

# Package ‘wrapr’

May 24, 2019

**Type** Package

**Title** Wrap R Tools for Debugging and Parametric Programming

**Version** 1.8.7

**Date** 2019-05-23

**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:  
'%.>%' 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-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]

Repository CRAN

Date/Publication 2019-05-24 04:20:03 UTC

## R topics documented:

add_name_column . . . . .	4
ApplyTo . . . . .	5
apply_left . . . . .	6
apply_left.default . . . . .	7
apply_left_default . . . . .	8
apply_right . . . . .	9
apply_right.default . . . . .	10
apply_right_S4 . . . . .	11
as.list.UnaryFn . . . . .	12
as.list.UnaryFnList . . . . .	13
as.UnaryFn . . . . .	14
as_dot_fn . . . . .	14
as_fn . . . . .	15
as_fnlist . . . . .	16
bquote_call . . . . .	16
bquote_function . . . . .	17
buildNameCallback . . . . .	18
build_frame . . . . .	19
c.UnaryFn . . . . .	20
checkColsFormUniqueKeys . . . . .	20
check_equiv_frames . . . . .	21
clean_fit_glm . . . . .	21
clean_fit_lm . . . . .	23
coalesce . . . . .	24
Collector . . . . .	25
DebugFn . . . . .	26
DebugFnE . . . . .	27
DebugFnW . . . . .	28
DebugFnWE . . . . .	29
DebugPrintFn . . . . .	30
DebugPrintFnE . . . . .	31
defineLambda . . . . .	32
dot_arrow . . . . .	33
draw_frame . . . . .	34
draw_framec . . . . .	35
evalb . . . . .	36
execute_parallel . . . . .	36
fnlist . . . . .	38
format.PartialFunction . . . . .	38
format.PartialNamedFn . . . . .	39
format.SrcFunction . . . . .	39
format.UnaryFnList . . . . .	40

grepdf . . . . .	40
grepv . . . . .	41
has_no_dup_rows . . . . .	42
invert_perm . . . . .	42
lambda . . . . .	43
lapplym . . . . .	44
let . . . . .	45
makeFunction_se . . . . .	46
mapsyms . . . . .	47
map_to_char . . . . .	48
map_upper . . . . .	49
match_order . . . . .	50
mk_formula . . . . .	50
mk_tmp_name_source . . . . .	52
named_map_builder . . . . .	53
orderv . . . . .	54
parLapplyLBm . . . . .	55
PartialFunction-class . . . . .	56
PartialNamedFn-class . . . . .	56
partition_tables . . . . .	56
pkgfn . . . . .	57
psagg . . . . .	58
qae . . . . .	59
qc . . . . .	60
qchar_frame . . . . .	61
qe . . . . .	62
qs . . . . .	63
reduceexpand . . . . .	64
restrictToNameAssignments . . . . .	65
run_package_tests . . . . .	65
run_wrapp_tests . . . . .	66
seqi . . . . .	67
sequence_as_function . . . . .	68
show,PartialFunction-method . . . . .	69
show,PartialNamedFn-method . . . . .	69
show,SrcFunction-method . . . . .	70
show,UnaryFnList-method . . . . .	70
sinterp . . . . .	71
sortv . . . . .	72
split_at_brace_pairs . . . . .	73
srcfn . . . . .	73
SrcFunction-class . . . . .	74
stop_if_dot_args . . . . .	74
strsplit_capture . . . . .	75
UnaryFn-class . . . . .	76
UnaryFnList-class . . . . .	76
uniques . . . . .	76
vapplym . . . . .	77

VectorizeM . . . . .	78
view . . . . .	79
wrapfn . . . . .	79
wrapr . . . . .	80
%in_block% . . . . .	81
%c% . . . . .	82
%dot% . . . . .	83
%p% . . . . .	83
%qc% . . . . .	84

<b>Index</b>	<b>85</b>
--------------	-----------

---

add_name_column	<i>Add list name as a column to a list of data.frames.</i>
-----------------	--

---

### 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**

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

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

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

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

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

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

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

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

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

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

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

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

```
## 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.

---

apply_left	<i>S3 dispatch on class of pipe_left_arg.</i>
------------	---

---

### Description

For formal documentation please see [https://github.com/WinVector/wrapr/blob/master/extras/wrapr\\_pipe.pdf](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**

