Package ‘wrapr’

Type          Package
Title         Wrap R Tools for Debugging and Parametric Programming
Version       2.0.6
Date          2020-12-06

URL           https://github.com/WinVector/wrapr,
               https://winvector.github.io/wrapr/

Maintainer    John Mount <jmount@win-vector.com>

BugReports    https://github.com/WinVector/wrapr/issues

Description   Tools for writing and debugging R code. Provides:
               ‘%>%’ dot-pipe (an ‘S3’ configurable pipe), unpack/to (R style multiple assignment/return),
               ‘build_frame()/draw_frame()’ (‘data.frame’ example tools),
               ‘qc()’ (quoting concatenate),
               ‘:=’ (named map builder), ‘let()’ (converts non-standard evaluation interfaces to parametric standard
               evaluation interfaces, inspired by ‘gtools::strmacro()’ and ‘base::bquote()’), and more.

License       GPL-2 | GPL-3
Encoding      UTF-8
LazyData      true

RoxygenNote   7.1.1
Depends       R (>= 3.3.0)
Imports       utils, methods, stats
Suggests      parallel, knitr, graphics, rmarkdown, R.rsp, tinytest
VignetteBuilder knitr, R.rsp

ByteCompile   true

NeedsCompilation no

Author         John Mount [aut, cre],
               Nina Zumel [aut],
               Win-Vector LLC [cph]

Repository     CRAN

Date/Publication 2020-12-06 17:30:02 UTC
R topics documented:

add_name_column .................................................. 3
apply_left ........................................................... 4
apply_left.default .................................................. 6
apply_left_default .................................................. 7
apply_right ........................................................... 8
apply_right.default .................................................. 9
apply_right_S4 ........................................................ 10
as_named_list ........................................................ 11
bquote_call_args ..................................................... 13
bquote_function ....................................................... 14
buildNameCallback ................................................... 15
build_frame ............................................................ 16
checkColsFormUniqueKeys ........................................... 17
check_equiv_frames .................................................... 17
clean_fit_glm .......................................................... 18
clean_fit_lm ............................................................ 19
coalesce ................................................................. 20
DebugFn ................................................................. 22
DebugFnE ................................................................. 23
DebugFnW ................................................................. 24
DebugFnWE .............................................................. 25
DebugPrintFn ............................................................ 26
DebugPrintFnE ........................................................... 27
defineLambda ......................................................... 28
dot_arrow ................................................................. 29
draw_frame .............................................................. 30
draw_framec ............................................................. 31
evalb .......................................................... 32
execute_parallel ....................................................... 33
f ................................................................. 34
grepdf ................................................................. 35
grepv ................................................................. 36
has_no_dup_rows ....................................................... 37
invert_perm ............................................................ 37
lambda ................................................................. 38
lapplym ................................................................. 39
let ................................................................. 40
makeFunction_se ....................................................... 42
mapsyms ............................................................... 43
map_to_char .......................................................... 43
map_upper ............................................................. 44
match_order ........................................................... 45
mk_formula ............................................................. 46
mk_tmp_name_source .................................................. 47
named_map_builder ................................................... 48
orderv ................................................................. 49
add_name_column

Add list name as a column to a list of data.frames.

Description

Add list name as a column to a list of data.frames.

Usage

add_name_column(dlist, destinationColumn)
Arguments

   dlist  named list of data.frames
destinationColumn  character, name of new column to add

Value

   list of data frames, each of which as the new destinationColumn.

Examples

   dlist <- list(a = data.frame(x = 1), b = data.frame(x = 2))
   add_name_column(dlist, 'name')

apply_left  \( S3 \) dispatch on class of pipe_left_arg.

Description


Usage

   apply_left(
       pipe_left_arg,
       pipe_right_arg,
       pipe_environment,
       left_arg_name,
       pipe_string,
       right_arg_name
   )

Arguments

   pipe_left_arg  left argument.
   pipe_right_arg substitute(pipe_right_arg) argument.
   pipe_environment  environment to evaluate in.
   left_arg_name  name, if not NULL name of left argument.
   pipe_string  character, name of pipe operator.
   right_arg_name  name, if not NULL name of right argument.
apply_left

Value

result

See Also

apply_left.default

Examples

apply_left.character <- function(pipe_left_arg,
    pipe_right_arg,
    pipe_environment,
    left_arg_name,
    pipe_string,
    right_arg_name) {
    if(is.language(pipe_right_arg)) {
        wrapr::apply_left_default(pipe_left_arg,
            pipe_right_arg,
            pipe_environment,
            left_arg_name,
            pipe_string,
            right_arg_name)
    } else {
        paste(pipe_left_arg, pipe_right_arg)
    }
}
apply_left.default  

S3 dispatch on class of pipe_left_arg.

Description

Place evaluation of left argument in . and then evaluate right argument.

Usage

```r
## Default S3 method:
apply_left(
  pipe_left_arg,
  pipe_right_arg,
  pipe_environment,
  left_arg_name,
  pipe_string,
  right_arg_name
)
```

Arguments

- `pipe_left_arg`  left argument
- `pipe_right_arg` substitute(pipe_right_arg) argument
- `pipe_environment`  environment to evaluate in
- `left_arg_name`  name, if not NULL name of left argument.
- `pipe_string`  character, name of pipe operator.
- `right_arg_name`  name, if not NULL name of right argument.

Value

result

See Also

`apply_left`

Examples

```r
5 %.>% sin(.)
```
apply_left_default

S3 dispatch on class of pipe_left_arg.

Description

Place evaluation of left argument in . and then evaluate right argument.

Usage

apply_left_default(
  pipe_left_arg,
  pipe_right_arg,
  pipe_environment,
  left_arg_name,
  pipe_string,
  right_arg_name
)

Arguments

pipe_left_arg  left argument
pipe_right_arg  substitute(pipe_right_arg) argument
pipe_environment  environment to evaluate in
left_arg_name  name, if not NULL name of left argument.
pipe_string  character, name of pipe operator.
right_arg_name  name, if not NULL name of right argument.

Value

result

See Also

apply_left

Examples

5 %.>% sin(.)
apply_right  

S3 dispatch on class of pipe_right_argument.

Description

Triggered if right hand side of pipe stage was a name that does not resolve to a function. For formal documentation please see https://github.com/WinVector/wrapr/blob/master/extras/wrapr_pipe.pdf.

Usage

apply_right(
  pipe_left_arg,
  pipe_right_arg,
  pipe_environment,
  left_arg_name,
  pipe_string,
  right_arg_name
)

Arguments

pipe_left_arg  left argument
pipe_right_arg  right argument
pipe_environment  environment to evaluate in
left_arg_name  name, if not NULL name of left argument.
pipe_string  character, name of pipe operator.
right_arg_name  name, if not NULL name of right argument.

Value

result

See Also

apply_left, apply_right_S4

Examples

# simulate a function pointer
apply_right.list <- function(pipe_left_arg,
  pipe_right_arg,
  pipe_environment,
  left_arg_name,
  pipe_string,
apply_right.default

```r
right_arg_name) {
    pipe_right_arg$f(pipe_left_arg)
}
```

```r
f <- list(f=sin)
2 %.>% f
f$f <- cos
2 %.>% f
```

---

Default apply_right implementation.

Description

Default apply_right implementation: S4 dispatch to apply_right_S4.

Usage

```r
## Default S3 method: apply_right
apply_right(
    pipe_left_arg,
    pipe_right_arg,
    pipe_environment,
    left_arg_name,
    pipe_string,
    right_arg_name
)
```

Arguments

- `pipe_left_arg`: left argument
- `pipe_right_arg`: pipe_right_arg argument
- `pipe_environment`: environment to evaluate in
- `left_arg_name`: name, if not NULL name of left argument.
- `pipe_string`: character, name of pipe operator.
- `right_arg_name`: name, if not NULL name of right argument.

Value

result

See Also

`apply_left, apply_right, apply_right_S4`
Examples

# simulate a function pointer
apply_right.list <- function(pipe_left_arg,
   pipe_right_arg,
   pipe_environment,
   left_arg_name,
   pipe_string,
   right_arg_name) {
   pipe_right_arg$f(pipe_left_arg)
}

f <- list(f=sin)
2 %>% f
f$f <- cos
2 %>% f

apply_right_S4

S4 dispatch method for apply_right.

Description

Intended to be generic on first two arguments.

Usage

apply_right_S4(
   pipe_left_arg,
   pipe_right_arg,
   pipe_environment,
   left_arg_name,
   pipe_string,
   right_arg_name
)

Arguments

pipe_left_arg   left argument
pipe_right_arg  pipe_right_arg argument
pipe_environment environment to evaluate in
left_arg_name   name, if not NULL name of left argument.
pipe_string     character, name of pipe operator.
right_arg_name  name, if not NULL name of right argument.
as_named_list

Value

result

See Also

apply_left, apply_right

Examples

a <- data.frame(x = 1)
b <- data.frame(x = 2)

# a %.>% b # will (intentionally) throw

setMethod(
  "apply_right_S4",
  signature("data.frame", "data.frame"),
  function(pipe_left_arg,
    pipe_right_arg,
    pipe_environment,
    left_arg_name,
    pipe_string,
    right_arg_name) {
      rbind(pipe_left_arg, pipe_right_arg)
    })

a %.>% b # should equal data.frame(x = c(1, 2))

as_named_list

Capture named objects as a named list.

Description

Build a named list from a sequence of named arguments of the form NAME, or NAME = VALUE. This is intended to shorten forms such as list(a = a, b = b) to as_named_list(a, b).

Usage

as_named_list(...)

Arguments

... argument names (must be names, not strings or values) plus possible assigned values.
Value

a named list mapping argument names to argument values

Examples

```r
a <- data.frame(x = 1)
b <- 2

str(as_named_list(a, b))

as_named_list(a, x = b, c = 1 + 1)

# an example application for this function is managing saving and
# loading values into the workspace.
if(FALSE) {
  # remotes::install_github("WinVector/wrapr")
  library(wrapr)

  a <- 5
  b <- 7
  do_not_want <- 13

  # save the elements of our workspace we want
  saveRDS(as_named_list(a, b), 'example_data.RDS')

  # clear values out of our workspace for the example
  rm(list = ls())
  ls()
  # notice workspace environment now empty

  # read back while documenting what we expect to
  # read in
  unpack[a, b] <- readRDS('example_data.RDS')

  # confirm what we have, the extra unpack is a side
  # effect of the []<- notation. To avoid this instead
  # use one of:
  # unpack(readRDS('example_data.RDS'), a, b)
  # readRDS('example_data.RDS') %>% unpack(. , a, b)
  # readRDS('example_data.RDS') %>% unpack[a, b]
  ls()
  # notice do_not_want is not present

  print(a)

  print(b)
}
```
bquote_call_args

Treat ... argument as bquoted-values.

