# 10. Input-Validierung

Robuste Funktionen mit informativen Fehlermeldungen schreiben

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", "assertthat", "cli")) {
  if (!require(pkg, character.only = TRUE)) install.packages(pkg)
}

library(tidyverse)
```

## Einleitung

Jede Funktion macht Annahmen über ihre Inputs: ein Vektor sollte numerisch sein, ein Dataframe sollte bestimmte Spalten enthalten, ein Wert sollte positiv sein. Wenn diese Annahmen verletzt werden, wirft R entweder einen kryptischen Fehler tief in der Funktion – oder schlimmer noch – produziert stillschweigend falsche Ergebnisse. Beides kostet Debugging-Zeit und untergräbt das Vertrauen.

Man betrachte, was passiert, wenn man eine Faktor-Spalte an eine Funktion übergibt, die Zahlen erwartet:

```
summarize_column <- function(data, col) {
  values <- data[[col]]
  centered <- values - mean(values)  # error surfaces here, not at entry
  sum(centered^2) / (length(centered) - 1)
}

summarize_column(iris, "Species")
```

```
Warning in mean.default(values): Argument ist weder numerisch noch boolesch:
gebe NA zurück
```

```
Warning in Ops.factor(values, mean(values)): '-' ist nicht sinnvoll für
Faktoren
```

```
[1] NA
```

Die Fehlermeldung erwähnt `mean()` und “not meaningful for factors”, aber das eigentliche Problem ist, dass eine nicht-numerische Spalte übergeben wurde. Mit Input-Validierung fängt man das sofort ab:

```
summarize_column <- function(data, col) {
  values <- data[[col]]
  if (!is.numeric(values)) {
    stop(glue::glue("Column '{col}' must be numeric, but is {class(values)[1]}"))
  }
  centered <- values - mean(values)
  sum(centered^2) / (length(centered) - 1)
}

summarize_column(iris, "Species")
```

```
Error in summarize_column(iris, "Species"): Column 'Species' must be numeric, but
is factor
```

Kapitel 9 hat `stop()`, `stopifnot()` und `match.arg()` für defensives Programmieren eingeführt. Dieses Kapitel erweitert das Toolkit: `warning()` und `message()` für nicht-fatale Signale, das assertthat-Paket für lesbare Assertions, das cli-Paket für formatierte Fehlermeldungen und `tryCatch()` für fehlertolerante Weiterverarbeitung.

# Wiederholung: stop() und stopifnot()

Da Kapitel 9 diese Funktionen ausführlich behandelt hat, hier nur eine kurze Zusammenfassung mit Fokus auf ihre **Einschränkungen**.

`stop()` gibt volle Kontrolle über die Fehlermeldung. `stopifnot()` ist kompakter, aber seine automatisch generierten Meldungen sind schwer lesbar:

```
validate_proportion <- function(x) {
  stopifnot(is.numeric(x))
  stopifnot(all(x >= 0 & x <= 1, na.rm = TRUE))
  x
}

validate_proportion(c(0.2, 1.5, 0.8))
```

```
Error in validate_proportion(c(0.2, 1.5, 0.8)): all(x >= 0 & x <= 1, na.rm = TRUE)
ist nicht TRUE
```

Die Meldung `all(x >= 0 & x <= 1, na.rm = TRUE) is not TRUE` liest sich wie Code statt wie eine Erklärung. Eine benutzerfreundliche Meldung würde sagen “Found values outside the range [0, 1]”. Diese Lesbarkeits-Lücke motiviert die später besprochenen Pakete assertthat und cli.

Als Faustregel: `stopifnot()` für interne Assertions verwenden, die nur Entwickler sehen, und `stop()` (oder `cli_abort()`) für benutzerseitige Validierung, wo die Qualität der Meldung wichtig ist.

# warning() und message()

Nicht jedes Problem sollte die Ausführung stoppen. R bietet `warning()` für Situationen, in denen etwas wahrscheinlich nicht stimmt, und `message()` für rein informativen Output.

## warning(): Etwas könnte nicht stimmen

Eine Warnung signalisiert, dass die Funktion ein Ergebnis produziert hat, der Aufrufer aber über ein mögliches Problem informiert werden sollte:

```
column_means <- function(data) {
  numeric_cols <- data %>% select(where(is.numeric))

  na_counts <- numeric_cols %>%
    summarize(across(everything(), \(x) sum(is.na(x)))) %>%
    pivot_longer(everything(), names_to = "column", values_to = "n_na") %>%
    filter(n_na > 0)
```

```
  if (nrow(na_counts) > 0) {
    cols_with_na <- na_counts %>%
      mutate(label = glue::glue("{column} ({n_na})")) %>%
      pull(label) %>%
      paste(collapse = ", ")
    warning(glue::glue("NAs removed in: {cols_with_na}"), call. = FALSE)
  }

  numeric_cols %>%
    summarize(across(everything(), \(x) mean(x, na.rm = TRUE)))
}

column_means(airquality)
```

```
Warning: NAs removed in: Ozone (37), Solar.R (7)
```

```
     Ozone  Solar.R     Wind     Temp     Month       Day
1 42.12931 185.9315 9.957516 77.88235 6.993464 15.80392
```

Das Argument `call. = FALSE` unterdrückt den Funktionsaufruf in der Warnung, was den Output sauberer macht.

## message(): Informativer Output

Eine Message ist rein informativ – nichts ist falsch, man hält nur den Nutzer auf dem Laufenden:

```
standardize <- function(data) {
  n_cols <- sum(sapply(data, is.numeric))
  n_skipped <- ncol(data) - n_cols
  message(glue::glue("Standardizing {n_cols} numeric columns, skipping {n_skipped}
non-numeric"))

  data %>%
    mutate(across(where(is.numeric), \(x) (x - mean(x, na.rm = TRUE)) / sd(x, na.rm
= TRUE)))
}

result <- standardize(iris)
```

```
Standardizing 4 numeric columns, skipping 1 non-numeric
```

## Unterdrücken und das richtige Signal wählen

Nutzer können Warnungen und Messages selektiv unterdrücken mit `suppressWarnings()` und `suppressMessages()`. Das funktioniert nur, weil `warning()` und `message()` separate Signalmechanismen verwenden – hätte man stattdessen `cat()` benutzt, gäbe es keine Möglichkeit, den Output programmatisch zu unterdrücken.

Die Wahl zwischen `stop()`, `warning()` und `message()` hängt vom Schweregrad ab:

| Signal | Fatal? | Verwenden wenn… |
|---|---|---|
| `stop()` | Ja | Die Funktion *kann* kein valides Ergebnis produzieren |
| `warning()` | Nein | Das Ergebnis *könnte* problematisch sein |

| Signal | Fatal? | Verwenden wenn… |
|---|---|---|
| `message()` | Nein | Alles ist in Ordnung, nur zur Info |

Ein praktischer Test: Wenn jemand die Funktion in `suppressWarnings()` einwickelt und ein falsches Ergebnis bekommt, hätte das Signal ein Fehler sein sollen, keine Warnung.

**Übung: Funktion mit Warnungen**

Man schreibe eine Funktion `safe_mean()`, die einen numerischen Vektor `x` entgegennimmt, prüft ob er tatsächlich numerisch ist (mit Fehler abbrechen falls nicht), eine **Warnung** ausgibt falls NAs vorhanden sind (mit Angabe der Anzahl), und den Mittelwert mit `na.rm = TRUE` zurückgibt.

Zum Testen: `airquality$Ozone` (37 NAs) und `c(1, 2, 3)` (keine NAs).

**Lösung**

