Package: pkgutils (via r-universe)

August 26, 2024

Title Functions for building functions

Version 0.1.0

Description A suite of tools for helping in function and package development. This includes functions that check variables for consistency, or help parse out inputs to functions. This has a particular focus on inputs that use formula or `tidyselect` interfaces to define the parts of a dataframe that we might be interested in.

License MIT + file LICENSE

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.2.3

Imports rlang, utils, stringr, glue, tibble, dplyr, purrr, tidyr, magrittr, lubridate, tidyselect, digest, cli

Suggests knitr, pillar, rmarkdown, testthat (>= 3.0.0), ggplot2

VignetteBuilder knitr

Config/testthat/edition 3

URL https://bristol-vaccine-centre.github.io/pkgutils/index.html, https://github.com/bristol-vaccine-centre/pkgutils

BugReports https://github.com/bristol-vaccine-centre/pkgutils/issues

Depends R (>= 3.4.0)

Repository https://bristol-vaccine-centre.r-universe.dev

RemoteUrl https://github.com/bristol-vaccine-centre/pkgutils

RemoteRef 0.1.0

RemoteSha 9f04dfc234354fe556f5de1b8cd117a48bc39829

Contents

`as.var_grp_df` *The* `var_grp_df` *dataframe subtype*

Description

This is like a grouped data frame but with 3 grouping dimensions. These are labelled z, y, and x and relate to as z (i.e. group, or cohort), y (i.e. subgroup, or response) and x (i.e. data). In some configurations, only z and x are non-empty. The purpose of this is to make some group / subgroup data operations consistent. An example is running multiple models across different bootstraps from example.

Usage

```
as.var_grp_df(df, z, y, x)
```

Arguments

df	a dataframe
z	the z columns (e.g. cohort) as a list of columns
y	the y columns (e.g. response) as a list of columns
x	the x columns (e.g. predictor) as a list of columns

Examples

```
tmp = as.var_grp_df(iris,
  c("Species"),
  c("Sepal.Width", "Sepal.Length"),
  c("Petal.Width", "Petal.Length"))
# print.var_grp_df(tmp)
glimpse.var_grp_df(tmp)
```

---

check_consistent *Check function parameters are conform to a set of rules*

Description

If the parameters of a function are given in some combination but have an interdependency (e.g. different parameterisations of a probability distribution) or a constraint (like `x>0`) this function can simulataneously check all interrelations are satisfied and report on all the not conformant features of the parameters.

Usage

```
check_consistent(..., .env = rlang::caller_env())
```

Arguments

...	a set of rules to check either as `x=y+z`, or `x>y`. Single `=` assignment is checked for equality using `identical` otherwise the expressions are evaluated and checked they all are true. This for consistency with `resolve_missing` which only uses assignment, and ignores logical expressions.
.env	the environment to check in

Value

nothing, throws an informative error if the checks fail.

Examples

```
testfn = function(pos, neg, n) {
  check_consistent(pos=n-neg, neg=n-pos, n=pos+neg, n>pos, n>neg)
}

testfn(pos = 1:4, neg=4:1, n=rep(5,4))
try(testfn(pos = 1:4, neg=5:2, n=rep(5,4)))
```

---

check_date *Checks a set of variables can be coerced to a date and coerces them*

Description

Checks a set of variables can be coerced to a date and coerces them

Usage

```
check_date(
  ...,
  .message = "`{param}` is not a date: ({err}).",
  .env = rlang::caller_env()
)
```

Arguments

`...`	Arguments passed on to `base::as.Date` `x` an object to be converted.
`.message`	a glue spec containing `{param}` as the name of the parameter and `{err}` the cause fo the error
`.env`	the environment to check (defaults to calling environment)

Value

nothing. called for side effects. throws error if not all variables can be coerced.