```

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**

```
## 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\_left\_default *S3 dispatch on class of pipe\_left\_arg.*

---

**Description**

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

**Usage**

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



**Arguments**

pipe\_left\_arg left argument  
 pipe\_right\_arg substitute(pipe\_right\_arg) argument  
 pipe\_environment  
                   environment to evaluate in  
 left\_arg\_name name, if not NULL name of left argument.  
 pipe\_string character, name of pipe operator.  
 right\_arg\_name name, if not NULL name of right argument.

**Value**

result

**See Also**

[apply\\_left](#)

**Examples**

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

---

apply_right	<i>S3 dispatch on class of pipe_right_argument.</i>
-------------	---

---

**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](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)
}

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

---

`apply_right.default` *Default apply\_right implementation.*

---

**Description**

Default `apply_right` implementation: S4 dispatch to `apply_right_S4`.

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

**Value**

result

**See Also**

[apply\\_left](#), [apply\\_right](#), [apply\\_right\\_S4](#)

**Examples**

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

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

---

apply\_right\_S4

*S4 dispatch method for apply\_right.*

---

**Description**

Intended to be generic on first two arguments.

**Usage**

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

**Arguments**

pipe\_left\_arg left argument  
 pipe\_right\_arg pipe\_right\_arg argument  
 pipe\_environment  
                   environment to evaluate in  
 left\_arg\_name name, if not NULL name of left argument.  
 pipe\_string character, name of pipe operator.  
 right\_arg\_name name, if not NULL name of right argument.

**Value**

result

**See Also**

[apply\\_left](#), [apply\\_right](#)

**Examples**

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

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

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

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

---

as.list.UnaryFn      *Get list of primitive unary fns.*

---

**Description**

Get list of primitive unary fns.

**Usage**

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

**Arguments**

x                      UnaryFn derived classe to extract  
 ...                    not used.

**Value**

list of non UnaryFnList functions

**Examples**

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

---

as.list.UnaryFnList    *Get list of primitive unary fns.*

---

**Description**

Get list of primitive unary fns.

**Usage**

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

**Arguments**

x	UnaryFn derived classe to extract
...	not used.

**Value**

list of non UnaryFnList functions

**Examples**

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

---

as.UnaryFn	<i>Convert a list of UnaryFns into a UnaryFn.</i>
------------	---

---

**Description**

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

**Usage**

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

**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	<i>Convert an unevaluted pipeline into a function.</i>
-----------	--

---

**Description**

Convert an unevaluted 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 <- as_dot_fn(sin(.) %.>% cos(.))
f(1:3)

g <- as_dot_fn(. %.>% sin(.) %.>% cos(.))
g(1:3)
```

---

as\_fn

---

*Convert a pipeable object into a function.*


---

**Description**

Convert a pipeable object into a function of "."

**Usage**

```
as_fn(pipeable, env = parent.frame())
```

**Arguments**

pipeable	a wrapr dot-pipe pipeable object
env	environment to work in.

**Details**

Note: writes "." into env.

**Value**

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

**Examples**

```
p <- pkgfn("base::sin", "x")
f <- as_fn(p)
f(5)
```

---

as_fnlist	<i>Wrap a list of UnaryFns as a UnaryFnList.</i>
-----------	--

---

**Description**

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

**Usage**

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

**Arguments**

items	list of UnaryFn 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	<i>Treat ... call argument as bquoted-values.</i>
-------------	---

---

**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.



**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	<i>Adapt a function to use bquote on its arguments.</i>
-----------------	---

---

**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**

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

```
f1 <- function(x) { substitute(x) }
f2 <- bquote_function(f1)
```

```
arg <- as.name("USER_ARG")
f2(arg) # returns arg
f2(. (arg)) # returns USER_ARG
```

---

buildNameCallback	<i>Build a custom writeback function that writes state into a user named variable.</i>
-------------------	--

---

### 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	<i>Build a data.frame from the user's description.</i>
-------------	--

---

### 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**

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

**Arguments**

... UnaryFn derived classes to combine

**Value**

UnaryFn representing the sequence

**Examples**

```
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**

```
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**

Please see <http://www.win-vector.com/blog/2014/05/trimming-the-fat-from-glm-models-in-r/> for discussion.

