Package ‘wrapr’

July 24, 2019

Type Package
Title Wrap R Tools for Debugging and Parametric Programming
Version 1.8.9
Date 2019-07-24
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),
'let()' (converts non-standard evaluation interfaces to parametric standard
evaluation interfaces, inspired by 'gtools:strmacro()' and 'base::bquote()'),
'build_frame()'/draw_frame()' ('data.frame' example tools),
'qc()' (quoting concatenate),
'':=' (named map builder),
and more.
License GPL-2 | GPL-3
Encoding UTF-8
LazyData true
RoxygenNote 6.1.1
Depends R (>= 3.2.1)
Imports utils, methods, graphics, stats
Suggests parallel, knitr, rmarkdown, RUnit, R.rsp
VignetteBuilder knitr, R.rsp
ByteCompile true
NeedsCompilation no
Author John Mount [aut, cre],
Nina Zumel [aut],
Win-Vector LLC [cph]
R topics documented:

add_name_column .................................................. 2
ApplyTo ............................................................... 2
apply_left ............................................................ 4
apply_left.default .................................................. 5
apply_left_default ................................................... 6
apply_right ........................................................... 7
apply_right.default .................................................. 8
apply_right_S4 ....................................................... 10
as.list.UnaryFn ..................................................... 11
as.list.UnaryFnList ................................................ 12
as.UnaryFn ........................................................... 12
as_dot_fn ............................................................. 13
as_fn  ................................................................. 14
as_fnlist ............................................................... 14
bquote_call ............................................................ 15
bquote_function ...................................................... 16
buildNameCallback .................................................. 17
build_frame ........................................................... 18
c.UnaryFn ............................................................. 19
checkColsFormUniqueKeys ......................................... 19
check_equiv_frames .................................................. 20
clean_fit_glm ......................................................... 20
clean_fit_lm ........................................................... 22
coalesce ............................................................... 23
Collector ............................................................... 24
DebugFn ................................................................. 25
DebugFnE ............................................................... 26
DebugFnW ............................................................... 27
DebugFnWE ............................................................. 28
DebugPrintFn ......................................................... 29
DebugPrintFnE ....................................................... 30
defineLambda .......................................................... 31
dot_arrow ............................................................... 32
draw_frame ............................................................ 33
draw_framec ........................................................... 34
evalb ................................................................. 35
execute_parallel ..................................................... 35
fnlist ................................................................. 37
format.PartialFunction ............................................. 37
format.PartialNamedFn ............................................. 38
format.SrcFunction ................................................ 38
format.UnaryFn ...................................................... 39
R topics documented:

grepdf ................................................................. 39
grepv ................................................................. 40
has_no_dup_rows ..................................................... 41
invert_perm ........................................................... 41
lambda ................................................................. 42
lapplym ............................................................... 43
let ................................................................. 44
makeFunction_se ..................................................... 45
mapsyms ............................................................. 47
map_to_char .......................................................... 47
map_upper ............................................................ 48
match_order ........................................................... 49
mk_formula ............................................................ 50
mk_tmp_name_source .................................................. 51
named_map_builder .................................................. 52
orderv ............................................................... 53
parLapplyLBm ........................................................ 54
PartialFunction-class ............................................... 55
PartialNamedFn-class ............................................... 55
partition_tables ..................................................... 55
pkgfn ............................................................... 56
psagg ............................................................... 57
qae ............................................................... 58
qe ............................................................... 59
qchar_frame .......................................................... 60
qe ............................................................... 61
qs ............................................................... 62
reduceexpand .......................................................... 63
restrictToNameAssignments ........................................ 64
run_package_tests .................................................... 65
run_wrapr_tests ....................................................... 66
seqi ............................................................... 67
sequence_as_function ................................................ 67
show,PartialFunction-method ....................................... 68
show,PartialNamedFn-method ....................................... 69
show,SrcFunction-method ........................................... 69
show,UnaryFnList-method .......................................... 70
sinterp .............................................................. 70
sortv ............................................................... 71
split_at_brace_pairs ................................................ 72
srcfn .............................................................. 73
SrcFunction-class .................................................... 73
stop_if_dot_args ...................................................... 74
strsplit_capture ...................................................... 74
UnaryFn-class ......................................................... 75
UnaryFnList-class ..................................................... 75
uniques ............................................................. 76
vapplym .............................................................. 77
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')
ApplyTo

Apply a single argument function to its argument.

