8. Eigene Funktionen

Wiederverwendbaren Code schreiben und tidyeval verstehen

Dr. Paul Schmidt

Warum eigene Funktionen?

Einer der wichtigsten Schritte auf dem Weg vom R-Anwender zum R-Programmierer ist das Schreiben eigener Funktionen. Das Grundprinzip dahinter ist einfach: Wenn man denselben Code mehr als zweimal kopiert und eingefügt hat, sollte man eine Funktion daraus machen. Diese Regel wird oft als **DRY-Prinzip** bezeichnet — “Don’t Repeat Yourself”.

Das Schreiben von Funktionen hat mehrere handfeste Vorteile. Erstens kann man der Funktion einen aussagekräftigen Namen geben, der sofort verrät, was der Code tut. Zweitens muss man bei Änderungen nur eine einzige Stelle im Code anpassen, nicht jede Kopie. Drittens eliminiert man Fehler, die beim Kopieren und Einfügen entstehen — etwa wenn man vergisst, einen Variablennamen an einer Stelle zu ändern. Und viertens kann man Funktionen projektübergreifend wiederverwenden.

Betrachten wir folgenden Code, der Spalten auf einen Wertebereich von 0 bis 1 skaliert:

```
mtcars %>%
  select(mpg, hp, wt, qsec) %>%
  mutate(
    mpg = (mpg - min(mpg, na.rm = TRUE)) / (max(mpg, na.rm = TRUE) - min(mpg, na.rm
= TRUE)),
    hp = (hp - min(hp, na.rm = TRUE)) / (max(hp, na.rm = TRUE) - min(hp, na.rm =
TRUE)),
    wt = (wt - min(wt, na.rm = TRUE)) / (max(hp, na.rm = TRUE) - min(wt, na.rm =
TRUE)),
    qsec = (qsec - min(qsec, na.rm = TRUE)) / (max(qsec, na.rm = TRUE) - min(qsec,
na.rm = TRUE))
  ) %>%
  head()
```

```
                      mpg        hp        wt      qsec
Mazda RX4         0.4510638 0.2049470 -2.157895 0.2333333
Mazda RX4 Wag     0.4510638 0.2049470 -2.654971 0.3000000
Datsun 710        0.5276596 0.1448763 -1.573099 0.4892857
Hornet 4 Drive    0.4680851 0.2049470 -3.317739 0.5880952
Hornet Sportabout 0.3531915 0.4346290 -3.756335 0.3000000
Valiant           0.3276596 0.1872792 -3.795322 0.6809524
```

Dieser Code ist nicht nur lang und repetitiv, er enthält auch einen subtilen Fehler — lässt er sich finden? Bei so viel Wiederholung ist es fast unvermeidlich, dass sich Tippfehler einschleichen. Eine Funktion löst beide Probleme.

> **Weiterführende Ressourcen**
>
> Dieses Kapitel orientiert sich stark am hervorragenden Kapitel 25: Functions aus “R for Data Science” (2. Auflage). Für eine noch tiefere Behandlung von Tidy Evaluation empfehlen wir die Programming with dplyr Vignette.

Grundlagen: function()

Syntax und Aufbau

Eine Funktion in R besteht aus drei Teilen: einem **Namen**, den **Argumenten** und dem **Body** (Funktionskörper). Die grundlegende Syntax sieht so aus:

```
funktionsname <- function(argument1, argument2) {
  # Body: Der Code, der ausgeführt wird
  ergebnis <- argument1 + argument2
  ergebnis
}
```

Wenden wir das auf unser Skalierungsproblem an. Der sich wiederholende Teil ist die Formel `(x - min(x)) / (max(x) - min(x))`. Das einzige, was sich ändert, ist die Variable — das wird unser Argument:

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

Testen wir die Funktion:

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

```
[1] 0.0 0.5 1.0
```

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

```
[1] 0.00 0.25 0.50   NA 1.00
```

Jetzt wird unser ursprünglicher Code deutlich lesbarer und kürzer:

```
mtcars %>%
  select(mpg, hp, wt, qsec) %>%
  mutate(
    mpg = rescale01(mpg),
    hp = rescale01(hp),
    wt = rescale01(wt),
    qsec = rescale01(qsec)
  ) %>%
  head()
```

```
                      mpg        hp        wt      qsec
Mazda RX4         0.4510638 0.2049470 0.2830478 0.2333333
Mazda RX4 Wag     0.4510638 0.2049470 0.3482485 0.3000000
Datsun 710        0.5276596 0.1448763 0.2063411 0.4892857
Hornet 4 Drive    0.4680851 0.2049470 0.4351828 0.5880952
Hornet Sportabout 0.3531915 0.4346290 0.4927129 0.3000000
Valiant           0.3276596 0.1872792 0.4978266 0.6809524
```

