Package: optree (via r-universe)

June 2, 2026

Title Hierarchical Runtime Configuration Management

Version 0.1.1

Description Provides tools for managing nested, multi-level configuration systems with runtime mutability, type validation, and default value management. Supports creating hierarchical options managers with customizable validators for scalar and vector types (numeric, character, logical), enumerated values, bounded ranges, and complex structures like XY pairs. Options can be dynamically modified at runtime while maintaining type safety through validator functions, and easily reset to their default values when needed.

License MIT + file LICENSE

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.3

Depends R (>= 4.1.0)

Suggests testthat (>= 3.0.0), knitr, rmarkdown

VignetteBuilder knitr

Config/testthat/edition 3

URL https://optree.bangyou.me/, https://github.com/byzheng/optree

BugReports https://github.com/byzheng/optree/issues

Repository https://byzheng.r-universe.dev

Date/Publication 2026-03-03 00:28:18 UTC

RemoteUrl https://github.com/byzheng/optree

RemoteRef HEAD

RemoteSha a464c9e2d05ac358e85ad9dc0c04b306bc8f59f0

Contents

`create_options_manager`

Create a hierarchical, mutable options manager

Description

`create_options_manager()` creates a runtime configuration manager that supports **nested options**, **group validation**, and **resetting to defaults**. It is ideal for managing complex, interdependent settings in R packages or projects.

Usage

```
create_options_manager(defaults, validators = list())
```

Arguments

`defaults`	A named list specifying the default values of the options. Nested lists can be used to represent hierarchical groups of related options.
`validators`	An optional named list of functions used to validate options. Each function should take a single argument (the value being set) and throw an error if the value is invalid. Names correspond to option paths, e.g., `"thermaltime"` for a top-level group.

Details

This manager allows you to safely store and update **related groups of options**. For example, a `thermaltime` group might have `x` and `y` vectors that must always have the same length. Using validators ensures that these relationships are maintained whenever options are updated.

The manager supports **merge-aware updates**, meaning that if a nested list is provided, only the specified elements are updated while others are preserved.

Dot-separated path notation: The `set()` function now accepts path strings like `"phenology.thermaltime.y" = c(0, 25, 0)`, which are automatically converted to nested lists internally. This provides a more concise syntax for updating deeply nested options without reconstructing the entire hierarchy.

Transactional updates: If validation fails during a `set()` call, all changes are rolled back and the options remain in their previous state. This ensures that the options manager is always in a consistent state.

Value

A list with three functions:

`get(name = NULL)` Retrieve the current value of an option. Use a dot-separated string for nested options, e.g., `"thermaltime.x"`. If `name` is NULL, returns all current options.

`set(...)` Update one or more options by name. Accepts named arguments in two formats: (1) dot-separated paths like `"phenology.thermaltime.y" = ...` or (2) nested lists like `thermaltime = list(x = ..., y = ...)`. Both styles can be mixed in a single call. Validators are automatically applied if provided.

`reset()` Reset all options to their default values.

Examples

```
# Define a validator for a group
thermaltime_validator <- function(value) {
    if (!is.list(value) || !all(c("x", "y") %in% names(value))) {
        stop("thermaltime must be a list with both x and y")
    }
   if (length(value$x) != length(value$y)) stop("thermaltime x and y must have same length")
}

# Create a manager
canola <- create_options_manager(
    defaults = list(
        thermaltime = list(x = c(2, 30, 35), y = c(0, 28, 0)),
        frost_threshold = 0
    ),
    validators = list(
        "thermaltime" = thermaltime_validator
    )
)

# Access and update (both methods work)
canola$get("thermaltime.x")

# Method 1: Use dot-separated path strings (concise!)
canola$set("thermaltime.y" = c(0, 25, 0))
canola$set("thermaltime.x" = c(5, 25, 40))

# Method 2: Use nested list (traditional way)
canola$set(thermaltime = list(x = c(5, 25, 40), y = c(0, 20, 0)))

# Method 3: Mix both styles in one call
canola$set(
  "thermaltime.x" = c(10, 30, 45),
  frost_threshold = -2
)

# Reset to defaults
canola$reset()
```

v_character_scalar *Validator for Character Scalar Values*

Description

v_character_scalar() returns a validator function that checks if a value is a single character value. This is useful as a validator function for options managers created with create_options_manager().

Usage

```
v_character_scalar()
```

Value

A validator function that takes a value x and raises an error if:

- x is not a single character value
- x is an empty string

Examples

```
# Create a validator for non-empty character scalars
validator <- v_character_scalar()

# Valid input
validator("hello")

# Invalid inputs (would raise errors)
try(validator(c("hello", "world")))  # vector, not scalar
try(validator(123))  # numeric, not character
```

v_enum *Validator for Enumerated Character Values*

Description

v_enum() returns a validator function that checks if a value is a single character value matching one of a predefined set of choices. This is useful for options that must be one of several allowed values.

Usage

```
v_enum(choices)
```

Arguments

choices	A character vector of allowed values.