Description

If x is a UnaryFn instance this function returns a new UnaryFnList representing the composite function c(f, x) which is interpreted as the function x(f(.)) (composition from left to right). Otherwise evaluate f(x) (application from left to right).

Usage

```r
ApplyTo(f, x, env = parent.frame())
```

## S4 method for signature 'UnaryFnList,UnaryFnList'

```r
ApplyTo(f, x, env = parent.frame())
```

## S4 method for signature 'UnaryFnList,UnaryFn'

```r
ApplyTo(f, x, env = parent.frame())
```

## S4 method for signature 'UnaryFn,UnaryFnList'

```r
ApplyTo(f, x, env = parent.frame())
```

## S4 method for signature 'UnaryFn,UnaryFn'

```r
ApplyTo(f, x, env = parent.frame())
```

## S4 method for signature 'UnaryFnList,ANY'

```r
ApplyTo(f, x, env = parent.frame())
```

## S4 method for signature 'PartialNamedFn,ANY'

```r
ApplyTo(f, x, env = parent.frame())
```

## S4 method for signature 'PartialNamedFn,UnaryFnList'

```r
ApplyTo(f, x, 
    env = parent.frame())
```

## S4 method for signature 'PartialNamedFn,UnaryFn'

```r
ApplyTo(f, x, env = parent.frame())
```

## S4 method for signature 'PartialFunction,ANY'

```r
ApplyTo(f, x, env = parent.frame())
```

## S4 method for signature 'PartialFunction,UnaryFnList'

```r
ApplyTo(f, x, 
    env = parent.frame())
```

## S4 method for signature 'PartialFunction,UnaryFn'

```r
ApplyTo(f, x, env = parent.frame())
```
apply_left

## S4 method for signature 'SrcFunction,ANY'
ApplyTo(f, x, env = parent.frame())

## S4 method for signature 'SrcFunction,UnaryFnList'
ApplyTo(f, x, env = parent.frame())

## S4 method for signature 'SrcFunction,UnaryFn'
ApplyTo(f, x, env = parent.frame())

Arguments

- **f** object of S4 class derived from UnaryFn.
- **x** argument.
- **env** environment to work in.

Value

f(x) if x is not a UnaryFn else f composed with x.

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.
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)
  }
}

setMethod(
  wrapr::apply_right_S4,
  signature = c(pipe_left_arg = "character", pipe_right_arg = "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

"a" %>% toupper()

q <- "z"
"a" %>% q
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(.)
```
**Description**

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

**Usage**

```r
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**

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

apply_right.default

Default apply_right implementation.

Description

Default apply_right implementation: S4 dispatch to apply_right_S4.

Usage

```r
## 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` : 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

```r
# 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

```r
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.

### Value

result
as.list.UnaryFn

See Also

apply_left, apply_right

Examples

```r
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.list.UnaryFn  

Get list of primative unary fns.

Description

Get list of primative unary fns.

Usage

```r
## S3 method for class 'UnaryFn'
as.list(x, ...)
```

Arguments

- `x`: UnaryFn derived classe to extract
- `...`: not used.

Value

list of non UnaryFnList functions
Examples

```r
as.list(pkgfn("base::sin", "x"))
as.list(c(pkgfn("base::sin", "x"), pkgfn("base::cos", "x")))
```

Description
Get list of primitive unary fns.