Argumente mit und ohne Defaults

Funktionen können beliebig viele Argumente haben. Argumente ohne Default-Wert sind **erforderlich**, Argumente mit Default-Wert sind **optional**:

```
# na.rm hat einen Default-Wert, x nicht
my_mean <- function(x, na.rm = FALSE) {
  sum(x, na.rm = na.rm) / length(x)
```

```
}

my_mean(c(1, 2, 3))
```

```
[1] 2
```

```
my_mean(c(1, 2, NA), na.rm = TRUE)
```

```
[1] 1
```

Bei der Reihenfolge gilt: Erforderliche Argumente zuerst, optionale danach. Das wichtigste Argument (meist die Daten) kommt an erste Stelle — das ermöglicht die nahtlose Integration in Pipe-Ketten.

Rueckgabewerte

R-Funktionen geben automatisch das Ergebnis der letzten Zeile zurück. Man kann auch explizit `return()` verwenden, was besonders bei frühem Abbruch nützlich ist:

```
# Implizite Rückgabe (letzte Zeile)
add_one <- function(x) {
  x + 1
}

# Explizite Rückgabe mit return()
safe_divide <- function(x, y) {
  if (y == 0) {
    return(NA_real_)
  }
  x / y
}

safe_divide(10, 2)
```

```
[1] 5
```

```
safe_divide(10, 0)
```

```
[1] NA
```

Die Konvention ist: `return()` nur für frühe Abbrüche verwenden. Am Ende der Funktion ist die implizite Rückgabe üblicher und lesbarer.

Das Ellipsis-Argument (...)

Das spezielle Argument `...` (drei Punkte, auch “Ellipsis” genannt) erlaubt es, beliebig viele zusätzliche Argumente an eine andere Funktion durchzureichen:

```
# Alle zusätzlichen Argumente werden an mean() weitergegeben
my_summary <- function(x, ...) {
  c(
    mean = mean(x, ...),
    sd = sd(x, ...)
  )
}

# Ohne na.rm
my_summary(c(1, 2, 3))
```

```
mean   sd
   2    1
```

```
# Mit na.rm = TRUE (wird an mean() und sd() durchgereicht)
my_summary(c(1, 2, NA), na.rm = TRUE)
```

```
     mean        sd
1.5000000 0.7071068
```

Das ist besonders nützlich, wenn man Wrapper-Funktionen schreibt und nicht alle möglichen Argumente der inneren Funktion explizit auflisten möchte.

Übung: Standardabweichung vom Mittelwert

Schreibe eine Funktion `cv()`, die den Variationskoeffizienten (Standardabweichung geteilt durch Mittelwert) berechnet. Die Funktion sollte ein optionales `na.rm`-Argument haben.

```
cv(c(1, 2, 3, 4, 5))
cv(c(1, 2, NA, 4, 5), na.rm = TRUE)
```

Lösung

```
cv <- function(x, na.rm = FALSE) {
  sd(x, na.rm = na.rm) / mean(x, na.rm = na.rm)
}

cv(c(1, 2, 3, 4, 5))
```

```
[1] 0.5270463
```

```
cv(c(1, 2, NA, 4, 5), na.rm = TRUE)
```

```
[1] 0.6085806
```

Drei Funktionstypen

Das R for Data Science Buch unterscheidet drei nützliche Kategorien von Funktionen, die man häufig schreibt.