```
safe_mean <- function(x) {
  if (!is.numeric(x)) {
    stop(glue::glue("x must be numeric, not {class(x)[1]}"), call. = FALSE)
  }

  n_na <- sum(is.na(x))
  if (n_na > 0) {
    warning(glue::glue("{n_na} NA value(s) removed before computing mean"),
call. = FALSE)
  }

  mean(x, na.rm = TRUE)
}

safe_mean(airquality$Ozone)
```

```
Warning: 37 NA value(s) removed before computing mean
```

```
[1] 42.12931
```

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

```
[1] 2
```

# Strukturierte Validierung mit assertthat

Das assertthat-Paket liegt zwischen `stopifnot()` (kompakt aber kryptisch) und `stop()` (lesbar aber wortreich). Seine `assert_that()`-Funktion funktioniert wie `stopifnot()`, erzeugt aber menschenlesbare Fehlermeldungen:

```
library(assertthat)

validate_proportion <- function(x) {
  assert_that(is.numeric(x))
  assert_that(all(x >= 0 & x <= 1, na.rm = TRUE))
  x
```

```
}

validate_proportion("hello")
```

```
Error: x is not a numeric or integer vector
```

## Eingebaute Hilfsfunktionen

Das Paket bietet Typ-Prüfungsfunktionen mit klaren Fehlermeldungen:

```
assert_that(is.string("hello"))       # single character string
```

```
[1] TRUE
```

```
assert_that(is.number(42))            # single numeric value
```

```
[1] TRUE
```

```
assert_that(is.flag(TRUE))            # single logical value
```

```
[1] TRUE
```

```
assert_that(has_name(mtcars, "mpg"))  # name exists in object
```

```
[1] TRUE
```

```
assert_that(not_empty(c(1, 2, 3)))    # non-empty
```

```
[1] TRUE
```

```
# Failure produces a clear message
assert_that(has_name(mtcars, "horsepower"))
```

```
Error: mtcars does not have all of these name(s): 'horsepower'
```

## see_if() und on_failure()

`see_if()` prüft eine Bedingung ohne die Ausführung zu stoppen und gibt `TRUE` / `FALSE` mit einem Message-Attribut zurück. `on_failure()` ermöglicht es, eigene Fehlermeldungen für selbst geschriebene Prüffunktionen zu definieren.

⚠ Fortgeschritten: on_failure() (zum Aufklappen)

Der `on_failure()`-Mechanismus nutzt Metaprogrammierung, um Fehlermeldungen anzupassen. Dies ist eine Nischenfunktion, die die meisten Anwender nicht benötigen:

```
is_positive <- function(x) is.numeric(x) && all(x > 0, na.rm = TRUE)

on_failure(is_positive) <- function(call, env) {
  n_bad <- sum(eval(call$x, env) <= 0, na.rm = TRUE)
  glue::glue("{deparse(call$x)} contains {n_bad} non-positive value(s)")
}

assert_that(is_positive(c(1, -2, 3, -4)))
```

```
Error: c(1, -2, 3, -4) contains 2 non-positive value(s)
```

> **Hinweis**
>
> Das assertthat-Paket befindet sich im Wartungsmodus (letztes CRAN-Update 2019). Für neuen Code empfehlen sich `cli::cli_abort()` und `rlang::abort()` – sie bieten reichhaltigere Fehlermeldungen mit Inline-Formatierung und werden aktiv weiterentwickelt.

# Informative Fehler mit cli

Das cli-Paket bietet modernen, formatierten Output für R. Seine Funktionen `cli_abort()`, `cli_warn()` und `cli_inform()` ersetzen `stop()`, `warning()` und `message()` mit zwei Vorteilen: Inline-Markup und automatische Wertformatierung.

## Grundlegende Verwendung

Die cli-Funktionen akzeptieren einen Character-Vektor, bei dem jedes Element eine Zeile wird. Benannte Elemente bekommen spezielle Bullet-Präfixe – `"x"` für Probleme, `"i"` für Info, `"!"` für Warnungen:

```
library(cli)

validate_age <- function(age) {
  if (!is.numeric(age)) {
    cli_abort(c(
      "{.arg age} must be numeric.",
      "x" = "You supplied a {.cls {class(age)}} vector."
    ))
  }
  if (any(age < 0 | age > 150, na.rm = TRUE)) {
    cli_abort(c(
      "{.arg age} must be between 0 and 150.",
      "i" = "Found {sum(age < 0 | age > 150, na.rm = TRUE)} out-of-range value(s)."
    ))
  }
  age
}

validate_age("twenty")
```

```
Error in `validate_age()`:
! `age` must be numeric.
✖ You supplied a <character> vector.
```