Usage

```r
## S3 method for class 'UnaryFnList'
as.list(x, ...)
```

Arguments

- `x` UnaryFn derived class to extract
- `...` not used.

Value
list of non UnaryFnList functions

Examples

```r
as.list(pkgfn("base::sin", "x"))
as.list(c(pkgfn("base::sin", "x"), pkgfn("base::cos", "x")))
```

Description
Convert a list of UnaryFns into a UnaryFn.

Usage

```r
as.UnaryFn(items, env = parent.frame())
```
as_dot_fn

Arguments

items list of UnaryFn derived instances.
env environment to work in.

Value

UnaryFnList

See Also

pkgfn, wrapfn, srcfn

Examples

f <- as.UnaryFn(list(pkgfn("base::sin", "x"), pkgfn("base::cos", "x")))
cat(format(f))
1:3 %.>% f

as_dot_fn

Convert an unevaluated pipeline into a function.

Description

Convert an unevaluated pipeline into a function of ".".

Usage

as_dot_fn(pipeline, env = parent.frame())

Arguments

pipeline a un-evaluated wrapr pipeline.
env environment to work in.

Details

Note: writes "." into env.

Value

single function with signature (., env = parent.frame())
Examples

\[ f \leftarrow \text{as\_dot\_fn}(\sin(.) \ %>% \ \cos(.)) \]
\[ f(1:3) \]

\[ g \leftarrow \text{as\_dot\_fn}(\ . \ %>% \ \sin(.) \ %>% \ \cos(.)) \]
\[ g(1:3) \]

\[
\text{as\_fn} \quad \text{Convert a pipeable object into a function.}
\]

Description

Convert a pipeable object into a function of ".".

Usage

\[
\text{as\_fn}(\text{pipeable}, \text{env} = \text{parent.frame()} )
\]

Arguments

\[
\begin{align*}
\text{pipeable} & \quad \text{a \textit{wrapr} dot-pipe pipeable object} \\
\text{env} & \quad \text{environment to work in.}
\end{align*}
\]

Details

Note: writes "." into env.

Value

single function with signature (., env = parent.frame())

Examples

\[
\begin{align*}
p & \leftarrow \text{pkgfn}(\text{"base::sin"}, \ "x") \\
f & \leftarrow \text{as\_fn}(p) \\
f(5)
\end{align*}
\]
**as_fnlist**

*Wrap a list of UnaryFns as a UnaryFnList.*

**Description**

Unary functions are evaluated in left to right or first to last order.

**Usage**

```
as_fnlist(items, env = parent.frame())
```

**Arguments**

- `items`: list ofUnaryFn derived instances.
- `env`: environment to work in.

**Value**

UnaryFnList

**See Also**

pkgfn, wrapfn, srcfn

**Examples**

```
f <- as_fnlist(list(pkgfn("base::sin", "x"), pkgfn("base::cos", "x")))
cat(format(f))
1:3 %>% f
```

---

**bquote_call**

*Treat ... call argument as bquoted-values.*

**Description**

bquote_call re-writes calls.

**Usage**

```
bquote_call(call, env = parent.frame())
```

**Arguments**

- `call`: result of match.call()
- `env`: environment to perform lookups in.
**bquote_function**

**Details**
Note: eagerly evaluates argument and writes them into the function’s executing environment.

**Value**
altered call

**See Also**
bquote_function, bquote_call_args

---

**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.

**Usage**
bquote_function(fn)

**Arguments**

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

**Value**
new function.

**See Also**
bquote_call_args

**Examples**

```r
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)
```
**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

```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

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 )
c.UnaryFn  

Combine UnaryFns

Description
Combine UnaryFns

Usage
```r
## S3 method for class 'UnaryFn'
c(...)
```

Arguments
- `...` UnaryFn derived classes to combine

Value
UnaryFn representing the sequence

Examples
```r
c(pkgfn("base::sin", "x"), pkgfn("base::cos", "x"))
```

checkColsFormUniqueKeys  

Check that a set of columns form unique keys.

Description
For local data.frame only.

Usage
```r
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)