**Usage**

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

**Arguments**

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

**Value**

list(model=model, summary=summary)

**Examples**

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

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

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

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

---

clean_fit_lm	<i>Fit a stats::lm without carrying back large structures.</i>
--------------	--

---

## Description

Please see <http://www.win-vector.com/blog/2014/05/trimming-the-fat-from-glm-models-in-r/> for discussion.

## Usage

```
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

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

res_lm <- clean_fit_lm("y", c("x1", "x2"),
                      mk_data_example(1))
length(serialize(res_lm$model, NULL))

res_lm <- clean_fit_lm("y", c("x1", "x2"),
                      mk_data_example(10000))
length(serialize(res_lm$model, NULL))
```

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

---

coalesce	<i>Coalesce values (NULL/NA on left replaced by values on the right).</i>
----------	---

---

## 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)
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))

```

---

DebugFn	<i>Capture arguments of exception throwing function call for later debugging.</i>
---------	---

---

**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 <code>globalenv()</code> 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)
```

---

DebugFnE	<i>Capture arguments and environment of exception throwing function call for later debugging.</i>
----------	---

---

**Description**

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

**Usage**

```
DebugFnE(saveDest, fn, ...)
```

**Arguments**

saveDest	where to write captured state (determined by type): NULL random temp file, character temp file, name <code>globalenv()</code> 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/c9c327d10a91dcf9a3370d30dff8ac3d>. Please see: vignette("DebugFnW", package="wrapp").

### 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 <code>globalenv()</code> 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**

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

**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
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	<i>Capture arguments and environment of exception throwing function call for later debugging.</i>
---------------	---

---

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

---

**Description**

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

**Usage**

```
pipe_left_arg %>% pipe_right_arg
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 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](https://github.com/WinVector/wrapr/blob/master/extras/wrapr_pipe.pdf). For a base-R step-debuggable pipe please try the Bizarro Pipe <http://www.win-vector.com/blog/2017/01/using-the-bizarro-pipe-to-debug-magrittr-pipelines-in-r/>. %>.% and %.% are synonyms.

**Value**

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

**Functions**

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

**Examples**

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

---

draw_frame	<i>Render a simple data.frame in build_frame format.</i>
------------	--

---

**Description**

Render a simple data.frame in build\_frame format.

**Usage**

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

**Arguments**

x                    data.frame (with atomic types).  
 ...                 not used for values, forces later arguments to bind by name.  
 time\_format        character, format for "POSIXt" classes.  
 formatC\_options    named list, options for formatC()- used on numerics.  
 adjust\_for\_auto\_indent    integer additional after first row padding

**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	<i>Render a simple data.frame in qchar_frame format.</i>
-------------	--

---

### 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	<i>eval(bquote(expr)) shortcut.</i>
-------	-------------------------------------

---

**Description**

eval(bquote(expr)) shortcut.

**Usage**

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

**Arguments**

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

**Value**

eval(bquote(expr))

**Examples**

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

---

execute_parallel	<i>Execute f in parallel partitioned by partition_column.</i>
------------------	---

---

**Description**

Execute f in parallel partitioned by partition\_column, see [partition\\_tables](#) for details.

**Usage**

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

**Arguments**

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

**Value**

list of f evaluations.

**See Also**

[partition\\_tables](#)

**Examples**

```
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)
}
```

---

fnlist	<i>Wrap a list of functions as a function.</i>
--------	--

---

**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	<i>format step</i>
------------------------	--------------------

---

**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**

```
## 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**

```
## 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**

```
## 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**

```
grepdf(pattern, x, ..., ignore.case = FALSE, perl = FALSE,
        value = FALSE, fixed = FALSE, useBytes = FALSE, invert = FALSE)
```

**Arguments**

pattern	passed to <a href="#">grep</a>
x	data.frame to work with
...	force later arguments to be passed by name
ignore.case	passed to <a href="#">grep</a>
perl	passed to <a href="#">grep</a>
value	passed to <a href="#">grep</a>
fixed	passed to <a href="#">grep</a>
useBytes	passed to <a href="#">grep</a>
invert	passed to <a href="#">grep</a>



**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	<i>Return a vector of matches.</i>
-------	------------------------------------

---

**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 <a href="#">grep</a> .
x	character vector to match to, passed to <a href="#">grep</a> .
...	not used, forced later arguments to bind by name.
ignore.case	logical, passed to <a href="#">grep</a> .
perl	logical, passed to <a href="#">grep</a> .
fixed	logical, passed to <a href="#">grep</a> .
useBytes	logical, passed <a href="#">grep</a> .
invert	passed to <a href="#">grep</a> .

