10. Input Validation

Writing robust functions with informative error messages

Dr. Paul Schmidt

To install and load all packages used in this chapter, run the following code:

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

library(tidyverse)
```

Introduction

Every function makes assumptions about its inputs: a vector should be numeric, a dataframe should contain certain columns, a value should be positive. When these assumptions are violated, R either throws a cryptic error deep inside the function or – worse – silently produces wrong results. Both outcomes waste debugging time and erode trust.

Consider what happens when you pass a factor column to a function that expects numbers:

```r
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
```

The error mentions `mean()` and “not meaningful for factors”, but the real problem is that someone passed a non-numeric column. With input validation, we catch this immediately:

```r
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
```

Chapter 9 introduced `stop()`, `stopifnot()`, and `match.arg()` for defensive programming. This chapter expands the toolkit: `warning()` and `message()` for non-fatal signals, the assertthat package for readable assertions, the cli package for formatted error messages, and `tryCatch()` for graceful error recovery.

Review: stop() and stopifnot()

Since Chapter 9 covered these in detail, here is just a brief recap focusing on their **limitations**.

`stop()` gives full control over the error message. `stopifnot()` is more compact but its auto-generated messages are hard to read:

```r
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
```

The message `all(x >= 0 & x <= 1, na.rm = TRUE) is not TRUE` reads like code rather than an explanation. A human-friendly message would say “Found values outside the range [0, 1]”. This readability gap motivates the assertthat and cli packages discussed later.

As a rule of thumb: use `stopifnot()` for internal assertions that only developers will see, and `stop()` (or `cli_abort()`) for user-facing validation where message quality matters.

warning() and message()

Not every problem should halt execution. R provides `warning()` for situations where something is probably wrong, and `message()` for purely informational output.

warning(): Something Might Be Off

A warning signals that the function produced a result, but the caller should be aware of a potential issue:

```r
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
```

The `call. = FALSE` argument suppresses the function call in the warning, making the output cleaner.

message(): Informational Output

A message is purely informational – nothing is wrong, you are just keeping the user informed:

```
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
```

Suppressing and Choosing the Right Signal

Users can selectively silence warnings and messages with `suppressWarnings()` and `suppressMessages()`. This only works because `warning()` and `message()` use separate signaling mechanisms – if you had used `cat()` instead, users would have no way to suppress the output programmatically.

The choice between `stop()`, `warning()`, and `message()` comes down to severity:

Signal	Fatal?	Use when…
`stop()`	Yes	The function *cannot* produce a valid result
`warning()`	No	The result *might* be problematic
`message()`	No	Everything is fine, just FYI

A practical test: if someone wraps your function in `suppressWarnings()` and gets a wrong result, your signal should have been an error, not a warning.

Exercise: Function with Warnings

Write a function `safe_mean()` that takes a numeric vector `x`, checks that it is actually numeric (stop with error if not), issues a **warning** if NAs are present (reporting how many), and returns the mean with `na.rm = TRUE`.

Test with `airquality$Ozone` (37 NAs) and `c(1, 2, 3)` (no NAs).

Solution

```r
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
```

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

```
[1] 2
```

Structured Validation with assertthat

The assertthat package sits between `stopifnot()` (compact but cryptic) and `stop()` (readable but verbose). Its `assert_that()` function works like `stopifnot()` but generates human-readable error messages:

```r
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
```

Built-in Helpers

The package provides type-checking functions with clear error messages:

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

```
[1] TRUE
```

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

```
[1] TRUE
```

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

```
[1] TRUE
```

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

```
[1] TRUE
```

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

```
[1] TRUE
```

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

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

see_if() and on_failure()

`see_if()` checks a condition without stopping execution, returning `TRUE` / `FALSE` with a message attribute. `on_failure()` lets you define custom error messages for your own check functions.

⚠ Advanced: on_failure() (click to expand)

The `on_failure()` mechanism uses metaprogramming to customize error messages. This is a niche feature that most users will not need:

```r
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)
```

ℹ Note

The assertthat package is in maintenance mode (last CRAN update 2019). For new code, `cli::cli_abort()` and `rlang::abort()` are recommended — they provide richer error messages with inline formatting and are actively maintained.

Informative Errors with cli

The cli package provides modern, formatted output for R. Its `cli_abort()`, `cli_warn()`, and `cli_inform()` replace `stop()`, `warning()`, and `message()` with two advantages: inline markup and automatic value formatting.

Basic Usage

The cli functions accept a character vector where each element becomes a line. Named elements get special bullet prefixes – `"x"` for problems, `"i"` for info, `"!"` for warnings:

```r
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.
```

```r
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

Curly braces format values according to their role in the message:

Markup	Purpose	Markup	Purpose
`{.arg name}`	Argument	`{.val value}`	A value
`{.var name}`	Variable	`{.cls class}`	Class name
`{.code code}`	Code snippet	`{.fn name}`	Function name

The `?` operator handles pluralization, and R expressions are interpolated just like in `glue::glue()`:

```r
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() and cli_inform()

The same markup works for warnings and messages:

```r
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
```

Exercise: Rewrite stop() to cli_abort()

The following BMI calculator uses basic `stop()` messages. Rewrite the validation to use `cli_abort()` with inline markup. Each error should have a header line and at least one `"x"` bullet.

