

Package: svBase (via r-universe)

October 9, 2026

Type Package

Version 1.7.4

Title 'SciViews::R' - Base Functions

Description Functions to manipulate the three main classes of ``data frames" for 'SciViews::R': data.frame, data.table and tibble. Allow to select the preferred one, and to convert more carefully between the three, taking care of correct presentation of row names and data.table's keys. More homogeneous way of creating these three data frames and of printing them on the R console.

Maintainer Philippe Grosjean <phgrosjean@sciviews.org>

Depends R (>= 4.2.0)

Imports cli (>= 3.6.4), collapse (>= 2.0.12), compiler (>= 4.2.0), data.table (>= 1.15.4), data.trame (>= 0.9.0), dplyr (>= 1.1.4), Hmisc (>= 5.2.3), lifecycle (>= 1.0.4), rlang (>= 1.1.1), tibble (>= 3.2.1), tidyr (>= 1.3.0), utils (>= 4.2.0), zeallot (>= 0.1.0)

Suggests dtplyr (>= 1.3.1), covr (>= 3.5.0), knitr (>= 1.42), rmarkdown (>= 2.21), spelling (>= 2.2.1), testthat (>= 3.0.0)

Remotes SciViews/data.trame

NeedsCompilation yes

License MIT + file LICENSE

URL <https://github.com/SciViews/svBase>,
<https://www.sciviews.org/svBase/>,
<https://sciviews.r-universe.dev/svBase>

BugReports <https://github.com/SciViews/svBase/issues>

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.2

VignetteBuilder knitr

Encoding UTF-8

Language en-US

ByteCompile yes

Config/testthat/edition 3

Config/testthat/parallel true

Config/pak/sysreqs cmake make libicu-dev libuv1-dev

Repository <https://sciviews.r-universe.dev>

Date/Publication 2026-03-03 09:28:28 UTC

RemoteUrl <https://github.com/SciViews/svBase>

RemoteRef HEAD

RemoteSha aa006d4c5624b2725d479f96084181dfe0fe3010

Contents

svBase-package	3
aka	3
alt_assign	5
as_dtx	7
collect_dtx	9
data_dot_mechanism	10
dtx	11
dtx_rows	13
expr_funs	14
f_	15
formula_masking	16
formula_select	17
fstat_functions	18
gettext_	20
is_dtx	23
labelise	24
prepare_data_dot	26
process_error_msg	29
retarget	30
section	31
speedy_functions	32
stop_	37
subtable	39
tidy_functions	41

Index	44
--------------	-----------

Description

The {svBase} package sets up the way data frames (with objects like R base's **data.frame**, **data.table** and tibble **tbl_df**) are managed in SciViews::R. The user can select the class of object it uses by default and many other SciViews::R functions return that format. Conversion from one to the other is made easier, including for the management of **data.frame**'s row names or **data.table**'s keys. Also homogeneous ways to create a data frame or to print it are also provided.

Important functions

- `dtx()` creates a data frame in the preferred format, with the following functions `dtbl()`, `dtf()` and `dttt()` that force respectively the creation of a data frame in one of the specified three formats. Use `getOption("SciViews.as_dtx", default = as_dtt)` to specify which function to use to convert into the preferred format.

Author(s)

Maintainer: Philippe Grosjean <phgrosjean@sciviews.org> ([ORCID](#))

See Also

Useful links:

- <https://github.com/SciViews/svBase>
- <https://www.sciviews.org/svBase/>
- <https://sciviews.r-universe.dev/svBase>
- Report bugs at <https://github.com/SciViews/svBase/issues>

Description

When a function or object is renamed, the link to its original help page is lost in R. Using `aka()` (also known as) with correct `alias=` allows to keep track of the original help page and get it with `?.obj`. Moreover, one can also populate a short man page with description, `seealso` and `example` or add a short comment message that is displayed at the same time in the R console.

Usage

```

aka(
  obj,
  alias = NULL,
  comment = "",
  description = NULL,
  seealso = NULL,
  example = NULL,
  url = NULL
)

## S3 method for class 'aka'
print(
  x,
  hyperlink_type = getOption("hyperlink_type", default = .hyperlink_type()),
  ...
)

## S3 method for class 'aka'
str(object, ...)

```

Arguments

<code>obj</code>	An R object.
<code>alias</code>	The full qualified name of the alias object whose help page should be retained as <code>pkg::name</code> . If <code>NULL</code> (by default), use <code>obj</code> .
<code>comment</code>	A comment to place in <code>obj</code> (will also be printed when calling <code>.?obj</code>).
<code>description</code>	A description of the function for the sort man page.
<code>seealso</code>	A character string of functions in the form <code>fun</code> or <code>pkg::fun</code> .
<code>example</code>	A character string with code for a short example.
<code>url</code>	An http or https URL pointing to the help page for the function on the Internet.
<code>x</code>	An aka object
<code>hyperlink_type</code>	The type of hyperlink supported. The default value should be <code>ok</code> . Use <code>"none"</code> to suppress hyperlink, <code>"href"</code> for <code>http(s)://</code> link that opens a web page, or <code>"help"</code> in RStudio to open the corresponding help page in the Help tab.
<code>...</code>	Further arguments (not used yet)
<code>object</code>	An aka object

Value

The original `obj` with the `comment` attribute set or replaced with `comment =` plus a `src` attribute set to `alias =` and `description`, `seealso`, `example`, and `url` attributes also possibly populated. If the object is a function, its class becomes **aka** and **function**.

Examples

```
# Say you prefer is.true() similar to is.na() or is.null()
# but R provides isTRUE().
library(svBase)
# Also defining a short man page
is.true <- aka(isTRUE, description = "Check if an object is TRUE.",
  seealso = c("is.false", "logical"), example = c("is.true(TRUE)", "is.true(FALSE)", "is.true(NA)"))
# This way, you still got access to the right help page for is.true()
## Not run:
.?is.true

## End(Not run)
```

alt_assign

Alternate assignment (multiple and/or collect results from dplyr)

Description

These alternate assignment operators can be used to perform multiple assignment (also known as destructuring assignment). These are imported from the {zeallot} package (see the corresponding help page at [zeallot::operator](#) for complete description). They also performs a [dplyr::collect\(\)](#) allowing to get results from dplyr extensions like {dtplyr} for data.tables, or {dbplyr} for databases. Finally these two assignment operators also make sure that the preferred data frame object is returned by using [default_dtx\(\)](#).

Usage

```
value %>% x

x %<-% value

## Default S3 method:
collect(x, ...)
```

Arguments

value	The object to be assigned.
x	A name, or a name structure for multiple (deconstructing) assignment, or any object that does not have a specific [dplyr::collect()] method for collect.default().
...	further arguments passed to the method (not used for the default one)

Details

These assignation operator are overloaded to get interesting properties in the context of {tidyverse} pipelines and to make sure to always return our preferred data frame object (data.frame, data.table, or tibble). Thus, before being assigned, value is modified by calling [dplyr::collect\(\)](#) on it and by applying [default_dtx\(\)](#).

Value

These operators invisibly return value. `collect.default()` simply return `x`.

Examples

```
# The alternate assignment operator performs three steps:
# 1) Collect results from dbplyr or dtplyr
library(dplyr)
library(data.table)
library(dtplyr)
library(svBase)
dtt <- data.table(x = 1:5, y = rnorm(5))
dtt |>
  mutate(x2 = x^2) |>
  select(x2, y) ->
  res

print(res)
class(res) # This is a data frame

dtt |>
  lazy_dt() |>
  mutate(x2 = x^2) |>
  select(x2, y) ->
  res

print(res)
class(res) # This is NOT a data frame

# Same pipeline, but assigning with %>%
dtt |>
  lazy_dt() |>
  mutate(x2 = x^2) |>
  select(x2, y) %>%
  res

print(res)
class(res) # res is the preferred data frame (data.table by default)

# 2) Convert data frame in the chosen format using default_dtx()
dtf <- data.frame(x = 1:5, y = rnorm(5))
class(dtf)
res %<-% dtf
class(res) # A data.table by default
# but it can be changed with options("SciViews.as_dtx")

# 3) If the zeallot syntax is used, make multiple assignment
c(X, Y) %<-% dtf # Variables of dtf assigned to different names
X
Y

# The %>% is meant to be used in pipelines, otherwise it does the same
```

as_dtx	<i>Coerce objects into data.frames, data.frames, data.tables, tibbles or matrices</i>
--------	---

Description

Objects are coerced into the desired class. For `as_dtx()`, the desired class is obtained from `getOption("SciViews.as_dtx")`, with a default value producing a `data.frame` object. If the data are grouped with `dplyr::group_by()`, the resulting data frame is also `dplyr::ungroup()`ed in the process.

