Character-String) und gibt den Wert zurück, der dem passenden Fall zugeordnet ist:

```
describe_month <- function(month_num) {
  season <- switch(as.character(month_num),
    "5"  = "Spring",
    "6"  = "Early Summer",
    "7"  = "Summer",
    "8"  = "Late Summer",
    "9"  = "Early Fall",
    "Unknown season"
  )
  glue::glue("Month {month_num}: {season}")
}
```

```
describe_month(7)
```

```
Month 7: Summer
```

```
describe_month(5)
```

```
Month 5: Spring
```

```
describe_month(12)
```

```
Month 12: Unknown season
```

Der letzte unbenannte Wert (`"Unknown season"`) dient als Standardwert, der zurückgegeben wird, wenn kein Fall passt. Das macht `switch()` besonders nützlich innerhalb von Funktionen, in denen ein Argument zwischen einer Handvoll vordefinierter Optionen auswählt:

```
summarize_stat <- function(x, stat = "mean") {
  switch(stat,
    "mean"   = mean(x, na.rm = TRUE),
    "median" = median(x, na.rm = TRUE),
    "sd"     = sd(x, na.rm = TRUE),
    stop(glue::glue("Unknown statistic: '{stat}'"))
  )
}

summarize_stat(airquality$Temp, "mean")
```

```
[1] 77.88235
```

```
summarize_stat(airquality$Temp, "median")
```

```
[1] 79
```

Der `stop()` -Aufruf in der Standardposition ist ein gängiges Muster: Er erzeugt einen informativen Fehler, wenn ein unerwarteter Wert übergeben wird, anstatt stillschweigend `NULL` zurückzugeben.

Was sind Listen?

Atomare Vektoren: Nur ein Typ

Bevor Listen eingeführt werden, lohnt es sich, zu rekapitulieren, wie atomare Vektoren funktionieren. Ein atomarer Vektor ist eine Folge von Werten, die alle denselben Typ teilen:

```
num_vec <- c(1.5, 2.3, 4.1)       # double
int_vec <- 1:5                     # integer
chr_vec <- c("a", "b", "c")       # character
log_vec <- c(TRUE, FALSE, TRUE)    # logical
```

Wenn Typen in einem einzelnen Vektor gemischt werden, konvertiert R stillschweigend alles zum allgemeinsten Typ:

```
mixed <- c(1, "two", TRUE)
mixed
```

```
[1] "1"    "two"  "TRUE"
```

```
class(mixed)
```

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

Alles wurde zu Character, weil Character der allgemeinste Typ ist. Diese “alle Elemente müssen übereinstimmen”-Einschränkung macht Vektoren effizient, begrenzt aber auch, was ein einzelner Vektor enthalten kann.

Listen: Jeder Typ, jede Länge

Eine Liste hebt diese Einschränkung auf. Jedes Element einer Liste kann einen anderen Typ und eine andere Länge haben — eine Zahl, ein Character-Vektor, ein ganzer Dataframe oder sogar eine weitere Liste:

```
my_list <- list(
  numbers = c(10, 20, 30),
  greeting = "Hello",
  flag = TRUE,
  data = head(mtcars, 3)
)

my_list
```

```
$numbers
[1] 10 20 30

$greeting
[1] "Hello"

$flag
[1] TRUE

$data
               mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4     21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710    22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
```

Eine hilfreiche Analogie ist ein **Zug**: Jeder Waggon (Element) kann völlig unterschiedliche Fracht transportieren. Ein Vektor hingegen gleicht eher einem Förderband, auf dem jedes Element dieselbe Form haben muss.

Benannte Elemente werden dringend empfohlen, weil sie den Code selbstdokumentierend machen. Unbenannte Listen funktionieren aber ebenfalls:

```
unnamed <- list(1:3, "text", FALSE)
unnamed
```

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

[[2]]
[1] "text"

[[3]]
[1] FALSE
```

Zugriff auf Listenelemente

Daten aus einer Liste zu extrahieren ist eine der häufigsten Verwirrungsquellen für R-Anfänger. Es gibt drei Operatoren, und die Unterscheidung zwischen ihnen ist wichtig.

[[— Ein einzelnes Element extrahieren

Doppelte Klammern `[[` extrahieren ein einzelnes Element aus der Liste. Das Ergebnis ist das Element selbst, keine Liste:

```
my_list[["numbers"]]
```

```
[1] 10 20 30
```

```
class(my_list[["numbers"]])
```

```
[1] "numeric"
```

Mit der Zug-Analogie: `[[` öffnet einen Waggon und nimmt die Fracht heraus.

[— Eine Teil-Liste extrahieren

Einfache Klammern `[` extrahieren eine Teilmenge der Liste. Das Ergebnis ist immer eine **Liste**, auch wenn sie nur ein Element enthält:

```
my_list["numbers"]
```

```
$numbers
[1] 10 20 30
```

```
class(my_list["numbers"])
```

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

Mit der Zug-Analogie: `[` koppelt einen oder mehrere Waggons ab, behält sie aber als (kleineren) Zug.

$ — Kurzschreibweise für benannte Elemente

Das Dollarzeichen `$` ist eine praktische Kurzform für `[[` bei benannten Elementen:

```
my_list$greeting
```

```
[1] "Hello"
```

Es ist äquivalent zu `my_list[["greeting"]]`, erfordert aber weniger Tipparbeit. Der `$`-Operator unterstützt auch Partial Matching (z.B. würde `my_list$gre` funktionieren), wobei man sich darauf nicht verlassen sollte, da es den Code fragil macht.

Der entscheidende Unterschied

Die Unterscheidung zwischen `[` und `[[` bereitet vielen Anfängern Schwierigkeiten. Ein visueller Vergleich hilft:

```r
# [[ returns the element (a numeric vector)
str(my_list[["numbers"]])
```

```
 num [1:3] 10 20 30
```

```r
# [ returns a list containing that element
str(my_list["numbers"])
```

```
List of 1
 $ numbers: num [1:3] 10 20 30
```

Wenn man mit den extrahierten Daten rechnen möchte (z.B. den Mittelwert der Zahlen berechnen), braucht man fast immer `[[` oder `$`. Wenn man eine Teilmenge einer Liste an eine andere Funktion übergeben möchte, die eine Liste erwartet, verwendet man `[`.

Übung: Eine Liste erstellen und darauf zugreifen

1. Erstelle eine Liste namens `my_data` mit drei benannten Elementen:
   - `measurements`: der numerische Vektor `c(4.2, 5.1, 3.8, 6.0, 4.7)`
   - `cars_sample`: die ersten 5 Zeilen von `mtcars`
   - `note`: der Character-String `"Collected on day 1"`
2. Extrahiere `measurements` mit `$` und berechne den Mittelwert.
3. Extrahiere `cars_sample` mit `[[` und zeige nur die `mpg`-Spalte.
4. Verwende `[`, um eine Teil-Liste mit `measurements` und `note` zu erstellen. Überprüfe mit `class()`, dass das Ergebnis eine Liste ist.

Lösung

```r
# 1. Create the list
my_data <- list(
  measurements = c(4.2, 5.1, 3.8, 6.0, 4.7),
  cars_sample = head(mtcars, 5),
  note = "Collected on day 1"
)

# 2. Extract and compute
mean(my_data$measurements)
```

```
[1] 4.76
```

```r
# 3. Extract dataframe column
my_data[["cars_sample"]]$mpg
```

```
[1] 21.0 21.0 22.8 21.4 18.7
```

```r
# 4. Sub-list
sub <- my_data[c("measurements", "note")]
class(sub)
```

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

```r
str(sub)
```

```
List of 2
 $ measurements: num [1:5] 4.2 5.1 3.8 6 4.7
 $ note        : chr "Collected on day 1"
```

Verschachtelte Listen

Listen können andere Listen enthalten und so hierarchische Strukturen beliebiger Tiefe erzeugen:

```r
experiment <- list(
  metadata = list(
    researcher = "Dr. Smith",
    date = "2025-03-15"
  ),
  results = list(
```

```
    treatment_A = c(4.2, 3.8, 5.1),
    treatment_B = c(6.7, 7.2, 6.9)
  )
)
```

Um auf verschachtelte Elemente zuzugreifen, verkettet man die Extraktionsoperatoren:

```
# The researcher name
experiment[["metadata"]][["researcher"]]
```

```
[1] "Dr. Smith"
```

```
# Equivalent with $
experiment$metadata$researcher
```

```
[1] "Dr. Smith"
```

```
# The first value of treatment B
experiment$results$treatment_B[1]
```

```
[1] 6.7
```

Die Funktion `str()` ist unverzichtbar, um die Struktur verschachtelter Listen zu verstehen:

```
str(experiment)
```

```
List of 2
 $ metadata:List of 2
  ..$ researcher: chr "Dr. Smith"
  ..$ date      : chr "2025-03-15"
 $ results :List of 2
  ..$ treatment_A: num [1:3] 4.2 3.8 5.1
  ..$ treatment_B: num [1:3] 6.7 7.2 6.9
```

`str()` liefert eine kompakte Zusammenfassung, die den Typ und die ersten Werte jedes Elements zeigt, eingerückt entsprechend der Verschachtelungshierarchie. Für große oder tief verschachtelte Listen begrenzt das Argument `max.level`, wie tief `str()` absteigt:

```
str(experiment, max.level = 1)
```

```
List of 2
 $ metadata:List of 2
 $ results :List of 2
```

Listen in der Praxis

Listen sind nicht nur ein Lehrkonzept — sie tauchen ständig in der täglichen R-Arbeit auf. Drei häufige Situationen verdeutlichen dies.

Modellobjekte sind Listen

Wenn man ein lineares Modell mit `lm()` anpasst, ist das Ergebnis eine Liste. Sie enthält die Koeffizienten, Residuen, angepassten Werte und vieles mehr:

```
fit <- lm(mpg ~ wt, data = mtcars)
typeof(fit)
```

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

```
names(fit)
```

```
 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"            "df.residual"
 [9] "xlevels"       "call"          "terms"         "model"
```

Einzelne Komponenten können mit `$` extrahiert werden:

```
# Coefficients
fit$coefficients
```

```
(Intercept)          wt
  37.285126   -5.344472
```

```
# First 10 residuals
fit$residuals[1:10]
```

```
        Mazda RX4     Mazda RX4 Wag        Datsun 710    Hornet 4 Drive
       -2.2826106        -0.9197704        -2.0859521         1.2973499
Hornet Sportabout           Valiant        Duster 360         Merc 240D
       -0.2001440        -0.6932545        -3.9053627         4.1637381
         Merc 230          Merc 280
        2.3499593         0.2998560
```

Die Funktion `summary()` gibt wiederum eine weitere Liste mit zusätzlich berechneten Statistiken wie R-Quadrat zurück:

```
fit_summary <- summary(fit)
fit_summary$r.squared
```

```
[1] 0.7528328
```

```
fit_summary$coefficients
```

```
             Estimate Std. Error   t value     Pr(>|t|)
(Intercept) 37.285126   1.877627 19.857575 8.241799e-19
wt          -5.344472   0.559101 -9.559044 1.293959e-10
```

split() gibt eine benannte Liste zurueck

Die Funktion `split()` teilt einen Dataframe in eine benannte Liste von Teil-Dataframes auf, einen pro Gruppe:

```r
by_cyl <- split(mtcars, mtcars$cyl)
names(by_cyl)
```

```
[1] "4" "6" "8"
```

```r
# The 6-cylinder cars
head(by_cyl[["6"]], 3)
```

```
                mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4      21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag  21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Hornet 4 Drive 21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
```

Dieses Muster wird in Kombination mit `lapply()` oder `purrr::map()` besonders mächtig, die in Kapitel 11 behandelt werden.

JSON-Daten und APIs

Bei der Arbeit mit Daten aus Web-APIs ist die Antwort typischerweise JSON, das R in eine verschachtelte Listenstruktur parst. Das Verständnis der Navigation durch Listen ist daher essenziell für jeden, der mit externen Datenquellen arbeitet. Obwohl dieses Kapitel JSON nicht im Detail behandelt, sind die Fähigkeiten zum Zugriff auf verschachtelte Listenelemente direkt anwendbar.

Übung: Modell-Output extrahieren

1. Passe ein lineares Modell an: `lm(mpg ~ wt + hp, data = mtcars)`.
2. Extrahiere den R-Quadrat-Wert aus der Modellzusammenfassung.
3. Extrahiere die Koeffizientenschätzungen (Intercept, wt, hp) als benannten numerischen Vektor.
4. Extrahiere die ersten 6 Residuen.
5. Kombiniere R-Quadrat, die drei Koeffizientenschätzungen und die Anzahl der Beobachtungen in einem einzelnen Tibble mit einer Zeile.

Hinweis: Die Anzahl der Beobachtungen kann über die Länge von `fit$residuals` bestimmt werden.

Lösung

```
# 1. Fit the model
fit <- lm(mpg ~ wt + hp, data = mtcars)