Description

bquote_call_args is a helper to allow the user to write functions with bquote-enabled argument substitution. Uses convention that := is considered a alias for =. Re-writes call args to evaluate expr with bquote .() substitution. Including .(-x) promoting x's value from character to a name, which is called "quote negation" (hence the minus-sign).

Usage

bquote_call_args(call, env = parent.frame())

Arguments

call result of match.call()
env environment to perform lookups in.

Value

name list of values

See Also

bquote_function

Examples

f <- function(q, ...) {
  env = parent.frame()
  # match.call() best called in function context.
  captured_call <- match.call()
  captured_args <- bquote_call_args(captured_call, env)
  captured_args
}

z <- "x"
y <- 5
qv <- 3

# equivalent to f(3, x = 5)
f(.qv), .(z) := .(y))

# equivalent to f(q = 7)
qname <- 'q'
f(.(qname) := 7)
bquote_function

Adapt a function to use bquote on its arguments.

Description

bquote_function is for adapting a function defined elsewhere for bquote-enabled argument substitution. Re-write call to evaluate expr with bquote .() substitution. Uses convention that := is considered a alias for =. Including .(-x) promoting x’s value from character to a name, which is called "quote negation" (hence the minus-sign).

Usage

bquote_function(fn)

Arguments

fn function to adapt, must have non-empty formals().

Value

new function.

See Also

bquote_call_args

Examples

if(requireNamespace('graphics', quietly = TRUE)) {
  angle = 1:10
  variable <- as.name("angle")
  plotb <- bquote_function(graphics::plot)
  plotb(x = .(variable), y = sin(.variable))
}

f1 <- function(x) { substitute(x) }
f2 <- bquote_function(f1)
arg <- "USER_ARG"
f2(arg) # returns arg
f2(.arg) # returns "USER_ARG" (character)
f2(.(-arg)) # returns USER_ARG (name)
Build a custom writeback function that writes state into a user named variable.

### Description

Build a custom writeback function that writes state into a user named variable.

### Usage

```r
buildNameCallback(varName)
```

### Arguments

- **varName**: character where to write captured state

### Value

writeback function for use with functions such as `DebugFnW`

### Examples

```r
# user function
f <- function(i) { (1:10)[[i]] }
# capture last error in variable called "lastError"
writeBack <- buildNameCallback('lastError')
# wrap function with writeBack
df <- DebugFnW(writeBack,f)
# capture error (Note: tryCatch not needed for user code!)
tryCatch(
  df(12),
  error = function(e) { print(e) })
# examine error
str(lastError)
# redo call, perhaps debugging
tryCatch(
  do.call(lastError$fn_name, lastError$args),
  error = function(e) { print(e) })
```
**build_frame**  
*Build a data.frame from the user’s description.*

**Description**

A convenient way to build a data.frame in legible transposed form. Position of first "|" (or other infix operator) determines number of columns (all other infix operators are aliases for ",,"). Names are de-referenced.

**Usage**

```r
declare_frame(..., cf_eval_environment = parent.frame())
```

**Arguments**

- `...`: cell names, first infix operator denotes end of header row of column names.
- `cf_eval_environment`: environment to evaluate names in.

**Value**

character data.frame

**See Also**

draw_frame, qchar_frame

**Examples**

```r
tc_name <- "training"
x <- build_frame(
  "measure", tc_name, "validation" |
  "minus binary cross entropy", 5, -7 |
  "accuracy", 0.8, 0.6)
print(x)
str(x)
cat(draw_frame(x))
```

```r
build_frame(
  "x" |
  -1 |
  2 )
```
checkColsFormUniqueKeys

Check that a set of columns form unique keys.

Description

For local data.frame only.

Usage

checkColsFormUniqueKeys(data, keyColNames)

Arguments

data data.frame to work with.
keyColNames character array of column names to check.

Value

logical TRUE if the rows of data are unique addressable by the columns named in keyColNames.

Examples

d <- data.frame(key = c('a', 'a', 'b'), k2 = c(1, 2, 2))
checkColsFormUniqueKeys(d, 'key') # should be FALSE
checkColsFormUniqueKeys(d, c('key', 'k2')) # should be TRUE

check_equiv_frames

Check two data.frames are equivalent after sorting columns and rows.

Description

Confirm two dataframes are equivalent after reordering columns and rows.

Usage

check_equiv_frames(d1, d2, ..., tolerance = sqrt(.Machine$double.eps))

Arguments

d1 data.frame 1
d2 data.frame 2
... force later arguments to bind by name
tolerance numeric comparision tolerance
clean_fit_glm

Fit a stats::glm without carrying back large structures.

Description

Please see https://win-vector.com/2014/05/30/trimming-the-fat-from-glm-models-in-r/ for discussion.

Usage

clean_fit_glm(
  outcome,
  variables,
  data,
  ...,  
  family,
  intercept = TRUE,
  outcome_target = NULL,
  outcome_comparator = "==",
  weights = NULL,
  env = baseenv()
)

Arguments

  outcome character, name of outcome column.
  variables character, names of variable columns.
  data data.frame, training data.
  ... not used, force later arguments to be used by name
  family passed to stats::glm()
  intercept logical, if TRUE allow an intercept term.
  outcome_target scalar, if not NULL write outcome==outcome_target in formula.
  outcome_comparator one of "==", "!=" ,">", "<=", ">", only use of outcome_target is not NULL.
  weights passed to stats::glm()
  env environment to work in.

Value

  list(model=model, summary=summary)
Examples

```r
mk_data_example <- function(k) {
  data.frame(
    x1 = rep(c("a", "a", "b", "b"), k),
    x2 = rep(c(0, 0, 0, 1), k),
    y = rep(1:4, k),
    yC = rep(c(FALSE, TRUE, TRUE, TRUE), k),
    stringsAsFactors = FALSE)
}

res_glm <- clean_fit_glm("yC", c("x1", "x2"),
                         mk_data_example(1),
                         family = binomial)
length(serialize(res_glm$model, NULL))

res_glm <- clean_fit_glm("yC", c("x1", "x2"),
                         mk_data_example(10000),
                         family = binomial)
length(serialize(res_glm$model, NULL))

predict(res_glm$model,
        newdata = mk_data_example(1),
        type = "response")
```

---

clean_fit_lm  

*Fit a stats::lm without carrying back large structures.*

Description


Usage

```r
clean_fit_lm(  
  outcome,  
  variables,  
  data,  
  ...,  
  intercept = TRUE,  
  weights = NULL,  
  env = baseenv()
)
```
Coalesce

Arguments

- `outcome`: character, name of outcome column.
- `variables`: character, names of variable columns.
- `data`: data.frame, training data.
- `...`: not used, force later arguments to be used by name.
- `intercept`: logical, if TRUE allow an intercept term.
- `weights`: passed to stats::glm()
- `env`: environment to work in.

Value

list(model=model, summary=summary)

Examples

```r
mk_data_example <- function(k) {
  data.frame(
    x1 = rep(c("a", "a", "b", "b"), k),
    x2 = rep(c(0, 0, 0, 1), k),
    y = rep(1:4, k),
    yC = rep(c(FALSE, TRUE, TRUE, TRUE), k),
    stringsAsFactors = FALSE)
}
res_lm <- clean_fit_lm("y", c("x1", "x2"),
  mk_data_example(1))
length(serialize(res_lm$model, NULL))
res_lm <- clean_fit_lm("y", c("x1", "x2"),
  mk_data_example(10000))
length(serialize(res_lm$model, NULL))
predict(res_lm$model,
  newdata = mk_data_example(1))
```

Description

This is a simple "try to take values on the left, but fall back to the right if they are not available" operator. It is inspired by SQL coalesce and the notation is designed to evoke the idea of testing and the C# ?? null coalescing operator. NA and NULL are treated roughly equally: both are replaced regardless of available replacement value (with some exceptions). The exceptions are: if the left hand side is a non-zero length vector we preserve the vector type of the left-hand side and do not assign any values that vectors can not hold (NULLs and complex structures) and do not replace with a right argument list.
Usage
  coalesce(coalesce_left_arg, coalesce_right_arg)

  coalesce_left_arg %?% coalesce_right_arg

Arguments
  coalesce_left_arg
    vector or list.
  coalesce_right_arg
    vector or list.

Details
This operator represents a compromise between the desire to replace length zero structures and
NULL/NA values and the desire to preserve the first argument's structure (vector versus list). The
order of operations has been chosen to be safe, convenient, and useful. Length zero lists are not
treated as NULL (which is consistent with R in general). Note for non-vector operations on condi-
tions we recommend looking into isTRUE, which solves some problems even faster than coalesce
style operators.

When length(coalesce_left_arg)<=0 then return coalesce_right_arg if length(coalesce_right_arg)>0,
otherwise return coalesce_left_arg. When length(coalesce_left_arg)>0: assume coalesce_left_arg
is a list or vector and coalesce_right_arg is a list or vector that is either the same length as co-
alesce_left_arg or length 1. In this case replace NA/NULL elements of coalesce_left_arg with
the corresponding elements of coalesce_right_arg (re-cycling coalesce_right_arg when it is length 1).

Value
  coalesce_left_arg with NA elements replaced.

Functions
  • %?%: coalesce operator

Examples
  c(NA, NA, NA) %?% 5       # returns c(5, 5, 5)
  c(1, NA, NA) %?% list(5)  # returns c(1, 5, 5)
  c(1, NA, NA) %?% list(list(5)) # returns c(1, NA, NA)
  c(1, NA, NA) %?% c(NA, 20, NA) # returns c(1, 20, NA)
  NULL %?% list()       # returns NULL
  NULL %?% c(1, NA)     # returns c(1, NA)
  list(1, NULL, NULL) %?% c(3, 4, NA)   # returns list(1, 4, NA_real_)
  list(1, NULL, NULL, NA, NA) %?% list(2, NULL, NA, NULL, NA) # returns list(1, NULL, NA, NULL, NA)
  c(1, NA, NA) %?% list(1, 2, list(3)) # returns c(1, 2, NA)
  c(1, NA) %?% list(1, NULL)       # returns c(1, NA)
  c() %?% list(1, NA, NULL)      # returns list(1, NA, NULL)
  c() %?% c(1, NA, 2)            # returns c(1, NA, 2)
**DebugFn**  
*Capture arguments of exception throwing function call for later debugging.*

**Description**
Run fn, save arguments on failure. Please see: vignette("DebugFnW",package="wrapr").

**Usage**
```r
DebugFn(saveDest, fn, ...)  
```

**Arguments**
- `saveDest` where to write captured state (determined by type): NULL random temp file, character temp file, name globalenv() variable, and function triggers callback.
- `fn` function to call
- `...` arguments for fn

**Value**
`fn(...)` normally, but if `fn(...)` throws an exception save to `saveDest` RDS of list `r` such that `do.call(r$fn,r$args)` repeats the call to `fn` with `args`.