Usage

```
as_dtx(x, ..., rownames = NULL, keep.key = TRUE, byref = FALSE)
as_dtrm(x, ..., rownames = NULL, keep.key = TRUE, byref = FALSE)
as_dtf(x, ..., rownames = NULL, keep.key = TRUE, byref = NULL)
as_dtt(x, ..., rownames = NULL, keep.key = TRUE, byref = FALSE)
as_dtbl(x, ..., rownames = NULL, keep.key = TRUE, byref = NULL)
default_dtx(x, ..., rownames = NULL, keep.key = TRUE, byref = FALSE)

## S3 method for class 'tbl_df'
as.matrix(x, row.names = NULL, optional = FALSE, ...)

as_matrix(x, rownames = NULL, ...)
```

Arguments

<code>x</code>	An object.
<code>...</code>	Further arguments passed to the methods (not used yet).
<code>rownames</code>	The name of the column with row names. If <code>NULL</code> , it is assessed from <code>getOption("SciViews.dtx.rownames")</code> .
<code>keep.key</code>	Do we keep the <code>data.table</code> key into a "key" attribute or do we restore <code>data.table</code> or <code>data.frame</code> key from the attribute?
<code>byref</code>	If <code>TRUE</code> , the object is modified by reference when converted into a <code>data.table</code> or a <code>data.frame</code> (faster, but not conventional). This is <code>FALSE</code> by default, or <code>NULL</code> if the argument does not apply in the context.
<code>row.names</code>	Same as <code>rownames</code> , but for base R functions.
<code>optional</code>	logical, If <code>TRUE</code> , setting row names and converting column names to syntactically correct names is optional.

Value

The coerced object. For `as_dtx()`, the coercion is determined from `getOption("SciViews.as_dtx")` which must return one of the four other `as_dt...` functions (`as_dtrm` by default). The `default_dtx()` does the same as `as_dtx()` if the object is a `data.frame`, a `data.frame`, a `data.table`, or a `tibble`, but it return the unmodified object for any other class (including subclassed data frames). This is a convenient function to force conversion only between those four objects classes.

Note

Use `as_matrix()` instead of `base::as.matrix()`: it has different default arguments to better account for rownames in `data.table` and `tibble`!

Examples

```
# A data.frame
dtf <- dtf(
  x = 1:5,
  y = rnorm(5),
  f = letters[1:5],
  l = sample(c(TRUE, FALSE), 5, replace = TRUE))

# Convert into a tibble
(dtbl <- as_dtbl(dtf))
# Since row names are trivial (1 -> 5), a .rownames column is not added

dtf2 <- dtf
rownames(dtf2) <- letters[1:5]
dtf2

# Now, the conversion into a tibble adds .rownames
(dtbl2 <- as_dtbl(dtf2))
# and data frame row names are set again when converted back to dtf
as_dtf(dtbl2)

# It also work for conversions data.frame <-> data.table
(dtt2 <- as_dtt(dtf2))
as_dtf(dtt2)
# or data.frame <-> data.trame
(dtrm2 <- as_dtrm(dtf2))
as_dtf(dtrm2)

# It does not work when converting a tibble or a data.table into a matrix
# with as.matrix()
as.matrix(dtbl2)
# ... but as_matrix() does the job!
as_matrix(dtbl2)

# The name for row in dtrm, dtt and dtbl is in:
# (data.frame's row names are converted into a column with this name)
getOption("SciViews.dtx.rownames", default = ".rownames")
```

```

# Convert into the preferred data frame object (data.frame by default)
(dtx2 <- as_dtx(dtf2))
class(dtx2)

# The default data frame object used:
getOption("SciViews.as_dtx", default = as_dtrm)

# default_dtx() does the same as as_dtx(),
# but it also does not change other objects
# So, it is safe to use whatever the object you pass to it
(dtx2 <- default_dtx(dtf2))
class(dtx2)
# Any other object than data.frame, data.table or tbl_df
# is not converted
res <- default_dtx(1:5)
class(res)
# No conversion if the data frame is subclassed
dtf3 <- dtf2
class(dtf3) <- c("subclassed", "data.frame")
class(default_dtx(dtf3))

# data.table keys are converted into a 'key' attribute and back
library(data.table)
setkey(dtt2, 'x')
haskey(dtt2)
key(dtt2)

(dtf3 <- as_dtf(dtt2))
attributes(dtf3)
# Key is restored when converted back into a data.table (also from a tibble)
(dtt3 <- as_dtt(dtf3))
haskey(dtt3)
key(dtt3)

# Grouped tibbles are ungrouped with as_dtbl() or as_dtx()/default_dtx()!
mtcars |> dplyr::group_by(cyl) -> mtcars_grouped
class(mtcars_grouped)
mtcars2 <- as_dtbl(mtcars_grouped)
class(mtcars2)

```

collect_dtx

Force computation of a lazy tidyverse object

Description

When {dplyr} or {tidyr} verbs are applied to a **data.table** or a database connection, they do not output data frames but objects like **dtplyr_step** or **tbl_sql** that are called lazy data frames. The actual process is triggered by using `as_dtx()`, or more explicitly with `dplyr::collect()` which coerces the result to a **tibble**. If you want the default {svBase} data frame object instead, use `collect_dtx()`, or if you want a specific object, use one of the other variants.

Usage

```
collect_dtx(x, ...)

collect_dtrm(x, ...)

collect_dtf(x, ...)

collect_dtt(x, ...)

collect_dtbl(x, ...)
```

Arguments

x A data.frame, data.table, tibble or a lazy data frame (dtplyr_step, tbl_sql...).

... Arguments passed on to methods for `dplyr::collect()`.

Value

A data frame (data.frame, data.table or tibble's tbl_df), the default version for `collect_dtx()`.

Examples

```
# Assuming the default data frame for svBase is a data.table
mtcars_dtt <- as_dtt(mtcars)
library(dplyr)
library(dtplyr)
# A lazy data frame, not a "real" data frame!
mtcars_dtt |> lazy_dt() |> select(mpg:disp) |> class()
# A data frame
mtcars |> select(mpg:disp) |> class()
# A data table
mtcars_dtt |> select(mpg:disp) |> class()
# A tibble, always!
mtcars_dtt |> lazy_dt() |> select(mpg:disp) |> collect() |> class()
# The data frame object you want, default one specified for svBase
mtcars_dtt |> lazy_dt() |> select(mpg:disp) |> collect_dtx() |> class()
```

data_dot_mechanism *The data-dot mechanism*

Description

The **data-dot** mechanism injects automatically `.data = .` in the call to a function when it detects it is necessary (most of the time, when `.data=` is missing, or a unnamed first argument is not suitable as `.data`, i.e., it is not a **data.frame**). This is useful to avoid having to avoid writing `.` "everywhere" in your functions when you use the explicit pipe operator `%>.%`, or with `. = ...` constructs. The data-dot mechanism may fail with an error message if it cannot inject `.` as `.data=`, or when `.` is not found. It may also be prohibited if the variable `.SciViews.implicit.data.dot` is set to `,0 FALSE` (see examples).