# 2. R-squared
fit_summary <- summary(fit)
r_sq <- fit_summary$r.squared
r_sq
```

```
[1] 0.8267855
```

```
# 3. Coefficients
coefs <- fit$coefficients
coefs
```

```
(Intercept)          wt          hp
37.22727012 -3.87783074 -0.03177295
```

```
# 4. First 6 residuals
fit$residuals[1:6]
```

```
        Mazda RX4     Mazda RX4 Wag        Datsun 710    Hornet 4 Drive
       -2.5723294        -1.5834826        -2.4758187         0.1349799
Hornet Sportabout           Valiant
        0.3727334        -2.3738163
```

```
# 5. Summary tibble
tibble(
  r_squared = r_sq,
  intercept = coefs[["(Intercept)"]],
  coef_wt   = coefs[["wt"]],
  coef_hp   = coefs[["hp"]],
  n_obs     = length(fit$residuals)
)
```

```
# A tibble: 1 × 5
  r_squared intercept coef_wt coef_hp n_obs
      <dbl>     <dbl>   <dbl>   <dbl> <int>
1     0.827      37.2   -3.88 -0.0318    32
```

Von Listen zu Dataframes

In vielen Workflows ist der letzte Schritt nach der Arbeit mit Listen die Umwandlung der Ergebnisse in einen tidy Dataframe. Je nach Struktur der Liste gibt es verschiedene Ansätze.

Für einfache Listen, in denen jedes Element ein Skalar oder ein Vektor gleicher Länge ist, funktioniert die direkte Umwandlung:

```
results <- list(
  group = c("A", "B", "C"),
  mean  = c(4.2, 5.8, 3.1),
  sd    = c(0.9, 1.2, 0.7)
)

as_tibble(results)
```

```
# A tibble: 3 × 3
  group  mean    sd
  <chr> <dbl> <dbl>
1 A       4.2   0.9
2 B       5.8   1.2
3 C       3.1   0.7
```

Für Listen von Dataframes (wie die Ausgabe von `split()`) kombiniert `bind_rows()` aus dplyr diese zurück zu einem einzelnen Dataframe. Das Argument `.id` erstellt eine Spalte, die festhält, aus welchem Listenelement jede Zeile stammt:

```
by_cyl <- split(mtcars, mtcars$cyl)

bind_rows(by_cyl, .id = "cyl") %>%
  head(6)
```

```
                mpg cyl  disp hp drat    wt  qsec vs am gear carb
Datsun 710     22.8   4 108.0 93 3.85 2.320 18.61  1  1    4    1
Merc 240D      24.4   4 146.7 62 3.69 3.190 20.00  1  0    4    2
Merc 230       22.8   4 140.8 95 3.92 3.150 22.90  1  0    4    2
Fiat 128       32.4   4  78.7 66 4.08 2.200 19.47  1  1    4    1
Honda Civic    30.4   4  75.7 52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla 33.9   4  71.1 65 4.22 1.835 19.90  1  1    4    1
```

Diese Umwandlungen sind ein wiederkehrendes Muster im Workflow, der in Kapitel 9 (Funktionen schreiben, die Listen zurückgeben) und Kapitel 11 (Funktionen auf viele Eingaben anwenden und die Ergebnisse sammeln) eingeführt wird.

Zusammenfassung

Dieses Kapitel hat zwei grundlegende R-Konzepte eingeführt: Kontrollfluss für Entscheidungen im Code und Listen als flexible Datenstruktur.

ℹ Wichtige Erkenntnisse

1. **Bedingte Logik:** `if` / `else` für skalare Entscheidungen, `ifelse()` für einfache vektorisierte Entscheidungen, `case_when()` für Multi-Kategorie-Entscheidungen, und `switch()` zum Abgleich mit diskreten Werten.
2. **Listen:** R's flexibelste Datenstruktur — jedes Element kann einen anderen Typ und eine andere Länge haben. Erstellt mit `list()`, Zugriff mit `$`, `[[` oder `[`.
3. **Listen in der Praxis:** Viele R-Funktionen geben Listen zurück (z.B. `lm()`, `t.test()`). Das Verständnis des Listenzugriffs mit `$` und `[[` ist essenziell für die Extraktion von Modellergebnissen.
4. **Listen zu Dataframes:** `as_tibble()` konvertiert einfache Listen, `bind_rows(.id = )` kombiniert Listen von Dataframes.
5. **Nächste Schritte:**
   - Kapitel 9: Funktionen schreiben — Kontrollfluss und Listenkonstruktion in wiederverwendbare Einheiten verpacken.
   - Kapitel 11: Iteration — Funktionen auf Listen von Eingaben anwenden mit `for`-Schleifen und `purrr::map()`.

Bibliography