Package 'wrapr'

June 13, 2018

Type Package

Title Wrap R Tools for Debugging and Parametric Programming

Version 1.5.0

Date 2018-06-13

URL https://github.com/WinVector/wrapr,
http://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:
'let()'
(converts non-standard evaluation interfaces to parametric standard
evaluation interfaces),
'%.>%' dot-pipe (an 'S3' configurable pipe),
'build_frame()'/'draw_frame()' ('data.frame' example tools),
'qc()' (quoting concatenate),
':=' (named map builder),
'DebugFnW()' (capture function context on error for debugging),
and more.

License GPL-3

Encoding UTF-8

LazyData true

RoxygenNote 6.0.1

Depends R (>= 3.2.1)

Imports utils

Suggests parallel, testthat, knitr, rmarkdown

VignetteBuilder knitr

ByteCompile true

NeedsCompilation no

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

Repository CRAN

Date/Publication 2018-06-13 13:02:55 UTC

R topics documented:

`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

For formal documentation please see https://github.com/WinVector/wrapr/blob/master/extras/wrapr_pipe.pdf.

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.

Value

result

See Also

apply_left.default

Examples

```
# collects from left to right
apply_left.character <- function(pipe_left_arg,
                                 pipe_right_arg,
                                 pipe_environment,
                                 left_arg_name,
                                 pipe_string,
                                 right_arg_name) {
  paste(pipe_left_arg, pipe_right_arg)
}

"a" %.>% 5 %.>% 7
```

`apply_left.default` *S3 dispatch on class of pipe_left_arg.*

Description

Place evalation of left argument in `.` and then evaluate right argument.

Usage

```
## 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

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

apply_right *S3 dispatch on class of pipe_right_argument.*

Description

Triggered if right hand side 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	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_right.default

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.default` *S3 dispatch on type of pipe_right_argument.*

Description

Triggered if right hand side was a name that does not resolve to a function. Default implementation is re-dispatch through `apply_left`. Currently this is not thought to be a common execution case.

Usage

```
## Default S3 method:
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`	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`, `apply_right`

Examples

```
v <- list(1, 2)
f <- function(z) { format(z) }
f %.>% v
```

buildNameCallback *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

```
buildNameCallback(varName)
```

Arguments

varName	character where to write captured state

Value

writeback function for use with functions such as DebugFnW

Examples

```
# 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 (non-empty) data.frame.*

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

```
build_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

```
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))

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

`coalesce` *Coalesce values (NULL/NA on left replaced by values on the right).*

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 conditions 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 coalesce_left_arg or length 1. In this case replace NA/NULL elements of coalesce_left_arg with 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)
# list() %?% list(1, NA, NULL) # throws an error.
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

```
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(rfn,rargs) repeats the call to fn with args.

See Also

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

Examples

```
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(rfn,rargs) repeats the call to fn with args.

See Also

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

Examples

```
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)


f <- function(i) { (1:10)[[i]] }
curEnv <- environment()
writeBack <- function(sit) {
   assign('lastError', sit, envir=curEnv)
}
attr(writeBack,'name') <- 'writeBack'
df <- DebugFnW(writeBack,f)
tryCatch(
   df(12),
   error = function(e) { print(e) })
str(lastError)
```

`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

```
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(rfn,rargs) repeats the call to fn with args.

See Also

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

Examples

```
saveDest <- paste0(tempfile('debug'),'.RDS')
f <- function(i) { (1:10)[[i]] }
# correct run
DebugPrintFn(saveDest, f, 5)
# now re-run
# capture error on incorrect run
tryCatch(
   DebugPrintFn(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)
```

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

```
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(rfn,rargs) repeats the call to fn with args.

See Also

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

Examples

```
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$args, envir=situation$env)
# 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

Examples

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

---

dot_arrow *Pipe operator ("dot arrow").*

Description

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

Usage

```
pipe_left_arg %.>% pipe_right_arg

pipe_left_arg %>.% pipe_right_arg
```

Arguments

pipe_left_arg	left argument expression (substituted into .)
pipe_right_arg	right argument expession (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 "pipled into nothing cases" such as `5 %.>% sin()`, many of these checks can be turned off by adding braces.

For some discussion, please see http://www.win-vector.com/blog/2017/07/in-praise-of-syntactic-sugar/. For some more examples, please see the package README https://github.com/WinVector/wrapr. For formal documentation please see https://github.com/WinVector/wrapr/blob/master/extras/wrapr_pipe.pdf. %>.% and %.>% are synonyms.

Value

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

Functions

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

Examples

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

---

draw_frame *Render a data.frame in build_frame format.*

Description

Render a data.frame in build_frame format.