See Also

[prepare_data_dot\(\)](#)

Examples

```
# Here is how you create a data-dot function
my_subset <- function(.data = (.), i, j) {
  # This makes it a data-dot function
  if (!prepare_data_dot(.data))
    return(recall_with_data_dot())

  # Code of the function
  # Second argument (i here) must not be a data.frame to avoid confusion
  message(".env has ", paste(names(.env), collapse = ", "))
  .data[i, j]
}
dtf1 <- data.frame(x = 1:3, y = 4:6)
my_subset(dtf1, 1, 'y')
# If .data is in '.', it can be omitted
.= dtf1
my_subset(1, 'y')

# This mechanism is potentially confusing. You can inactivate it anywhere:
.SciViews.implicit.data.dot <- FALSE
# This time next call is wrong
try(my_subset(1, 'y'))
# You must indicate '.' explicitly in that case:
my_subset(., 1, 'y')
rm(.SciViews.implicit.data.dot) # Reactivate it
my_subset(1, 'y') # Implicit again
# Note that, if you have not defined '.' and try to use it, you got
# an error:
rm(.)
try(my_subset(1, 'y'))
```

dtx

Create a data frame (data.frame, base's data.frame, data.table or tibble's tbl_df)

Description

Create a data frame (data.frame, base's data.frame, data.table or tibble's tbl_df)

Usage

```
dtx(..., .name_repair = c("check_unique", "unique", "universal", "minimal"))

dtrm(..., .name_repair = c("check_unique", "unique", "universal", "minimal"))
```

```
dtbl(..., .name_repair = c("check_unique", "unique", "universal", "minimal"))

dtf(..., .name_repair = c("check_unique", "unique", "universal", "minimal"))

dtt(..., .name_repair = c("check_unique", "unique", "universal", "minimal"))
```

Arguments

... A set of name-value pairs. The content of the data frame. See [tibble::tibble\(\)](#) for more details on the way dynamic-dots are processed.

.name_repair The way problematic column names are treated, see also [tibble::tibble\(\)](#) for details.

Value

A data frame as a **data.frame** object for [dtrm\(\)](#), a **tbl_df** object for [dtbl\(\)](#), a **data.frame** for [dtf\(\)](#) or a **data.table** for [dtt\(\)](#).

Note

`data.frame`, `data.table` and `tibble`'s `tbl_df` do not use row names. However, you can add a column named `.rownames` (by default), or the name that is in `getOption("SciViews.dtx.rownames")` and it will be automatically set as row names when the object is converted into a `data.frame` with [as_dtf\(\)](#). For [dtf\(\)](#), just create a column of this name and it is directly used as row names for the resulting `data.frame` object.

See Also

[dtx_rows\(\)](#), [is_dtx\(\)](#), [collect_dtx\(\)](#)

Examples

```
dtrm1 <- dtrm(
  x = 1:5,
  y = rnorm(5),
  f = letters[1:5],
  l = sample(c(TRUE, FALSE), 5, replace = TRUE)
)
class(dtrm1)

dtbl1 <- dtbl(
  x = 1:5,
  y = rnorm(5),
  f = letters[1:5],
  l = sample(c(TRUE, FALSE), 5, replace = TRUE)
)
class(dtbl1)

dtf1 <- dtf(
  x = 1:5,
  y = rnorm(5),
```

```

    f = letters[1:5],
    l = sample(c(TRUE, FALSE), 5, replace = TRUE)
  )
  class(dtf1)

  dtt1 <- dtt(
    x = 1:5,
    y = rnorm(5),
    f = letters[1:5],
    l = sample(c(TRUE, FALSE), 5, replace = TRUE))
  class(dtt1)

  # Using dtx(), one construct the preferred data frame object
  # (a data.trame by default, can be changed with options(SciViews.as_dtx = ...))
  dtx1 <- dtx(
    x = 1:5,
    y = rnorm(5),
    f = letters[1:5],
    l = sample(c(TRUE, FALSE), 5, replace = TRUE))
  class(dtx1) # data.trame by default

  # Use dtx_rows() to easily create a data frame:
  dtx2 <- dtx_rows(
    ~x, ~y, ~f,
    1, 3, 'a',
    2, 4, 'b'
  )
  dtx2
  class(dtx2)

  # This is how you specify row names for dtf (data.frame)
  dtf(x = 1:3, y = 4:6, .rownames = letters[1:3])

```

dtx_rows

*Row-wise creation of a data frame***Description**

The presentation of the data (see examples) is easier to read than with the traditional column-wise entry in `dtx()`. This could be used to enter small tables in R, but do not abuse of it!

Usage

```
dtx_rows(...)
```

```
dtrm_rows(...)
```

```
dtf_rows(...)
```

```
dtt_rows(...)
```

```
dtbl_rows(...)
```

Arguments

... Specify the structure of the data frame by using formulas for variable names like `~x` for variable `x`. Then, use one argument per value in the data frame. It is possible to unquote with `!!` and to unquote-splice with `!!!`.

Value

A data frame of class **data.frame** for `dtrm_rows()`, **data.frame** for `dtf_rows()`, **data.table** for `dtb_rows()`, tibble **tbl_df** for `dtbl_rows()` and the default object with `dtx_rows()`.

Examples

```
df <- dtx_rows(
  ~x, ~y, ~group,
  1, 3, "A",
  6, 2, "A",
  10, 4, "B"
)
df
```

expr_funs

Find all functions in an Expression

Description

Return a character vector containing the name of all the functions in an expression or a call.

Usage

```
expr_funs(expr, max.names = -1L, unique = FALSE, exclude.names = NULL)
```

Arguments

<code>expr</code>	an expression or call from which the name of the function are to be extracted.
<code>max.names</code>	the maximum number of names to be returned. -1 indicates no limit (other than vector size limits).
<code>unique</code>	a logical value which indicates whether duplicate names should be removed from the value.
<code>exclude.names</code>	a character vector with names to exclude, or NULL for none.

Details

The c code is adapted from base R code `do_allnames()` (the later one allows to extract either variables, or variables + functions, but not functions alone).

Value

A character vector with the extracted function names.

Examples

```
# A formula where some names are simultaneously functions and variables
ff <- ~z(x, y, z, TRUE, "test", l = 4) + (y(z, x(1)) + y(2))
all.vars(ff)
all.names(ff, unique = TRUE)
expr_funs(ff)
expr_funs(ff, unique = TRUE)
expr_funs(ff, exclude.names = "~")
```

f_	<i>Convert a string into a formula and/or change the associated environment</i>
----	---

Description

This function is used to transform a name like 'var' into a formula ~var.

Usage

```
f_(x, env = parent.frame())
```

Arguments

x	Either a name (character string) or a formula
env	The environment to associate with the formula

Value

A formula

Examples

```
f_('var')
```

formula_masking	<i>Formula-masking interface using either standard evaluation, or formulas</i>
-----------------	--

Description

The formula-masking is a little bit like the data-masking used in dplyr, except it uses formulas for non-standard evaluation of the arguments, otherwise, it uses standard evaluation. It allows to use both approaches (standard and non-standard) within the same function. Moreover, the intention of the user and the possible non-standard evaluation of the arguments are clearer through formulas.

Usage

```
formula_masking(
  ...,
  .max.args = NULL,
  .must.be.named = FALSE,
  .make.names = FALSE,
  .no.se = FALSE,
  .no.se.msg = gettext("Standard evaluation is not allowed."),
  .envir = parent.frame(2L),
  .frame = parent.frame(),
  .verbose = .op$verbose
)
```

Arguments

<code>...</code>	Arguments to be processed by formula-masking.
<code>.max.args</code>	The maximum allowed arguments in
<code>.must.be.named</code>	If TRUE, all arguments must be named.
<code>.make.names</code>	If TRUE, unnamed arguments are named automatically.
<code>.no.se</code>	If TRUE, standard evaluation is not allowed.
<code>.no.se.msg</code>	The message to be used if standard evaluation is not allowed.
<code>.envir</code>	The environment where to expand formulas (possibly superseded by the environment attached to the first formula).
<code>.frame</code>	The frame where the focus in the calling stack should be set in error messages (not used yet).
<code>.verbose</code>	If TRUE, some additional information about formulas substitution is printed.

Value

A list with components:

- `dots`: A list of arguments, either expressions (if standard evaluation was used) or expressions extracted from the right-hand side of the formulas (if formulas were used).
- `env`: The environment where the expressions should be evaluated.
- `are_formulas`: TRUE if formulas were used, FALSE if standard evaluation was used.