**See Also**
`dump.frames`, `DebugFn`, `DebugFnW`, `DebugFnWE`, `DebugPrintFn`, `DebugFnE`, `DebugPrintFnE`

**Examples**
```r
saveDest <- paste0(tempfile("debug"),".RDS")
f <- function(i) { (1:10)[[i]] }
# correct run
DebugFn(saveDest, f, 5)
# now re-run
# capture error on incorrect run
tryCatch(
  DebugFn(saveDest, f, 12),
  error = function(e) { print(e) })
# examine details
situation <- readRDS(saveDest)
str(situation)
# fix and re-run
situation$args[[1]] <- 6
do.call(situation$fn_name,situation$args)
# clean up
file.remove(saveDest)
```
DebugFnE

Capture arguments and environment of exception throwing function call for later debugging.

Description

Run fn, save arguments, and environment on failure. Please see: vignette("DebugFnW", package="wrapr").

Usage

DebugFnE(saveDest, fn, ...)

Arguments

- `saveDest` where to write captured state (determined by type): NULL random temp file, character temp file, name globalenv() variable, and function triggers callback.
- `fn` function to call
- `...` arguments for fn

Value

`fn(...)` normally, but if `fn(...)` throws an exception save to `saveDest` RDS of list r such that `do.call(r$fn,r$args)` repeats the call to `fn` with args.

See Also

dump.frames, DebugFn, DebugFnW, DebugFnWE, DebugPrintFn, DebugFnE, DebugPrintFnE

Examples

```r
saveDest <- paste0(tempfile('debug'), '.RDS')
f <- function(i) { (1:10)[[i]] }
# correct run
DebugFnE(saveDest, f, 5)
# now re-run
# capture error on incorrect run
tryCatch(
  DebugFnE(saveDest, f, 12),
  error = function(e) { print(e) })
# examine details
situation <- readRDS(saveDest)
str(situation)
# fix and re-run
situation$args[[1]] <- 6
do.call(situation$fn, situation$args, envir=situation$env)
# clean up
file.remove(saveDest)
```
DebugFnW

Wrap a function for debugging.

Description

Wrap fn, so it will save arguments on failure.

Usage

DebugFnW(saveDest, fn)

Arguments

saveDest where to write captured state (determined by type): NULL random temp file, character temp file, name globalenv() variable, and function triggers callback.

fn function to call

Value

wrapped function that saves state on error.

See Also

dump.frames, DebugFn, DebugFnW, DebugFnWE, DebugPrintFn, DebugFnE, DebugPrintFnE Operator idea from: https://gist.github.com/nassimhaddad/c9c327d10a91dcf9a3370d30dff8ac3d. Please see: vignette("DebugFnW",package="wrapr").

Examples

saveDest <- paste0(tempfile('debug'),'.RDS')
f <- function(i) { (1:10)[[i]] } 
df <- DebugFnW(saveDest,f) # correct run
df(5) # now re-run # capture error on incorrect run tryCatch(
  df(12),
  error = function(e) { print(e) }) # examine details
situation <- readRDS(saveDest)
str(situation) # fix and re-run
situation$args[[1]] <- 6
do.call(situation$fn,situation$args) # clean up
file.remove(saveDest)
DebugFnWE

Wrap function to capture arguments and environment of exception throwing function call for later debugging.

Description

Wrap fn, so it will save arguments and environment on failure. Please see: vignette("DebugFnW", package="wrapr").

Usage

DebugFnWE(saveDest, fn, ...)

Arguments

saveDest where to write captured state (determined by type): NULL random temp file, character temp file, name globalenv() variable, and function triggers callback.

fn function to call

... arguments for fn

Value

wrapped function that captures state on error.

See Also

dump.frames, DebugFn, DebugFnW, DebugFnWE, DebugPrintFn, DebugFnE, DebugPrintFnE

Idea from: https://gist.github.com/nassimhaddad/c9c327d10a91dcf9a3370d30dff8ac3d
Examples

```r
saveDest <- paste0(tempfile('debug'), '.RDS')
f <- function(i) { (1:10)[[i]] }
df <- DebugFnWE(saveDest, f)
# correct run
df(5)
# now re-run
# capture error on incorrect run
tryCatch(
  df(12),
  error = function(e) { print(e) })
# examine details
situation <- readRDS(saveDest)
str(situation)
# fix and re-run
situation$args[[1]] <- 6
do.call(situation$fn, situation$args, envir=situation$env)
# clean up
file.remove(saveDest)
```

---

**DebugPrintFn**

Capture arguments of exception throwing function call for later debugging.

Description

Run fn and print result, save arguments on failure. Use on systems like `ggplot()` where some calculation is delayed until `print()`. Please see: vignette("DebugFnW", package="wrapr").

Usage

`DebugPrintFn(saveDest, fn, ...)`

Arguments

- `saveDest` where to write captured state (determined by type): NULL random temp file, character temp file, name `globalenv()` variable, and function triggers callback.
- `fn` function to call
- `...` arguments for `fn`

Value

`fn(...)` normally, but if `fn(...)` throws an exception save to `saveDest` RDS of list `r` such that `do.call(r$fn, r$args)` repeats the call to `fn` with `args`. 
**DebugPrintFnE**

Capture arguments and environment of exception throwing function call for later debugging.

**Description**

Run fn and print result, save arguments and environment on failure. Use on systems like ggplot() where some calculation is delayed until print(). Please see: vignette("DebugFnW",package="wrapr").

**Usage**

```r
DebugPrintFnE(saveDest, fn, ...)
```

**Arguments**

- **saveDest**
  - where to write captured state (determined by type): NULL random temp file, character temp file, name globalenv() variable, and function triggers callback.

- **fn**
  - function to call

- **...**
  - arguments for fn

**Value**

fn(...) normally, but if fn(...) throws an exception save to saveDest RDS of list r such that do.call(r$fn,r$args) repeats the call to fn with args.

**Examples**

```r
saveDest <- paste0(tempfile('debug'), '.RDS')
f <- function(i) { (1:10)[[i]] }
# correct run
DebugPrintFnE(saveDest, f, 5)
# now re-run
# capture error on incorrect run
tryCatch(
  DebugPrintFnE(saveDest, f, 12),
  error = function(e) { print(e) })
# examine details
situation <- readRDS(saveDest)
str(situation)
# fix and re-run
situation$args[[1]] <- 6
do.call(situation$fn, situation$args)
# clean up
file.remove(saveDest)
```
defineLambda

Define lambda function building function.

Description

Use this to place a copy of the lambda-symbol function builder in your workspace.

Usage

```
defineLambda(envir = parent.frame(), name = NULL)
```

Arguments

- `envir` environment to work in.
- `name` character, name to assign to (defaults to Greek lambda).

See Also

```
lambda, makeFunction_se, named_map_builder
```
**dot_arro**

**Examples**

```r
defineLambda()
  # ls()
```

---

**Pipe operator ("dot arrow", "dot pipe" or "dot arrow pipe").**

---

**Description**

Defined as roughly: 

```
a %>.% b ~ { . <-a; b }; (with visible .-side effects).
```

**Usage**

```
pipe_left_arg %>% pipe_right_arg
```

**Arguments**

- `pipe_left_arg` left argument expression (substituted into .)
- `pipe_right_arg` right argument expression (presumably including .)

**Details**

The pipe operator has a couple of special cases. First: if the right hand side is a name, then we try to de-reference it and apply it as a function or surrogate function.

The pipe operator checks for and throws an exception for a number of "piped into nothing cases" such as 

```
5 %>.% sin()
```

many of these checks can be turned off by adding braces.


The dot arrow pipe has S3/S4 dispatch (please see [https://journal.r-project.org/archive/2018/RJ-2018-042/index.html](https://journal.r-project.org/archive/2018/RJ-2018-042/index.html)). However as the right-hand side of the pipe is normally held unevaluated, we don’t know the type except in special cases (such as the right-hand side being referred to by a name or variable). To force the evaluation of a pipe term, simply wrap it in `.()`.

**Value**

```
eval({ . <- pipe_left_arg; pipe_right_arg });
```
Functions

• %.>%: dot arrow
• %>%.: alias for dot arrow
• %%.: alias for dot arrow

Examples

# both should be equal:
cos(exp(sin(4)))
4 %.>% sin(.) %.>% exp(.) %.>% cos(.)

f <- function() { sin }
# returns f() ignoring dot, not what we want
5 %.>% f()
# evaluates f() early then evaluates result with .-substitution rules
5 %.>% .(f())

draw_frame

Render a simple data.frame in build_frame format.

Description

Render a simple data.frame in build_frame format.

Usage

draw_frame(
  x,
  ..., 
  time_format = "%Y-%m-%d %H:%M:%S",
  formatC_options = list(),
  adjust_for_auto_indent = 2
)

Arguments

x data.frame (with atomic types).
... not used for values, forces later arguments to bind by name.
time_format character, format for "POSIXt" classes.
formatC_options named list, options for formatC()- used on numerics.
adjust_for_auto_indent integer additional after first row padding
**draw_framec**

**Value**

character

**See Also**

build_frame, qchar_frame

**Examples**

```r
tc_name <- "training"
x <- build_frame(  
  "measure" , tc_name, "validation", "idx" |  
  "minus binary cross entropy", 5 , 7 , 1L |  
  "accuracy"   , 0.8 , 0.6 , 2L )
print(x)
cat(draw_frame(x))
```

---

draw_framec  

*Render a simple data.frame in qchar_frame format.*

**Description**

Render a simple data.frame in qchar_frame format.

