Package 'fuj'

October 8, 2026

Type Package

Title Functions and Utilities for Jordan

Version 0.3.0

Maintainer Jordan Mark Barbone <jmbarbone@gmail.com>

Description Provides core functions and utilities for packages and other code developed by Jordan Mark Barbone.

License MIT + file LICENSE

Encoding UTF-8

Language en-US

Depends R (>= 4.1)

Suggests spelling, testthat (>= 3.0.0)

URL https://jmbarbone.github.io/fuj/, https://github.com/jmbarbone/fuj

BugReports https://github.com/jmbarbone/fuj/issues

Config/testthat/edition 3

Config/roxygen2/version 8.1.0

NeedsCompilation no

Author Jordan Mark Barbone [aut, cph, cre] (ORCID: <https://orcid.org/0000-0001-9788-3628>)

Repository CRAN

Date/Publication 2026-10-08 07:50:09 UTC

Contents

alias_arithmetic 2
alias_extract 3
collapse 3
colons 4
conditions 5
delay 6

encode . . . 7
exattr . . . 8
flip . . . 9
fp . . . 10
hot . . . 11
if_null . . . 12
include . . . 13
list0 . . . 15
match_arg . . . 15
match_ext . . . 17
muffle . . . 18
names . . . 19
negate . . . 20
new_condition . . . 21
os . . . 22
quick_df . . . 23
require_namespace . . . 24
struct . . . 25
vap . . . 26
verbose . . . 30
yes_no . . . 32

Index **33**

`alias_arithmetic` *Arithmetic wrappers*

Description

Arithmetic wrappers

Value

See base::Arithmetic

Examples

```
      add(7, 2) # +
 subtract(7, 2) # -
 multiply(7, 2) # *
   divide(7, 2) # /
raise_power(7, 2) # ^
  remainder(7, 2) # %%
 divide_int(7, 2) # %/%
```

alias_extract *Extract and replace aliases*

Description

Extract and replace aliases

Value

See base::Extract

Examples

```
df <- quick_df(list(a = 1:5, b = 6:10))
# alias of `[`
subset1(df, 1)
subset1(df, 1, )
subset1(df, , 1)
subset1(df, , 1, drop = FALSE)

# alias of `[[`
subset2(df, 1)
subset2(df, 1, 2)

# alias of `$`
subset3(df, a)
subset3(df, "b")
subset3(df, "foo")

# alias of `[<-`
subassign1(df, "a", , 2)
```

collapse *Collapse*

Description

Simple wrapper for concatenating strings

Usage

```
collapse(..., sep = "")
```

Arguments

...	one or more R objects, to be converted to character vectors.
sep	a character string to separate the terms. Not NA_character_.

Value

A `character` vector of concatenated values. See `base::paste()` for more details.

Examples

```
collapse(1:10)
collapse(list("a", b = 1:2))
collapse(quick_df(list(a = 1:3, b = 4:6)), sep = "-")
```

`colons` *Colons*

Description

Get an object from a namespace

Usage

```
package %::% name

package %:::% name

package %colons% name
```

Arguments

`package`	Name of the package
`name`	Name to retrieve

Details

The functions mimic the use of `::` and `:::` for extracting values from namespaces. `%colons%` is an alias for `%::%`.

Value

The variable `name` from package `package`

WARNING

To reiterate from other documentation: it is not advised to use `:::` in your code as it will retrieve non-exported objects that may be more likely to change in their functionality that exported objects.

See Also

```
help("::")
```

Examples

```
identical("base" %::% "mean", base::mean)
"fuj" %:::% "colons_example" # unexported value
```

---

conditions *Default Conditions*

Description

Default Conditions

Usage

```
message_condition(..., class = NULL)

msg(..., class = NULL)

verbose_message(message, call = NULL)

error_condition(..., class = NULL)

err(..., class = NULL)

input_error(message = "invalid input", ...)

value_error(message = "invalid value", ...)

class_error(message = "invalid class", ...)

type_error(message = "invalid type", ...)

interactive_error(message = "must be used in an interactive session", ...)

namespace_error(package)

development_error(..., package = find_package())

defunct_error(..., package = find_package())

internal_error(
  message = c(sprintf("An internal error occurred in '%s'.", package),
    "  Please report this to the package maintainer."),
  ...,
  package = find_package()
)
```

```
warning_condition(..., class = NULL)

wrn(..., class = NULL)

input_warning(message = "invalid input", ...)

value_warning(message = "invalid value", ...)

class_warning(message = "invalid class", ...)

interactive_warning(message = "must be used in an interactive session", ...)

namespace_warning(package)

development_warning(..., package = find_package())

internal_warning(
  message = c(sprintf("An internal error occurred in '%s'.", package),
    "  Please report this to the package maintainer."),
  ...,
  package = find_package()
)

deprecated_warning(..., package = find_package())
```

Arguments

class	A class names for the condition; message_condition(), warning_condition(), error_condition() will always use classes "message", "warning", "error", respectively.
message, ...	A character vector of message components; message is used for anything with a default message.
call	The call that generated the condition
package	Package name. Default will attempt to find the package by the calling environment, which must return a character value. This will also be passed to the package argument in new_condition()

---

delay *Delay the evaluation of an expression until the end of a function or block.*

Description

Delay the evaluation of an expression until the end of a function or block.