See Also

`formula_select()`, `rlang::is_formula()`, `rlang::f_lhs()`, `rlang::f_rhs()`

Examples

```
# TODO...
```

formula_select	<i>Formula-select is similar to tidy-select in the tidyverse, but using formulas</i>
----------------	--

Description

The formula-select interface allows to give arguments in the tidy-select style withing formulas, or as standard-evaluated arguments. It thus combines both approaches within the same function, and makes it clearer which are the intention: standard evaluation, or non-standard evaluation when formulas are used.

Usage

```
formula_select(
  ...,
  .fast.allowed.funs = NULL,
  .max.args = NULL,
  .must.be.named = FALSE,
  .make.names = FALSE,
  .no.se = FALSE,
  .no.se.msg = gettext("Standard evaluation is not allowed."),
  .envir = parent.frame(2L),
  .frame = parent.frame()
)
```

Arguments

<code>...</code>	Arguments to be processed by formula-masking.
<code>.fast.allowed.funs</code>	A character vector of function names that are allowed for a fast treatment (usually though collapse functions). If any other function is used, the slower tidy-select mechanism is used, see tidyselect::eval_select() .
<code>.max.args</code>	The maximum allowed arguments in <code>...</code>
<code>.must.be.named</code>	If TRUE, all arguments must be named.
<code>.make.names</code>	If TRUE, unnamed arguments are named automatically.
<code>.no.se</code>	If TRUE, standard evaluation is not allowed.
<code>.no.se.msg</code>	The message to be used if standard evaluation is not allowed.

<code>.envir</code>	The environment where to expand formulas (possibly superseded by the environment attached to the first formula).
<code>.frame</code>	The frame where the focus in the calling stack should be set in error messages (not used yet).

Value

A list with components:

- `dots`: the processed arguments (formulas are turned into expressions)
- `are_formulas`: whether the arguments were formulas
- `env`: The environment where the expressions should be evaluated.
- `fastselect`: whether fast selection can be used

See Also

`formula_masking()`, <https://tidyselect.r-lib.org/articles/syntax.html> or `vignette("syntax", package = "tidyselect")` for a technical description of the rules of evaluation.

Examples

```
# TODO...
```

<code>fstat_functions</code>	<i>Fast (flexible and friendly) statistical functions (mainly from collapse) for matrix-like and data frame objects</i>
------------------------------	---

Description

The fast statistical function, or fast-flexible-friendly statistical functions are prefixed with "f". These vectorized functions supersede the no-f functions, bringing the capacity to work smoothly on matrix-like and data frame objects. Most of them are defined in the {collapse} package. For instance, base `mean()` operates on a vector, but not on a data frame. A matrix is recognized as a vector and a single mean is returned. On the contrary, `collapse::fmean()` calculates one mean per column. It does the same for a data frame, and it does so usually quicker than base functions. No need for `colMeans()`, a separate function to do so. Fast statistical functions also recognize grouping with `collapse::fgroup_by()`, `sgroup_by()` or `dplyr::group_by()` and calculate the mean by group in this case. Again, no need for a different function like `stats::ave()`. Finally, these functions also have a `TRA=` argument that computes, for instance, if `TRA = "-"`, $(x - f(x))$ very efficiently (for instance to calculate residuals by subtracting the mean). Another particularity is the `na.rm=` argument that is `TRUE` by default, while it is `FALSE` by default for `mean()`. These are generic functions with methods for **matrix**, **data.frame**, **grouped_df** and a **default** method used for simple numeric vectors. Most of them are defined in the {collapse} package, but there are a couple more here, together with an alternate syntax to replace `TRA=` with `%_f%`.

Usage

```
list_fstat_functions()

fn(x, ...)

fna(x, ...)

x %replacef% expr

x %replace_fillf% expr

x %-f% expr

x %+f% expr

x %--f% expr

x %/f% expr

x %/*100f% expr

x %*f% expr

x %modf% expr

x %-modf% expr
```

Arguments

x	A numeric vector, matrix, data frame or grouped data frame (class 'grouped_df').
...	Further arguments passed to the method, like <code>w=</code> , a numeric vector of (non-negative) weights that may contain missing values, or <code>TRA=</code> , a quoted operator indicating the transformation to perform: "replace" to get a vector of same size of x with results, "replace_fill" idem but also replace missing data, "-" to subtract, "+" to add, "--" to subtract and add the global statistic, "/" to divide, "%" to divide and multiply by 100 (percent), "*" to multiply, "%%" to take the modulus (remainder from division by the statistic) and "-%" to subtract modulus (i.e., to floor the data by the statistic), see collapse::TRA() . Also <code>na.rm=</code> , a logical indicating if we skip missing values in x if TRUE (by default). If FALSE for any missing data in x, NA is returned. For details and other arguments, see the corresponding help page in the collapse package.
expr	The expression to evaluate as RHS of the <code>%_f%</code> operators.

Value

The number of all observations for `fn()` or the number of missing observations for `fna()`. `list_fstat_functions()` returns a list of all the known fast statistical functions.

Note

The page [collapse::fast-statistical-functions](#) gives more details. `fn()` count all observations, including NAs, `fna()` counts only NAs, where `collapse::fnobs()` counts non-missing observations. Instead of `TRA=` one can use the `%__f%` functions where `__` is `replace`, `replace_fill`, `-`, `+`, `++`, `/`, `/*100` for `TRA="%", *`, `mod` for `TRA="%%"`, or `-mod` for `TRA="-%%"`. See example.

Examples

```
library(collapse)
data(iris)
iris_num <- iris[, -5] # Only numerical variables
mean(iris$Sepal.Length) # OK, but mean(iris_num) does not work)
colMeans(iris_num)
# Same
fmean(iris_num)
# Idem, but mean by group for all 4 numerical variables
iris |> fgroup_by(Species) |> fmean()
# Residuals (x - mean(x)) by group
iris |> fgroup_by(Species) |> fmean(TRA = "-")
# The same calculation, in a little bit more expressive way
iris |> fgroup_by(Species) %-f% fmean()
# or:
iris_num %-f% fmean(g = iris$Species)
```

 gettext_

Translate messages in a different language than the one in the R session

Description

Message translation in R is obtained with `base::gettext()` or `base::ngettext()`. But, there is no way to specify that one needs translated messages in a different language than the current one in R. Here are alternate functions that have an additional `lang=` argument allowing to do so. If the `lang=` argument is not provided in the call, they use the language defined in the R session. It is useful to define a different language, for instance, to keep R error and warning messages in English, but to generate translation for tables and figures in a different language in a report.

Usage

```
gettext_(..., domain = NULL, trim = TRUE, lang = get_sciviews_lang())

gettextf_(fmt, ..., domain = NULL, trim = TRUE, lang = get_sciviews_lang())

ngettext_(n, msg1, msg2, domain = NULL)

get_language(unset = "en")

set_language(lang, unset = get_language())
```

```

get_sciviews_lang(unset = get_language())

set_sciviews_lang(lang, unset = "en")

check_lang(lang, allow_uppercase = FALSE)

test_gettext_lang(lang = get_sciviews_lang(), n = 1)

```

Arguments

...	one of more character vectors.
domain	the 'domain' for the translation, a character string or NULL; see base::gettext() for more details. For <code>ngettext_()</code> , it should combine the domain and the lang, like "domain/lang" (e.g., "NULL/en_US" or "R-stats/fr"). This is a workaround to define the language, because base version of that function does not allow additional arguments and we have to remain compatible here.
trim	logical indicating if the white space trimming should happen.
lang	the target language (could be two lowercase letters, e.g., "en" for English, "fr" for French, "de" for German, etc.). One can also further specify variants, e.g., "en_US", or "en_GB", or even "fr_FR.UTF-8". For <code>get_sciviews_lang()</code> and <code>set_sciviews_lang()</code> , it is the secondary language. For the other functions, it is the R session language that is used by default. One can specify the alternate SciViews language globally with either the <code>SCIVIEWS_LANG</code> environment variable, or with the R option <code>SciViews_lang</code> , but it is a better practice to use <code>set_sciviews_lang()</code> in the R session. If missing, NULL, or "", the default is used from <code>unset</code> . For the SciViews language, uppercase letters are accepted, and they mean "translate more" (typically, factor and ordered levels are also translated, for instance).
fmt	a character vector of format strings, each of up to 8192 bytes.
n	a non-negative integer.
msg1	the message to be used in English for <code>n = 1</code> .
msg2	the message to be used in English for <code>n = 0, 2, 3, ...</code>
unset	The default language to use if not defined yet, "en" (English) by default for regular R language, and the currently defined R language for the alternate SciViews language.
allow_uppercase	logical indicating if uppercase letters are allowed for the first two letters of the language code (FALSE by default, but should be TRUE for the SciViews language).