Vektor-Funktionen

Vektor-Funktionen nehmen einen oder mehrere Vektoren als Input und geben einen Vektor zurück. Sie lassen sich weiter unterteilen in **Mutate-Funktionen** (Output hat gleiche Länge wie Input) und **Summary-Funktionen** (Output hat Länge 1).

```
# Mutate-Funktion: gleiche Länge wie Input
z_score <- function(x) {
  (x - mean(x, na.rm = TRUE)) / sd(x, na.rm = TRUE)
}

z_score(c(1, 2, 3, 4, 5))
```

```
[1] -1.2649111 -0.6324555  0.0000000  0.6324555  1.2649111
```

```r
# Summary-Funktion: ein Wert als Output
coef_variation <- function(x, na.rm = FALSE) {
  sd(x, na.rm = na.rm) / mean(x, na.rm = na.rm)
}

coef_variation(c(1, 2, 3, 4, 5))
```

```
[1] 0.5270463
```

Dataframe-Funktionen

Dataframe-Funktionen nehmen einen Dataframe als Input und geben einen Dataframe zurück. Sie sind typischerweise Wrapper um dplyr-Verben:

```r
# Beispiel einer Dataframe-Funktion
filter_extreme <- function(df, var, threshold = 2) {
  df %>%
    filter(abs(as.vector(scale({{ var }}))) > threshold)
}

# Autos mit extremem Verbrauch (> 2 SD vom Mittelwert)
mtcars %>%
  filter_extreme(mpg) %>%
  select(mpg, hp, wt)
```

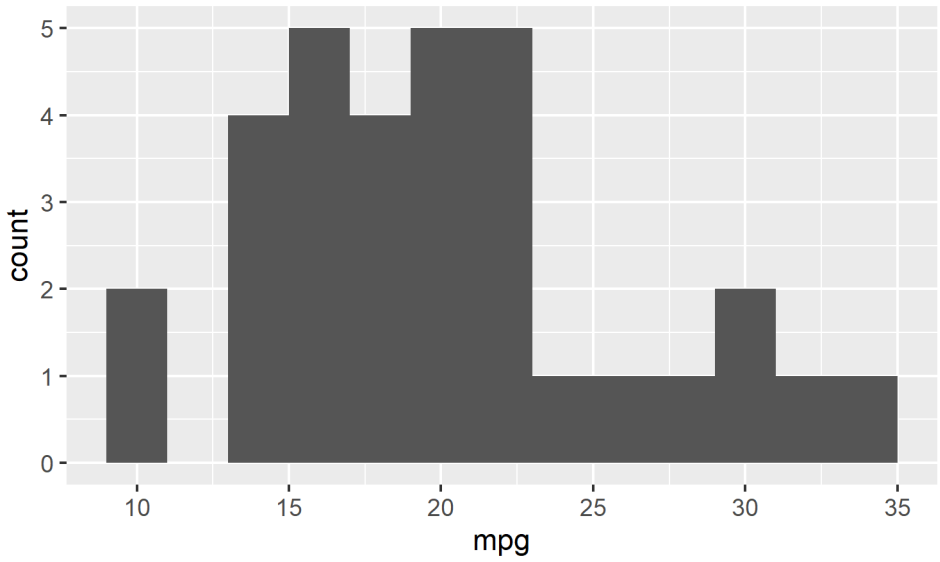
```
                mpg hp    wt
Fiat 128       32.4 66 2.200
Toyota Corolla 33.9 65 1.835
```

Plot-Funktionen

Plot-Funktionen nehmen einen Dataframe und geben einen ggplot zurück:

```r
# Beispiel einer Plot-Funktion
histogram <- function(df, var, binwidth = NULL) {
  df %>%
    ggplot(aes(x = {{ var }})) +
    geom_histogram(binwidth = binwidth)
}

mtcars %>% histogram(mpg, binwidth = 2)
```



Die `{ }`-Syntax im Plot-Beispiel wird im Abschnitt über Tidy Evaluation genauer erklärt.

Defensive Programmierung

Gute Funktionen prüfen ihre Eingaben und geben verständliche Fehlermeldungen aus. Das spart Debugging-Zeit und macht den Code robuster.

stop() fuer Fehlermeldungen

Die Funktion `stop()` bricht die Ausführung ab und zeigt eine Fehlermeldung an:

```
calculate_bmi <- function(weight_kg, height_m) {
  if (!is.numeric(weight_kg) || !is.numeric(height_m)) {
    stop("weight_kg und height_m muessen numerisch sein")
  }
  if (any(height_m <= 0)) {
    stop("height_m muss positiv sein")
  }
  weight_kg / height_m^2
}

calculate_bmi(70, 1.75)
```

```
[1] 22.85714
```

```
calculate_bmi(70, "groß")
```

```
Error in calculate_bmi(70, "groß"): weight_kg und height_m muessen numerisch sein
```

stopifnot() fuer schnelle Checks

Für einfache Bedingungen ist `stopifnot()` kompakter:

```
calculate_bmi <- function(weight_kg, height_m) {
  stopifnot(is.numeric(weight_kg), is.numeric(height_m))
  stopifnot(all(height_m > 0))

  weight_kg / height_m^2
}

calculate_bmi(70, 0)
```

```
Error in calculate_bmi(70, 0): all(height_m > 0) ist nicht TRUE
```