```
validate_age(c(25, 30, -5, 200))
```

```
Error in `validate_age()`:
! `age` must be between 0 and 150.
ℹ Found 2 out-of-range value(s).
```

## Inline Markup

Geschweifte Klammern formatieren Werte entsprechend ihrer Rolle in der Meldung:

| Markup | Zweck | Markup | Zweck |
|---|---|---|---|
| `{.arg name}` | Argument | `{.val value}` | Ein Wert |
| `{.var name}` | Variable | `{.cls class}` | Klassenname |
| `{.code code}` | Code-Snippet | `{.fn name}` | Funktionsname |

Der `?`-Operator übernimmt die Pluralisierung, und R-Ausdrücke werden genau wie in `glue::glue()` interpoliert:

```
check_columns <- function(data, required_cols) {
  missing <- setdiff(required_cols, names(data))
  if (length(missing) > 0) {
    cli_abort(c(
      "Required column{?s} missing from {.arg data}.",
      "i" = "Missing: {.val {missing}}."
    ))
  }
}

check_columns(mtcars, c("mpg", "horsepower", "torque"))
```

```
Error in post_process_plurals(pstr, values): Cannot pluralize without a quantity
```

## cli_warn() und cli_inform()

Dasselbe Markup funktioniert auch für Warnungen und Messages:

```
safe_divide <- function(x, y) {
  if (any(y == 0, na.rm = TRUE)) {
    cli_warn(c(
      "Division by zero encountered.",
      "i" = "{sum(y == 0)} element{?s} of {.arg y} {?is/are} zero.",
      "i" = "Returning {.val {Inf}} for those positions."
    ))
  }
  x / y
}

safe_divide(c(10, 20, 30), c(2, 0, 5))
```

```
Warning: Division by zero encountered.
i 1 element of `y` is zero.
i Returning Inf for those positions.
```

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

### Übung: stop() zu cli_abort() umschreiben

Der folgende BMI-Rechner verwendet einfache `stop()`-Meldungen. Man schreibe die Validierung um, sodass `cli_abort()` mit Inline-Markup verwendet wird. Jeder Fehler sollte eine Kopfzeile und mindestens einen `"x"`-Bullet haben.

```
calc_bmi <- function(weight_kg, height_m) {
  if (!is.numeric(weight_kg)) stop("weight_kg must be numeric")
  if (!is.numeric(height_m)) stop("height_m must be numeric")
  if (length(weight_kg) != length(height_m)) stop("Lengths must match")
  if (any(weight_kg <= 0, na.rm = TRUE)) stop("weight_kg must be positive")
  if (any(height_m <= 0, na.rm = TRUE)) stop("height_m must be positive")
  weight_kg / height_m^2
}
```

Zum Testen: `calc_bmi("80", 1.80)`, `calc_bmi(c(70, 80), c(1.70, 1.75, 1.80))` und `calc_bmi(c(70, -5), c(1.70, 1.80))`.

**Lösung**

```
calc_bmi <- function(weight_kg, height_m) {
  if (!is.numeric(weight_kg)) {
    cli_abort(c(
      "{.arg weight_kg} must be numeric.",
      "x" = "You supplied a {.cls {class(weight_kg)}} vector."
    ))
  }
  if (!is.numeric(height_m)) {
    cli_abort(c(
      "{.arg height_m} must be numeric.",
      "x" = "You supplied a {.cls {class(height_m)}} vector."
    ))
  }
  if (length(weight_kg) != length(height_m)) {
    cli_abort(c(
      "{.arg weight_kg} and {.arg height_m} must have the same length.",
      "x" = "{.arg weight_kg} has {length(weight_kg)} element{?s}, {.arg
height_m} has {length(height_m)}."
    ))
  }
  if (any(weight_kg <= 0, na.rm = TRUE)) {
    cli_abort(c(
      "{.arg weight_kg} must contain only positive values.",
      "x" = "Found {sum(weight_kg <= 0, na.rm = TRUE)} non-positive value{?s}."
    ))
  }
  if (any(height_m <= 0, na.rm = TRUE)) {
    cli_abort(c(
      "{.arg height_m} must contain only positive values.",
      "x" = "Found {sum(height_m <= 0, na.rm = TRUE)} non-positive value{?s}."
    ))
  }
  weight_kg / height_m^2
}