Usage

```
delay(expr, envir = parent.frame(), add = TRUE, after = FALSE)
```

Arguments

expr	An expression to be evaluated later.
envir	The environment in which to evaluate the expression.
add	if TRUE, add expr to be executed after any previously set expressions (or before if after is FALSE); otherwise (the default) expr will overwrite any previously set expressions.
after	if add is TRUE and after is FALSE, then expr will be added on top of the expressions that were already registered. The resulting last in first out order is useful for freeing or closing resources in reverse order.

Value

None, called for its side-effects

Examples

```
local({
  x <- 1L
  delay(message("x = ", x))
  x <- 2L
})
```

---

encode *Encode/factor*

Description

Low-level encoding and factor building

Usage

```
encode(x, from, to, strict = FALSE, exclude = NULL)

fact(x, levels = NULL, exclude = NULL)
```

Arguments

x	A vector of values
from, to	Vectors of the same length. Values in x are matched to from and replaced with the corresponding value in to.
strict	If TRUE, values in x that are not matched to from will be replaced with NA. If FALSE, they will be left unchanged.

exclude	Values in x that will not be matched when recoding; exclude will take priority over from in encode().
levels	A vector of unique values. If NULL, the unique values in x are used.

Details

encode() is a general purpose function for replacing values in a vector.

fact() is a low-level function for building factor vectors. It does not perform any sort()ing of levels (unlink base::factor()). Re-leveling can be done by applying encode() to the levels of a factor object.

Value

encode() A vector of the same length as x with values replaced according to from and to.

fact() A factor vector with levels corresponding to the unique values in x (or levels if provided).

Examples

```
fact(strsplit("factor function", "")[[1L]], exclude = " ")

# encode() can be used to for the same utility as factor(x, levels, labels)
# (note: applying to levels can be more efficient)
(x <- fact(strsplit("jordan", "")[[1L]]))
levels(x) <- encode(levels(x), from = c("a", "o"), to = "*")
x
```

---

exattr *Exact attributes*

Description

Get the exact attributes of an object

Usage

```
exattr(x, which)

x %attr% which
```

Arguments

x	an object whose attributes are to be accessed.
which	a non-empty character string specifying which attribute is to be accessed.

Value

See base::attr()

Examples

```
foo <- struct(list(), "foo", aa = TRUE)
  attr(foo, "a")  # TRUE : partial match successful
exattr(foo, "a")  # NULL : partial match failed
exattr(foo, "aa") # TRUE : exact match
```

`flip` *Flip*

Description

Flip an object.

Usage

```
flip(x, ...)

## Default S3 method:
flip(x, ...)

## S3 method for class 'matrix'
flip(x, by = c("rows", "columns"), keep_rownames = NULL, ...)

## S3 method for class 'data.frame'
flip(x, by = c("rows", "columns"), keep_rownames = NULL, ...)
```

Arguments

`x`	An object
`...`	Additional arguments passed to methods
`by`	Flip by `"rows"` or `"columns"` (partial matches accepted)
`keep_rownames`	Logical, if `TRUE` will not reset row names; `NULL`

Value

A `vector` of values, equal length of `x` that is reversed or a `data frame` with flipped rows/columns

Examples

```
flip(letters[1:3])
flip(seq.int(9, -9, by = -3))
flip(head(iris))
flip(head(iris), keep_rownames = TRUE)
flip(head(iris), by = "col")
```

fp *File path operations*

Description

`is_path()` checks for either a `file_path` class or an `fs_path`, the latter useful for the `fs` package.

`file_path()` is an alias for `fp()` and `is_file_path()` is an alias for `is_path()`.

`set_file_ext()` changes the file extension of a file path, removing any existing extension first. `file_ext<-()` serves as an alias.

`np()` will normalize the file path, always using `"/"` as the path separator, and without signaling errors or warnings for non-existent paths.

Paths can be constructed with `/` and `+` methods:

```
fp("here") / "subdir" + "ext"
```

This will create a file path for `here/subdir.ext`, with the extension added after. **Note**: `fp()` can be replaced with `np()`.

Usage

```
fp(...)

np(...)

file_path(...)

norm_path(...)

is_path(x)

is_file_path(x)

set_file_ext(x, value, compression = TRUE)

file_ext(x, compression = TRUE) <- value
```

Arguments

`...`	Path components, passed to `base::file.path()`
`x`	An object to test
`value`	The new file extension. When `NULL`, removes the current extension
`compression`	If `TRUE`, also removes common compression extensions

Details

Lightweight file path functions

Value

- fp()/file_path(): A character vector with a "file_path" class
- np()/norm_path(): A character vector with a "file_path" class, with normalized paths
- is_path()/is_file_path(): A TRUE or FALSE value
- set_file_ext()/file_ext<-(): The file path with the updated extension

Examples

```
fp("here")
fp("~/there")
fp("back\\slash")
fp("remove//extra\\\\slashes")
fp("a", c("b", "c"), "d")

# supports / and +
(x <- fp("here") / "subdir" + "ext")

# change file extension
file_ext(x) <- "txt"
x
file_ext(x) <- ".txt.gz"
x
file_ext(x) <- NULL
x
```

---

hot *Hold or Toss*

Description

Hold or Toss

Usage

```
hold(x, i, na = c("drop", "keep"))

toss(x, i, na = c("keep", "drop"))
```

Arguments

x	A vector of values
i	An indication of which subset values to action. This can be a logical vector, an integer vector, or a function that takes x as an argument and returns a logical or integer vector
na	How to handle NA values in when p is a logical vector, or a function that returns a logical vector. hold() defaults to dropping NA values, while toss() defaults to keeping NA values. When p is an integer vector, NA values are always dropped.