**Usage**

draw_framec(x, ..., unquote_cols = character(0), adjust_for_auto_indent = 2)

**Arguments**

- `x` data.frame (with character types).
- `...` not used for values, forces later arguments to bind by name.
- `unquote_cols` character, columns to elide quotes from.
- `adjust_for_auto_indent` integer additional after first row padding.

**Value**

character

**See Also**

build_frame, qchar_frame
Examples

```r
controlTable <- wrapr::qchar_frame(
  "flower_part", "Length" , "Width" |
  "Petal" , Petal.Length , Petal.Width |
  "Sepal" , Sepal.Length , Sepal.Width )
cat(draw_framec(controlTable, unquote_cols = qc(Length, Width)))
```

---

evalb

Near eval(bquote(expr)) shortcut.

Description

Evaluate `expr` with `bquote .()` substitution. Including .(-x) promoting x’s value from character to a name, which is called "quote negation" (hence the minus-sign).

Usage

```r
evalb(expr, where = parent.frame())
```

Arguments

- **expr**: expression to evaluate.
- **where**: environment to work in.

Value

evaluated substituted expression.

Examples

```r
if(requireNamespace('graphics', quietly = TRUE)) {
  angle = 1:10
  variable <- as.name("angle")
  fn_name <- 'sin'
evalb( plot(x = .(variable), y = .(-fn_name)(.(variable))) )
}
```
execute_parallel

Execute f in parallel partitioned by partition_column.

**Description**

Execute f in parallel partitioned by partition_column, see partition_tables for details.

**Usage**

execute_parallel(
  tables,
  f,
  partition_column,
  ...,
  cl = NULL,
  debug = FALSE,
  env = parent.frame()
)

**Arguments**

- **tables**
  named map of tables to use.
- **f**
  function to apply to each tableset signature is function takes a single argument that is a named list of data.frames.
- **partition_column**
  character name of column to partition on
- **...**
  force later arguments to bind by name.
- **cl**
  parallel cluster.
- **debug**
  logical if TRUE use lapply instead of parallel::clusterApplyLB.
- **env**
  environment to look for values in.

**Value**

list of f evaluations.

**See Also**

partition_tables

**Examples**

```r
if(requireNamespace("parallel", quietly = TRUE)) {
  cl <- parallel::makeCluster(2)
  d <- data.frame(x = 1:5, g = c(1, 2, 2, 2))
```
f <- function(dl) {
  d <- dl$d
  d$s <- sqrt(d$x)
  d
}

r <- execute_parallel(list(d = d), f,
  partition_column = "g",
  cl = cl) %.>%
do.call(rbind, .) %.>%
print(.)

parallel::stopCluster(cl)
}

---

f.  

*Build an anonymous function of dot.*

---

**Description**

Build an anonymous function of dot.

**Usage**

f.(body, env = parent.frame())

**Arguments**

- **body**: function body
- **env**: environment to work in.

**Value**

user defined function.

**See Also**

lambda, defineLambda, named_map_builder, makeFunction_se

**Examples**

f <- f.(sin(.) %.>% cos(.))
7 %.>% f
grepdf

Grep for column names from a data.frame

Description

Grep for column names from a data.frame

Usage

```r
grepdf(
  pattern,
  x,
  ...,  # force later arguments to be passed by name
  ignore.case = FALSE,
  perl = FALSE,
  value = FALSE,
  fixed = FALSE,
  useBytes = FALSE,
  invert = FALSE
)
```

Arguments

- `pattern`: passed to `grep`
- `x`: data.frame to work with
- `...`: force later arguments to be passed by name
- `ignore.case`: passed to `grep`
- `perl`: passed to `grep`
- `value`: passed to `grep`
- `fixed`: passed to `grep`
- `useBytes`: passed to `grep`
- `invert`: passed to `grep`

Value

column names of x matching grep condition.

See Also

grep, grepv
Examples

d <- data.frame(xa=1, yb=2)

# starts with
grepdf('^x', d)

# ends with
grepdf('b$', d)

---

**grepv**

*Return a vector of matches.*

Description

Return a vector of matches.

Usage

grepv(
  pattern,  
  x,        
  ...,      
  ignore.case = FALSE, 
  perl = FALSE,     
  fixed = FALSE,    
  useBytes = FALSE, 
  invert = FALSE
)

Arguments

- **pattern** character scalar, pattern to match, passed to grep.
- **x** character vector to match to, passed to grep.
- **...** not used, forced later arguments to bind by name.
- **ignore.case** logical, passed to grep.
- **perl** logical, passed to grep.
- **fixed** logical, passed to grep.
- **useBytes** logical, passed to grep.
- **invert** passed to grep.

Value

vector of matching values.
has_no_dup_rows

See Also
grep, grepdf

Examples

```r
grepv("x$", c("sox", "xor"))
```

has_no_dup_rows

Check for duplicate rows.

Description

Check a simple data.frame (no list or exotic rows) for duplicate rows.

Usage

```r
has_no_dup_rows(data)
```

Arguments

data data.frame

Value

TRUE if there are no duplicate rows, else FALSE.

invert_perm

Invert a permutation.

Description


Usage

```r
invert_perm(p)
```

Arguments

p vector of length n containing each of seq_len(n) exactly once.
Value

vector q such that p[q] == q[p] == seq_len(length(p))

Examples

```r
p <- c(4, 5, 7, 8, 9, 6, 1, 3, 2, 10)
q <- invert_perm(p)
p[q]
all.equal(p[q], seq_len(length(p)))
q[p]
all.equal(q[p], seq_len(length(p)))
```

---

**lambda**

*Build an anonymous function.*

Description

Mostly just a place-holder so lambda-symbol form has somewhere safe to hang its help entry.

Usage

```r
lambda(..., env = parent.frame())
```

Arguments

... formal parameters of function, unbound names, followed by function body (code/language).

env environment to work in

Value

user defined function.

See Also

defineLambda, makeFunction_se, named_map_builder

Examples

```r
#lambda-syntax: lambda(arg [, arg]*, body [, env=env])
# also works with lambda character as function name
# print(intToUtf8(0x03BB))

# example: square numbers
sapply(1:4, lambda(x, x^2))

# example more than one argument
```
lapplym

Memoizing wrapper for lapply.

Description

Memoizing wrapper for lapply.

Usage

lapplym(X, FUN, ...)

Arguments

X  list or vector of inputs
FUN  function to apply
...  additional arguments passed to lapply

Value

list of results.

See Also

VectorizeM, vapplym, parLapplyLBm

Examples

fs <- function(x) { x <- x[[1]]; print(paste("see", x)); sin(x) }
# should only print "see" twice, not 6 times
lapplym(c(0, 1, 1, 0, 0, 1), fs)
let  

Execute expr with name substitutions specified in alias.

Description

let implements a mapping from desired names (names used directly in the expr code) to names used in the data. Mnemonic: "expr code symbols are on the left, external data and function argument names are on the right."

Usage

```r
let(
    alias,
    expr,
    ...
)
```

Arguments

- **alias**: mapping from free names in expr to target names to use (mapping have both unique names and unique values).
- **expr**: block to prepare for execution.
- **...**: force later arguments to be bound by name.
- **envir**: environment to work in.
- **subsMethod**: character substitution method, one of 'langsubs' (preferred), 'subsubs', or 'stringsubs'.
- **strict**: logical if TRUE names and values must be valid un-quoted names, and not dot.
- **eval**: logical if TRUE execute the re-mapped expression (else return it).
- **debugPrint**: logical if TRUE print debugging information when in stringsubs mode.

Details

Please see the wrapr vignette for some discussion of let and crossing function call boundaries: vignette('wrapr', 'wrapr'). For formal documentation please see https://github.com/WinVector/wrapr/blob/master/extras/wrapr_let.pdf. Transformation is performed by substitution, so please be wary of unintended name collisions or aliasing.

Something like let is only useful to get control of a function that is parameterized (in the sense it take column names) but non-standard (in that it takes column names from non-standard evaluation argument name capture, and not as simple variables or parameters). So wrapr::let is not useful for
non-parameterized functions (functions that work only over values such as `base::sum`), and not useful for functions take parameters in straightforward way (such as `base::merge`'s "by" argument). `dplyr::mutate` is an example where we can use a `let` helper. `dplyr::mutate` is parameterized (in the sense it can work over user supplied columns and expressions), but column names are captured through non-standard evaluation (and it rapidly becomes unwieldy to use complex formulas with the standard evaluation equivalent `dplyr::mutate_`). alias can not include the symbol ".".

The intent from is from the user perspective to have (if `a <- 1; b <- 2`): `let(c(z = 'a'), z+b)` to behave a lot like `eval(substitute(z+b,c(z=quote(a))))`.

`let` deliberately checks that it is mapping only to legal R names; this is to discourage the use of `let` to make names to arbitrary values, as that is the more properly left to R's environment systems. `let` is intended to transform "tame" variable and column names to "tame" variable and column names. Substitution outcomes that are not valid simple R variable names (produced with out use of back-ticks) are forbidden. It is suggested that substitution targets be written `ALL_CAPS` style to make them stand out.

`let` was inspired by `gtools:strmacro()`. Please see `https://github.com/WinVector/wrapr/blob/master/extras/MacrosInR.md` for a discussion of macro tools in R.

### Value

result of expr executed in calling environment (or expression if eval==FALSE).

### See Also

`bquote`, `do.call`

### Examples

```r
d <- data.frame(
  Sepal_Length=c(5.8,5.7),
  Sepal_Width=c(4.0,4.4),
  Species='setosa')
mapping <- qc(
  AREA_COL = Sepal_area,
  LENGTH_COL = Sepal_Length,
  WIDTH_COL = Sepal_Width
)

# let-block notation
let(
  mapping,
  d %.>%
  transform(., AREA_COL = LENGTH_COL * WIDTH_COL)
)

# Note: in packages can make assignment such as:
# AREA_COL <- LENGTH_COL <- WIDTH_COL <- NULL
# prior to code so targets don't look like unbound names.
```
makeFunction_se

Build an anonymous function.

Description

Build an anonymous function.

Usage

makeFunction_se(params, body, env = parent.frame())

Arguments

params formal parameters of function, unbound names.
body substituted body of function to map arguments into.
env environment to work in.

Value

user defined function.

See Also

lambda, defineLambda, named_map_builder

Examples

```r
f <- makeFunction_se(as.name('x'), substitute({x*x}))
f(7)
```

```r
g <- makeFunction_se(c(as.name('x'), as.name('y')), substitute({ x + 3*y })))
g(1,100)
```
mapsyms

Map symbol names to referenced values if those values are string scalars (else throw).