calc_bmi("80", 1.80)
```

```
Error in `calc_bmi()`:
! `weight_kg` must be numeric.
✖ You supplied a <character> vector.
```

```
calc_bmi(c(70, 80), c(1.70, 1.75, 1.80))
```

```
Error in `calc_bmi()`:
! `weight_kg` and `height_m` must have the same length.
✖ `weight_kg` has 2 elements, `height_m` has 3.
```

```
calc_bmi(c(70, -5), c(1.70, 1.80))
```

```
Error in `calc_bmi()`:
! `weight_kg` must contain only positive values.
✖ Found 1 non-positive value.
```

# tryCatch() und withCallingHandlers()

Die bisherigen Werkzeuge dienen dem *Erzeugen* von Fehlern und Warnungen. Manchmal muss man sie jedoch *behandeln* – Fehler abfangen und fehlertolerant weiterarbeiten, statt abzustürzen.

## Grundmuster

`tryCatch()` wertet einen Ausdruck aus und führt eine Handler-Funktion aus, wenn eine Condition ausgelöst wird:

```
result <- tryCatch(
  log("abc"),
  error = function(e) {
    message(glue::glue("Caught an error: {e$message}"))
    NA_real_
  }
)
```

```
Caught an error: Nicht-numerisches Argument für mathematische Funktion
```

```
result
```

```
[1] NA
```

## Praxisbeispiel: Robustes Datei-Einlesen

```
safe_read_csv <- function(path) {
  tryCatch(
    read_csv(path, show_col_types = FALSE),
    error = function(e) {
      warning(glue::glue("Could not read '{path}': {e$message}"), call. = FALSE)
      NULL
    }
  )
}

result <- safe_read_csv("nonexistent_file.csv")
```

```
Warning: Could not read 'nonexistent_file.csv': 'nonexistent_file.csv' does not
exist in current working directory:
'C:/Users/PaulSchmidt-BioMathG/AppData/Local/Temp/RtmpeIZo1W/filee82c241717ec/
content/r_more'.
```

```
result
```

```
NULL
```

Die Funktion gibt `NULL` zurück statt abzustürzen, was sie sicher in Pipelines verwendbar macht, in denen einige Dateien fehlen könnten.

## Warnungen und mehrere Conditions behandeln

Man kann Handler für verschiedene Condition-Typen registrieren:

```
carefully <- function(expr) {
  tryCatch(
    expr,
    error = function(e) glue::glue("ERROR: {e$message}"),
    warning = function(w) glue::glue("WARNING: {w$message}")
  )
}

carefully(log(10))
```

```
[1] 2.302585
```

```
carefully(log(-1))
```

```
WARNING: NaNs wurden erzeugt
```

```
carefully(log("abc"))
```

```
ERROR: Nicht-numerisches Argument für mathematische Funktion
```

Zu beachten ist, dass wenn ein `warning`-Handler in `tryCatch()` feuert, er das Abschließen des ursprünglichen Ausdrucks verhindert. Wenn man Warnungen protokollieren und trotzdem das Ergebnis erhalten möchte, verwendet man stattdessen `withCallingHandlers()`:

⚠ Fortgeschritten: withCallingHandlers() (zum Aufklappen)

Anders als `tryCatch()` führt `withCallingHandlers()` den Handler aus, ohne die ursprüngliche Berechnung abzubrechen. Das ist nützlich, wenn man Warnungen sammeln und trotzdem das Ergebnis erhalten möchte. Der `<<-`-Operator weist einer Variablen in der übergeordneten Umgebung zu, und `invokeRestart("muffleWarning")` unterdrückt die Warnung nach dem Protokollieren:

```
logged_warnings <- character(0)

result <- withCallingHandlers(
  {
    x <- as.numeric(c("1", "abc", "3"))
    mean(x, na.rm = TRUE)
  },
  warning = function(w) {
    logged_warnings <<- c(logged_warnings, w$message)
    invokeRestart("muffleWarning")
  }
)

result
```

```
[1] 2
```

```
logged_warnings
```

```
[1] "NAs durch Umwandlung erzeugt"
```

Für die meisten Anwendungsfälle reicht `tryCatch()` aus. `withCallingHandlers()` braucht man nur, wenn die Ausführung nach einer Warnung fortgesetzt werden soll.

## Brücke zu purrr

Wenn man über viele Elemente iteriert, führt Kapitel 11 `safely()` und `possibly()` aus purrr ein – Convenience-Wrapper um `tryCatch()`, die für die Verwendung mit `map()` konzipiert sind:

```
# tryCatch wrapper written manually
safe_log <- function(x) tryCatch(log(x), error = function(e) NA_real_)
map_dbl(list(1, "a", 3), safe_log)
```

```r
# Same thing with possibly()
map_dbl(list(1, "a", 3), possibly(log, otherwise = NA_real_))
```

### Übung: Robustes Datei-Einlesen

Man schreibe eine Funktion `try_read_csv()`, die einen Dateipfad entgegennimmt, versucht ihn mit `read_csv()` einzulesen, und `NULL` mit einer `message()` zurückgibt, falls die Datei nicht gelesen werden kann.

Zum Testen: eine temporäre CSV-Datei mit `write_csv()` und `tempfile()` erstellen, dann die Funktion sowohl mit der echten Datei als auch mit einem falschen Pfad aufrufen.

### Lösung

```r
try_read_csv <- function(path) {
  tryCatch(
    read_csv(path, show_col_types = FALSE),
    error = function(e) {
      message(glue::glue("Failed to read '{path}': {e$message}"))
      NULL
    }
  )
}

# Create a temporary test file
tmp_file <- tempfile(fileext = ".csv")
write_csv(mtcars %>% head(5), tmp_file)

# Test with existing file
try_read_csv(tmp_file)
```

```
# A tibble: 5 × 11
    mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1  21       6   160   110  3.9   2.62  16.5     0     1     4     4
2  21       6   160   110  3.9   2.88  17.0     0     1     4     4
3  22.8     4   108    93  3.85  2.32  18.6     1     1     4     1
4  21.4     6   258   110  3.08  3.22  19.4     1     0     3     1
5  18.7     8   360   175  3.15  3.44  17.0     0     0     3     2
```

```r
# Test with non-existent file
try_read_csv("this_file_does_not_exist.csv")
```

```
Failed to read 'this_file_does_not_exist.csv': 'this_file_does_not_exist.csv'
does not exist in current working directory: 'C:/Users/PaulSchmidt-BioMathG/
AppData/Local/Temp/RtmpeIZo1W/filee82c241717ec/content/r_more'.
```