Value

- `hold()` **retains** values in `x` matched against `i`
- `toss()` **removes** values in `x` matched against `i`

Examples

```
x <- c(1, NA, 3, 4, Inf, 6)
twos <- function(x) x %% 2 == 0
hold(x, twos) # 4, 6
toss(x, twos) # 1, 3, NA, Inf

hold(x, twos, na = "keep") # NA, 4, Inf, 6
toss(x, twos, na = "drop") # 1, 3

i <- c(1:3, NA)
x <- letters[1:5]
hold(x, i)
toss(x, i)
```

---

if_null *Default value for* `NULL` *or no length*

Description

Replace if `NULL` or not length

Usage

```
x %||% y

x %|||% y

x %len% y
```

Arguments

`x`, `y`	If `x` is `NULL` returns `y`; otherwise `x`

Details

A mostly copy of `rlang`'s `%||%` except does not use `rlang::is_null()`, which, currently, calls the same primitive `base::is.null()` function.

Note: `%||%` is copied from `{base}` if available (**R** versions >= 4.4)

Value

`x` if it is not `NULL` or has length, depending on check

Examples

```
# replace NULL (for R < 4.4)
NULL %||% 1L
2L   %||% 1L

# replace empty
""       %|||% 1L
NA       %|||% 1L
double() %|||% 1L
NULL     %|||% 1L

# replace no length
logical() %len% TRUE
FALSE     %len% TRUE
```

`include` *Include exports in Search Path*

Description

`include()` checks whether or not the namespace has been loaded to the `base::search()` path. It uses the naming convention `include:{package}` to denote the differences from loading via `base::library()` or `base::require()`. When `exports` is `NULL`, the environment is detached from the search path if found. When `exports` is not `NULL`,

Note: This function has the specific purpose of affecting the search path. Use `options(fuj.verbose = TRUE)` or `options(verbose = TRUE)` for more information.

Usage

```
include(package, exports = NULL, lib = .libPaths(), pos = 2L, warn = NULL)
```

Arguments

`package`	A package name. This can be given as a base::name or a character string. See section `package` class handling.
`exports`	A character vector of exports. When named, these exports will be aliases as such.
`lib`	See `lib.loc` in `base::loadNamespace()`.
`pos`	An integer specifying the position in the `base::search()` path to attach the new environment.
`warn`	See `warn.conflicts` in `base::attach()`, generally. The default `NULL` converts all `messagess` with masking errors to `verboseMessages`, `TRUE` converts to `includeConflictsWarning` messages, `NA` uses `packageStartupMessages`, and `FALSE` silently ignores conflicts.

Details

Include (attach) a package and specific exports to Search Path

Value

The attached environment, invisibly.

package class handling

When package is a base::name or base::AsIs, assumed an installed package. When package is a file path (via `is_path()`) then package is assumed a file path. When just a string, a viable path is checked first; if it doesn't exist, then it is assumed a package.

When the package is `base::source()`'d the name of the environment defaults to the base name of `x` (file extension removed). However, if the object `.AttachName` is found in the sourced file, then that is used as the environment name for the `base::search()` path.

Note: `include()` won't try to *attach* an environment a second time, however, when package is a path, it must be `base::source()`ed each time to check for the `.AttachName` object. If there are any side effects, they will be repeated each time `include(path)` is called.

Examples

```
# include(package) will ensure that the entire package is attached
include(fuj)
head(ls("include:fuj"), 20)
detach("include:fuj", character.only = TRUE)

# include a single export
include(fuj, "collapse")

# include multiple exports, and alias
include(fuj, c(
  no_names = "remove_names",
  match_any = "any_match"
))

# include an export where the alias has a warn conflict
include(fuj, c(attr = "exattr"))

# note that all 4 exports are included
ls("include:fuj")

# all exports are the same
identical(collapse, fuj::collapse)
identical(no_names, fuj::remove_names)
identical(match_any, fuj::any_match)
identical(attr, fuj::exattr)
```

list0 *Listing for dots*

Description

Tries to not complain about empty arguments

Usage

```
list0(...)

lst(...)
```

Arguments

...	Arguments to collect in a list

Details

If `options(fuj.list.active = FALSE)` is set to prior to package loading (i.e., prior to `.onLoad()` called), this function becomes an alias for `base::list()`, disabling the special behavior.