Details

To prepare your package for translation with these functions, you should import `gettext_()`, `gettextf_()` and `ngettext_()` from `svBase`. Then, you define `gettext <- gettext_`, `gettextf <- gettextf_` and `ngettext <- ngettext_` somewhere in your package. To prepare translation

strings, you change the current directory of your R console to the base folder of the sources of your package and you issue `tools::update_pkg_po(".")` in R (or you include it in the tests: for an example, see `tests/testthat/test-translations.R` in the source of the `svBase` package). Then, you perform the translation for different languages with, say, **poEdit**, and recompile your package.

Value

A character vector with translated messages for the `gettext...` functions.

`test_gettext_lang()` just serves to test and demonstrate the translation in a given language.

`get_language()` and `get_sciviews_lang()` return the current language. `set_language()` and `set_sciviews_lang()` return the previous language invisibly (with an attribute `attr(*, "ok")` a **logical** indicating success.

`check_lang()` validates a `lang=` argument by returning `TRUE` invisibly, otherwise, it `stop()`s.

See Also

[base::gettext\(\)](#), [base::gettextf\(\)](#), [tools::update_pkg_po\(\)](#).

Examples

```
get_language()
get_sciviews_lang()

old_lang <- set_language("fr") # Switch to French for R language
old_sv_lang <- set_sciviews_lang("fr") # Switch to French for SciViews also

# R look for messages to be translated into gettext() calls, not gettext_()
# So, rename accordingly in your package:
gettext <- svBase::gettext_
gettextf <- svBase::gettextf_
ngettext <- svBase::ngettext_

# Retrieve strings in same language
gettext("empty model supplied", "incompatible dimensions",
  domain="R-stats", lang = "fr")

# Retrieve strings in different languages
gettext("empty model supplied", "incompatible dimensions",
  domain="R-stats", lang = "en")
gettext("empty model supplied", "incompatible dimensions",
  domain="R-stats", lang = "de")

# Try to get strings translated in an unknown language (just return the strings)
gettext("empty model supplied", "incompatible dimensions",
  domain="R-stats", lang = "xx")

# Test with some translations from the svMisc package itself:
svBase::test_gettext_lang()
svBase::test_gettext_lang("fr", n = 1)
svBase::test_gettext_lang("fr", n = 2)
svBase::test_gettext_lang("en", n = 1)
```

```
svBase::test_gettext_lang("en", n = 2)

# Restore original languages
set_language(old_lang)
set_sciviews_lang(old_sv_lang)
rm(old_lang, old_sv_lang, gettext, gettextf, ngettext)

# In case you must check if a lang= argument gets a correct value:
check_lang("en")
check_lang("en_US.UTF-8")
# Only for SciViews language!
check_lang("FR", allow_uppercase = TRUE)
# But these are incorrect
try(check_lang("EN"))
try(check_lang(""))
try(check_lang(NA_character_))
try(check_lang(NULL))
try(check_lang(42))
try(check_lang(c("en", "fr")))
try(check_lang("Fr", allow_uppercase = TRUE))
```

is_dtx	<i>Test if the object is a data frame (data.trame, data.frame, data.table or tibble)</i>
--------	--

Description

Test if the object is a data frame (data.trame, data.frame, data.table or tibble)

Usage

```
is_dtx(x, strict = TRUE)

is_dtrm(x, strict = TRUE)

is_dtf(x, strict = TRUE)

is_dtt(x, strict = TRUE)

is_dtbl(x, strict = TRUE)
```

Arguments

x	An object
strict	Should this be strictly the corresponding class TRUE, by default, or could it be subclassed too (FALSE). With strict = TRUE, the grouped_df tibbles and grouped_ts tsibbles are also considered (tibbles or tsibbles where <code>dplyr::group_by()</code> was applied).

Value

These functions return TRUE if the object is of the correct class, otherwise they return FALSE. `is_dtx()` return TRUE if `x` is one of a `data.frame`, `data.table`, `tibble`, or `data.frame`.

Examples

```
# data(mtcars)
is_dtf(mtcars) # TRUE
is_dtx(mtcars) # Also TRUE
is_dtt(mtcars) # FALSE
is_dtbl(mtcars) # FALSE
is_dtrm(mtcars) # FALSE
# but...
is_dtt(as_dtt(mtcars)) # TRUE
is_dtx(as_dtt(mtcars)) # TRUE
is_dtbl(as_dtbl(mtcars)) # TRUE
is_dtx(as_dtbl(mtcars)) # TRUE
is_dtrm(as_dtrm(mtcars)) # TRUE
is_dtx(as_dtrm(mtcars)) # TRUE
is_dtx(as_dtbl(mtcars) |> dplyr::group_by(cyl)) # TRUE (special case)

is_dtx("some string") # FALSE
```

labelise

Set label (and units)

Description

Set the label, as well as the units attributes to an object. The label can be used for better display as plot axes labels, or as table headers in pretty-formatted R outputs. The units are usually associated to the label in axes labels for plots. `cl()` is a shortcut for concatenate (`c()`) and `labelise()`.

Usage

```
labelise(x, label, units = NULL, as_labelled = FALSE, ...)

labelize(x, label, units = NULL, as_labelled = FALSE, ...)

## Default S3 method:
labelise(x, label, units = NULL, as_labelled = FALSE, ...)

## S3 method for class 'data.frame'
labelise(x, label, units = NULL, as_labelled = FALSE, self = TRUE, ...)

cl(..., label = NULL, units = NULL, as_labelled = FALSE)

unlabelise(x, ...)
```

```

unlabelize(x, ...)

## Default S3 method:
unlabelise(x, ...)

## S3 method for class 'data.frame'
unlabelise(x, ..., self = FALSE)

```

Arguments

<code>x</code>	An object.
<code>label</code>	The character string to set as <code>label</code> attribute to <code>x</code> .
<code>units</code>	The units (optional) as a character string to set for <code>x</code> .
<code>as_labelled</code>	Should the object be converted as a labelled S3 object (no by default)? If you don't make labelled objects, subsetting the data will lead to a lost of <code>label</code> and <code>units</code> attributes for all variables. On the other hand, labelled objects are not always correctly handled by R code.
<code>...</code>	Further arguments: items to be concatenated in a vector using <code>c(...)</code> for <code>cl()</code> . Columns where <code>label</code> should be eliminated for <code>unlabelise()</code> .
<code>self</code>	Do we label the <code>data.frame</code> itself (<code>self = TRUE</code> , by default) or variables within that <code>data.frame</code> (<code>self = FALSE</code>)? In the later case, <code>label=</code> and <code>units=</code> must be either lists or character vectors of the same length as <code>x</code> , or be named with the names of several or all <code>x</code> variables. For <code>unlabelise()</code> , the default is <code>self = FALSE</code> .

Details

The same mechanism as the one used in package **Hmisc** is used here. However, **Hmisc** always add the **labelled** class to an object, while here, this is optional. Setting this class make the object more nicely printed, and subsettable without losing these attributes. But it conflicts with a class of the same name in package **haven**, used for other purposes. So, here, one can also opt not to set it, using `as_labelled = FALSE`.

Value

The `x` object plus a `label` attribute, and possibly, a `units` attribute.