Der Nachteil: Die automatisch generierten Fehlermeldungen sind weniger informativ als selbst formulierte.

match.arg() fuer kategorische Argumente

Wenn ein Argument nur bestimmte Werte annehmen darf, verwendet man `match.arg()`:

```
center <- function(x, type = c("mean", "median", "trimmed")) {
  type <- match.arg(type)

  switch(type,
    mean = mean(x, na.rm = TRUE),
    median = median(x, na.rm = TRUE),
    trimmed = mean(x, trim = 0.1, na.rm = TRUE)
  )
}

center(1:10, "mean")
```

```
[1] 5.5
```

```
center(1:10, "median")
```

```
[1] 5.5
```

```
center(1:10, "mena")
```

```
Error in match.arg(type): 'arg' sollte eines von '"mean", "median", "trimmed"' sein
```

Die erlaubten Werte werden im Default des Arguments definiert. `match.arg()` erlaubt auch partielle Übereinstimmung und gibt hilfreiche Fehlermeldungen bei falschen Eingaben.

Übung: Sichere Logarithmus-Funktion

Schreibe eine Funktion `safe_log()`, die:

1. Prüft, ob der Input numerisch ist
2. Prüft, ob alle Werte positiv sind
3. Bei negativen Werten eine hilfreiche Fehlermeldung gibt, die anzeigt, wie viele nicht-positive Werte vorhanden sind

```
safe_log(c(1, 10, 100))
safe_log(c(-1, 10, 100))
```

Lösung

```
safe_log <- function(x, base = exp(1)) {
  if (!is.numeric(x)) {
    stop("x muss numerisch sein, nicht ", typeof(x))
  }

  n_negative <- sum(x <= 0, na.rm = TRUE)
  if (n_negative > 0) {
    stop(
      glue::glue("x enthält {n_negative} Wert(e) <= 0. ",
                 "Logarithmus ist nur für positive Zahlen definiert.")
    )
  }

  log(x, base = base)
}

safe_log(c(1, 10, 100))
```

```
[1] 0.000000 2.302585 4.605170
```

```
safe_log(c(-1, 0, 10, 100))
```

```
Error in safe_log(c(-1, 0, 10, 100)): x enthält 2 Wert(e) <= 0. Logarithmus ist
nur für positive Zahlen definiert.
```

Funktionen im tidyverse: Tidy Evaluation

Sobald man Funktionen schreibt, die tidyverse-Verben wie `filter()`, `mutate()` oder `ggplot()` verwenden, stößt man auf ein besonderes Problem: Wie übergibt man Spaltennamen als Argumente?

Das Problem: Indirection

Betrachten wir diese naive Funktion:

```
grouped_mean <- function(df, group_var, mean_var) {
  df %>%
    group_by(group_var) %>%
    summarize(mean = mean(mean_var))
}

mtcars %>% grouped_mean(cyl, mpg)
```

```
Error in `group_by()`:
! Must group by variables found in `.data`.
✖ Column `group_var` is not found.
```

Die Funktion sucht nach Spalten namens `group_var` und `mean_var` — aber die gibt es nicht! Das Problem ist **Indirection**: dplyr verwendet **Data Masking**, um Spaltennamen ohne Anführungszeichen zu erlauben. Das ist praktisch im interaktiven Gebrauch, aber macht das Schreiben von Funktionen komplizierter.

> **ℹ Data Masking erklärt**
>
> Data Masking bedeutet, dass man `filter(df, x > 5)` schreiben kann statt `filter(df, df$x > 5)`. R sucht `x` zuerst in den Spalten des Dataframes, dann in der Umgebung. Das ist der Grund, warum `group_var` als Spaltenname interpretiert wird — nicht als Variable, die einen Spaltennamen enthält.