Value

A `list` of `...`

Examples

```
try(list(1, ))
list0(1, )
try(list(a = 1, ))
list0(a = 1, )
try(list(a = 1, , c = 3, ))
list0(a = 1, , c = 3, )
```

match_arg *Match arguments*

Description

Much like `base::match.arg()` with a few key differences:

- Will not perform partial matching (by default)
- Will not return error messages with curly quotations
- Special handling for `NULL` values
- Allows recoding of values via named lists or lists of formulas

Usage

```
match_arg(
  arg,
  choices,
  multiple = FALSE,
  partial = getOption("fuj.match_arg.partial", FALSE),
  null = c("error", "first", "all", "null")
)
```

Arguments

arg	The argument
choices	The available choices; named lists will return the name (a character) for when matched to the value within the list element. A list of formula objects (preferred) retains the LHS of the formula as the return value when matched to the RHS of the formula.
multiple	If TRUE allows multiple values to be returned
partial	If TRUE allows partial matching via base::pmatch()
null	Controls how arg = NULL is handled

Details

Argument matching for an argument

Value

A single value from arg matched on choices

Examples

```
fruits <- function(x = c("apple", "banana", "orange")) {
  match_arg(x)
}

fruits()         # apple
try(fruits("b")) # must be exact fruits("banana")

pfruits <- function(x = c("apple", "apricot", "banana")) {
  match_arg(x, partial = TRUE)
}
pfruits()          # apple
try(pfruits("ap")) # fuj:arg_match_error
pfruits("app")     # apple

afruits <- function(x = c("apple", "banana", "orange")) {
  match_arg(x, multiple = TRUE)
}

afruits() # apple, banana, orange
```

```
# can have multiple responses
how_much <- function(x = list(too_few = 0:2, ok = 3:5, too_many = 6:10)) {
  match_arg(x)
}

how_much(1)
how_much(3)
how_much(9)

# use a list of formulas instead
ls <- list(1L ~ 0:1, 2L, 3L ~ 3:5)
sapply(0:5, match_arg, choices = ls)
```

`match_ext` *Value matching - Extensions*

Description

Non matching alternatives and supplementary functions.

Usage

```
is_in(x, table)

is_out(x, table)

x %out% table

is_within(x, table)

x %wi% table

is_without(x, table)

x %wo% table

no_match(x, table)

any_match(x, table)
```

Arguments

`x`	vector or `NULL`: the values to be matched. Long vectors are supported.
`table`	vector or `NULL`: the values to be matched against. Long vectors are not supported.

Details

Contrast with base::match(), base::intersect(), and base::%in% The functions of %wi% and %wo% can be used in lieu of base::intersect() and base::setdiff(). The primary difference is that the base functions return only unique values, which may not be a desired behavior.

Value

- %out%: A logical vector of equal length of x, table
- %wo%, %wi%: A vector of values of x
- any_match(), no_match(): TRUE or FALSE
- is_in(): see base::%in%

Examples

```
1:10 %in% c(1, 3, 5, 9)
1:10 %out% c(1, 3, 5, 9)
letters[1:5] %wo% letters[3:7]
letters[1:5] %wi% letters[3:7]

# base functions only return unique values

          c(1:6, 7:2) %wo% c(3, 7, 12)  # -> keeps duplicates
  setdiff(c(1:6, 7:2),     c(3, 7, 12)) # -> unique values

          c(1:6, 7:2) %wi% c(3, 7, 12)  # -> keeps duplicates
intersect(c(1:6, 7:2),     c(3, 7, 12)) # -> unique values
```

---

muffle *Muffle messages*

Description

Aliases for base::suppressMessages() and base::suppressWarnings()

Usage

```
muffle(expr, fun, classes = "message")

wuffle(expr, fun, classes = "warning")
```

Arguments

expr	An expression to evaluate
fun	A function to *muffle* (or *wuffle*)
classes	A character vector if classes to suppress

Value

The result of `expr` or a `function` wrapping `fun`

Examples

```
# load function
foo <- function(...) {
  message("You entered :", paste0(...))
  c(...)
}

# wrap around function or muffle the function ti's
muffle(foo(1, 2))
muffle(fun = foo)(1, 2)
sapply(1:3, muffle(fun = foo))

# silence warnings
wuffle(as.integer("a"))
sapply(list(1, "a", "0", ".2"), wuffle(fun = as.integer))
```

`names` *Set names*

Description

Sets or removes names

Usage

```
set_names(x, nm = x)

remove_names(x)

x %names% nm

is_named(x, zero_ok = TRUE)
```

Arguments

`x`	A vector of values
`nm`	A vector of names
`zero_ok`	If `TRUE` allows use of `""` as a *special* name

Value

`x` with `nm` values assigned to names (if `x` is `NULL`, `NULL` is returned)

Examples

```
set_names(1:5)
set_names(1:5, c("a", "b", "c", "d", "e"))

x <- c(a = 1, b = 2)
remove_names(x)
x %names% c("c", "d")
is_named(x)
```

`negate` *Negations*

Description

`not()` is an *alias* for base::!. `negate()` is a variation of `base::Negate()` with an extra steps to preserve the original function and maintain the function formals.

Usage

```
not(x)

negate(fun)
```

Arguments

`x`	An object
`fun`	A function

Details

Negate an outcome or function

Value

`not()` See base::!