See Also

[label\(\)](#), [base::units\(\)](#)

Examples

```

# Labelise a vector:
x <- 1:10
x <- labelise(x, label = "A suite of integers", units = "cm")
x
# or, in a single operation:
x <- cl(1:10, label = "A suite of integers", units = "cm")

```

```

x
# Not adding the labelled class:
x <- cl(1:10, label = "Integers", units = "cm", as_labelled = FALSE)
x
# Unlabelling a labelled object
unlabelise(x)

# Labelise a data.frame
iris <- labelise(datasets::iris, "The famous iris dataset")
unlabelise(iris)
# but if you indicate self = FALSE, you can labelise variables within the
# data.frame (use a list or character vector of same length as x, or a
# named list or character vector):
iris <- labelise(iris, self = FALSE, label = list(
  Sepal.Length = "Length of the sepals",
  Petal.Length = "Length of the petals"
), units = c(rep("cm", 4), NA))
iris <- unlabelise(iris, self = FALSE)

```

prepare_data_dot

Prepare a data_dot function

Description

Prepare a function that uses the data-dot mechanism. In case the argument (usually, `.data = (.)`) is missing or is not a data frame in a call to a "data-dot" function, it is recalled after injection `.` as first argument.

Usage

```
prepare_data_dot(x, is_top_call = TRUE)
```

```
prepare_data_dot2(x, y, is_top_call = TRUE)
```

```

recall_with_data_dot(
  call,
  arg = ".data",
  value = quote((.)),
  env = parent.frame(2L),
  abort_msg = gettextf("`%s` must be a 'data.frame'.", arg),
  abort_msg2 = gettext("Implicit data-dot (.) not permitted"),
  abort_msg3 = gettext("Data-dot mechanism activated, but no `.` object found."),
  abort_msg4 =
    gettext("{.code {deparse(call0)}} rewritten as:\n{.code {deparse(call)}}"),
  abort_msg5 = gettextf("Define `.` before this call, or provide `%s` explicitly.",
    arg),
  abort_msg6 = gettextf("See {.help svBase::data_dot_mechanism} for more infos."),
  abort_frame = parent.frame()
)

```

```

)

recall_with_data_dot2(
  call,
  arg = "x",
  arg2 = "y",
  value = quote(.),
  env = parent.frame(2L),
  abort_msg = gettextf("`%s` and `%s` must both be 'data.frame'." , arg, arg2),
  abort_msg2 = gettext("Implicit data-dot (.) not permitted"),
  abort_msg3 = gettext("Data-dot mechanism activated, but no `.` object found."),
  abort_msg4 =
    gettext("{.code {deparse(call0)}} rewritten as:\n{.code {deparse(call)}}"),
  abort_msg5 = gettextf("Define `.` before this call, or provide `%s` explicitly.",
    arg),
  abort_msg6 = gettextf("See {.help svBase::data_dot_mechanism} for more infos."),
  abort_frame = parent.frame()
)

```

Arguments

<code>x</code>	An argument to check.
<code>is_top_call</code>	A logical indicating if this is a top-level call (TRUE by default) that should be focused in the call stack in case of an error.
<code>y</code>	A second argument.
<code>call</code>	A call object, usually a function call. Could be omitted, and in this case, <code>sys.call()</code> is invoked.
<code>arg</code>	The name of the argument to inject, usually <code>'data'</code> (default). For <code>prepare_data_dot2()</code> , it is <code>x</code> by default
<code>value</code>	The value to inject, usually the symbol <code>.</code> (default).
<code>env</code>	The environment where the evaluation of the data-dot-injected call should be evaluated (by default, <code>parent.frame(2L)</code> , should rarely be changed).
<code>abort_msg</code>	The message to use in case the <code>'data'</code> argument is wrong.
<code>abort_msg2</code>	An additional message to append to the error message in case data-dot-injection is not permitted (when <code>.SciViews.implicit.data.dot != TRUE</code> , see example).
<code>abort_msg3</code>	An error message when <code>.</code> is not found.
<code>abort_msg4</code>	Before and after data-dot replacement message.
<code>abort_msg5</code>	An additional explanation when <code>.</code> is not found.
<code>abort_msg6</code>	A hint to read the documentation of the data-dot mechanism.
<code>abort_frame</code>	The environment to use for the error message, by default, the caller environment (should rarely be changed).
<code>arg2</code>	The name of the second argument, <code>y</code> by default.

Details

The call is **not** checked if it is a correct function call. When called from within a function, passing `sys.call()` as call, it should be always correct. `prepare_data_dot2()` and `recall_with_data_dot2()` are similar, but for functions that have two first arguments that must be data frames, generally called `x` and `y`.

Value

TRUE if the preparation is correct for `prepare_data_dot()`, FALSE otherwise. The result from evaluating the data-dot-injected call for `recall_with_data_dot()`.

See Also

[data_dot_mechanism](#)

Examples

```
library(svBase)
# Use this (rename) function to get extra-info in the error message about the
# data-dot mechanism automatically
stop <- stop_

# Here is how you create a data-dot function
my_subset <- function(.data = (.), i, j) {
  if (!prepare_data_dot(.data))
    return(recall_with_data_dot())

  if (!is.numeric(i))
    stop("{.arg i} must be numeric") # Use this function
  if (i == 1)
    message(".env has ", paste(names(.env), collapse = ", "))
  .data[i, j]
}

dtf1 <- data.frame(x = 1:3, y = 4:6)
# The message shows the objects available in the function environment
my_subset(dtf1, 1, 'y')
# This is wrong
try(my_subset(dtf1, 'y'))
.= dtf1
my_subset(2, 'y')
# Error message with indication that the data-dot mechanism is activated
try(my_subset('y'))
# Data-dot mechanism not activate, but dot object used
try(my_subset(., 'y'))
# Wrong .data=
try(my_subset(.data = 'y'))

# When data-dot is not permitted...
.SciViews.implicit.data.dot <- FALSE
try(my_subset(2, 'y'))
```

```
rm(.SciViews.implicit.data.dot)

# When `.` is not found...
rm(.)
try(my_subset(2, 'y'))
```

process_error_msg	<i>Process error message (replace it with a better version)</i>
-------------------	---

Description

The tra list indicates which (partial) error message should be replaced by a more elaborate version, using the `cli::cli_abort()` syntax.

Usage

```
process_error_msg(msg, tra, fixed = TRUE)
```

Arguments

msg	The original error message.
tra	The translation list, where the names are the partial error and the content is the new error message(s).
fixed	Should a fixed or a regular expression comparison be done between the names of tra and the msg? Defaults to TRUE, which is faster. In case you need different behavior for different items, you can also supply a logical vector of the same length as tra.

Value

The replaced error message.

Examples

```
dir <- "/temp/dir"
file <- "~/some file.txt"
tra = list(
  "Directory not found" = c("The directory {.path {dir}} does not exist.",
    i = "Please create it first."),
  "file does not exist" = c("The file {.file {file}} does not exist.",
    i = "Check the path and try again.")
)
msg <- "The file does not exist"
try(cli::cli_abort(process_error_msg(msg, tra)))

msg <- "Directory not found on this machine"
try(cli::cli_abort(process_error_msg(msg, tra)))
```

retarget	<i>Retarget a formula to a different environment</i>
----------	--

Description

Change the environment of a formula to a specified environment.

Usage

```
retarget(x, env = parent.frame())
```

Arguments

x	A formula (or a quosure), or a list of formulas.
env	The environment to which the formula(s) should be retargeted. Defaults to the parent frame.

Value

The formula or list of formulas with the new environment set.

Examples

```
# A rather inefficient way to build a formula... for the sake of the demo!
make_formula <- function(x)
  as.formula(x)
f <- make_formula("x ~ log(y) + z")
f
f <- retarget(f)
f
# OK, but the environment associated to this formula is...
# the environment of the function:
rlang::f_env(f)

# With a list of formulas (local() creates a new environment):
fl <- local(list(y ~x^2, z~ sqrt(cos(x^2) + sin(x^2))))
fl # Not in GlobalEnv
retarget(fl) # Retargeted to GlobalEnv
```

section	<i>Create a section in a list (collection of functions and other objects).</i>
---------	--

Description

A section tags a list to sort its items. It is particularly useful when you create a collection of function (or other objects) to ease the access to these functions. Sections are displayed in printed and "str"ed versions of the list and are also functions that cut the list to the section content only. `get_section()` is the workhorse function that does the section extraction.

Usage

```
section(obj, title)

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

## S3 method for class 'section'
str(object, ...)

get_section(x, title)
```

Arguments

<code>obj</code>	A list object.
<code>title</code>	The title of the section. It must match the name of the list item. For a title "My section title", the name must be "0__MY_SECTION_NAME__" that is both a syntactically correct name and something that emphasizes the entry as a title.
<code>x</code>	A list containing the section
<code>...</code>	Further arguments (not used yet)
<code>object</code>	A list to use for section extraction

Value

A function that is able to extract the corresponding section from the list.