Value

A validator function that takes a value `x` and raises an error if:

- `x` is not a single character value
- `x` is not in the predefined `choices`

Examples

```
# Create a validator for one of several color choices
validator <- v_enum(choices = c("red", "green", "blue"))

# Valid inputs
validator("red")
validator("blue")

# Invalid inputs (would raise errors)
try(validator("yellow"))  # not in choices
try(validator(c("red", "blue")))  # vector, not scalar
try(validator(1))  # numeric, not character
```

`v_logical_scalar` *Validator for Logical Scalar Values*

Description

`v_logical_scalar()` returns a validator function that checks if a value is a single logical value. This is useful as a validator function for options managers created with `create_options_manager()`.

Usage

```
v_logical_scalar()
```

Value

A validator function that takes a value `x` and raises an error if `x` is not a single logical value.

Examples

```
# Create a validator for logical scalars
validator <- v_logical_scalar()

# Valid input
validator(TRUE)

# Invalid inputs (would raise errors)
try(validator(c(TRUE, FALSE))) # vector, not scalar
try(validator(1)) # numeric, not logical
```

v_numeric_range *Validator for Numeric Values Within a Range*

Description

v_numeric_range() returns a validator function that checks if a value is a single numeric value within a specified range. This is useful as a validator function for bounded numeric options in options managers created with create_options_manager().

Usage

```
v_numeric_range(min = -Inf, max = Inf)
```

Arguments

min	Minimum allowed value (inclusive). Defaults to -Inf (no lower bound).
max	Maximum allowed value (inclusive). Defaults to Inf (no upper bound).

Value

A validator function that takes a value x and raises an error if:

- x is not a single numeric value
- x is less than min or greater than max

Examples

```
# Create a validator for values between 0 and 1
validator <- v_numeric_range(min = 0, max = 1)

# Valid inputs
validator(0.5)
validator(0)
validator(1)

# Invalid inputs (would raise errors)
try(validator(-0.1)) # below minimum
try(validator(1.5)) # above maximum
try(validator(c(0.5, 0.7))) # vector, not scalar
```

v_numeric_scalar *Validator for Numeric Scalar Values*

Description

`v_numeric_scalar()` returns a validator function that checks if a value is a single numeric value. This is useful as a validator function for options managers created with `create_options_manager()`.

Usage

```
v_numeric_scalar()
```

Value

A validator function that takes a value `x` and raises an error if `x` is not a single numeric value.

Examples

```
# Create a validator for numeric scalars
validator <- v_numeric_scalar()

# Valid input
validator(42)

# Invalid inputs (would raise errors)
try(validator(c(1, 2, 3))) # vector, not scalar
try(validator("text")) # not numeric
```

v_numeric_vector *Validator for Numeric Vectors*

Description

`v_numeric_vector()` returns a validator function that checks if a value is a numeric vector meeting specified length and finiteness requirements. This is useful for options requiring numeric sequences or datasets.

Usage

```
v_numeric_vector(min_len = 1, finite = TRUE)
```

Arguments

`min_len`	Minimum length required for the vector. Defaults to 1.
`finite`	If TRUE (default), rejects non-finite values (Inf, -Inf, NaN). If FALSE, non-finite values are allowed.

Value

A validator function that takes a value `x` and raises an error if:

- `x` is not numeric
- `x` has fewer than `min_len` elements
- `x` contains NA values
- `finite` is TRUE and `x` contains non-finite values

Examples

```
# Create a validator for numeric vectors of at least length 3
validator <- v_numeric_vector(min_len = 3)

# Valid input
validator(c(1, 2, 3))
validator(c(0.5, 1.5, 2.5, 3.5))

# Invalid inputs (would raise errors)
try(validator(c(1, 2)))  # too short
try(validator(c(1, NA, 3)))  # contains NA
try(validator(c(1, Inf, 3)))  # contains non-finite value
try(validator("not numeric"))  # not numeric
```

---

v_xypair *Validator for XY Pair Lists*

Description

`v_xypair()` returns a validator function that checks if a value is a list with paired `x` and `y` components of equal length. This is useful for validating paired data structures in options managers created with `create_options_manager()`.

Usage

```
v_xypair(min_len = 1, max_len = NULL)
```

Arguments

`min_len`	Minimum length required for the `x` and `y` vectors. Defaults to 1.
`max_len`	Maximum length allowed for the `x` and `y` vectors. Defaults to `NULL` (no upper bound).

Value

A validator function that takes a value (typically a list with `x` and `y` components) and raises an error if:

- The value is not a list
- The list does not contain both `x` and `y` named elements
- Either `x` or `y` is NULL
- Either `x` or `y` is not an atomic vector
- Either `x` or `y` contains NA values
- The `x` and `y` vectors have different lengths
- The vectors are shorter than `min_len`
- The vectors are longer than `max_len` (when `max_len` is not `NULL`)

Examples

```
# Create a validator for XY pairs with minimum length 2
validator <- v_xypair(min_len = 2)

# Valid input
validator(list(x = c(1, 2, 3), y = c(10, 20, 30)))

# Invalid inputs (would raise errors)
try(validator(list(x = c(1), y = c(10))))
try(validator(list(x = c(1, 2), y = c(10, 20, 30)))) # different lengths
try(validator(list(x = c(1, NA), y = c(10, 20)))) # contains NA
try(validator(list(x = c(1, 2)))) # missing y
```

Index