`negate()` A `function`, which is the negation of `fun`

Examples

```
identical(not(TRUE), FALSE)

different <- negate(identical)
different(formals(identical), formals(identical))
```

new_condition *New condition*

Description

Template for a new condition. See more at base::conditions

Usage

```
new_condition(
  message = "",
  class = "fuj_condition",
  type = c("condition", "error", "warning", "message"),
  ...,
  call = NULL,
  package = find_package(),
  msg,
  pkg
)
```

Arguments

message	A message to print
class	Character string of a single condition class. If class does not end with the value used in class, the suffix is appended with an underscore (_). This can be ignored if passing class as an base::AsIs vector (i.e., I("my_class")); additionally, base::AsIs objects do not have package prepended to the name.
type	The type (additional class) of condition: error", "warning", "message", or NA, which is treated as NULL.
...	Ignored
call	A call expression
package	Control or adding package name to condition. If TRUE will try to get the current package name (via .packageName) from, presumably, the developmental package. If FALSE or NULL, no package name is prepended to the condition class as a new class. Otherwise, a package can be explicitly set with a single length character.
msg	Deprecated, see message
pkg	Deprecated, see package

Details

The use of .packageName when package = TRUE may not be valid during active development. When the attempt to retrieve the .packageName object is unsuccessful, the error is quietly ignored. However, this should be successful once the package is build and functions can then utilize this created object.

Value

A condition with the classes specified from class and type

Examples

```
# empty condition
x <- new_condition(
  "informative error message",
  class = "foo",
  type = "error"
 )
try(stop(x))

# with pkg
x <- new_condition("msg", class = "foo", type = "error", package = "bar")
# class contains multiple identifiers, including a "bar:fooError"
class(x)
# message contains package information at the end
try(stop(x))
```

---

os *Determine operating systems*

Description

Determine operating systems

Usage

```
is_windows()

is_macos()

is_linux()
```

Value

TRUE or FALSE

Examples

```
is_windows()
is_macos()
is_linux()
```

quick_df *Quick DF*

Description

This is a speedier implementation of base::as.data.frame() but does not provide the same sort of checks. It should be used with caution.

Usage

```
quick_df(x = NULL)

empty_df()

quick_dfl(...)
```

Arguments

x	A list or NULL (see return)
...	Columns as tag = value (passed to lst())

Value

A data.frame; if x is NULL a data.frame with 0 rows and 0 columns is returned (similar to calling base::data.frame() but faster). empty_df() returns a data.frame with 0 rows and 0 columns.

Examples

```
# unnamed will use make.names()
x <- list(1:10, letters[1:10])
quick_df(x)

# named is preferred
names(x) <- c("numbers", "letters")
quick_df(x)

# empty data.frame
empty_df() # or quick_df(NULL)
```

require_namespace *Require namespace*

Description

`require_namespace()` is ultimately a *check* which will produce an error on the first package that is not available or meets version requirements. Although this returns `TRUE`, it is not intended to be used in a conditional statement. Future version may return `invisible()`. For conditional checks use `available_namespace()`, which will return a named logical instead.

`is_namespace_available()` is an alias for `available_namespace()`.

Usage

```
require_namespace(package, ...)

available_namespace(package, ...)

is_namespace_available(package, ...)
```

Arguments

package, ...	Package names

Details

Checks if a package is available and optionally meets a specific version requirement.

Value

- `require_namespace()` `TRUE` (invisibly) if found; otherwise errors
- `available_namespace()` A named `logical` vector of same length as input. Vector names are packages and values are `TRUE` if the package is available and meets the version requirement, otherwise `FALSE`.

Examples

```
isTRUE(require_namespace("base")) # returns invisibly
try(require_namespace("1package")) # (using a purposefully bad name)
require_namespace("base", "utils")
try(require_namespace("base >= 3.5", "utils > 4.0", "fuj == 0.0"))

# no error check
fuj0 <- if (!available_namespace("fuj == 0.0")) {
  "fuj 0.0 does not exist"
}
fuj0
```

struct *Simple structures*

Description

Create simple structures

Usage

```
struct(x, class, ..., .keep_attr = FALSE)
```

Arguments

x	An object; if NULL, coerced to base::list()
class	A vector of classes; can also be NULL
...	Named attributes to set to x; overwrites any attributes in x even if defined in .keep_attr
.keep_attr	Control for keeping attributes from x: TRUE will retain all attributes from x; a character vector will pick out specifically defined attributes to retain; otherwise only attributes defined in ... will be used

Details

Unlike base::structure() this does not provide additional checks for special names, performs no base::storage.mode() conversions for factors (x therefor has to be an integer), attributes from x are not retained, and class is specified outside of other attributes and assigned after base::attributes() is called.

Essentially, this is just a wrapper for calling base::attributes() then base::class().

Note that base::structure() provides a warning when the first argument is NULL. struct() does not. The coercion from NULL to base::list() is done, and documented, in base::attributes().

Value

An object with class defined as class and attributes ...