**Value**

vector of matching values.

**See Also**

[grep](#), [grepdf](#)

**Examples**

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

---

has_no_dup_rows	<i>Check for duplicate rows.</i>
-----------------	----------------------------------

---

**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	<i>Invert a permutation.</i>
-------------	------------------------------

---

**Description**

For a permutation  $p$  build  $q$  such that  $p[q] == q[p] == \text{seq\_len}(\text{length}(p))$ . 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 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-syntax: lambda(arg [, arg]*, body [, env=env])
# also works with lambda character as function name
# print(intToUtf8(0x03BB))

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

# example more than one argument
```

```
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)
```

---

lapplym

*Memoizing wrapper for lapply.*

---

## Description

Memoizing wrapper for lapply.

## Usage

```
lapplym(X, FUN, ...)
```

## Arguments

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

## Value

list of results.

## See Also

[VectorizeM](#), [vapplym](#), [parLapplyLBm](#)

## Examples

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

---

let	<i>Execute expr with name substitutions specified in alias.</i>
-----	---

---

### 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 `wrpr` vignette for some discussion of `let` and crossing function call boundaries: `vignette('wrpr', 'wrpr')`. For formal documentation please see [https://github.com/WinVector/wrpr/blob/master/extras/wrpr\\_let.pdf](https://github.com/WinVector/wrpr/blob/master/extras/wrpr_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 `wrpr: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.
```

**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	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**

```
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	<i>Map symbol names to referenced values if those values are string scalars (else throw).</i>
---------	---

---

**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
...	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	<i>Map up-cased symbol names to referenced values if those values are string scalars (else throw).</i>
-----------	--

---

**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	<i>Match one order to another.</i>
-------------	------------------------------------

---

**Description**

Build a permutation  $p$  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_formula	<i>Construct a formula.</i>
------------	-----------------------------

---

**Description**

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

**Usage**

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

## 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.

## Details

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

## Value

a formula object

## See Also

[reformulate](#), [update.formula](#)

## Examples

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

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

mk\_tmp\_name\_source      *Produce a temp name generator with a given prefix.*

---

### Description

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

### Usage

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

### Arguments

<code>prefix</code>	character, string to prefix temp names with.
<code>...</code>	force later argument to be bound by name.
<code>alphabet</code>	character, characters to choose from in building ids.
<code>size</code>	character, number of characters to build id portion of names from.
<code>sep</code>	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	<i>Named map builder.</i>
-------------------	---------------------------

---

### 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)

```

---

orderv	<i>Order by a list of vectors.</i>
--------	------------------------------------

---

### Description

Produce an ordering permutation from a list of vectors. Essentially a non-... interface to [order](#).

### Usage

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

### Arguments

columns	list of atomic columns to order on, can be a <code>data.frame</code> .
...	not used, force later arguments to bind by name.
na.last	(passed to <a href="#">order</a> ) 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 <a href="#">order</a> ) 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 <a href="#">order</a> ) 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 <a href="#">sort</a> .

### 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[order(d), , drop = FALSE]
```

---

parLapplyLBm

*Memoizing wrapper for parLapplyLB*

---

## Description

Memoizing wrapper for parLapplyLB

## Usage

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

## Arguments

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

## Value

list of results.

## See Also

[parLapplyLB](#), [lapplym](#), [VectorizeM](#), [vapplym](#)

## Examples

```
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

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

### Arguments

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

### Value

list of names maps of data.frames partitioned by partition\_column.



**See Also**[execute\\_parallel](#)**Examples**

```
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))
```

---

 pkgfn

---

*Wrap the name of a function as a function.*


---

**Description**

Wrap the name of a function as a function.

**Usage**

```
pkgfn(fname, arg_name = ".", args = list())
```

**Arguments**

fname	character, function name in fname or package::fname format.
arg_name	character, name of argument to assign.
args	named list of additional arguments and values.