Arguments

d1        data.frame 1

d2        data.frame 2

Value

logical TRUE if equivalent

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

Description


Usage

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

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

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$glm$gmodel, NULL))
res(glm) <- clean_fit_glm("yC", c("x1", "x2"),
mk_data_example(10000),
  family = binomial)
length(serialize(res$glm$gmodel, NULL))
predict(res$glm$gmodel,
  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())
```

**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))
```
coalesce

```r
predict(res_lm$model,
    newdata = mk_data_example(1))
```

---

**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**

```r
coalesce(coalesce_left_arg, coalesce_right_arg)
```

```r
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)
c() %?% list(1, NA, NULL) # returns list(1, NA, NULL)
c() %?% c(1, NA, 2) # returns c(1, NA, 2)

Collector

Build a collector that can capture all pipe stages to the right.

Description

Build a collector that can capture all pipe stages to the right, using bquote()-(). escaping.

Usage

Collector()

Value

a Collector list-object.

Examples

phase <- 0.1
Collector() %>% sin(.) %>% cos(. + .(phase))
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(r$fn,r$args) 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)
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

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/c9c327d10a91df9a3370d30d0ac3d . 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

```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**

```r
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`. 
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
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

```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`. 
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

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

---

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

---

**Description**

Defined as roughly:  
\[ a \%>.% b \sim \{ . \leftarrow a; b \} \]  
(with visible .-side effects).

**Usage**

- `pipe_left_arg %>.% pipe_right_arg`
- `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 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.

For some more examples, please see the package README[https://github.com/WinVector/wrapr](https://github.com/WinVector/wrapr).  
%>.% and %.>% are synonyms.

**Value**

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

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

Examples

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

---

draw_frame

Render a simple data.frame in build_frame format.

Description

Render a simple data.frame in build_frame format.

Usage

```r
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

Value

character

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))

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

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

**evalb**

*eval(bquote(expr)) shortcut.*

**Description**

`eval(bquote(expr))` shortcut.

**Usage**

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

**Arguments**

- `...` expression to evaluate (one argument).
- `where` environment to work in.

**Value**

`eval(bquote(expr))`

**Examples**

```r
angle = 1:10
variable <- as.name("angle")
evalb(plot(x = .(variable), y = sin(.(variable))))
```

### execute_parallel

**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**

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

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tables</td>
<td>named map of tables to use.</td>
</tr>
<tr>
<td>f</td>
<td>function to apply to each tableset signature is function takes a single argument that is a named list of data.frames.</td>
</tr>
<tr>
<td>partition_column</td>
<td>character name of column to partition on</td>
</tr>
<tr>
<td>...</td>
<td>force later arguments to bind by name.</td>
</tr>
<tr>
<td>cl</td>
<td>parallel cluster.</td>
</tr>
<tr>
<td>debug</td>
<td>logical if TRUE use lapply instead of parallel::clusterApplyLB.</td>
</tr>
<tr>
<td>env</td>
<td>environment to look for values in.</td>
</tr>
</tbody>
</table>

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, 1, 2, 2 ,2))
  f <- function(dl) {
    d <- d$dl
    d$s <- sqrt(d$x)
    d
  }
  r <- execute_parallel(list(d = d), f,
                        partition_column = "g",
                        cl = cl) %>%
    do.call(rbind, .) %>%
    print(.)

  parallel::stopCluster(cl)
}
```
fnlist

Wrap a list of functions as a function.

Description

Unary functions are evaluated in left to right or first to last order.

Usage

fnlist(...)  

Arguments

...  

UnaryFn derived instances.