Description

Map symbol names to referenced values if those values are string scalars (else throw).

Usage

mapsyms(...)

Arguments

... symbol names mapping to string scalars

Value

map from original symbol names to new names (names found in the current environment)

See Also

let

Examples

x <- 'a'
y <- 'b'
print(mapsyms(x, y))
d <- data.frame(a = 1, b = 2)
let(mapsyms(x, y), d$x + d$y)

map_to_char

format a map.

Description

format a map.

Usage

map_to_char(mp, ..., sep = " ", assignment = "=", quote_fn = base::shQuote)
Arguments

- **mp** named vector or list
- **sep** separator suffix, what to put after commas
- **assignment** assignment string
- **quote_fn** string quoting function

Value

character formatted representation

See Also

dput, capture.output

Examples

```r
map_upper

# Examples

cat(map_to_char(c('a' = 'b', 'c' = 'd')))
cat(map_to_char(c('a' = 'b', 'd', 'e' = 'f')))
cat(map_to_char(c('a' = 'b', 'd' = NA, 'e' = 'f')))
cat(map_to_char(c(1, NA, 2)))
```

Description

Map upcased symbol names to referenced values if those values are string scalars (else throw).

Usage

map_upper(...)

Arguments

... symbol names mapping to string scalars

Value

map from original symbol names to new names (names found in the current environment)

See Also

let
**match_order**

Match one order to another.

Description


Usage

match_order(ids1, ids2)

Arguments

ids1

unique vector of ids.

ids2

unique vector of ids with sort(ids1)==sort(ids2).

Value

p integers such that ids1[p] == ids2

Examples

```r
ids1 <- c(4, 5, 7, 8, 9, 6, 1, 3, 2, 10)
ids2 <- c(3, 6, 4, 8, 5, 7, 1, 9, 10, 2)
p <- match_order(ids1, ids2)
ids1[p]
all.equal(ids1[p], ids2)
# note base::match(ids2, ids1) also solves this problem
```
Description

Safely construct a simple Wilkinson notation formula from the outcome (dependent variable) name and vector of input (independent variable) names.

Usage

```
mk_formula(
  outcome, # character scalar, name of outcome or dependent variable.
  variables, # character vector, names of input or independent variables.
  ..., # not used, force later arguments to bind by name.
  intercept = TRUE, # logical, if TRUE allow an intercept term.
  outcome_target = NULL, # scalar, if not NULL write outcome==outcome_target in formula.
  outcome_comparator = "==", # one of "==", "!="", ">="", "<="", ">", "<", only use of outcome_target is not NULL.
  env = baseenv(), # environment to use in formula (unless extra_values is non empty, then this is a parent environment).
  extra_values = NULL, # if not empty extra values to be added to a new formula environment containing env.
  as_character = FALSE # if TRUE return formula as a character string.
)
```

Arguments

- `outcome`: character scalar, name of outcome or dependent variable.
- `variables`: character vector, names of input or independent variables.
- `...`: not used, force later arguments to bind by name.
- `intercept`: logical, if TRUE allow an intercept term.
- `outcome_target`: scalar, if not NULL write outcome==outcome_target in formula.
- `outcome_comparator`: one of ">="", "!="", ">="", "<="", ">", "<", only use of outcome_target is not NULL.
- `env`: environment to use in formula (unless extra_values is non empty, then this is a parent environment).
- `extra_values`: if not empty extra values to be added to a new formula environment containing env.
- `as_character`: if TRUE return formula as a character string.

Details

Note: outcome and variables are each intended to be simple variable names or column names (or .). They are not intended to specify interactions, I()-terms, transforms, general expressions or other complex formula terms. Essentially the same effect as `reformulate`, but trying to avoid the `paste` currently in `reformulate` by calling `update.formula` (which appears to work over terms). Another
reasonable way to do this is just `paste(outcome,paste(variables,collapse = " + "),sep = " ~ ").

Care must be taken with later arguments to functions like `lm()` whose help states: "All of weights, subset and offset are evaluated in the same way as variables in formula, that is first in data and then in the environment of formula." Also note `env` defaults to `baseenv()` to try and minimize reference leaks produced by the environment captured by the formal ending up stored in the resulting model for `lm()` and `glm()`. For behavior closer to `as.formula()` please set the `env` argument to `parent.frame()`.

Value

a formula object

See Also

`reformulate`, `update.formula`

Examples

```r
f <- mk_formula("mpg", c("cyl", "disp"))
print(f)
(model <- lm(f, mtcars))
format(model$terms)

f <- mk_formula("cyl", c("wt", "gear"), outcome_target = 8, outcome_comparator = ">=")
print(f)
(model <- glm(f, mtcars, family = binomial))
format(model$terms)
```

---

`mk_tmp_name_source`  Produce a temp name generator with a given prefix.

Description

Returns a function f where: f() returns a new temporary name, f(remove=vector) removes names in vector and returns what was removed, f(dumpList=TRUE) returns the list of names generated and clears the list, f(peek=TRUE) returns the list without altering anything.

Usage

```r
mk_tmp_name_source(
  prefix = "tmpnam",
  ..., 
  alphabet = as.character(0:9),
  size = 20,
  sep = "_"
)
```
Arguments

- **prefix**: character, string to prefix temp names with.
- ... force later argument to be bound by name.
- **alphabet**: character, characters to choose from in building ids.
- **size**: character, number of characters to build id portion of names from.
- **sep**: character, separator between temp name fields.

Value

name generator function.

Examples

```r
f <- mk_tmp_name_source('ex')
print(f())
nm2 <- f()
print(nm2)
f(remove=nm2)
print(f(dumpList=TRUE))
```

named_map_builder  

Named map builder.

Description

Set names of right-argument to be left-argument, and return right argument. Called from `:=` operator.

Usage

```r
named_map_builder(targets, values)
```

`:=`(targets, values)

```r
targets %:=% values
```

Arguments

- **targets**: names to set.
- **values**: values to assign to names (and return).

Value

values with names set.
See Also

lambda, defineLambda, makeFunction_se

Examples

c('a' := '4', 'b' := '5')
# equivalent to: c(a = '4', b = '5')

c('a', 'b') := c('1', '2')
# equivalent to: c(a = '1', b = '2')

# the important example
name <- 'a'
name := '5'
# equivalent to: c('a' = '5')

orderv

Order by a list of vectors.

Description

Produce an ordering permutation from a list of vectors. Essentially a non-... interface to order.

Usage

orderv(
columns,
...,
na.last = TRUE,
decreasing = FALSE,
method = c("auto", "shell", "radix")
)

Arguments

columns list of atomic columns to order on, can be a data.frame.
... not used, force later arguments to bind by name.
na.last (passed to order) for controlling the treatment of NAs. If TRUE, missing values in the data are put last; if FALSE, they are put first; if NA, they are removed.
decreasing (passed to order) logical. Should the sort order be increasing or decreasing? For the "radix" method, this can be a vector of length equal to the number of arguments in ... For the other methods, it must be length one.
method (passed to order) the method to be used: partial matches are allowed. The default ("auto") implies "radix" for short numeric vectors, integer vectors, logical vectors and factors. Otherwise, it implies "shell". For details of methods "shell", "quick", and "radix", see the help for sort.
pack

Description
This function packs values given by name into a named list.

Usage
pack(..., .wrapr_private_var_env = parent.frame())

Arguments
... values to pack, these should be specified by name (not as constants).
.wrapr_private_var_env
   environment to evaluate in

Value
named list of values

See Also
unpack
Examples

```r
x <- 1
y <- 2
pack(x, y)  # list(x = 1, y = 2)

pack(a = x, y)  # list(a = 1, y = 2)
pack(a = 5, y)  # list(a = 5, y = 2)
pack(1, 2)  # list('1' = 1, '2' = 2)

v <- pack(x = 8, y = 9)  # list(x = 8, y = 9)
v -> unpack[x, y]
print(x)  # 8
print(y)  # 9
```

parLapplyLBm  

*Memoizing wrapper for parLapplyLB*

Description

Memoizing wrapper for parLapplyLB

Usage

```r
parLapplyLBm(cl = NULL, X, fun, ..., chunk.size = NULL)
```

Arguments

- `cl`  
  - cluster object
- `X`  
  - list or vector of inputs
- `fun`  
  - function to apply
- `...`  
  - additional arguments passed to lapply
- `chunk.size`  
  - passed to parallel::parLapplyLB

Value

list of results.

See Also

`parLapplyLB, lapplym, VectorizeM, vapplym`
Examples

```r
if(requireNamespace("parallel", quietly = TRUE)) {
  cl <- parallel::makeCluster(2)
  fs <- function(x) { x <- x[[1]]; Sys.sleep(1); sin(x) }
  # without memoization should take 1000 seconds
  lst <- parLapplyLBm(cl, c(rep(0, 1000), rep(1, 1000)), fs)
  parallel::stopCluster(cl)
}
```

---

**partition_tables**

*Partition as set of tables into a list.*

**Description**

Partition a set of tables into a list of sets of tables. Note: removes rownames.

**Usage**

```r
partition_tables(
  tables_used, 
  partition_column, 
  ..., 
  source_usage = NULL, 
  source_limit = NULL, 
  tables = NULL, 
  env = NULL 
)
```

**Arguments**

- `tables_used` character, names of tables to look for.
- `partition_column` character, name of column to partition by (tables should not have NAs in this column).
- `...` force later arguments to bind by name.
- `source_usage` optional named map from tables_used names to sets of columns used.
- `source_limit` optional numeric scalar limit on rows wanted every source.
- `tables` named map from tables_used names to data.frames.
- `env` environment to also look for tables named by tables_used

**Value**

list of names maps of data.frames partitioned by partition_column.
See Also

execute_parallel

Examples

```r
d1 <- data.frame(a = 1:5, g = c(1, 1, 2, 2, 2))
d2 <- data.frame(x = 1:3, g = 1:3)
d3 <- data.frame(y = 1)
partition_tables(c("d1", "d2", "d3"), "g", tables = list(d1 = d1, d2 = d2, d3 = d3))
```

pipe_impl  

_Pipe dispatch implementation._

Description

This is a helper for implementing additional pipes.