Die Standardloesung: Curly-Curly

Seit rlang 0.4.0 (2019) gibt es eine elegante Lösung: den **Embracing-Operator** `{ }` (auch “curly-curly” genannt). Er sagt dplyr: “Schau nicht nach einer Spalte mit diesem Namen, sondern schau in diese Variable hinein”:

```
grouped_mean <- function(df, group_var, mean_var) {
  df %>%
    group_by({{ group_var }}) %>%
    summarize(mean = mean({{ mean_var }}), .groups = "drop")
}

mtcars %>% grouped_mean(cyl, mpg)
```

```
# A tibble: 3 × 2
    cyl  mean
  <dbl> <dbl>
1     4  26.7
2     6  19.7
3     8  15.1
```

Die Regel ist einfach: **Jedes Argument, das an eine tidyverse-Funktion weitergegeben wird, die Data Masking oder Tidy Selection verwendet, muss embraced werden.**

Woher weiß man, welche Funktionen das sind? Die Dokumentation verrät es: Man sucht nach `<data-masking>` (für Funktionen wie `filter()`, `mutate()`, `summarize()`) oder `<tidy-select>` (für Funktionen wie `select()`, `rename()`, `across()`).

```
# Flexible Summary-Funktion
summary_stats <- function(df, var) {
  df %>%
    summarize(
      n = n(),
      mean = mean({{ var }}, na.rm = TRUE),
      sd = sd({{ var }}, na.rm = TRUE),
      min = min({{ var }}, na.rm = TRUE),
      max = max({{ var }}, na.rm = TRUE)
    )
}

mtcars %>% summary_stats(mpg)
```

```
   n     mean       sd  min  max
1 32 20.09062 6.026948 10.4 33.9
```

```
mtcars %>% group_by(cyl) %>% summary_stats(mpg)
```

```
# A tibble: 3 × 6
    cyl     n  mean    sd   min   max
  <dbl> <int> <dbl> <dbl> <dbl> <dbl>
1     4    11  26.7  4.51  21.4  33.9
2     6     7  19.7  1.45  17.8  21.4
3     8    14  15.1  2.56  10.4  19.2
```

Übung: Proportionen zählen

Schreibe eine Funktion `count_prop()`, die wie `count()` funktioniert, aber zusätzlich eine Spalte `prop` mit dem Anteil hinzufügt.

```
# Gewünschtes Ergebnis:
mtcars %>% count_prop(cyl)
# # A tibble: 3 × 3
#     cyl     n  prop
#   <dbl> <int> <dbl>
# 1     4    11 0.344
# 2     6     7 0.219
# 3     8    14 0.438
```

Lösung

```
count_prop <- function(df, var, sort = FALSE) {
  df %>%
    count({{ var }}, sort = sort) %>%
    mutate(prop = n / sum(n))
}

mtcars %>% count_prop(cyl)
```

```
  cyl  n    prop
1   4 11 0.34375
2   6  7 0.21875
3   8 14 0.43750
```

Dynamische Spaltennamen mit dem Walrus-Operator

Was, wenn man nicht nur eine Spalte *lesen*, sondern eine Spalte mit dynamischem Namen *erstellen* möchte? Der normale `=`-Operator erlaubt links nur feste Namen. Hier kommt `:=` ins Spiel (der “Walrus-Operator”):

```
# Funktion, die eine neue Spalte mit dynamischem Namen erstellt
standardize <- function(df, var) {
  df %>%
    mutate(
      "{{ var }}_z" := ({{ var }} - mean({{ var }}, na.rm = TRUE)) /
                       sd({{ var }}, na.rm = TRUE)
    )
}

mtcars %>%
  select(mpg, cyl) %>%
  standardize(mpg) %>%
  head()
```

```
                   mpg cyl      mpg_z
Mazda RX4         21.0   6  0.1508848
Mazda RX4 Wag     21.0   6  0.1508848
Datsun 710        22.8   4  0.4495434
Hornet 4 Drive    21.4   6  0.2172534
Hornet Sportabout 18.7   8 -0.2307345
Valiant           18.1   6 -0.3302874
```

Die Syntax `"{{ var }}_z" :=` kombiniert die glue-artige String-Interpolation mit dem Walrus-Operator. Das `{ var }` im String wird durch den Variablennamen ersetzt.