```
NULL
```

```r
# Clean up
file.remove(tmp_file)
```

```
[1] TRUE
```

# Vergleichstabelle

| Funktion | Fatal? | Eigene Meldung? | Paket | Am besten für |
|---|---|---|---|---|
| `stop()` | Ja | Ja (manuell) | base | Einfache, klare Fehler |
| `stopifnot()` | Ja | Nein (auto) | base | Schnelle interne Assertions |
| `warning()` | Nein | Ja (manuell) | base | Nicht-fatale Probleme |
| `message()` | Nein | Ja (manuell) | base | Informativer Output |
| `assert_that()` | Ja | Semi-auto | assertthat | Lesbare Typ-Prüfungen |
| `cli_abort()` | Ja | Ja (formatiert) | cli | Benutzerseitige Fehler mit Markup |
| `cli_warn()` | Nein | Ja (formatiert) | cli | Benutzerseitige Warnungen mit Markup |
| `cli_inform()` | Nein | Ja (formatiert) | cli | Benutzerseitige Messages mit Markup |
| `tryCatch()` | N/A | N/A | base | Fehler abfangen und recovern |

Für neuen Code empfiehlt sich eine praktische Kombination: `cli_abort()` / `cli_warn()` für benutzerseitige Funktionen, `stopifnot()` für interne Prüfungen und `tryCatch()` für fehlertolerante Weiterverarbeitung.

# Best Practices

**Am Funktionsrand validieren.** Input-Validierung gehört an den Anfang einer Funktion, vor jegliche Berechnung. Dieses “Fail Fast”-Prinzip bedeutet, dass die Funktion sofort mit einer klaren Meldung stoppt, anstatt teure Arbeit zu verrichten und später mit einer kryptischen Meldung zu scheitern.

**Spezifisch beschreiben, was schiefgegangen ist.** Eine gute Fehlermeldung beantwortet zwei Fragen: was ist falsch und was wurde erwartet. Man vergleiche `stop("Invalid input")` mit `cli_abort("{.arg weight_kg} must be numeric, not {.cls {class(weight_kg)}}.")` – die zweite Variante sagt dem Aufrufer genau, wie das Problem zu beheben ist.

**Den richtigen Schweregrad verwenden.** Fehler für Probleme, die ein valides Ergebnis verhindern, Warnungen für Situationen, in denen das Ergebnis valide aber potenziell unerwartet ist, und Messages für rein informativen Output.

**Interne Hilfsfunktionen nicht über-validieren.** Gründliche Validierung sollte öffentlichen Funktionen vorbehalten bleiben. Interne Helfer, die nur aus dem eigenen validierten Code aufgerufen werden, können ihren Inputs vertrauen:

```r
# Public function - validates inputs
calculate_stats <- function(data, col) {
  if (!is.data.frame(data)) cli_abort("{.arg data} must be a data frame.")
  if (!col %in% names(data)) cli_abort("Column {.val {col}} not found in {.arg
data}.")

  values <- data[[col]]
  list(center = compute_center(values), spread = compute_spread(values))
}

# Internal helper - no validation needed
compute_center <- function(x) {
  c(mean = mean(x, na.rm = TRUE), median = median(x, na.rm = TRUE))
}
```

**Für Menschen formatieren.** cli für benutzerseitige Meldungen verwenden, wo Lesbarkeit zählt, und `stopifnot()` für entwicklerseitige Assertions, wo Kürze wichtig ist.

# Zusammenfassung

Dieses Kapitel hat das Spektrum der Input-Validierungs- und Fehlerbehandlungs-Werkzeuge in R behandelt.

> ℹ Wichtige Erkenntnisse
>
> 1. **Signal-Schweregrad:** `stop()` / `cli_abort()` für fatale Fehler, `warning()` / `cli_warn()` für nicht-fatale Probleme, `message()` / `cli_inform()` für informativen Output.
> 2. **Früh validieren:** Inputs am Anfang von benutzerseitigen Funktionen prüfen (“Fail Fast”). Interne Helfer, die aus bereits validiertem Code aufgerufen werden, können ihren Inputs vertrauen.
> 3. **Spezifisch sein:** Gute Fehlermeldungen beantworten “Was ist schiefgegangen?” und “Was wurde erwartet?”. Das cli-Paket bietet Inline-Markup (`{.arg}`, `{.cls}`, `{.val}`) für formatierte, informative Meldungen.
> 4. **Fehler-Recovery:** `tryCatch()` fängt Fehler ab und gibt einen Fallback-Wert zurück. Man verwendet es, um Funktionen in Pipelines robust zu machen, in denen einzelne Inputs fehlschlagen können.
> 5. **Praktische Kombination:** `cli_abort()` / `cli_warn()` für benutzerseitige Funktionen, `stopifnot()` für interne Assertions, `tryCatch()` für fehlertolerante Weiterverarbeitung.

# Bibliography