Usage

```r
pipe_impl(pipe_left_arg, pipe_right_arg, pipe_environment, pipe_string = NULL)
```

Arguments

- `pipe_left_arg`: possibly unevaluated left argument.
- `pipe_right_arg`: possibly unevaluated right argument.
- `pipe_environment`: environment to evaluate in.
- `pipe_string`: character, name of pipe operator.

Value

`result`

Examples

```r
# Example: how wrapr pipe is implemented

print("%>%")

# Example: create a value that causes pipelines to record steps.

# inject raw values into wrapped/annotated world
```
unit_recording <- function(x, recording = paste(as.expression(substitute(x)), collapse = '\n')) {
  res <- list(value = x, recording = recording)
  class(res) <- "recording_value"
  res
}

# similar to bind or >>=
# (takes U, f:U -> V to M(f(U)), instead of
#  U, f:U -> M(V) to M(f(U)))
# so similar to a functor taking
#  f:U -> V to f':M(U) -> M(V)
# followed by application.
apply_left.recording_value <- function(
  pipe_left_arg,
  pipe_right_arg,
  pipe_environment,
  left_arg_name,
  pipe_string,
  right_arg_name) {
  force(pipe_environment)
  tmp <- wrapr::pipe_impl(
    pipe_left_arg = pipe_left_arg$value,
    pipe_right_arg = pipe_right_arg,
    pipe_environment = pipe_environment,
    pipe_string = pipe_string)
  unit_recording(
    tmp,
    paste0(pipe_left_arg$recording,
      ' %>% ',
      paste(as.expression(pipe_right_arg), collapse = '\n')))
}

# make available on standard S3 search path
assign('apply_left.recording_value',
  apply_left.recording_value,
  envir = .GlobalEnv)

unpack[value, recording] := 3 %>%
  unit_recording(.) %>%
  sin(.) %>%
  cos(.)

print(value)
print(recording)

# clean up
rm(envir = .GlobalEnv, list = 'apply_left.recording_value')
Description

Take a vector or list and return the first element (pseudo-aggregation or projection). If the argument length is zero or there are different items throw in an error.

Usage

psagg(x, ..., strict = TRUE)

Arguments

- **x**: should be a vector or list of items.
- **...**: force later arguments to be passed by name
- **strict**: logical, should we check value uniqueness.

Details

This function is useful in some split by column situations as a safe and legible way to convert vectors to scalars.

Value

x[[1]] (or throw if not all items are equal or this is an empty vector).

Examples

d <- data.frame(
  group = c("a", "a", "b"),
  stringsAsFactors = FALSE)
dl <- lapply(
  split(d, d$group),
  function(di) {
    data.frame(
      # note: di$group is a possibly length>1 vector!
      # pseudo aggregate it to the value that is
      # constant for each group, confirming it is constant.
      group_label = psagg(di$group),
      group_count = nrow(di),
      stringsAsFactors = FALSE
    )
  })
do.call(rbind, dl)
qae

Quote assignment expressions (name = expr, name := expr, name %==% expr).

Description

Accepts arbitrary un-parsed expressions as assignments to allow forms such as "Sepal_Long := Sepal.Length >= 2 * Sepal.Width". (without the quotes). Terms are expressions of the form "lhs := rhs", "lhs = rhs", "lhs %==% rhs".

Usage

qae(...)

Arguments

... assignment expressions.

Details

qae() uses bquote().() quasiquotation escaping notation, and .(~) "string quotes, string to name" notation.

Value

array of quoted assignment expressions.

See Also

qc, qe

Examples

ratio <- 2

exprs <- qae(Sepal_Long := Sepal.Length >= ratio * Sepal.Width,
              Petal_Short = Petal.Length <= 3.5)
print(exprs)

exprs <- qae(Sepal_Long := Sepal.Length >= .(ratio) * Sepal.Width,
              Petal_Short = Petal.Length <= 3.5)
print(exprs)

# library("rqdatatable")
# datasets::iris %>%
# extend_se(., exprs) %>%
# summary(.)
The qc() function is intended to help quote user inputs.

Usage

qc(..., .wrapr_private_var_env = parent.frame())

Arguments

... items to place into an array

.value wrapr_private_var_env
environment to evaluate in

Details

qc() a convenience function allowing the user to elide excess quotation marks. It quotes its arguments instead of evaluating them, except in the case of a nested call to qc() or c(). Please see the examples for typical uses both for named and un-named character vectors.

qc() uses bquote().() quasiquotation escaping notation.

Value

quoted array of character items

See Also

qe, qae, bquote

Examples

a <- "x"
qc(a) # returns the string "a" (not "x")
qc(.(a)) # returns the string "x" (not "a")
qc(.(a) := a) # returns c("x" = "a")
qc("a") # return the string "a" (not "\"a\"")
qc(sin(x)) # returns the string "sin(x)"
qc(a, qc(b, c)) # returns c("a", "b", "c")
qc(a, c("b", "c")) # returns c("a", "b", "c")

qc(x=a, qc(y=b, z=c)) # returns c(x="a", y="b", z="c")

c('x'=a', wrapr::qc('y'='b', 'z'='c')) # returns c(x="a", y="b", z="c")

qc(a : c(a=1, b=2)) # returns c(a.a = "1", a.b = "2")

qc(a := c(a=1, b=2)) # returns c(a.a = "1", a.b = "2")

c(a = c(a="1", b="2")) # returns c(a.a = "1", a.b = "2")

qchar_frame

Build a quoted data.frame.

Description

A convenient way to build a character data.frame in legible transposed form. Position of first "|
(or other infix operator) determines number of columns (all other infix operators are aliases for ".")
Names are treated as character types.

Usage

qchar_frame(...)

Arguments

... cell names, first infix operator denotes end of header row of column names.

Details

qchar_frame() uses bquote() .() quasiquotation escaping notation. Because of this using dot as a
name in some places may fail if the dot looks like a function call.

Value

character data.frame

See Also

draw_frame, build_frame
Examples

```r
loss_name <- "loss"
x <- qchar_frame(
  measure, training, validation |
  "minus binary cross entropy", .(loss_name), val_loss |
  accuracy, acc, val_acc )
print(x)
str(x)
cat(draw_frame(x))

qchar_frame(
  x |
  1 |
  2 ) %>% str(.)
```

Description

Accepts arbitrary un-parsed expressions as to allow forms such as "Sepal.Length >= 2 * Sepal.Width" (without the quotes).

Usage

```r
qe(...)
```

Arguments

```r
... assignment expressions.
```

Details

`qe()` uses `bquote()` .() quasiquotation escaping notation, and .(~) "string quotes, string to name" notation.

Value

array of quoted assignment expressions.

See Also

`qc`, `qae`
Examples

```r
ratio <- 2

exprs <- qe(Sepal.Length >= ratio * Sepal.Width,
             Petal.Length <= 3.5)
print(exprs)

exprs <- qe(Sepal.Length >= .(ratio) * Sepal.Width,
             Petal.Length <= 3.5)
print(exprs)
```

qs

Quote argument as a string.

Description

qs() uses bquote().() quasiquotation escaping notation.

Usage

```r
qs(s)
```

Arguments

- `s`: expression to be quoted as a string.

Value

character

Examples

```r
x <- 7
qs(a == x)
qs(a == .(x))
```
Use function to reduce or expand arguments.

Description

\[ x .|.| f \] stands for \( f(x[[1]], x[[2]], \ldots, x[[\text{length}(x)]]). \] \( v .|.| x \) also stands for \( f(x[[1]], x[[2]], \ldots, x[[\text{length}(x)]]). \)

The two operators are the same, the variation just allowing the user to choose the order they write things. The mnemonic is: "data goes on the dot-side of the operator."

Usage

\[
\begin{align*}
f & \%|.| args \\
args & %.|% f
\end{align*}
\]

Arguments

- **f**: function.
- **args**: argument list or vector, entries expanded as function arguments.

Details

Note: the reduce operation is implemented by `do.call()`, so has standard R named argument semantics.

Value

\( f(args) \) where args elements become individual arguments of \( f \).

Functions

- \%|.|: \( f \) reduce args
- \%.|%: \( f \) expand args

See Also

`do.call`, `list`, `c`

Examples

```r
args <- list('prefix_1', c(1:3), '_suffix')
args %|.| paste0
# prefix_1_suffix prefix_2_suffix prefix_3_suffix
paste0 %.|% args
# prefix_1_suffix prefix_2_suffix prefix_3_suffix
```
**restrictToNameAssignments**

*Restrict an alias mapping list to things that look like name assignments*

**Description**

Restrict an alias mapping list to things that look like name assignments

**Usage**

```r
restrictToNameAssignments(alias, restrictToAllCaps = FALSE)
```

**Arguments**

- `alias` mapping list
- `restrictToAllCaps` logical, if true only use all-capitalized keys

**Value**

string to string mapping

**Examples**

```r
alias <- list(region = 'east', str = "seven")
aliasR <- restrictToNameAssignments(alias)
print(aliasR)
```

---

**seqi**

*Increasing whole-number sequence.*

**Description**

Return an increasing whole-number sequence from `a` to `b` inclusive (return `integer(0)` if none such). Allows for safe iteration.

**Usage**

```r
seqi(a, b)
```

**Arguments**

- `a` scalar lower bound
- `b` scalar upper bound
Value
whole number sequence

Examples

# print 3, 4, and then 5
for(i in seqi(3, 5)) {
  print(i)
}

# empty
for(i in seqi(5, 2)) {
  print(i)
}

si

Dot substitution string interpolation.

Description
String interpolation using bquote-stype .() notation. Pure R, no C/C++ code called. sinterp and si are synonyms.

Usage

si(
  str,
  ..., 
  envir = parent.frame(),
  enclos = parent.frame(),
  match_pattern = "\\.\\\{([^()]\+)(\\([\^()]*)\\)\}\+\+\",
  removal_patterns = c("\\.\\(\", "\\)\$")
)

Arguments

str character string to be substituted into
... force later arguments to bind by name
envir environment to look for values
enclos enclosing evaluation environment
match_pattern regexp to find substitution targets.
removal_patterns regexps to remove markers from substitution targets.
Details


Value

modified strings

See Also

strsplit_capture, sinterp

Examples

```r
x <- 7
si("x is .x, x+1 is .(x+1)\n.x is odd is .x%%2 == 1")

# Because matching is done by a regular expression we
# can not use arbitrary depths of nested parenthesis inside
# the interpolation region. The default regexp allows
# one level of nesting (and one can use () in place
# of parens in many places).
si("sin(x*(x+1)) is .(sin(x*{x+1}))")

# We can also change the delimiters,
# in this case to !! through the first whitespace.
si(c("x is !!x , x+1 is !!x+1 \n!!x is odd is !!x%%2==1"),
  match_pattern = "!![^[:space:]]+[[:space:]]?",
  removal_patterns = c("^!!", "[[:space:]]?$")
```

sinterp

Dot substitution string interpolation.