**Value**

PartialNamedFn

**See Also**[fnlist](#), [wrapfn](#), [srcfn](#)**Examples**

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

---

psagg

*Pseudo aggregator.*

---

## Description

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

## Usage

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

## Arguments

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

## Details

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

## Value

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

## Examples

```
d <- data.frame(
  group = c("a", "a", "b"),
  stringsAsFactors = FALSE)
d1 <- 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, d1)
```

---

qae	<i>Quote assignment expressions (name = expr, name := expr, name %:= % expr).</i>
-----	---

---

### Description

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

### Usage

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

### Arguments

... assignment expressions.

### Details

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

### Value

array of quoted assignment expressions.

### See Also

[qc](#), [qe](#)

### Examples

```
ratio <- 2

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

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

# library("rqdatatable")
# datasets::iris %.>%
# extend_se(., exprs) %.>%
# summary(.)
```

---

`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  
.wrapr_private_var_env  
              environment to evaluate in
```

## Details

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

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

## Value

quoted array of character items

## See Also

[qe](#), [qae](#), [bquote](#)

## Examples

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

```

qc(a, c("b", "c")) # returns c("a", "b", "c")

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

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	<i>Build a quoted data.frame.</i>
-------------	-----------------------------------

---

### Description

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

### Usage

```
qchar_frame(...)
```

### Arguments

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

### Details

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

### Value

character data.frame

### See Also

[draw\\_frame](#), [build\\_frame](#)

**Examples**

```

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(.)

```

---

 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.

**Details**

qe() uses bquote() .() quasiquote escaping notation.

**Value**

array of quoted assignment expressions.

**See Also**

[qc](#), [qae](#)

**Examples**

```
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**

```
qs(s)
```

**Arguments**

s                    expression to be quoted as a string.

**Value**

character

**Examples**

```
x <- 7

qs(a == x)

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))

# # 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

```
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**

```
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	<i>Run wrapr package tests.</i>
-----------------	---------------------------------

---

**Description**

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

**Usage**

```
run_wrapp_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	<i>Increasing whole-number sequence.</i>
------	--

---

**Description**

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

**Usage**

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

**Arguments**

a scalar lower bound

b scalar upper bound

**Value**

whole number sequence

**Examples**

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

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

---

sequence\_as\_function *Convert a sequence of expressions into a function.*

---

**Description**

Convert a sequence of expressions into a function.

**Usage**

```
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**

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

---

show,PartialFunction-method  
*S4 print method*

---

**Description**

S4 print method

**Usage**

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

**Arguments**

object            item to print

---

show,PartialNamedFn-method  
*S4 print method*

---

**Description**

S4 print method

**Usage**

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

**Arguments**

object            item to print

---

show,SrcFunction-method  
*S4 print method*

---

**Description**

S4 print method

**Usage**

```
## 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

<code>str</code>	charater string to be substituted into
<code>...</code>	force later arguments to bind by name
<code>envir</code>	environemnt to look for values
<code>enclos</code>	enclosing evaluation environment
<code>match_pattern</code>	regexp to find substitution targets.
<code>removal_patterns</code>	regexps to remove markers from substitution targets.

## Details

See also <https://CRAN.R-project.org/package=R.utils>, <https://CRAN.R-project.org/package=rprintf>, and <https://CRAN.R-project.org/package=glue>.

## Value

modified strings

## Examples

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

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

# We can also change the delimiters,
# in this case to !! through the first whitespace.
```

```
sinterp(c("x is !!x , x+1 is !!x+1\n!!x 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

```
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 <a href="#">order</a> ) 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 <a href="#">order</a> ) 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 <a href="#">order</a> ) 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 <a href="#">sort</a> .

### 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 exprssion as a function.*

---

**Description**

Wrap the source for an exprssion as a function.

**Usage**

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

**Arguments**

expr\_src            character, source code of expression.  
arg\_name            characer, name of argument to assign.  
args                named list of additional arguments and values.

**Value**

SrcFunction

**See Also**[fnlist](#), [pkgfn](#), [wrapfn](#)**Examples**

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

---

SrcFunction-class	<i>Code text as a new partial function.</i>
-------------------	---

---

**Description**

Code text as a new partial function.

---

stop_if_dot_args	<i>Stop with message if dot_args is a non-trivial list.</i>
------------------	---

---

**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	<i>Split a string, keeping separator regions</i>
------------------	--

---

**Description**

Split a string, keeping separator regions

**Usage**

```
strsplit_capture(x, split, ..., ignore.case = FALSE, fixed = FALSE,
  perl = FALSE, useBytes = FALSE)
```

**Arguments**

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

**Value**

list of string segments annotated with is\_sep.