Value

UnaryFnList

See Also

pkgfn, wrapfn, srcfn

Examples

f <- fnlist(pkgfn("base::sin", "x"), pkgfn("base::cos", "x"))
cat(format(f))
1:3 %>% f

format.PartialFunction

format step

Description

format step

Usage

## S3 method for class 'PartialFunction'
format(x, ...)


Arguments

- `x` object to format
- `...` additional arguments (not used)

Value

character

format.PartialNamedFn

format step

Description

format step

Usage

```r
# S3 method for class 'PartialNamedFn'
format(x, ...)
```

Arguments

- `x` object to format
- `...` additional arguments (not used)

Value

character

format.SrcFunction

format step

Description

format step

Usage

```r
# S3 method for class 'SrcFunction'
format(x, ...)
```

Arguments

- `x` object to format
- `...` additional arguments (not used)
Value
character

---

format.UnaryFnList  format step

Description
format step

Usage
```r
## S3 method for class 'UnaryFnList'
format(x, ...)
```

Arguments
- `x`: object to format
- `...`: additional arguments (not used)

Value
character

---

grepdf  Grep for column names from a data.frame

Description
Grep for column names from a data.frame

Usage
```r
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.

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

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

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

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

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-syntx: 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
\$f <- \text{lambda}(x, y, x + y)\\ f(2, 4)\$

\# brace interface syntax
\$f <- x := (x^2)\\ f(5)\$

\# formula interface syntax: \([-\text{arg} | \text{arg}(-\text{arg})]+ := \{ \text{body} \}\\ f <- x - y := (x + 3 \times y)\\ f(5, 47)\$

\text{lapplym} \hspace{1cm} \text{Memoizing wrapper for lapply.}

**Description**

Memoizing wrapper for lapply.

**Usage**

\text{lapplym}(X, FUN, \ldots)

**Arguments**

- **X**: list or vector of inputs
- **FUN**: function to apply
- **\ldots**: additional arguments passed to lapply

**Value**

list of results.

**See Also**

\text{VectorizeM, vapplym, parLapplyLBm}

**Examples**

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

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

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

Description


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 (braces required for ":=" notation).
- 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)

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

mapsym

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

mapsym(...)

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
- ...: not used, force later arguments to bind by name.
- 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
**match_order**

*Match one order to another.*

**Description**


**Usage**

```r
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
```
Construct a formula.

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

Usage

```r
mk_formula(outcome, variables, ..., intercept = TRUE,
   outcome_target = NULL, outcome_comparator = "==", env = baseenv(),
   extra_values = NULL)
```

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.

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` function 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)
```

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(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

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

---

orderv  

Order by a list of vectors.

Description

Preduce an ordering permutation from a list of vectors. Essentially a non-... interface to \texttt{order}.

Usage

\begin{verbatim}
orderv(columns, ..., na.last = TRUE, decreasing = FALSE,  
method = c("auto", "shell", "radix"))
\end{verbatim}

Arguments

\begin{itemize}
\item \texttt{columns} list of atomic columns to order on, can be a \texttt{data.frame}.
\item \ldots not used, force later arguments to bind by name.
\item \texttt{na.last} (passed to \texttt{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.
\item \texttt{decreasing} (passed to \texttt{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 \ldots For the other methods, it must be length one.
\item \texttt{method} (passed to \texttt{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 \texttt{sort}.
\end{itemize}
Value

ordering permutation

See Also

order, sortv

Examples

d <- data.frame(x = c(2, 2, 3, 3, 1, 1), y = 6:1)
d[order(d$x, d$y), , drop = FALSE]
d[orderv(d), , drop = FALSE]
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)
}
```

PartialFunction-class

*Function with partial arguments as a new single argument function.*

Description

Function with partial arguments as a new single argument function.

PartialNamedFn-class

*Package qualified name of a function as a function.*

Description

Package qualified name of a function as a function.

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).
- `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

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))
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

```r
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

```r
qae(...)
```

Arguments

`...`  
assignment expressions.