Examples

```
a = c(1:4L)
b = c("1",NA,"3.3")
f = NULL
g = NA
check_numeric(a,b,f,g)

c = c("dfsfs")
try(check_numeric(c,d, mean))
```

check_integer *Checks a set of variables can be coerced to integer and coerces them*

Description

N.B. This only works for the specific environment (to prevent weird side effects)

Usage

```
check_integer(
  ...,
  .message = "`{param}` is not an integer ({err}).",
  .env = rlang::caller_env()
)
```

Arguments

...	a list of symbols
.message	a glue spec containing {param} as the name of the parameter and {err} the cause fo the error
.env	the environment to check (defaults to calling environment)

Value

nothing. called for side effects. throws error if not all variables can be coerced.

Examples

```
a = c(1:4)
b = c("1",NA,"3")
f = NULL
g = NA
check_integer(a,b,f,g)

c = c("dfsfs")
e = c(1.0,2.3)
try(check_integer(c,d,e, mean))
```

check_numeric *Checks a set of variables can be coerced to numeric and coerces them*

Description

N.B. This only works for the specific environment (to prevent weird side effects)

Usage

```
check_numeric(
  ...,
  .message = "`{param}` is non-numeric ({err}).",
  .env = rlang::caller_env()
)
```

Arguments

...	a list of symbols
.message	a glue spec containing {param} as the name of the parameter and {err} the cause fo the error
.env	the environment to check (defaults to calling environment)

Value

nothing. called for side effects. throws error if not all variables can be coerced.

Examples

```
a = c(1:4L)
b = c("1",NA,"3.3")
f = NULL
g = NA
check_numeric(a,b,f,g)

c = c("dfsfs")
try(check_numeric(c,d, mean))
```

col_syms *Column names as symbols*

Description

Column names as symbols

Usage

```
col_syms(df)
```

Arguments

df	a dataframe

Value

a list of symbols

Examples

```
intersect(col_syms(iris), ensyms2(tidyselect::starts_with("S"), .tidy=iris))
```

---

ensyms2 *Convert a parameter into a list of symbols*

Description

Used within a function this allows for a list of columns to be given as a parameter to the parent function in a number of flexible ways. A list of unquoted symbols, a list of quoted strings, a tidyselect syntax (assuming the parent function has a dataframe is its first argument) or as a formula.

Usage

```
ensyms2(
  x,
  .as = c("symbol", "character"),
  .side = c("rhs", "lhs"),
  .tidy = FALSE
)
```

Arguments

x	one of a list of symbols, a list of strings, a `tidyselect` expression, or a formula
.as	the type of output desired: (`symbol` or `character`)
.side	the desired side of formulae output: (`lhs` or `rhs`); this is only relevant if `x` is a formula (or list of formulae)
.tidy	is this being called in the context of a "tidy" style function. I.e. one that takes a dataframe as the main parameter? (Default is `FALSE`)

Value

either a list of symbols or a character vector of the symbols

Examples

```
# TODO: convert these to tests
eg = function(df, vars, ...) {
  vars = ensyms2(vars, ..., .tidy=TRUE)
  print(vars)
}

eg(iris, c(Sepal.Width, Species, Sepal.Length))
eg(iris, c("Sepal.Width", "Species", "Sepal.Length", "extra"))
eg(iris, "Sepal.Width")
eg(iris, Sepal.Width)
eg(iris, dplyr::vars(Sepal.Width))
eg(iris, dplyr::vars(Sepal.Width, Species, Sepal.Length))
eg(iris, list(Sepal.Width, Species, Sepal.Length))
eg(iris, list("Sepal.Width", "Species", "Sepal.Length"))
eg(iris, tidyselect::starts_with("Sepal"))
eg(iris, Species ~ Sepal.Width + Sepal.Length)
eg(iris, Species ~ Sepal.Width + Sepal.Length, .side = "lhs")
eg(iris, . ~ Sepal.Width + Sepal.Length, .side = "lhs")
eg(iris, Sepal.Width + Sepal.Length ~ .)
eg(iris, c(~ Sepal.Width + Sepal.Length, ~ Petal.Width + Petal.Length))

try(eg(iris, c(~ .)))
eg(iris, list(~ Sepal.Width + Sepal.Length, ~ Petal.Width + Petal.Length))

# In a way this shouldn't work, but does:
eg(iris, c(~ Sepal.Width + Sepal.Length, Petal.Width + Petal.Length))

# injection support:
subs = ensyms2(c("Sepal.Width", "Species", "Sepal.Length"))

# this must be injected as a single thing as the parameter x but actually it
# turns out to be just the same as supplying a list of symbols as the bare
# parameter
# eg(iris,!!subs)
# ensyms2(!!subs)
# same as:
# eg(iris,subs)
# ensyms2(subs)
```