Usage

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

Arguments

x	data.frame (atomic types, with at least 1 row and 1 column).
...	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.

Value

chracter

See Also

build_frame, qchar_frame

Examples

```
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))
```

`execute_parallel` *Execute f in parallel partition ed by partition_column.*

Description

Execute f in parallel partiation 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

```
if(requireNamespace("parallel", quietly = TRUE)) {
  cl <- parallel::makeCluster(2)

  d <- data.frame(x = 1:5, g = c(1, 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)
}
```

`grepdf` *Grep for column names from a* `data.frame`

Description

Grep for column names from a `data.frame`

Usage

```
grepdf(pattern, x, ..., 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.

Examples

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

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

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

`invert_perm` *Invert a permuation.*

Description

For a permutation p build q such that p[q] == q[p] == seq_len(length(p)). Please see http://www.win-vector.com/blog/2017/05/on-indexing-operators-and-composition/ and http://www.win-vector.com/blog/2017/09/permutation-theory-in-action/.

Usage

```
invert_perm(p)
```

Arguments

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

Value

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

Examples

```
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

```
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

```
#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 argumnet
f <- lambda(x, y, x+y)
f(2,4)

# brace interface syntax
f <- x := { x^2 }
f(5)

# formula interface syntax: [~arg|arg(~arg)+] := { body }
f <- x~y := { x + 3 * y }
f(5, 47)
```

`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

```
let(alias, expr, ..., envir = parent.frame(), subsMethod = "langsubs",
  strict = TRUE, eval = TRUE, debugPrint = FALSE)
```

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.

Value

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

Examples

```
d <- data.frame(Sepal_Length=c(5.8,5.7),
                Sepal_Width=c(4.0,4.4),
                Species='setosa',
                rank=c(1,2))

RANKCOLUMN <- NULL # optional, make sure macro target does not look like unbound variable.
GROUPCOLUMN <- NULL # optional, make sure macro target does not look like unbound variable.
mapping = c(RANKCOLUMN= 'rank', GROUPCOLUMN= 'Species')
let(alias = mapping,
    expr = {
       # Notice code here can be written in terms of known or concrete
       # names "RANKCOLUMN" and "GROUPCOLUMN", but executes as if we
       # had written mapping specified columns "rank" and "Species".

       # restart ranks at zero.
       dres <- d
       dres$RANKCOLUMN <- dres$RANKCOLUMN - 1 # notice, using $ not [[]]

       # confirm set of groups.
       groups <- unique(d$GROUPCOLUMN)
    },
    debugPrint = TRUE
    )
print(groups)
print(length(groups))
print(dres)
```

---

makeFunction_se *Build an anonymous function.*

Description

Developed from: http://www.win-vector.com/blog/2016/12/the-case-for-using-in-r/comment-page-1/#comment-66399, https://github.com/klmr/functional#a-concise-lambda-syntax, https://github.com/klmr/functional/blob/master/lambda.r Called from := operator.

Usage

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

Arguments

params	formal parameters of function, unbound names.
body	subsituted body of function to map arguments into (braces required for ":=" notation).
env	environment to work in.

Value

user defined function.

See Also

lambda, defineLambda, named_map_builder

Examples

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

f <- x := { x*x }
f(7)

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

g <- c(x,y) := { 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

```
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)))
```

`map_upper` *Map up-cased symbol names to referenced values if those values are string scalars (else throw).*

Description

Map up-cased 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`

Examples

```
x <- 'a'
print(map_upper(x))
d <- data.frame(a = "a_val")
let(map_upper(x), paste(d$X, x))
```

`match_order` *Match one order to another.*

Description

Build a pemutation p such that ids1[p] == ids2. See http://www.win-vector.com/blog/2017/09/permutation-theory-in-action/.

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

```
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
```

`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

```
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

```
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

```
named_map_builder(names, values)

":="(names, values)

names %:=% values
```

Arguments

names	names to set.
values	values to assign names to (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')

# fn version:
#  applied when right side is {}
#  or when left side is of class formula.

g <- x~y := { x + 3*y }
g(1,100)

f <- ~x := x^2
f(7)

f <- x := { sqrt(x) }
f(7)
```

`partition_tables` *Partition as set of talbes into a list.*

Description

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

Usage

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

Arguments

`tables_used`	charater, 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

```
d <- data.frame(a = 1:5, g = c(1, 1, 2, 2, 2))
partition_tables("d", "g", tables = list(d = d))
```

---

pipe_step *pipe_step*

Description

Out of date, please use apply_left instead.