Examples

```
#TODO...
```

speedy_functions	<i>Speedy functions (mainly from collapse and data.table) to manipulate data frames</i>
------------------	---

Description

[Deprecated]

These function are deprecated to the benefit of the functions whose name ends with an underscore _ (e.g., sselect() -> `svTidy::select_()`) in the svTidy package.

The Tidyverse defines a coherent set of tools to manipulate data frames that use a non-standard evaluation and sometimes require extra care. These functions, like `dplyr::mutate()` or `dplyr::summarise()` are defined in the {dplyr} and {tidyr} packages. The {collapse} package provides a couple of functions with similar interface, but with different and much faster code. For instance, `collapse::fselect()` is similar to `dplyr::select()`, or `collapse::fsummarise()` is similar to `dplyr::summarise()`. Not all functions are implemented, arguments and argument names differ, and the behavior may be very different, like `collapse::frename()` which uses `old_name = new_name`, while `dplyr::rename()` uses `new_name = old_name`! The speedy functions all are prefixed with an "s", like `smutate()`, and build on the work initiated in {collapse} to propose a series of paired functions with the tidy ones. So, `smutate()` and `dplyr::mutate()` are "speedy" and "tidy" counterparts and they are used in a very similar, if not identical way. This notation using a "s" prefix is there to draw the attention on their particularities. Their classes are **function** and **speedy_fn**. Avoid mixing tidy, speedy and non-tidy/speedy functions in the same pipeline. **This is a global page to present all the speedy functions in one place.** It is not meant to be a clear and detailed help page of all individual "s" functions. Please, refer to the corresponding help page of the non-"s" paired function for more details! You can use the {svMisc}'s `?.smutate` syntax to go to the help page of the non-"s" function with a message.

Usage

```
list_speedy_functions()

sgroup_by(.data, ...)

sungroup(.data, ...)

srename(.data, ...)

srename_with(.data, .fn, .cols = everything(), ...)

sfilter(.data, ...)

sfilter_ungroup(.data, ...)

sselect(.data, ...)

smutate(.data, ..., .keep = "all")
```

```
smutate_ungroup(.data, ..., .keep = "all")

stransmute(.data, ...)

stransmute_ungroup(.data, ...)

ssummarise(.data, ...)

sfull_join(x, y, by = NULL, suffix = c(".x", ".y"), copy = FALSE, ...)

sleft_join(x, y, by = NULL, suffix = c(".x", ".y"), copy = FALSE, ...)

sright_join(x, y, by = NULL, suffix = c(".x", ".y"), copy = FALSE, ...)

sinner_join(x, y, by = NULL, suffix = c(".x", ".y"), copy = FALSE, ...)

sbind_rows(..., .id = NULL)

scount(
  x,
  ...,
  wt = NULL,
  sort = FALSE,
  name = NULL,
  .drop = dplyr::group_by_drop_default(x),
  sort_cat = TRUE,
  decreasing = FALSE
)

stally(
  x,
  wt = NULL,
  sort = FALSE,
  name = NULL,
  sort_cat = TRUE,
  decreasing = FALSE
)

sadd_count(
  x,
  ...,
  wt = NULL,
  sort = FALSE,
  name = NULL,
  .drop = NULL,
  sort_cat = TRUE,
  decreasing = FALSE
)
```

```
)

sadd_tally(
  x,
  wt = NULL,
  sort = FALSE,
  name = NULL,
  sort_cat = TRUE,
  decreasing = FALSE
)

sbind_cols(
  ...,
  .name_repair = c("unique", "universal", "check_unique", "minimal")
)

sarrange(.data, ..., .by_group = FALSE)

spull(.data, var = -1, name = NULL, ...)

sdistinct(.data, ..., .keep_all = FALSE)

sdrop_na(data, ...)

sreplace_na(data, replace, ...)

spivot_longer(data, cols, names_to = "name", values_to = "value", ...)

spivot_wider(data, names_from = name, values_from = value, ...)

suncount(data, weights, .remove = TRUE, .id = NULL)

sunite(data, col, ..., sep = "_", remove = TRUE, na.rm = FALSE)

sseparate(
  data,
  col,
  into,
  sep = "[^[:alnum:]]+",
  remove = TRUE,
  convert = FALSE,
  ...
)

sseparate_rows(data, ..., sep = "[^[:alnum:]].]", convert = FALSE)

sfill(data, ..., .direction = c("down", "up", "downup", "updown"))
```

```

sextract(
  data,
  col,
  into,
  regex = "([[:alnum:]]+)",
  remove = TRUE,
  convert = FALSE,
  ...
)

```

Arguments

<code>.data</code>	A data frame (data.frame, data.table or tibble's <code>tbl_df</code>)
<code>...</code>	Arguments dependent to the context of the function and most of the time, not evaluated in a standard way (cf. the tidyverse approach).
<code>.fn</code>	A function to use.
<code>.cols</code>	The list of the column where to apply the transformation. For the moment, only all existing columns, which means <code>.cols = everything()</code> is implemented
<code>.keep</code>	Which columns to keep. The default is "all", possible values are "used", "unused", or "none" (see dplyr::mutate()).
<code>x</code>	A data frame (data.frame, data.table or tibble's <code>tbl_df</code>).
<code>y</code>	A second data frame.
<code>by</code>	A list of names of the columns to use for joining the two data frames.
<code>suffix</code>	The suffix to the column names to use to differentiate the columns that come from the first or the second data frame. By default it is <code>c(".x", ".y")</code> .
<code>copy</code>	This argument is there for compatibility with the "t" matching functions, but it is not used here.
<code>.id</code>	The name of the column for the origin id, either names if all other arguments are named, or numbers.
<code>wt</code>	Frequency weights. Can be NULL or a variable. Use data masking.
<code>sort</code>	If TRUE largest group will be shown on top.
<code>name</code>	The name of the new column in the output (n by default, and no existing column must have this name, or an error is generated). ⁴
<code>.drop</code>	Are levels with no observations dropped (TRUE by default).
<code>sort_cat</code>	Are levels sorted (TRUE by default).
<code>decreasing</code>	Is sorting done in decreasing order (FALSE by default)?
<code>.name_repair</code>	How should the name be "repaired" to avoid duplicate column names? See dplyr::bind_cols() for more details.
<code>.by_group</code>	Logical. If TRUE rows are first arranger by the grouping variables in any. FALSE by default.
<code>var</code>	A variable specified as a name, a positive or a negative integer (counting from the end). The default is -1 and returns last variable.
<code>.keep_all</code>	If TRUE keep all variables in <code>.data</code> .

<code>data</code>	A data frame, or for <code>replace_na()</code> a vector or a data frame.
<code>replace</code>	If <code>data</code> is a vector, a unique value to replace NAs, otherwise, a list of values, one per column of the data frame.
<code>cols</code>	A selection of the columns using tidy-select syntax, see <code>tidyr::pivot_longer()</code> .
<code>names_to</code>	A character vector with the name or names of the columns for the names.
<code>values_to</code>	A string with the name of the column that receives the values.
<code>names_from</code>	The column or columns containing the names (use tidy selection and do not quote the names).
<code>values_from</code>	Idem for the column or columns that contain the values.
<code>weights</code>	A vector of weight to use to "uncount" data.
<code>.remove</code>	If TRUE, and <code>weights</code> is the name of a column, that column is removed from data.
<code>col</code>	The name quoted or not of the new column with united variable.
<code>sep</code>	Separator to use between values for united or separated columns.
<code>remove</code>	If TRUE the initial columns that are separated are also removed from data.
<code>na.rm</code>	If TRUE, NAs are eliminated before uniting the values.
<code>into</code>	Name of the new column to put separated variables. Use NA for items to drop.
<code>convert</code>	If 'TRUE' resulting values are converted into numeric, integer or logical.
<code>.direction</code>	Direction in which to fill missing data: "down" (by default), "up", or "downup" (first down, then up), "updown" (the opposite).
<code>regex</code>	A regular expression used to extract the desired values (use one group with (and) for each element of <code>into</code>).