`escalate` *Cause warnings to create an error*

Description

The opposite of `suppressWarnings()`. This will immediately error if a warning if thrown by `expr`. This is useful to track down the source of a random and to prevent Rs permissive approach to data transformations. It is also useful to identify where in the code a intermittent `rlang` warning is being issued once every 8 hours.

Usage

```
escalate(expr)
```

Arguments

expr	expression to evaluate

Value

the evaluated expression or an error

Examples

```
try(escalate(as.integer("ASDAS")))

try(escalate(rlang::warn("test", .frequency="regularly", .frequency_id = "asdasdasasdd")))
try(escalate(rlang::warn("test", .frequency="regularly", .frequency_id = "asdasdasasdd")))
try(escalate(rlang::warn("test", .frequency="regularly", .frequency_id = "asdasdasasdd")))
try(escalate(rlang::warn("test", .frequency="regularly", .frequency_id = "asdasdasasdd")))

# options("rlib_warning_verbosity"=NULL)
# options("rlib_warning_verbosity"="verbose")
# "lifecycle_verbosity"="warning"
```

---

get_fn_args *Fully evaluate the arguments from a function call as a named list.*

Description

Used within a function this provides access to the actual arguments provided during invocation of the parent function, plus any default values. The parameters are evaluated eagerly before being returned (so symbols and expressions must resolve to real values.)

Usage

```
get_fn_args(env = rlang::caller_env(), missing = TRUE)
```

Arguments

env	the environment to check (default `rlang::caller_env()`)
missing	include missing parameters in list (default `TRUE`)?

Value

a named list of the arguments of the enclosing function

Examples

```
ftest = function(a,b,c="default",...) {
  tmp = get_fn_args()
  tmp
}

ftest(a=1, b=2)

# missing param `b` - empty values are returned just as a name in the environment
# with no value but which can be checked for as if in the environment.
tmp = ftest(a=1)
class(tmp$b)
rlang::is_missing(tmp$b)
b = 1
rlang::is_missing(tmp$b)

# extra param `d` and default parameter `c`
ftest(a=1, b=2, d="another")

# does not work
try(ftest(a=1, b=2, d=another))
# does work
tmp = ftest( a=1, d= as.symbol("another") )
# also does work
another =5
ftest( a=1, d= another)

# Filter out missing values

ftest2 = function(a,b,c="default",...) {
  tmp = get_fn_args(missing=FALSE)
  tmp
}

ftest2(a=1)
```

---

get_fn_name *Get the name of a function*

Description

Functions may be named or anonymous. When functions are used as a parameter, for error reporting it is sometimes useful to be able to refer to the function by the name it is given when it is defined. Sometimes functions can have multiple names.

Usage

```
get_fn_name(fn = rlang::caller_fn(), fmt = "%s", collapse = "/")
```

Arguments

fn	a function definition (defaults to the function from which get_fn_name is called)
fmt	passed to sprintf with the function name e.g. %s() will append brackets
collapse	passed to paste0 in the case of multiple matching functions. set this to NULL if you want the multiple function names as a vector.

Value

the name of the function or "<unknown>" if not known

Examples

```
# detecting the name when function used as a parameter. This is the
# primary use case for `get_fn_name`
testfn2 = function(fn) {
  message("called with function: ",get_fn_name(fn))
}

testfn2(mean)
testfn2(utils::head)
testfn2(testfn2)

# detecting the name of a calling function, an unusual use case as this is
# normally known to the user.
testfn = function() {
  message(get_fn_name(fmt="%s(...)")," is a function")
}

`test fn 2` = testfn
test_fn_3 = testfn
testfn()
```

---

optional_fn *Get an optional function without triggering a CRAN warning*

Description

You want to use a function if it is installed but don't want it to be installed as part of your package and you don't want to reference it as part of the Imports or Suggests fields in a package DESCRIPTION.