Details

`qae()` uses `bquote() .()` quasiquotation escaping notation.

Value

array of quoted assignment expressions.

See Also

`qc`, `qe`
qc

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(.)

qc

Quoting version of c() array concatenate.

Description

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

.nowrapr_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

gq, qae, bquote
Examples

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

```r
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.
Values

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.

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))
```
**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**

```r
f %|% args
```

```r
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**

```r
# 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))

# # simulate 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**

```r
alias <- list(region= 'east', str= "'seven'")
aliasR <- restrictToNameAssignments(alias)
print(aliasR)
```
run_package_tests  Run package tests.

Description

For all files with names of the form "^test_.+\.R$" in the package directory unit_tests run all functions with names of the form "^test_.+$" as RUnit tests. Attaches RUnit and pkg, requires RUnit. Stops on error.

Usage

```r
run_package_tests(pkg, ..., verbose = TRUE,
    package_test_dirs = "unit_tests", test_dirs = character(0),
    stop_on_issue = TRUE, stop_if_no_tests = TRUE,
    require_RUnit_attached = FALSE, require_pkg_attached = TRUE,
    rngKind = "Mersenne-Twister", rngNormalKind = "Inversion")
```

Arguments

- `pkg` character, name of package to test.
- `...` not used, force later arguments to bind by name.
- `verbose` logical, if TRUE print more.
- `package_test_dirs` directory names to look for in the installed package.
- `test_dirs` paths to look for tests in.
- `stop_on_issue` logical, if TRUE stop after errors or failures.
- `stop_if_no_tests` logical, if TRUE stop if no tests were found.
- `require_RUnit_attached` logical, if TRUE require RUnit be attached before testing.
- `require_pkg_attached` logical, if TRUE require pkg be attached before testing.
- `rngKind` pseudo-random number generator method name.
- `rngNormalKind` pseudo-random normal generator method name.

Details

Based on https://github.com/RcppCore/Rcpp/blob/master/tests/doRUnit. R. This version is GPL-3, works derived from it must be distributed GPL-3.

Value

RUnit test results (invisible).
run_wrapr_tests  Run wrapr package tests.

Description

Run the tests included with the wrapr package (assumes wrapr attached).

Usage

run_wrapr_tests(..., verbose = TRUE, package_test_dirs = "unit_tests",
  test_dirs = character(0), stop_on_issue = TRUE,
  stop_if_no_tests = TRUE, require_RUnit_attached = FALSE,
  require_pkg_attached = TRUE, rngKind = "Mersenne-Twister",
  rngNormalKind = "Inversion")

Arguments

... not used, force later arguments to bind by name.
verbose logical, if TRUE print more.
package_test_dirs directory names to look for in the installed package.
test_dirs paths to look for tests in.
stop_on_issue logical, if TRUE stop after errors or failures.
stop_if_no_tests logical, if TRUE stop if no tests were found.
require_RUnit_attached logical, if TRUE require RUnit be attached before testing.
require_pkg_attached logical, if TRUE require pkg be attached before testing.
rngKind pseudo-random number generator method name.
rngNormalKind pseudo-random normal generator method name.

Value

RUnit test results (invisible).
**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**

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

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

---

**sequence_as_function**  
*Convert a sequence of expressions into a function.*

**Description**

Convert a sequence of expressions into a function.

**Usage**

```r
sequence_as_function(dot_seq, env = parent.frame())
```
Arguments

- `dot_seq` list of expressions.
- `env` environment to work in.

Details

Note: not for steps that intend side-effects or have references to items in non-standard environments.