**Examples**

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

---

UnaryFn-class	<i>Functions that take a single argument</i>
---------------	--

---

**Description**

Functions that take a single argument

---

UnaryFnList-class	<i>List of Unary functions taken in order.</i>
-------------------	--

---

**Description**

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

---

uniques	<i>Strict version of unique (without ...).</i>
---------	--

---

**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 <code>base::unique</code> .
MARGIN	passed to <code>base::unique</code> .
fromLast	passed to <code>base::unique</code> .

**Value**

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

**Examples**

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

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

---

vapplym	<i>Memoizing wrapper for vapply.</i>
---------	--------------------------------------

---

**Description**

Memoizing wrapper for vapply.

**Usage**

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

**Arguments**

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

**Value**

vector of results.

**See Also**

[VectorizeM](#), [lapplym](#)

**Examples**

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

---

 VectorizeM

*Memoizing wrapper to base::Vectorize()*


---

**Description**

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

**Usage**

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

**Arguments**

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

**Details**

Only sensible for pure side-effect free deterministic functions.

**Value**

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

**See Also**

[Vectorize](#), [vapplym](#), [lapplym](#)

**Examples**

```
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	<i>Invoke a spreadsheet like viewer when appropriate.</i>
------	---

---

**Description**

Invoke a spreadsheet like viewer when appropriate.

**Usage**

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

**Arguments**

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

**Value**

invoke view or format object

**Examples**

```
view(mtcars)
```

---

wrapfn	<i>Wrap the source for an exprssion as a function.</i>
--------	--

---

**Description**

Wrap the source for an exprssion as a function.

**Usage**

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

**Arguments**

fn	function.
arg_name	characer, name of argument to assign.
args	named list of adittional arguments and values.

**Value**

PartialFunction

**See Also**

[pkgfn](#), [fnlist](#), [srcfn](#)

**Examples**

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

---

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-0qttT0-o9](https://youtu.be/iKLGxzzm9Hk?list=PLAKBwakacHbQp_Z66asDnjn-0qttT0-o9)
- Debug wrapper video: <https://youtu.be/zFEC9-1XSN8?list=PLAKBwakacHbQT51nPHex1on3YNCCmggZA>.



---

%in_block%	<i>Inline let-block notation.</i>
------------	-----------------------------------

---

### Description

Inline version of let-block.

### Usage

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, e2)
```

## Examples

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

---

`%dot%` *Inline dot product.*

---

**Description**

Inline dot product.

**Usage**

`e1 %dot% e2`

**Arguments**

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

**Value**

`c(e1, c2)`

**Examples**

`c(1,2) %dot% c(3, 5)`

---

`%p%` *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"
```

---

%qc%

*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 %qc% 5:6
```

```
c("a", "b") %qc% d
```

```
a %qc% b %qc% c
```

# Index

- `:=` (named\_map\_builder), 53
- `%.>%` (dot\_arrow), 33
- `%.%` (dot\_arrow), 33
- `%:=%` (named\_map\_builder), 53
- `%>.%` (dot\_arrow), 33
- `%?%` (coalesce), 24
- `%c%`, 82
- `%dot%`, 83
- `%in_block%`, 81
- `%p%`, 83
- `%qc%`, 84
  
- `add_name_column`, 4
- `apply_left`, 6, 8–12
- `apply_left.default`, 7, 7
- `apply_left_default`, 8
- `apply_right`, 9, 11, 12
- `apply_right.default`, 10
- `apply_right_S4`, 10, 11, 11
- `ApplyTo`, 5
- `ApplyTo`, PartialFunction, ANY-method (ApplyTo), 5
- `ApplyTo`, PartialFunction, UnaryFn-method (ApplyTo), 5
- `ApplyTo`, PartialFunction, UnaryFnList-method (ApplyTo), 5
- `ApplyTo`, PartialNamedFn, ANY-method (ApplyTo), 5
- `ApplyTo`, PartialNamedFn, UnaryFn-method (ApplyTo), 5
- `ApplyTo`, PartialNamedFn, UnaryFnList-method (ApplyTo), 5
- `ApplyTo`, SrcFunction, ANY-method (ApplyTo), 5
- `ApplyTo`, SrcFunction, UnaryFn-method (ApplyTo), 5
- `ApplyTo`, SrcFunction, UnaryFnList-method (ApplyTo), 5
- `ApplyTo`, UnaryFn, UnaryFn-method (ApplyTo), 5
  