```r
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
}
```

Test with: `calc_bmi("80", 1.80)`, `calc_bmi(c(70, 80), c(1.70, 1.75, 1.80))`, and `calc_bmi(c(70, -5), c(1.70, 1.80))`.

ℹ Solution

```r
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.
```

```r
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.
```

```r
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() and withCallingHandlers()

The tools above are about *producing* errors and warnings. Sometimes you need to *handle* them – catching errors and recovering gracefully instead of crashing.

Basic Pattern

`tryCatch()` evaluates an expression and runs a handler function if a condition is raised:

```r
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
```

```r
result
```

```
[1] NA
```

Practical Example: Robust File Reading

```r
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/filee82c5cfa6047/
content/r_more'.
```

```r
result
```

```
NULL
```

The function returns `NULL` instead of crashing, making it safe to use in pipelines where some files might be missing.

Handling Warnings and Multiple Conditions

You can register handlers for different condition types:

```r
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
```

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

```
WARNING: NaNs wurden erzeugt
```

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

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

Note that when a `warning` handler fires in `tryCatch()`, it prevents the original expression from completing. If you need to log warnings while still getting the result, use `withCallingHandlers()` instead:

⚠ Advanced: withCallingHandlers() (click to expand)

Unlike `tryCatch()`, `withCallingHandlers()` runs the handler without aborting the original computation. This is useful when you want to collect warnings while still getting the result. The `<<-` operator assigns to a variable in the parent environment, and `invokeRestart("muffleWarning")` suppresses the warning after logging it:

```
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"
```

For most use cases, `tryCatch()` is sufficient. Reach for `withCallingHandlers()` only when you need to continue execution after a warning.

Bridge to purrr

When iterating over many elements, Chapter 11 introduces `safely()` and `possibly()` from purrr – convenience wrappers around `tryCatch()` designed for use with `map()`:

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

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

Exercise: Robust File Reading

Write a function `try_read_csv()` that takes a file path, tries to read it with `read_csv()`, and returns `NULL` with a `message()` if the file cannot be read.

Test by creating a temporary CSV with `write_csv()` and `tempfile()`, then calling your function on both the real file and a fake path.

Solution

```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/filee82c5cfa6047/content/r_more'.
```

```
NULL
```

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

```
[1] TRUE
```

Comparison Table

Function	Fatal?	Custom message?	Package	Best for
`stop()`	Yes	Yes (manual)	base	Simple, clear errors

Function	Fatal?	Custom message?	Package	Best for
`stopifnot()`	Yes	No (auto)	base	Quick internal assertions
`warning()`	No	Yes (manual)	base	Non-fatal problems
`message()`	No	Yes (manual)	base	Informational output
`assert_that()`	Yes	Semi-auto	assertthat	Readable type checks
`cli_abort()`	Yes	Yes (formatted)	cli	User-facing errors with markup
`cli_warn()`	No	Yes (formatted)	cli	User-facing warnings with markup
`cli_inform()`	No	Yes (formatted)	cli	User-facing messages with markup
`tryCatch()`	N/A	N/A	base	Catching and recovering from errors

For most new code, a practical combination is: `cli_abort()` / `cli_warn()` for user-facing functions, `stopifnot()` for internal checks, and `tryCatch()` for graceful error recovery.

Best Practices

Validate at function boundaries. Input validation belongs at the top of a function, before any computation. This “fail fast” principle means the function stops immediately with a clear message rather than doing expensive work and failing later with a cryptic one.

Be specific about what went wrong. A good error message answers two questions: what is wrong and what was expected. Compare `stop("Invalid input")` with `cli_abort("{.arg weight_kg} must be numeric, not {.cls {class(weight_kg)}}.")` – the second tells the caller exactly how to fix the problem.

Use the right severity. Errors for problems that prevent a valid result, warnings for situations where the result is valid but potentially unexpected, and messages for purely informational output.

Don’t over-validate internal helpers. Reserve thorough validation for public-facing functions. Internal helpers called only from your own validated code can trust their inputs:

```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}.")
```

```r
  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))
}
```

Format for humans. Use cli for user-facing messages where readability matters, and `stopifnot()` for developer-facing assertions where brevity matters.

Summary

This chapter covered the spectrum of input validation and error handling tools in R.

Key Takeaways

1. **Signal Severity:** `stop()` / `cli_abort()` for fatal errors, `warning()` / `cli_warn()` for non-fatal problems, `message()` / `cli_inform()` for informational output.
2. **Validate Early:** Check inputs at the top of user-facing functions ("fail fast"). Internal helpers called from already-validated code can trust their inputs.
3. **Be Specific:** Good error messages answer "what went wrong?" and "what was expected?". The cli package provides inline markup (`{.arg}`, `{.cls}`, `{.val}`) for formatted, informative messages.
4. **Error Recovery:** `tryCatch()` catches errors and returns a fallback value. Use it to make functions robust in pipelines where some inputs may fail.
5. **Practical Combination:** `cli_abort()` / `cli_warn()` for user-facing functions, `stopifnot()` for internal assertions, `tryCatch()` for graceful recovery.

Bibliography