Spalten als Strings: .data Pronoun

Manchmal hat man Spaltennamen als Strings — etwa aus einer Konfigurationsdatei oder Benutzereingabe. Hier verwendet man das `.data`-Pronoun:

```
# Spaltenname kommt als String
summarize_column <- function(df, col_name) {
  df %>%
    summarize(mean = mean(.data[[col_name]], na.rm = TRUE))
}

summarize_column(mtcars, "mpg")
```

```
      mean
1 20.09062
```

```
# Nützlich für Iteration über Spaltennamen
col_names <- c("mpg", "hp", "wt")
map(col_names, ~ summarize_column(mtcars, .x))
```

```
[[1]]
      mean
1 20.09062

[[2]]
      mean
1 146.6875

[[3]]
     mean
1 3.21725
```

Fortgeschritten: enquo() und !!

Die `{ }`-Syntax ist eine Kurzschreibweise für eine Kombination aus `enquo()` und `!!`. In den meisten Fällen braucht man die explizite Form nicht, aber es gibt Situationen, wo sie nötig ist — zum Beispiel wenn man den Variablennamen als String extrahieren möchte.

Hier dieselbe Funktion in beiden Schreibweisen:

```
# Mit {{ }} - die empfohlene Kurzform
grouped_mean_short <- function(df, group_var, mean_var) {
  df %>%
    group_by({{ group_var }}) %>%
    summarize(mean = mean({{ mean_var }}), .groups = "drop")
}

# Mit enquo() und !! - die explizite Form
grouped_mean_explicit <- function(df, group_var, mean_var) {
  group_var <- enquo(group_var)  # Argument einfangen
  mean_var <- enquo(mean_var)

  df %>%
    group_by(!!group_var) %>%    # Mit !! wieder einsetzen
    summarize(mean = mean(!!mean_var), .groups = "drop")
}

# Beide liefern dasselbe Ergebnis
mtcars %>% grouped_mean_short(cyl, mpg)
```

```
# A tibble: 3 × 2
    cyl  mean
  <dbl> <dbl>
1     4  26.7
2     6  19.7
3     8  15.1
```

```
mtcars %>% grouped_mean_explicit(cyl, mpg)
```

```
# A tibble: 3 × 2
    cyl  mean
  <dbl> <dbl>
1     4  26.7
2     6  19.7
3     8  15.1
```

enquo() fängt ein Argument ein, ohne es auszuwerten. **!!** (bang-bang) fügt den eingefangenen Ausdruck wieder ein.

Wann braucht man die explizite Form? Wenn man den Variablennamen als String extrahieren möchte:

```
# as_label() extrahiert den Namen als String - nur mit enquo() möglich
summary_with_label <- function(df, var) {
  var_quo <- enquo(var)
  var_name <- rlang::as_label(var_quo)

  df %>%
    summarize(
      variable = var_name,
      mean = mean(!!var_quo, na.rm = TRUE)
    )
}

mtcars %>% summary_with_label(mpg)
```

```
  variable     mean
1      mpg 20.09062
```

```
mtcars %>% summary_with_label(hp)
```

```
  variable     mean
1       hp 146.6875
```

Mehrere Spalten als Strings: syms() und !!!

Wenn man mehrere Spaltennamen als Character-Vektor hat und diese in einer tidyverse-Funktion verwenden möchte, braucht man `syms()` und `!!!`:

- **syms()** wandelt einen Character-Vektor in eine Liste von Symbolen um
- **!!!** (splice-Operator) entpackt diese Liste, sodass jedes Element einzeln übergeben wird

```
# Mehrere Gruppierungsvariablen als Character-Vektor
grouped_summary <- function(df, group_vars, summary_var) {
  # syms() wandelt c("cyl", "am") in list(sym("cyl"), sym("am")) um
  group_symbols <- syms(group_vars)

  df %>%
    # !!! entpackt die Liste: group_by(cyl, am) statt group_by(list(...))
    group_by(!!!group_symbols) %>%
    summarize(mean = mean({{ summary_var }}, na.rm = TRUE), .groups = "drop")
}

mtcars %>% grouped_summary(c("cyl", "am"), mpg)
```

```
# A tibble: 6 × 3
    cyl    am  mean
  <dbl> <dbl> <dbl>
1     4     0  22.9
2     4     1  28.1
3     6     0  19.1
4     6     1  20.6
5     8     0  15.0
6     8     1  15.4
```

Diese Technik ist besonders nützlich, wenn die Gruppierungsvariablen dynamisch bestimmt werden — etwa aus einer Konfiguration oder Benutzereingabe.