Usage

```
optional_fn(
  pkg,
  name,
  alt = function(...) {
     stop("function `", pkg, "::", name, "(...)` not available")
```

```
    }
)
```

Arguments

pkg	the package name (or the function name as "pkg::fn")
name	the function you wish to use (if not specified in pkg)
alt	an alternative function that can be used if the requested one is not available. The default throws an error if the package is not available, but a fallback can be used instead.

Value

the function you want if available or the alternative

Examples

```
# use openSSL if installed:
fn = optional_fn("openssl", "md5", alt = ~ digest::digest(.x, "md5"))

as.character(fn(as.raw(c(1,2,3))))

#' # this function does not exists and so the alternative is used instead.
fn3 = optional_fn("asdasdadsda::asdasdasd", ~ message("formula alternative"))
fn3()
```

---

recycle *Strictly recycle function parameters*

Description

recycle is called within a function and ensures the parameters in the calling function are all the same length by repeating them using rep. This function alters the environment from which it is called. It is stricter than R recycling in that it will not repeat vectors other than length one to match the longer ones, and it throws more informative errors.

Usage

```
recycle(..., .min = 1, .env = rlang::caller_env())
```

Arguments

...	the variables to recycle
.min	the minimum length of the results (defaults to 1)
.env	the environment to recycle within.

Details

NULL values are not recycled, missing values are ignored.

Value

the length of the longest variable

Examples

```
testfn = function(a, b, c) {
  n = recycle(a,b,c)
  print(a)
  print(b)
  print(c)
  print(n)
}

testfn(a=c(1,2,3), b="needs recycling", c=NULL)
try(testfn(a=c(1,2,3), c=NULL))

testfn(a=character(), b=integer(), c=NULL)

# inconsistent to have a zero length and a non zero length
try(testfn(a=c("a","b"), b=integer(), c=NULL))
```

---

resolve_missing *Resolve missing values in function parameters and check consistency*

Description

Uses relationships between parameters to iteratively fill in missing values. It is possible to specify an inconsistent set of rules or data in which case the resulting values will be picked up and an error thrown.

Usage

```
resolve_missing(
  ...,
  .env = rlang::caller_env(),
  .eval_null = TRUE,
  .error =
   "unable to infer missing variable(s): {.missing} using:\n{.constraints}\ngiven known variable(s): {.
)
```

Arguments

...	either a set of relationships as a list of x=y+z expressions
.env	the environment to check in (optional - defaults to caller_env())
.eval_null	the resolution defined missing variables as those that are not specified but we can also fill in values that are explicitly given as NULL or default to NULL if this is TRUE. This is the default.
.error	a glue spec defining the error message. This can use parameters .missing, .constraints, .present and .call to construct an error message.

Value

nothing. Alters the .env environment to fill in missing values or throws an informative error

Examples

```
# missing variables left with default value of NULL in function definition
testfn = function(pos, neg, n) {
  resolve_missing(pos=n-neg, neg=n-pos, n=pos+neg)
  return(tibble::tibble(pos=pos,neg=neg,n=n))
}

testfn(pos=1:4, neg = 4:1)
testfn(neg=1:4, n = 10:7)

try(testfn())

# not enough info to infer the missing variables
try(testfn(neg=1:4))

# the parameters given are inconsistent with the relationships defined.
try(testfn(pos=2, neg=1, n=4))
```

---

var_group *Extract a definition of column groups from function parameters*

Description

This is a supporting utility for functions that have a signature of function(df, ...) that operate on different groups of columns, and need the user to supply column groups in a simple way. There are 2 or 3 levels of column grouping that can be specified easily in this style of function, and they are generally referred to as z (i.e. group, or cohort), y (i.e. subgroup, or response) and x (i.e. data). In some configurations, only z and x are available.