Description


Usage

```r
sinterp(
  str,
  ..., 
  envir = parent.frame(),
  enclos = parent.frame(),
  match_pattern = "\\.\\{\([^()\+]*\)\}\{\([^()\+]*\)\}+\}+", 
  removal_patterns = c("\\\\\\(", "\\$")
)
```
sinterp

Arguments

str character string(s) to be substituted into
...
force later arguments to bind by name
envir environment to look for values
enclos enclosing evaluation environment
match_pattern regexp to find substitution targets.
removal_patterns 
regexps to remove markers from substitution targets.

Details


Value

modified strings

See Also

strsplit_capture, si

Examples

x <- 7
sinterp("x is .(x), x+1 is .(x+1)\n.(x) is odd is .(x%%2 == 1)"")
# Because matching is done by a regular expression we
# can not use arbitrary depths of nested parenthesis inside
# the interpolation region. The default regexp allows
# one level of nesting (and one can use () in place
# of parens in many places).
sinterp("\(\sin(\(x\*(x+1))\)\) is .(\(\sin(\(x\*(x+1))\))\)")
# We can also change the delimiters,
# in this case to !! through the first whitespace.
sinterp(c("x is !!x \ x+1 is !!x+1 \n!!x \ x is odd is !!x%%2==1"),
  match_pattern = '!![^[:space:]]+[^[:space:]]?",
  removal_patterns = c("!!", "[[:space:]]?$"))
sortv  

Sort a data.frame.

Description

Sort a data.frame by a set of columns.

Usage

```r
sortv(
  data,
  colnames,
  ...,  
  na.last = TRUE,
  decreasing = FALSE,
  method = c("auto", "shell", "radix")
)
```

Arguments

- `data`: data.frame to sort.
- `colnames`: column names to sort on.
- `...`: not used, force later arguments to bind by name.
- `na.last`: (passed to `order`) for controlling the treatment of NAs. If TRUE, missing values in the data are put last; if FALSE, they are put first; if NA, they are removed.
- `decreasing`: (passed to `order`) logical. Should the sort order be increasing or decreasing? For the "radix" method, this can be a vector of length equal to the number of arguments in .... For the other methods, it must be length one.
- `method`: (passed to `order`) the method to be used: partial matches are allowed. The default ("auto") implies "radix" for short numeric vectors, integer vectors, logical vectors and factors. Otherwise, it implies "shell". For details of methods "shell", "quick", and "radix", see the help for `sort`.

Value

ordering permutation

See Also

- `orderv`

Examples

```r
d <- data.frame(x = c(2, 2, 3, 3, 1, 1), y = 6:1)
sortv(d, c("x", "y"))
```
**split_at_brace_pairs**

*Split strings at -pairs.*

**Description**

Split strings at -pairs.

**Usage**

```r
split_at_brace_pairs(s, open_symbol = "{", close_symbol = ")")
```

**Arguments**

- `s`: string or list of strings to split.
- `open_symbol`: symbol to start marking.
- `close_symbol`: symbol to end marking.

**Value**

array or list of split strings.

**Examples**

```r
split_at_brace_pairs("(x) + y + (z)")
```

**stop_if_dot_args**

*Stop with message if dot_args is a non-trivial list.*

**Description**

Generate a stop with a good error message if the dots argument was a non-trivial list. Useful in writing functions that force named arguments.

**Usage**

```r
stop_if_dot_args(dot_args, msg = "")
```

**Arguments**

- `dot_args`: substitute(list(...)) from another function.
- `msg`: character, optional message to prepend.
Value

NULL or stop()

Examples

```r
f <- function(x, ..., inc = 1) {
  stop_if_dot_args(substitute(list(...)), "f")
  x + inc
}

f(7)
f(7, inc = 2)
tryCatch(
  f(7, 2),
  error = function(e) { print(e) }
)
```

strsplit_capture

Split a string, keeping separator regions

Description

Split a string, keeping separator regions

Usage

```r
strsplit_capture(
  x,
  split,
  ...,  # force later arguments to bind by name
  ignore.case = FALSE,
  fixed = FALSE,
  perl = FALSE,
  useBytes = FALSE
)
```

Arguments

- `x`: character string to split (length 1 vector)
- `split`: split pattern
- `...`: force later arguments to bind by name
- `ignore.case`: passed to `gregexpr`
- `fixed`: passed to `gregexpr`
- `perl`: passed to `gregexpr`
- `useBytes`: passed to `gregexpr`
Value

list of string segments annotated with is_sep.

See Also

sinterp, si

Examples

strsplit_capture("x is .(x) and x+1 is .(x+1)", "\\.\(([^()]+)\\)\")

to

Unpack or bind values by names into the calling environment, eager eval (no-dot) variation.

Description

Unpacks or binds values into the calling environment, eager eval (no-dot) variation. Uses bquote escaping. NULL is a special case that is unpacked to all targets. NA targets are skipped. All non-NA target names must be unique.

Usage

to(...)

Arguments

... argument names to write to

Details

Note: when using []<- notation, a reference to the unpacker object is written into the unpacking environment as a side-effect of the implied array assignment. := assignment does not have this side-effect. Array-assign form can not use the names: .., wrapr_private_self, value, or to. function form can not use the names: . or wrapr_private_value. For more details please see here https://win-vector.com/2020/01/20/unpack-your-values-in-r/.

Related work includes Python tuple unpacking, zeallot's arrow, and vadr::bind.

Value

a UnpackTarget
Examples

# named unpacking
# looks like assignment: DESTINATION = NAME_VALUE_USING

d <- data.frame(x = 1:2,
                 g=c('test', 'train'),
                 stringsAsFactors = FALSE)

to[train_set = train, test_set = test] := split(d, d$g)

# train_set and test_set now correctly split
print(train_set)
print(test_set)
rm(list = c('train_set', 'test_set'))

# named unpacking NEWNAME = OLDNAME implicit form
# values are matched by name, not index

to[train, test] := split(d, d$g)

print(train)
print(test)
rm(list = c('train', 'test'))

# pipe version (notice no dot)

split(d, d$g) %.>% to(train, test)

print(train)
print(test)
rm(list = c('train', 'test'))

# Note: above is wrapr dot-pipe, piping does not currently work with
# magrittr pipe due to magrittr's introduction of temporary
# intermediate environments during evaluation.

# bquote example

train_col_name <- 'train'
test_col_name <- 'test'
to[train = .(train_col_name), test = .(test_col_name)] := split(d, d$g)

print(train)
print(test)
rm(list = c('train', 'test'))

---

uniques  Strict version of unique (without ...).

Description

Check that ... is empty and if so call base::unique(x, incomparables = incomparables, MARGIN = MARGIN, fromLast = fromLast) (else throw an error)

Usage

uniques(x, ..., incomparables = FALSE, MARGIN = 1, fromLast = FALSE)
unpack

Arguments

x items to be compared.
...
not used, checked to be empty to prevent errors.
incomparables passed to base::unique.
MARGIN passed to base::unique.
fromLast passed to base::unique.

Value

base::unique(x, incomparables = incomparables, MARGIN = MARGIN, fromLast = fromLast)

Examples

x = c("a", "b")
y = c("b", "c")

# task: get unique items in x plus y
unique(c(x, y)) # correct answer
unique(x, y) # oops forgot to wrap arguments, quietly get wrong answer
tryCatch(
    uniques(x, y), # uniques catches the error
error = function(e) { e })
uniques(c(x, y)) # uniques works like base::unique in most case
Details

Note: when using []<- notation, a reference to the unpacker object is written into the unpacking environment as a side-effect of the implied array assignment. := assignment does not have this side-effect. Array-assign form can not use the names: .., wrapr_private_self, value, or unpack. Function form can not use the names: . or wrapr_private_value. For more details please see here https://win-vector.com/2020/01/20/unpack-your-values-in-r/.

Related work includes Python tuple unpacking, zeallot’s arrow, and vadr::bind.

Value

value passed in (invisible)

See Also

pack

Examples

# named unpacking
# looks like assignment: DESTINATION = NAME_VALUE_USING
d <- data.frame(x = 1:2,
    g=c('test', 'train'),
    stringsAsFactors = FALSE)
unpack[train_set = train, test_set = test] := split(d, d$g)
# train_set and test_set now correctly split
print(train_set)
print(test_set)
rm(list = c('train_set', 'test_set'))

# named unpacking NEWNAME = OLDNAME implicit form
# values are matched by name, not index
unpack[train, test] := split(d, d$g)
print(train)
print(test)
rm(list = c('train', 'test'))

# function version
unpack(split(d, d$g), train, test)
print(train)
print(test)
rm(list = c('train', 'test'))

# pipe version
split(d, d$g) %.>% unpack(., train, test)
print(train)
print(test)
rm(list = c('train', 'test'))

# Note: above is wrapr dot-pipe, piping does not currently work with
# magrittr pipe due to magrittr’s introduction of temporary
# intermediate environments during evaluation.
# bquote example
train_col_name <- `train'
test_col_name <- `test'
unpack(split(d, d$g), train = .(train_col_name), test = .(test_col_name))
print(train)
print(test)
rm(list = c('train', 'test'))

---

vapplym

Memoizing wrapper for vapply.

Description

Memoizing wrapper for vapply.

Usage

vapplym(X, FUN, FUN.VALUE, ..., USE.NAMES = TRUE)

Arguments

X list or vector of inputs
FUN function to apply
FUN.VALUE type of vector to return
... additional arguments passed to lapply
USE.NAMES passed to vapply

Value

vector of results.

See Also

VectorizeM, lapplym

Examples

fs <- function(x) { x <- x[[1]]; print(paste("see", x)); sin(x) }
# should only print "see" twice, not 6 times
vapplym(c(0, 1, 1, 0, 0, 1), fs, numeric(1))
**VectorizeM**  
*Memoizing wrapper to base::Vectorize()*

**Description**

Build a wrapped function that applies to each unique argument in a vector of arguments once.