pick() fuer Tidy Selection in Data-Masking-Kontext

Manchmal möchte man Tidy Selection (wie bei `select()`) innerhalb einer Data-Masking-Funktion (wie `group_by()`) verwenden. Hier hilft `pick()`:

```r
# Mehrere Gruppierungsspalten mit Tidy Selection
count_by <- function(df, ...) {
  df %>%
    group_by(pick(...)) %>%
    summarize(n = n(), .groups = "drop")
}

mtcars %>% count_by(cyl, am)
```

```
# A tibble: 6 × 3
    cyl    am     n
  <dbl> <dbl> <int>
1     4     0     3
2     4     1     8
3     6     0     4
4     6     1     3
5     8     0    12
6     8     1     2
```

```r
mtcars %>% count_by(starts_with("c"))
```

```
# A tibble: 9 × 3
    cyl  carb     n
  <dbl> <dbl> <int>
1     4     1     5
2     4     2     6
3     6     1     2
4     6     4     4
5     6     6     1
6     8     2     4
7     8     3     3
8     8     4     6
9     8     8     1
```

Wichtig: Bei `...` verwendet man `pick(...)` direkt, nicht `pick({{ ... }})`. Die `{ }`-Syntax ist nur für einzelne benannte Argumente gedacht.

Uebersicht: Wann welchen Ansatz?

Situation	Lösung	Beispiel
Spalte als “bare name” übergeben	`{ }`	`filter({{ var }} > 0)`
Spaltenname als String	`.data[[]]`	`summarize(mean = mean(.data[[col]]))`
Mehrere Spalten via `...`	`...` direkt durchreichen	`group_by(...)` oder `pick(...)`
Dynamischer Spaltenname erstellen	`:=`	`mutate("{{ var }}_new" := ...)`
Variablenname als String extrahieren	`enquo()` + `as_label()`	`as_label(enquo(var))`

Situation	Lösung	Beispiel
Liste von Strings zu Symbolen	`syms()` + `!!!`	`group_by(!!!syms(cols))`
Tidy Select in Data Masking	`pick()`	`group_by(pick(...))`

Übung: Flexible Filterung

Schreibe eine Funktion `filter_na()`, die alle Zeilen entfernt, in denen eine bestimmte Spalte `NA` ist.

```
# Test-Daten
test_df <- tibble(
  x = c(1, NA, 3),
  y = c("a", "b", NA)
)

test_df %>% filter_na(x)
test_df %>% filter_na(y)
```

Lösung

```
filter_na <- function(df, var) {
  df %>%
    filter(!is.na({{ var }}))
}

test_df <- tibble(
  x = c(1, NA, 3),
  y = c("a", "b", NA)
)

test_df %>% filter_na(x)
```

```
# A tibble: 2 × 2
      x y
  <dbl> <chr>
1     1 a
2     3 <NA>
```

```
test_df %>% filter_na(y)
```

```
# A tibble: 2 × 2
      x y
  <dbl> <chr>
1     1 a
2    NA b
```

Übung: Plot-Funktion mit dynamischem Titel

Erweitere die `histogram()`-Funktion so, dass der Titel automatisch den Variablennamen enthält:

```
mtcars %>% histogram(mpg, binwidth = 2)
# Sollte einen Titel wie "Histogramm von mpg" haben
```

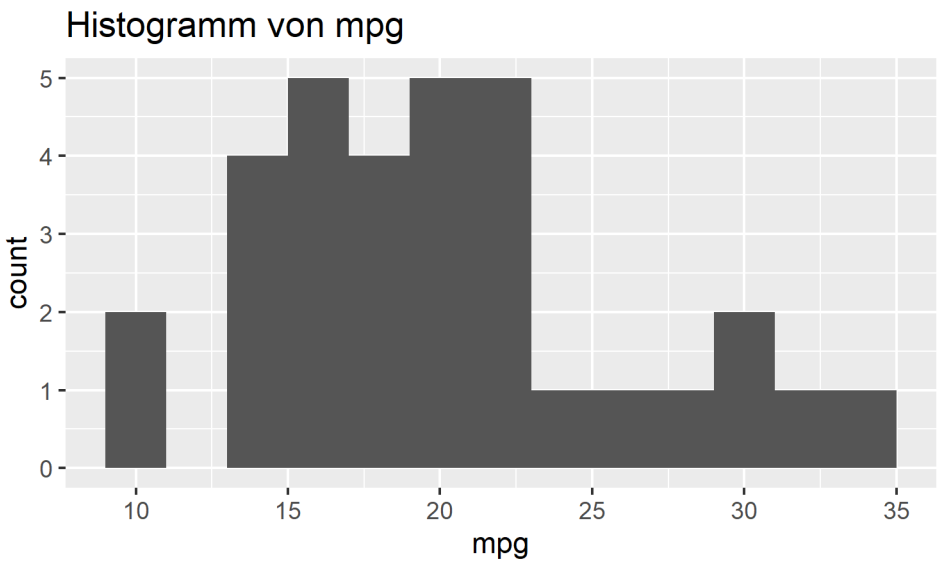
Hinweis: `rlang::englue()` oder die Kombination aus `enquo()` und `as_label()` verwenden.