- `ApplyTo`, UnaryFn, UnaryFnList-method (ApplyTo), 5
- `ApplyTo`, UnaryFnList, ANY-method (ApplyTo), 5
- `ApplyTo`, UnaryFnList, UnaryFn-method (ApplyTo), 5
- `ApplyTo`, UnaryFnList, UnaryFnList-method (ApplyTo), 5
- `as.list.UnaryFn`, 12
- `as.list.UnaryFnList`, 13
- `as.UnaryFn`, 14
- `as_dot_fn`, 14
- `as_fn`, 15
- `as_fnlist`, 16
  
- `bquote`, 46, 60
- `bquote_call`, 16
- `bquote_call_args`, 17
- `bquote_function`, 17, 17
- `build_frame`, 19, 34, 35, 61
- `buildNameCallback`, 18
  
- `c`, 64
- `c.UnaryFn`, 20
- `capture.output`, 49
- `check_equiv_frames`, 21
- `checkColsFormUniqueKeys`, 20
- `clean_fit_glm`, 21
- `clean_fit_lm`, 23
- `coalesce`, 24
- `Collector`, 25
  
- `DebugFn`, 26, 26, 27–29, 31, 32
- `DebugFnE`, 26, 27, 27, 28, 29, 31, 32
- `DebugFnW`, 18, 26–28, 28, 29, 31, 32
- `DebugFnWE`, 26–29, 29, 31, 32
- `DebugPrintFn`, 26–29, 30, 31, 32
- `DebugPrintFnE`, 26–29, 31, 31, 32
- `defineLambda`, 32, 43, 47, 53
- `do.call`, 46, 64

- dot\_arrow, 33
- dput, 49
- draw\_frame, 19, 34, 61
- draw\_framec, 35
- dump.frames, 26–29, 31, 32
- evalb, 36
- execute\_parallel, 36, 57
- fnlist, 38, 57, 74, 80
- format.PartialFunction, 38
- format.PartialNamedFn, 39
- format.SrcFunction, 39
- format.UnaryFnList, 40
- grep, 40–42
- grepdf, 40, 42
- grepv, 41, 41
- has\_no\_dup\_rows, 42
- invert\_perm, 42
- isTRUE, 24
- lambda, 32, 43, 47, 53
- lapplym, 44, 55, 77, 78
- let, 45, 48, 49, 81
- list, 64
- makeFunction\_se, 32, 43, 46, 53
- map\_to\_char, 48
- map\_upper, 49
- mapsyms, 47
- match\_order, 50
- mk\_formula, 50
- mk\_tmp\_name\_source, 52
- named\_map\_builder, 32, 43, 47, 53
- order, 54, 72
- orderv, 54, 72
- parLapplyLB, 55
- parLapplyLBm, 44, 55
- PartialFunction-class, 56
- PartialNamedFn-class, 56
- partition\_tables, 36, 37, 56
- pkgfn, 14, 16, 38, 57, 74, 80
- psagg, 58
- qae, 59, 60, 62
- qc, 59, 60, 62
- qchar\_frame, 19, 34, 35, 61
- qe, 59, 60, 62
- qs, 63
- reduceexpand, 64
- reformulate, 51
- restrictToNameAssignments, 65
- run\_package\_tests, 65
- run\_wrapr\_tests, 66
- seqi, 67
- sequence\_as\_function, 68
- show,PartialFunction-method, 69
- show,PartialNamedFn-method, 69
- show,SrcFunction-method, 70
- show,UnaryFnList-method, 70
- sinterp, 71
- sort, 54, 72
- sortv, 54, 72
- split\_at\_brace\_pairs, 73
- srcfn, 14, 16, 38, 57, 73, 80
- SrcFunction-class, 74
- stop\_if\_dot\_args, 74
- strsplit\_capture, 75
- sum, 64
- UnaryFn-class, 76
- UnaryFnList-class, 76
- uniques, 76
- update.formula, 51
- vapplym, 44, 55, 77, 78
- Vectorize, 78
- VectorizeM, 44, 55, 77, 78
- view, 79
- wrapfn, 14, 16, 38, 57, 74, 79
- wrapr, 80
- wrapr-package (wrapr), 80