Usage

```
var_group(df, ..., .infer_y = FALSE)
```

Arguments

df	a data frame which may be grouped
...	a specification for the groupings which may be one of: • A formula or list of formulae (e.g. y1 + y2 ~ x1 + x2, z:from df grouping). the . can be used to specify the rest of the columns, e.g. y1 + y2 ~ . • A list of symbols (x1, x2, ..., z:from df grouping, y:empty) • A list of quosures (e.g. dplyr::vars(x1,x2)) (x, z:from df grouping, y:empty) • One tidyselect specification (x, z:from df grouping, y:empty) • Two tidyselect specifications (x, y, z:from df grouping) • Three tidyselect specifications (x, y, z, N.B. df must be ungrouped for this to work) • Column names as strings (x, z:from df grouping, y:empty)
.infer_y	if only z and x is defined make y the rest of the dataframe columns

Value

a var_grp_df with defined z, y and x column groups, for use within the var_group_* framework.

Examples

```
tmp = iris %>% dplyr::group_by(Species) %>% var_group(. ~ Petal.Width + Sepal.Width)

tmp = iris %>% dplyr::group_by(Species) %>%
  var_group(tidyselect::starts_with("Sepal"),tidyselect::starts_with("Petal"))
```

---

var_group_compare *Cross compare subgroups of data to each other*

Description

This function helps construct group wise cross-correlation matrices and other between column comparisons from a dataframe. We assume we have a data with a major grouping and then data columns we wish to compare to each other. We specify the columns to compare to each other as a formula or as a tidyselect using a var_grp_df and using this we use these a set of columns to compare.

Usage

```
var_group_compare(var_grp_df, ..., .diagonal = FALSE)
```

Arguments

var_grp_df	a data frame with major and data groupings
...	a set of named functions. The functions must take 2 vectors of the type of the columns being compared and generate a single result (which may be a complex S3 object such as a lm). Such functions might be for example be chisq.test for factor columns or cor for numeric columns.
.diagonal	should a column be compared with itself? this is usually FALSE

Details

Although the examples here are functional we generally expect these to be wrapped within a function within a package where the comparisons are pre-defined, and the var_group framework is hidden from the user.

Value

a dataframe containing the major z groupings and unique binary combinations of y and x columnsas y and x columns. The named comparisons provided in ... form the other columns. If these are not primitive types this will be a list column.

Examples

```
iris %>% dplyr::group_by(Species) %>% var_group(~ .) %>%
  var_group_compare(
    correlation = cor
  )

ggplot2::diamonds %>% var_group(tidyselect::where(is.factor)) %>%
  var_group_compare(
    chi.p.value = ~ stats::chisq.test(.x,.y)$p.value
  )
```

---

var_group_count *The number of major groups (z categories) in a var_grp_df*

Description

The number of major groups (z categories) in a var_grp_df

Usage

```
var_group_count(var_grp_df)
```

Arguments

var_grp_df	the var_grp dataframe

Value

a count of groups

Examples

```
tmp = iris %>% dplyr::group_by(Species) %>% var_group(. ~ Petal.Width + Sepal.Width)
tmp %>% var_group_count()
```

var_group_formula *Export* var_group *metadata as a formula*

Description

Produces the y and x terms of a var_grp_df as a formula for potentially using in a model or another var_group

Usage

```
var_group_formula(var_grp_df)
```

Arguments

var_grp_df	a var_group dataframe

Value

a formula like y1 + y2 ~ x1 + x2 + ...

var_group_modify *Apply a function to each* z *group using* group_modify()

Description

Apply a function to each z group using group_modify()

Usage

```
var_group_modify(var_grp_df, .f, ..., .subgroup = TRUE, .progress = FALSE)
```

Arguments

var_grp_df	the var_grp dataframe
.f	a function with the signature function(x,y,z,...) if the default .subgroup=TRUE or of the form function(xy,z,...) if .subgroup=FALSE. If .subgroup=TRUE The function will be called once for each group and subgroup with the parameters x being the data as a dataframe with usually multiple rows, and y and z being single row dataframes contianing the current subgroup and group respectively. Is subgroup=FALSE then only the major grouping z is used and
...	Arguments passed on to dplyr::group_modify .data A grouped tibble .keep are the grouping variables kept in .x
.subgroup	in the grouped data frames also subgroup by the y columns
.progress	shoudl progress be reported with a progress bar.