Examples

```
x <- list(a = 1, b = 2)
# structure() retains the $names attribute of x but struct() does not
structure(x, class = "data.frame", row.names = 1L)
struct(x, "data.frame", row.names = 1L)
struct(x, "data.frame", row.names = 1L, names = names(x))

# structure() corrects entries for "factor" class
# but struct() demands the data to be an integer
structure(1, class = "factor", levels = "a")
try(struct(1, "factor", levels = "a"))
```

```
struct(1L, "factor", levels = "a")

# When first argument is NULL -- attributes() coerces
try(structure(NULL))    # NULL, no call to attributes()
struct(NULL, NULL)      # list(), without warning
x <- NULL
attributes(x) <- NULL
x                       # NULL
attributes(x) <- list() # struct() always grabs ... into a list
x                       # list()

# Due to the use of class() to assign class, you may experience some
# other differences between structure() and struct()
x <- structure(1, class = "integer")
y <- struct(1, "integer")
str(x)
str(y)

all.equal(x, y)

# Be careful about carrying over attributes
x <- quick_df(list(a = 1:2, b = 3:4))
# returns empty data.frame
struct(x, "data.frame", new = 1)

# safely changing names without breaking rownames
struct(x, "data.frame", names = c("c", "d")) # breaks
struct(x, "data.frame", names = c("c", "d"), .keep_attr = TRUE)
struct(x, "data.frame", names = c("c", "d"), .keep_attr = "row.names")

# safely adds comments
struct(x, "data.frame", comment = "hi", .keep_attr = TRUE)
struct(x, "data.frame", comment = "hi", .keep_attr = c("names", "row.names"))

# assignment in ... overwrites attributes
struct(x, "data.frame", names = c("var1", "var2"), .keep_attr = TRUE)
```

---

vap *Vector applies*

Description

Apply a function over elements of a `vector`.

Usage

```
vap(x, f, ...)

vapi(x, f, ...)
```

```
vap2(x, y, f, ...)

vap3(x, y, z, f, ...)

vapp(p, f, ...)

vap_vec(x, f, ...)

vap_lgl(x, f, ...)

vap_int(x, f, ...)

vap_dbl(x, f, ...)

vap_chr(x, f, ...)

vap_raw(x, f, ...)

vap_cpl(x, f, ...)

vap_date(x, f, ...)

vap_dttm(x, f, ...)

vapi_lgl(x, f, ...)

vapi_int(x, f, ...)

vapi_dbl(x, f, ...)

vapi_chr(x, f, ...)

vapi_raw(x, f, ...)

vapi_cpl(x, f, ...)

vapi_date(x, f, ...)

vapi_dttm(x, f, ...)

vap2_lgl(x, y, f, ...)

vap2_int(x, y, f, ...)

vap2_dbl(x, y, f, ...)

vap2_chr(x, y, f, ...)
```

```
vap2_raw(x, y, f, ...)

vap2_cpl(x, y, f, ...)

vap2_date(x, y, f, ...)

vap2_dttm(x, y, f, ...)

vap3_lgl(x, y, z, f, ...)

vap3_int(x, y, z, f, ...)

vap3_dbl(x, y, z, f, ...)

vap3_chr(x, y, z, f, ...)

vap3_raw(x, y, z, f, ...)

vap3_cpl(x, y, z, f, ...)

vap3_date(x, y, z, f, ...)

vap3_dttm(x, y, z, f, ...)

vapp_lgl(p, f, ...)

vapp_int(p, f, ...)

vapp_dbl(p, f, ...)

vapp_chr(p, f, ...)

vapp_raw(p, f, ...)

vapp_cpl(p, f, ...)

vapp_date(p, f, ...)

vapp_dttm(p, f, ...)

with_vap_progress(expr)

with_vap_indexed_errors(expr)

with_vap_handlers(expr)
```

Arguments

x, y, z	Values to map over

f	Function or specification of function to apply.
...	Additional arguments passed to f
p	A list of values (i.e., parameters) to map over
expr	The expression to evaluate.

Details

Like base::lapply(), base::mapply(), and family, the vap functions provide a means of applying a function to each element of a vector, and controlling return types. The vap family provides extra tools and controls, as well as *date* outputs (i.e., _date, _dttm variants that work with Date and POSIXct types).

- vap() uses a single x argument
- vapi() uses a single x argument and passes the names (when available, otherwise index) as the second argument
- vap2(), vap3() use two and three arguments, respectively
- vapp() uses a pairlist of arguments

Value

For vap(), vapi(), returns a list with length of x. For vap2(), vap3(), and vap(), return length is determined by how ... is recycled inside base::mapply().

All have type variants (e.g., vap_chr(), vapi_int(), vap3_dbl()) which return a vector of the corresponding class, with the same length of x, or following the same recycling rules as mapply() for multiple inputs. These returns are *coerced*, rather than *checked*, and may result in unexpected outputs. Likely, warnings or errors will be signaled accordingly.

vap_vec() is a variant of vap() that returns a *flattened* vector. This has similar behavior as base::sapply(), in that a list will be returned if the base::unlist()'d output has multiple values in an element.

with_vap_progress() sets an option vap.progress to TRUE for the duration of expr, which causes a progress bar to be displayed for any vap* calls inside expr.

with_vap_handlers() sets an options(vap.indexed_errors = TRUE) for the duration of expr, which causes errors inside vap calls to include the index at which the error occurred.