Lösung

```
histogram <- function(df, var, binwidth = NULL) {
  title <- rlang::englue("Histogramm von {{var}}")

  df %>%
    ggplot(aes(x = {{ var }})) +
    geom_histogram(binwidth = binwidth) +
    labs(title = title)
}

mtcars %>% histogram(mpg, binwidth = 2)
```



Best Practices und Style

Benennung

Funktionsnamen sollten Verben sein und klar beschreiben, was die Funktion tut:

```
# Gut: Verben, beschreibend
impute_missing()
calculate_bmi()
extract_coefficients()

# Schlecht: Zu kurz oder nicht beschreibend
f()
```

```
my_function()
do_stuff()
```

Argumentnamen sollten Substantive sein. Das Daten-Argument heißt typischerweise `df`, `data` oder `.data`.

Code-Formatierung

Immer geschweifte Klammern `{}` verwenden, auch bei einzeiligen Funktionen. Der Body wird mit zwei Leerzeichen eingerückt:

```
# Gut
add_one <- function(x) {
  x + 1
}

# Vermeiden
add_one <- function(x) x + 1
```

Dokumentation mit Roxygen

Wenn man ein R-Paket entwickelt, muss jede exportierte Funktion dokumentiert werden. Diese Dokumentation wird im **Roxygen-Format** geschrieben — spezielle Kommentare direkt über der Funktion, die mit `#'` beginnen. Beim Bauen des Pakets werden diese Kommentare automatisch in die formatierten Hilfsseiten umgewandelt, die man mit `?funktionsname` aufruft.

Aber auch wenn man gar kein Paket schreibt, sondern nur ein Skript mit ein paar Hilfsfunktionen, lohnt sich dieses Format. Statt unstrukturierte Kommentare neben die Funktion zu schreiben, kann man gleich das Roxygen-Format verwenden. Es ist übersichtlich, standardisiert, und falls die Funktion später doch in ein Paket wandert, ist die Dokumentation bereits fertig.

Die wichtigsten Roxygen-Tags:

- **Titel** (erste Zeile): Eine kurze, einzeilige Beschreibung der Funktion
- **Beschreibung** (nach Leerzeile): Ausführlichere Erklärung, was die Funktion tut
- `@param name`: Beschreibt ein Argument der Funktion
- `@return`: Beschreibt, was die Funktion zurückgibt
- `@examples`: Ausführbare Beispiele für die Verwendung

```
#' Berechne den Body Mass Index
#'
#' Diese Funktion berechnet den BMI aus Gewicht und Größe.
#' Bei Vektoren wird der BMI elementweise berechnet.
#'
#' @param weight_kg Gewicht in Kilogramm (numerischer Vektor).
#' @param height_m Größe in Metern (numerischer Vektor).
#'
#' @return Ein numerischer Vektor mit BMI-Werten.
#'
#' @examples
#' calculate_bmi(70, 1.75)
#' calculate_bmi(c(60, 80), c(1.60, 1.80))
calculate_bmi <- function(weight_kg, height_m) {
  stopifnot(is.numeric(weight_kg), is.numeric(height_m))
```

```
  stopifnot(all(height_m > 0))

  weight_kg / height_m^2
}
```

In RStudio und Positron kann man ein leeres Roxygen-Skelett automatisch einfügen lassen: Cursor in die Funktion setzen und **Code → Insert Roxygen Skeleton** wählen (oder `Ctrl+Alt+Shift+R`).

Bibliography