Value

the transformed data as a plain dataframe

Examples

```
tmp = iris %>% dplyr::group_by(Species) %>% var_group(. ~ Petal.Width + Petal.Length)

tmp2 = tmp %>% var_group_modify(
  ~ {
    Sys.sleep(0.02)
    return(.x %>% dplyr::count())
  },
  .progress=TRUE
)

tmp3 = tmp %>% var_group_modify(~ .x %>% dplyr::count(), .subgroup=FALSE)

# .f with 2 parameters:
tmp %>% var_group_modify(
  ~ {
    return(tibble::tibble(
      Sepal.Area = .y$Sepal.Length*.y$Sepal.Width,
      Max.Petal.Area = max(.x$Petal.Length*.x$Petal.Width),
      n = nrow(.x)
    ))
  }
) %>% dplyr::filter(n>1)
```

---

var_group_nest *Nest a* var_grp_df *by the* z *columns*

Description

Nest a var_grp_df by the z columns

Usage

```
var_group_nest(var_grp_df, .subgroup = FALSE, .key = "data")
```

Arguments

var_grp_df	the var_grp dataframe
.subgroup	in the nested data frames also group the y columns
.key	The name of the resulting nested column. Only applicable when ... isn't specified, i.e. in the case of df %>% nest(.by = x). If NULL, then "data" will be used by default.

Value

aa nested dataframe with z columns and a .key column with the y and x columns nested in it. The nested data will be grouped by y columns.

Examples

```
tmp = iris %>% dplyr::group_by(Species) %>% var_group(. ~ Petal.Width + Sepal.Width)
tmp2 = tmp %>% var_group_nest()
```

---

var_grps *Extract grouping info frm a* var_grp_df

Description

Extract grouping info frm a var_grp_df

Usage

```
var_grps(var_grp_df)
```

Arguments

var_grp_df	the dataframe

Value

a list of lists containing the x,y, and z column sets as symbol lists

---

var_grp_s3 var_grp_df *S3 Methods*

Description

var_grp_df S3 Methods

Usage

```
glimpse.var_grp_df(x, ...)

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

## S3 method for class 'var_grp_df'
print(x, ...)

is.var_grp_df(x, ...)
```

Arguments

x	a var_grp_df dataframe
...	passed to generic functions

Functions

- glimpse.var_grp_df(): glimpse
- format(var_grp_df): format
- print(var_grp_df): print
- is.var_grp_df(): is

---

var_has_groups *Does this* var_grp_df *have more than one major group?*

Description

Does this var_grp_df have more than one major group?

Usage

```
var_has_groups(var_grp_df)
```

Arguments

var_grp_df	a var_group dataframe

Value

boolean

---

var_subgroup_count *The number of major and sub groups (*z *and* y *categories) in a* var_grp_df

---

Description

The number of major and sub groups (z and y categories) in a var_grp_df

Usage

```
var_subgroup_count(var_grp_df, .stratified = FALSE)
```

Arguments

var_grp_df	the var_grp dataframe
.stratified	if TRUE return the subgroup count stratified by major groups as a dataframe

Value

a count of groups and subgroups

Examples

```
tmp = iris %>% dplyr::group_by(Species) %>% var_group(. ~ Petal.Width + Sepal.Width)
tmp %>% var_subgroup_count()
```

---

var_subgroup_nest *Nest a* var_grp_df *by the* z *and* y *columns*

Description

Nest a var_grp_df by the z and y columns

Usage

```
var_subgroup_nest(var_grp_df, .key = "data")
```

Arguments

var_grp_df	the var_grp dataframe
.key	The name of the resulting nested column. Only applicable when ... isn't specified, i.e. in the case of df %>% nest(.by = x). If NULL, then "data" will be used by default.

Value

a nested dataframe with z and y columns and a .key column with the x columns nested in it

Examples

```
tmp = iris %>% dplyr::group_by(Species) %>% var_group(. ~ Petal.Width + Sepal.Width)
tmp2 = tmp %>% var_group_nest()
```

Index