Usage

```
pipe_step(pipe_left_arg, pipe_right_arg, pipe_environment, pipe_name = NULL)
```

Arguments

pipe_left_arg
: left argument.

pipe_right_arg
: substitute(pipe_right_arg) argument.

pipe_environment
: environment to evaluate in.

pipe_name
: character, name of pipe operator.

Value

result

pipe_step.default *pipe_step*

Description

Out of date, please use `apply_left` instead.

Usage

```
## Default S3 method:
pipe_step(pipe_left_arg, pipe_right_arg, pipe_environment,
  pipe_name = NULL)
```

Arguments

`pipe_left_arg`	left argument.
`pipe_right_arg`	substitute(pipe_right_arg) argument.
`pipe_environment`	environment to evaluate in.
`pipe_name`	character, name of pipe operator.

Value

result

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.

Value

array of quoted assignment expressions.

See Also

qc, qe

Examples

```
exprs <- qae(Sepal_Long := Sepal.Length >= ratio * Sepal.Width,
             Petal_Short = Petal.Length <= 3.5)
print(exprs)
#ratio <- 2
#datasets::iris %.>%
#  seplyr::mutate_se(., exprs) %.>%
#  summary(.)
```

---

qc *Quoting version of c() array concatinator.*

Description

The qc() function is intended to help quote user inputs. It is 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() itself. Please see the examples for typical uses both for named and un-named character vectors.

Usage

```
qc(...)
```

Arguments

...	items to place into an array

Value

quoted array of character items

See Also

qe, qae

Examples

```
a <- "x"
qc(a) # returns the string "a" (not "x")

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

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

---

qchar_frame *Build a (non-empty) 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 infic 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.

Value

character data.frame

See Also

draw_frame, build_frame

Examples

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

qchar_frame(
  x |
  1 |
  2 )
```

qe *Quote expressions.*

Description

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

Usage

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

Arguments

...	assignment expressions.

Value

array of quoted assignment expressions.

See Also

qc, qae

Examples

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

qs *Quote a string.*

Description

Quote a string.

Usage

```
qs(s)
```

Arguments

s	expression to be quoted as a string.

Value

character

Examples

```
qs(a == "x")
```

`reduceexpand` *Use function to reduce or expand arguments.*

Description

The operators `%.|%` and `%|.%` are wrappers for `do.call`. These functions are used to pass arguments from a list to variadic function (such as `sum`). The operator symbols are meant to invoke non-tilted versions of APL's reduce and expand operators. Unevaluated expressions containing `%.|%`, `%|.%`, or `do.call` can be used simulate partial function application or simulate function Currying. The take-away is one can delegate all variadic argument construction to `list`, and manipulation to `c`.

Usage

```
f %|.% args

args %.|% f
```

Arguments

`f`	function.
`args`	argument list or vector, entries expanded as function arguments.

Value

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

Functions

- `%|.%`: f reduce args
- `%.|%`: args expand f

See Also

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

Examples

```
# basic examples
1:10 %.|% sum
1:10 %.|% base::sum
1:10 %.|% function(...) { sum(...) }

# simulate partial application of log(., base=2)
1:4 %.>% do.call(log, list(., base = 2))

# # simluate partial application with dplyr
# # can be used with dplyr/rlang as follows
# d <- data.frame(x=1, y=2, z=3)
# syms <- rlang::syms(c("x", "y"))
# d %.>% do.call(dplyr::select, c(list(.), syms))
```

`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

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

Arguments

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

Value

string to string mapping

Examples

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

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

```
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

```
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) }
)
```

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

Description

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

Usage

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

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, quitely 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
```

---

view *Invoke a spreadsheet like viewer when appropriate.*

Description

Invoke a spreadsheet like viewer when appropriate.

Usage

```
view(x, ..., title = paste(deparse(substitute(x)), collapse = " "), 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)
```

`wrapr` *wrapr: Wrap R Functions for Debugging and Parametric Programming*

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.

`wrapr_function` *wrapr_function*

Description

Out of date, please use `apply_right` instead.

Usage

```
wrapr_function(pipe_left_arg, pipe_right_arg, pipe_environment,
  pipe_name = NULL)
```

Arguments

pipe_left_arg	left argument.
pipe_right_arg	right argument (general object, not a function).
pipe_environment	environment to evaluate in.
pipe_name	character, name of pipe operator.

Value

result

---

wrapr_function.default *wrapr_function*

Description

Out of date, please use apply_right instead.

Usage

```
## Default S3 method:
wrapr_function(pipe_left_arg, pipe_right_arg,
  pipe_environment, pipe_name = NULL)
```

Arguments

pipe_left_arg	left argument.
pipe_right_arg	substitute(pipe_right_arg) argument.
pipe_environment	environment to evaluate in.
pipe_name	character, name of pipe operator.

Value

result

Index