Extras

Two helper functions are provided to set options for a progress bars (options(fuj.vap.progress)) and reporting an index during and error (options(fuj.vap.index_error)). Two wrapper functions are provided: with_vap_progress() and with_vap_handlers(), respectively; the latter may include other handlers in the future. These are not turned on by default (or rather, the option settings are set to FALSE within fuj) as they incur some additional overhead.

Examples

```
fruits <- c("apple", "banana", "pear")
vap(fruits, toupper)
vapi(fruits, function(x, i) paste0(i, ": ", toupper(x)))
vap_int(set_names(month.name, month.abb), nchar)
vap2_dbl(1:5, 6:10, `+`)
vapp_int(list(x = 1:5, y = 6:10, z = 11:15), sum)

# return type is coerced:
vap_int(1:5, paste0, "9")  # character -> integer
vap_date(month.name, nchar) # integer -> Date

# when f is a character or number, subsetting is performed
x <- list(list(a = 1, b = 3), list(a = 2, b = 4))
vap_int(x, "a")
vap_int(x, 2)

# wrap in [with_vap_progress()] to show a progress bar
invisible(
  with_vap_progress(
    vap(1:10 / 20, Sys.sleep)
  )
)

# wrap in [with_vap_handlers()] to report index on error
try(
  with_vap_handlers(
    vap(10:1, function(x) if (x == 3) stop("bad"))
  )
)
```

---

verbose *Verbose*

Description

Simple verbose condition handling

Usage

```
verbose(
  ...,
  .fill = getOption("fuj.verbose.fill"),
  .label = getOption("fuj.verbose.label"),
  .verbose = getOption("fuj.verbose", getOption("verbose"))
)

make_verbose(opt)
```

Arguments

...	A message to display. When ... is NULL (and only NULL), no message will display.
.fill	When TRUE, each new line will be prefixed with the verbose label (controlled through options("fuj.verbose.fill"))
.label	A label to prefix the message with (controlled through options("fuj.verbose.label"))
.verbose	When TRUE (or is a function when returns TRUE) prints out the message.
opt	An option to use in lieu of fun.verbose. Note: options("fuj.verbose") is temporarily set to isTRUE(getOption(opt)) when the function is evaluate, but is reset to its original value on exit.

Details

verbose() can be safely placed in scripts to signal additional message conditions. verbose() can be controlled with options("verbose") (the default) and an override, options("fuj.verbose"). The latter can be set to a function whose result will be used for conditional evaluation.

make_verbose() allows for the creation of a custom verbose function.

Value

None, called for its side-effects. When conditions are met, will signal a verbose_message condition.

Examples

```
op <- options(verbose = FALSE)
verbose("will not show")

options(verbose = TRUE)
verbose("message printed")
verbose("multiple lines ", "will be ", "combined")
options(op)

op <- options(fuj.verbose = function() TRUE)
verbose("function will evaluate")
verbose(NULL) # nothing
verbose(NULL, "something")
verbose(if (FALSE) {
  "`if` returns `NULL` when not `TRUE`, which makes for additional control"
})
options(op)

# make your own verbose
verb <- make_verbose("fuj.foo.bar")
verb("will not show")
options(fuj.foo.bar = TRUE)
verb("will show")
```

yes_no *Yes-no prompt*

Description

Prompts the user to make a yes/no selection

Usage

```
yes_no(..., na = NULL, n_yes = 1, n_no = 2, noninteractive_error = TRUE)
```

Arguments

...	text to display
na	Text for an NA response. When NULL, will not provide a possible NA response. When
n_yes, n_no	The number of yes/no selections
noninteractive_error	While TRUE, throws an error when the session is not interactive. If FALSE, will return NA instead.

Index

%:::% (colons), 4
%::% (colons), 4
%attr% (exattr), 8
%colons% (colons), 4
%len% (if_null), 12
%names% (names), 19
%out% (match_ext), 17
%wi% (match_ext), 17
%wo% (match_ext), 17
%out%, *18*
%wi%, *18*
%wo%, *18*

add (alias_arithmetic), 2
alias_arithmetic, 2
alias_extract, 3
any_match (match_ext), 17
any_match(), *18*
attributes, *25*
available_namespace
        (require_namespace), 24
available_namespace(), *24*

base::%in%, *18*
base::Arithmetic, *2*
base::as.data.frame(), *23*
base::AsIs, *14*, *21*
base::attach(), *13*
base::attr(), *8*
base::attributes(), *25*
base::class(), *25*
base::conditions, *21*
base::data.frame(), *23*
base::Extract, *3*
base::factor(), *8*
base::file.path(), *10*
base::intersect(), *18*
base::is.null(), *12*
base::lapply(), *29*
base::library(), *13*
base::list(), *15*, *25*
base::loadNamespace(), *13*
base::mapply(), *29*
base::match(), *18*
base::match.arg(), *15*
base::name, *13*, *14*
base::Negate(), *20*
base::paste(), *4*
base::pmatch(), *16*
base::require(), *13*
base::sapply(), *29*
base::search(), *13*, *14*
base::setdiff(), *18*
base::source(), *14*
base::storage.mode(), *25*
base::structure(), *25*
base::suppressMessages(), *18*
base::suppressWarnings(), *18*
base::unlist(), *29*

class_error (conditions), 5
class_warning (conditions), 5
collapse, 3
colons, 4
conditions, 5