Value

function with signature `.eval_environment = parent.frame()`

Examples

```r
seq <- Collector() %>% paste(., "a") %>% paste(., "b")
f <- sequence_as_function(seq)
f("x")
```

Description

S4 print method

Usage

```r
## S4 method for signature 'PartialFunction'
show(object)
```

Arguments

- `object` item to print
Description

S4 print method

Usage

```r
## S4 method for signature 'PartialNamedFn'
show(object)
```

Arguments

- `object`: item to print

Description

S4 print method

Usage

```r
## S4 method for signature 'SrcFunction'
show(object)
```

Arguments

- `object`: item to print
show,UnaryFnList-method

S4 print method

Description

S4 print method

Usage

## S4 method for signature 'UnaryFnList'
show(object)

Arguments

- **object**: item to print

sinterp

Dot substitution.

Description

String interpolation using `bquote`-style .() notation. Pure R, no C/C++ code called.

Usage

sinterp(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

See also [https://CRAN.R-project.org/package=R.utils](https://CRAN.R-project.org/package=R.utils), [https://CRAN.R-project.org/package=rprintf](https://CRAN.R-project.org/package=rprintf), and [https://CRAN.R-project.org/package=glue](https://CRAN.R-project.org/package=glue).
sortv

Value

modified strings

Examples

```r
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 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

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

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

split_at_brace_pairs("{x} + y + {z}")
**srcfn**

*Wrap the source for an expression as a function.*

**Description**

Wrap the source for an expression as a function.

**Usage**

`srcfn(expr_src, arg_name = ".", args = list())`

**Arguments**

- `expr_src`: character, source code of expression.
- `arg_name`: character, name of argument to assign.
- `args`: named list of additional arguments and values.

**Value**

SrcFunction

**See Also**

fnlist, pkgfn, wrapfn

**Examples**

```r
f <- srcfn(". + z", ",", args = list(z = 10))
cat(format(f))
1:3 %>% f
```

**SrcFunction-class**

*Code text as a new partial function.*

**Description**

Code text as a new partial function.
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) }
)

strsplit_capture  Split a string, keeping separator regions

Description

Split a string, keeping separator regions

Usage

strsplit_capture(x, split, ..., ignore.case = FALSE, fixed = FALSE, perl = FALSE, useBytes = FALSE)
UnaryFn-class

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.

Examples

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

UnaryFn-class

Functions that take a single argument

Description

Functions that take a single argument

UnaryFnList-class

List of Unary functions taken in order.

Description

Unary functions are evaluated in left to right or first to last order.
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)

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

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

# task: get unique items in x plus y
unique(c(x, y))  # correct answer
uniques(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
```
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

```r
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.

SIMPLOYEE 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

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))

view

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

```r
view(mtcars)
```

Description

Wrap the source for an expression as a function.

Usage

```r
wrapfn(fn, arg_name = ".", args = list())
```

Arguments

- **fn**: function.
- **arg_name**: character, name of argument to assign.
- **args**: named list of additional arguments and values.

Value

PartialFunction

See Also

pkgfn, fnlist, srcfn

Examples

```r
f <- wrapfn(sin, ".")
cat(format(f))
1:3 %>% f
```
wrapr

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=PLAKBwakacHbQT51nPHex1on3YNCCm

%in_block%

Inline let-block notation.

Description

Inline version of `let`-block.

Usage

```r
a %in_block% b
```

Arguments

- `a` (left argument) named character vector with target names as names, and replacement names as values.
- `b` (right argument) expression or block to evaluate under `let` substitution rules.

Value

evaluated block.
See Also

let

Examples

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

# 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.

---

**%c%**

*Inline list/array concatenate.*

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"

c(1,2) %dot% c(3, 5)
Inline character paste0.

Description

Inline character paste0.

Usage

e1 %p% e2

Arguments

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

Value

c(e1, c2)

Examples

"a" %p% "b"
c("a", "b") %p% "d"

Inline quoting list/array concatenate.

Description

Inline quoting list/array concatenate.

Usage

e1 %qc% e2

Arguments

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

Value

qc(e1, c2)
Examples

1:2 %qcc% 5:6

c("a", "b") %qcc% d

a %qcc% b %qcc% c