**Usage**

```r
VectorizeM(
    FUN,
    vectorize.args = arg.names,
    SIMPLIFY = TRUE,
    USE.NAMES = TRUE,
    UNLIST = FALSE
)
```

**Arguments**

- **FUN**: function to apply
- **vectorize.args**: a character vector of arguments which should be vectorized. Defaults to first argument of FUN. If set must be length 1.
- **SIMPLIFY**: logical or character string; attempt to reduce the result to a vector, matrix or higher dimensional array; see the simplify argument of sapply.
- **USE.NAMES**: logical; use names if the first ... argument has names, or if it is a character vector, use that character vector as the names.
- **UNLIST**: logical; if TRUE try to unlist the result.

**Details**

Only sensible for pure side-effect free deterministic functions.

**Value**

adapted function (vectorized with one call per different value).

**See Also**

Vectorize, vapplym, lapplym

**Examples**

```r
fs <- function(x) { x <- x[1]; print(paste("see", x)); sin(x) }
fv <- VectorizeM(fs)
# should only print "see" twice, not 6 times
fv(c(0, 1, 1, 0, 0, 1))
```
Invoke a spreadsheet like viewer when appropriate.

Description

Invoke a spreadsheet like viewer when appropriate.

Usage

view(x, ..., title = wrapr_deparse(substitute(x)), n = 200)

Arguments

x R object to view
...

force later arguments to bind by name.
title title for viewer
n number of rows to show

Value

invoke view or format object

Examples

view(mtcars)

Description

Provides DebugFnW() to capture function context on error for debugging, and let() which converts non-standard evaluation interfaces to parametric standard evaluation interfaces. DebugFnW() captures the calling function and arguments prior to the call causing the exception, while the classic options(error=dump.frames) form captures at the moment of the exception itself (thus function arguments may not be at their starting values). let() rebinds (possibly unbound) names to names.
Details

For more information:

- vignette('DebugFnW',package='wrapr')
- vignette('let',package='wrapr')
- vignette(package='wrapr')
- Website: https://github.com/WinVector/wrapr
- let video: https://youtu.be/iKLGxzzm9Hk?list=PLAKBwakacHbQp_Z66asDnjn-0qttTO-o9
- Debug wrapper video: https://youtu.be/zFEC9-1XSN8?list=PLAKBwakacHbQT51nPHex1on3YNCCmggZA.
Unpack or bind values into the calling environment.

Description

Unpacks or binds values into the calling environment. Uses bquote escaping. NULL is a special case that is unpacked to all targets. NA targets are skipped. All non-NA target names must be unique.

Usage

## S3 replacement method for class 'Unpacker'

```r
wrapr_private_self[...] <- value
```

Arguments

- `wrapr_private_self`:
  object implementing the feature, `wrapr::unpack`
- `...`
  names of to unpack to (can be escaped with bquote .() notation).
- `value`
  list to unpack into values, must have a number of entries equal to number of ... arguments

Details

Note: when using `[]<-` notation, a reference to the unpacker object is written into the unpacking environment as a side-effect of the implied array assignment. `:=` assignment does not have this side-effect. Array-assign form can not use the names: .., `wrapr_private_self`, `value`, or the name of the unpacker itself. For more details please see here [https://win-vector.com/2020/01/20/unpack-your-values-in-r/](https://win-vector.com/2020/01/20/unpack-your-values-in-r/).

Related work includes Python tuple unpacking, `zeallot`'s arrow, and `vadr::bind`.

Value

- `wrapr_private_self`

Examples

```r
# named unpacking
# looks like assignment: DESTINATION = NAME_VALUE_USING
d <- data.frame(x = 1:2,
               g=c('test', 'train'),
               stringsAsFactors = FALSE)
to[train_set = train, test_set = test] := split(d, d$g)
# train_set and test_set now correctly split
print(train_set)
print(test_set)
rm(list = c('train_set', 'test_set'))
```
# named unpacking \texttt{NEWNAME = OLDNAME} implicit form
# values are matched by name, not index
to[train, test] := split(d, d$g)
print(train)
print(test)
rm(list = c('train', 'test'))

# bquote example
train_col_name <- 'train'
test_col_name <- 'test'
to[train = .(train_col_name), test = .(test_col_name)] := split(d, d$g)
print(train)
print(test)
rm(list = c('train', 'test'))

---

%in_block% \hspace{1cm} \textit{Inline let-block notation.}

\textbf{Description}

Inline version of \texttt{let}-block.

\textbf{Usage}

\begin{verbatim}
a %in_block% b
\end{verbatim}

\textbf{Arguments}

\begin{itemize}
\item \texttt{a} (left argument) named character vector with target names as names, and replacement names as values.
\item \texttt{b} (right argument) expression or block to evaluate under let substitution rules.
\end{itemize}

\textbf{Value}

evaluated block.

\textbf{See Also}

\texttt{let}

\textbf{Examples}

\begin{verbatim}
d <- data.frame(
    Sepal_Length=c(5.8,5.7),
    Sepal_Width=c(4.0,4.4),
    Species='setosa')
\end{verbatim}
# let-block notation
let(
  qc(
    AREA_COL = Sepal_area,
    LENGTH_COL = Sepal_Length,
    WIDTH_COL = Sepal_Width
  ),
  d .>%
  transform(., AREA_COL = LENGTH_COL * WIDTH_COL)
)

# %in_block% notation
qc(
  AREA_COL = Sepal_area,
  LENGTH_COL = Sepal_Length,
  WIDTH_COL = Sepal_Width
) %in_block% {
  d .>%
  transform(., AREA_COL = LENGTH_COL * WIDTH_COL)
}

# Note: in packages can make assignment such as:
# AREA_COL <- LENGTH_COL <- WIDTH_COL <- NULL
# prior to code so targets don't look like unbound names.

---

%<s% Dot substitution string interpolation.

Description


Usage

str %<s% envir

Arguments

str character string to be substituted into
envir environment to look for values

Details

Value
modified strings

See Also
strsplit_capture, si

Examples

"x is .(x)" %s> list(x = 7)
Description
Inline list/array concatenate.

Usage
```
e1 %c% e2
```

Arguments
- **e1**: first, or left argument.
- **e2**: second, or right argument.

Value
```
c(e1, c2)
```

Examples
```
1:2 %c% 5:6
```
```
c("a", "b") %c% "d"
```

---

Description
Inline dot product.

Usage
```
e1 %dot% e2
```

Arguments
- **e1**: first, or left argument.
- **e2**: second, or right argument.

Value
```
c(e1, c2)
```
Examples

\texttt{c(1,2) \%dot\% c(3, 5)}

\texttt{\%p\%}

\textit{Inline character paste0.}

Description

Inline character paste0.

Usage

\texttt{e1 \%p\% e2}

Arguments

\begin{itemize}
  \item \texttt{e1} first, or left argument.
  \item \texttt{e2} second, or right argument.
\end{itemize}

Value

\texttt{c(e1, c2)}

Examples

\texttt{\"a\" \%p\% \"b\"}
\texttt{c("a", \"b\") \%p\% \"_d\"}

\texttt{\%qc\%}

\textit{Inline quoting list/array concatenate.}

Description

Inline quoting list/array concatenate.

Usage

\texttt{e1 \%qc\% e2}
Arguments

e1  first, or left argument.
e2  second, or right argument.

Value

\texttt{qc(e1, c2)}

Examples

\begin{verbatim}
1:2 %qc% 5:6
c(“a”, “b”) %qc% d
a %qc% b %qc% c
\end{verbatim}
Index

:= (named_map_builder), 48
[.] Unpacker, 76
[<-. Unpacker, 77
%>%(dot_arrow), 29
%%(dot_arrow), 29
%:=(named_map_builder), 48
%>%(dot_arrow), 29
%>(coalesce), 20
%<%(79
%c%, 81
%dot%, 81
%in_%block%, 78
%p%, 82
%qc%, 82
%s>%, 80

add_name_column, 3
apply_left, 4, 6–9, 11
apply_left.default, 5, 6
apply_left.default, 7
apply_right, 8, 9, 11
apply_right.default, 9
apply_right_S4, 8, 9, 10
as_named_list, 11

bquote, 41, 57
bquote_call_args, 13, 14
bquote_function, 13, 14
build_frame, 16, 31, 58
buildNameCallback, 15

c, 61
capture.output, 44
check_equiv_frames, 17
checkColsFormUniqueKeys, 17
clean_fit_glm, 18
clean_fit_lm, 19
coalesce, 20

debugFn, 22, 22, 23–25, 27, 28
DebugFnE, 22, 23, 24, 25, 27, 28
DebugFnW, 15, 22–24, 24, 25, 27, 28
DebugFnWE, 22–25, 25, 27, 28
DebugPrintFn, 22–25, 26, 27, 28
DebugPrintFnE, 22–25, 27, 27, 28
defineLambda, 28, 34, 38, 42, 49
do.call, 41, 61
dot_arrow, 29
dput, 44
draw_frame, 16, 30, 58
draw_framec, 31
dump.frames, 22–25, 27, 28
evalb, 32
execute_parallel, 33, 53

f., 34
grep, 35–37
grepdf, 35, 37
grepy, 35, 36

has_no_dup_rows, 37
invert_perm, 37
isTRUE, 21

lambda, 28, 34, 38, 42, 49
lapplym, 39, 51, 73, 74
let, 40, 43, 44, 78
list, 61

makeFunction_se, 28, 34, 38, 42, 49
map_to_char, 43
map_upper, 44
mapsyms, 43

match_order, 45
mk_formula, 46
mk_tmp_name_source, 47

named_map_builder, 28, 34, 38, 42, 48

84
INDEX

order, 49, 50, 66
orderv, 49, 66

pack, 50, 72
parLapplyLB, 51
parLapplyLBm, 39, 51
partition_tables, 33, 52
pipe_impl, 53
psagg, 54

qae, 56, 57, 59
qc, 56, 57, 59
qchar_frame, 16, 31, 58
qe, 56, 57, 59
qs, 60
reduceexpand, 61
reformulate, 46, 47
restrictToNameAssignments, 62

seqi, 62
si, 63, 65, 69, 80
sinterp, 64, 65, 69
sort, 49, 66
sortv, 50, 66
split_at_brace_pairs, 67
stop_if_dot_args, 67
strsplit_capture, 64, 65, 68, 80

to, 69
uniques, 70
unpack, 50, 71
update.formula, 46, 47

vapplym, 39, 51, 73, 74
Vectorize, 74
VectorizeM, 39, 51, 73, 74
view, 75

wrapr, 75