Date, *29*
defunct_error (conditions), 5
delay, 6
deprecated_warning (conditions), 5
development_error (conditions), 5
development_warning (conditions), 5
divide (alias_arithmetic), 2
divide_int (alias_arithmetic), 2

empty_df (quick_df), 23
empty_df(), *23*
encode, 7
encode(), *8*
err (conditions), 5

error_condition (conditions), 5
error_condition(), 6
exattr, 8

fact (encode), 7
fact(), 8
factors, 25
file_ext<- (fp), 10
file_path (fp), 10
file_path(), 10, 11
flip, 9
fp, 10
fp(), 10, 11
fuj, 29

hold (hot), 11
hold(), 11, 12
hot, 11

if_null, 12
include, 13
include(), 13, 14
input_error (conditions), 5
input_warning (conditions), 5
interactive_error (conditions), 5
interactive_warning (conditions), 5
internal_error (conditions), 5
internal_warning (conditions), 5
is_file_path (fp), 10
is_file_path(), 10, 11
is_in (match_ext), 17
is_in(), 18
is_linux (os), 22
is_macos (os), 22
is_named (names), 19
is_namespace_available
        (require_namespace), 24
is_namespace_available(), 24
is_out (match_ext), 17
is_path (fp), 10
is_path(), 10, 11, 14
is_windows (os), 22
is_within (match_ext), 17
is_without (match_ext), 17

list0, 15
Long vectors, 17
lst (list0), 15
lst(), 23

make_verbose (verbose), 30
make_verbose(), 31
mapply(), 29
match_arg, 15
match_ext, 17
message_condition (conditions), 5
message_condition(), 6
msg (conditions), 5
muffle, 18
multiply (alias_arithmetic), 2

NA_character_, 3
names, 19
namespace_error (conditions), 5
namespace_warning (conditions), 5
negate, 20
negate(), 20
new_condition, 21
new_condition(), 6
no_match (match_ext), 17
no_match(), 18
norm_path (fp), 10
norm_path(), 11
normalize, 10
normalized, 11
not (negate), 20
not(), 20
np (fp), 10
np(), 10, 11

os, 22

POSIXct, 29

quick_df, 23
quick_dfl (quick_df), 23

raise_power (alias_arithmetic), 2
remainder (alias_arithmetic), 2
remove_names (names), 19
require_namespace, 24
require_namespace(), 24

set_file_ext (fp), 10
set_file_ext(), 10, 11
set_names (names), 19
struct, 25
struct(), 25
subassign1 (alias_extract), 3
subassign2 (alias_extract), 3

subassign3 (alias_extract), 3
subset1 (alias_extract), 3
subset2 (alias_extract), 3
subset3 (alias_extract), 3
subtract (alias_arithmetic), 2

toss (hot), 11
toss(), *11*, *12*
type_error (conditions), 5

value_error (conditions), 5
value_warning (conditions), 5
vap, 26
vap(), *29*
vap2 (vap), 26
vap2(), *29*
vap2_chr (vap), 26
vap2_cpl (vap), 26
vap2_date (vap), 26
vap2_dbl (vap), 26
vap2_dttm (vap), 26
vap2_int (vap), 26
vap2_lgl (vap), 26
vap2_raw (vap), 26
vap3 (vap), 26
vap3(), *29*
vap3_chr (vap), 26
vap3_cpl (vap), 26
vap3_date (vap), 26
vap3_dbl (vap), 26
vap3_dbl(), *29*
vap3_dttm (vap), 26
vap3_int (vap), 26
vap3_lgl (vap), 26
vap3_raw (vap), 26
vap_chr (vap), 26
vap_chr(), *29*
vap_cpl (vap), 26
vap_date (vap), 26
vap_dbl (vap), 26
vap_dttm (vap), 26
vap_int (vap), 26
vap_lgl (vap), 26
vap_raw (vap), 26
vap_vec (vap), 26
vap_vec(), *29*
vapi (vap), 26
vapi(), *29*
vapi_chr (vap), 26
vapi_cpl (vap), 26
vapi_date (vap), 26
vapi_dbl (vap), 26
vapi_dttm (vap), 26
vapi_int (vap), 26
vapi_int(), *29*
vapi_lgl (vap), 26
vapi_raw (vap), 26
vapp (vap), 26
vapp_chr (vap), 26
vapp_cpl (vap), 26
vapp_date (vap), 26
vapp_dbl (vap), 26
vapp_dttm (vap), 26
vapp_int (vap), 26
vapp_lgl (vap), 26
vapp_raw (vap), 26
verbose, 30
verbose(), *31*
verbose_message (conditions), 5

warning_condition (conditions), 5
warning_condition(), *6*
with_vap_handlers (vap), 26
with_vap_handlers(), *29*
with_vap_indexed_errors (vap), 26
with_vap_progress (vap), 26
with_vap_progress(), *29*
wrn (conditions), 5
wuffle (muffle), 18

yes_no, 32