Value

See corresponding "non-s" function for the full help page with indication of the return values.

Note

The `ssummarise()` function does not support `n()` as does `dplyr::summarise()`. You can use `fn()` instead, but then, you must give a variable name as argument. The `fn()` alternative can also be used in `dplyr::summarise()` for homogeneous syntax between the two. From {dplyr}, the `dplyr::slice()` and `slice_xxx()` functions are not added yet because they are not available for {dbplyr}. Also `dplyr::anti_join()`, `dplyr::semi_join()` and `dplyr::nest_join()` are not implemented yet. From {tidyr} `tidyr::expand()`, `tidyr::chop()`, `tidyr::unchop()`, `tidyr::nest()`, `tidyr::unnest()`, `tidyr::unnest_longer()`, `tidyr::unnest_wider()`, `tidyr::hoist()`, `tidyr::pack()` and `tidyr::unpack()` are not implemented yet.

Examples

```
# TODO...
```

stop_	<i>Enhanced stop() and warning()</i>
-------	--------------------------------------

Description

stop_() is a wrapper around `cli::cli_abort()` that provides more control on the stop message and also provides nice formatting and glue interpolation. This version uses `gettext()` to translate the message, on the contrary to `cli::cli_abort()`. `warning_()` is similar to `warning()` but uses `call. = FALSE` by default. Finally, `stop_top_call()` allows to tag from where an error should be reported (see examples).

Usage

```
stop_(
  ...,
  call. = FALSE,
  domain = NULL,
  class = NULL,
  call = stop_top_call(2L),
  envir = parent.frame(),
  last_call = sys.call(-1L)
)

warning_(
  ...,
  call. = FALSE,
  immediate. = FALSE,
  noBreaks. = FALSE,
  domain = NULL
)

stop_top_call(nframe = 2L)

object_info(x)
```

Arguments

...	One or more character strings with the error or warning message(s). Name them '*' =, 'i' =, 'v' =, 'x' = or '!' = to format message items. First message item is considered to be '!' by default.
call.	Logical, whether to include the call in the warning message. Not used for stop_().
domain	see <code>gettext()</code> . If NA, messages will not be translated.
class	The subclass of the error condition message
call	The execution environment of a currently running function where the error should be reported from.

<code>envir</code>	The environment where to evaluate the glue expressions.
<code>last_call</code>	The last call issued by the user, used to check if a dot (.) object was invoked.
<code>immediate.</code>	Logical, whether to issue the warning immediately even if <code>getOption("warn") <= 0</code> . Note that this is not respected for condition objects.
<code>noBreaks.</code>	logical, indicating as far as possible the message should be output as a single line when <code>options(warn = 1)</code> .
<code>nframe</code>	The number of frames to go up the call stack to find the top call for stop condition messages.
<code>x</code>	An R object to describe.

Value

`stop_()` and `warning_()` are invoked for their side-effects, but `stop_()` actually stops execution of the current code. `stop_top_call()` return the top call to be used for stop condition messages.

See Also

[stop\(\)](#), [warning\(\)](#), [cli::cli_abort\(\)](#), [gettext\(\)](#)

Examples

```
# If you want to include the error messages in the translation strings in
# your package, you have to rename `stop_()` into `stop()` and `warning_()`
# into `warning()`, because [tools::xgettext2pot()] will only pick up the
# later ones.
stop <- stop_
warning <- warning_

# Call not integrated by default now
warning("just a test")
warning("just a test", call. = TRUE)

if (FALSE) {# Avoid running code that generates errors automatically
# Correctly formatted stop messages
n <- "some text"
stop("{.var n} must be a numeric vector",
  x = "You've supplied a {.cls {class(n)}} vector.")

n <- 1:18
stop("{.var n} must be a scalar numeric:",
  i = "There {?is/are} {length(n)} element{?s}.",
  x = "Indicate a single numeric, not: {n}.")

# When issued from within a function, the function call is used in the error
test1 <- function(x) {
  stop("{.var n} must be a scalar numeric:",
    i = "There {?is/are} {length(x)} element{?s}.")
}
test1(1:3)
```

```

# If another function calls `test1()`, error is still reported from test1:

test2 <- function(x) {
  test1(x)
}
test2(1:3)

# In such a case, it is better to report the error from `test2()`.
# You can do that by stating `._top_call_. <- TRUE` in the body of `test2()`.
test2 <- function(x) {
  .__top_call__. <- TRUE
  test1(x)
}
test2(1:3)
}# End of if(FALSE)

rm(stop, warning)

```

subsettable

Define a function as being 'subsettable' using \$ operator

Description

In case a textual argument allows for selecting the result, for instance, if `plot()` allows for several charts that you can choose with a `type=` or `which=`, making the function 'subsettable' also allows to indicate `fun$variant()`. The `subsettable_type2` variant is faster for only internal implementation of various types, while `subsettable_type` first searches for a function with name `.<generic>$type()`. See examples.

Usage

```

## S3 method for class 'subsettable_type'
x$name

## S3 method for class 'subsettable_type2'
x$name

## S3 method for class 'subsettable_which'
x$name

name_function_type(fun, method = NULL, type)

list_types(fun, method = NULL)

get_type(fun, method = NULL, type, stop.if.missing = TRUE)

args_type(fun, method = NULL, type)

```

```
## S3 method for class 'subsettable_type'
.DollarNames(x, pattern = "")
```

Arguments

x	A subsettable_type function.
name	The value to use for the type= argument.
fun	The name of the function (as a scalar string).
method	An optional method name (as a scalar string).
type	The type to select (as a scalar string).
stop.if.missing	If TRUE (default), an error is raised if the requested type is not found. If FALSE, NULL is returned instead.
pattern	A regular expression. Only matching names are returned.

Examples

```
# Simple selection of type with a switch inside the function itself
foo <- structure(function(x, type = c("histogram", "boxplot"), ...) {
  type <- match.arg(type, c("histogram", "boxplot"))
  switch(type,
    histogram = hist(x, ...),
    boxplot = boxplot(x, ...),
    stop("unknow type")
  )
}, class = c("function", "subsettable_type2"))
foo

# This function can be used as usual:
foo(rnorm(50), type = "histogram")
# ... but also this way:
foo$histogram(rnorm(50))
foo$boxplot(rnorm(50))

# A more complex use, where it is possible to define additional types easily.
# It also allow for completion after fun$... and completion of functions
# arguments, depending on the selected type (to avoid putting all arguments
# for all types together, otherwise, it is a mess)
head2 <- structure(function(data, n = 10, ..., type = "default") {
  # This was the old (static) approach: not possible to add a new type
  # without modifying the function head2()
  #switch(type,
  #  default = `.head2$default`(data, n = n, ...),
  #  fun = `.head2$fun`(data, n = n, ...)
  #)
  # This is the new (dynamic) approach
  get_type("head2", type = type)(data, n = n, ...)
}, class = c("subsettable_type", "function", "head2"))

# We define two types for head2(): default and fun
```

```

`head2_default` <- function(data, n = 10, ...) {
  head(data, n = n)
}

# Apply a fun on head() - just an example, not necessarily useful
`head2_fun` <- function(data, n = 10, fun = summary, ...) {
  head(data, n = n) |> fun(...)
}

head2(iris)
head2(iris, type = "default") # Idem
head2$default(iris) # Idem
head2$fun(iris) # The other type, with fun = summary()
head2$fun(iris, fun = str)

# Now, the completion (e.g., in RStudio or Positron)
# 1. Type head2$ and you got the list of available types
# 2. Select "default" then hit <tab>, you got the list of args for default
# 3. Do the same but select "fun", now you got the arguments for the fun type
# 4. Just write a new `.head2_<type>` function and <type> is automatically
#    integrated!

```

tidy_functions

Tidy functions (mainly from dplyr and tidyr) to manipulate data frames

Description

[Deprecated]

These function are deprecated to the benefit of the functions whose name ends with an underscore _ (e.g., select() -> [svTidy::select_\(\)](#)) in the [svTidy](#) package.

The Tidyverse defines a coherent set of tools to manipulate data frames that use a non-standard evaluation and sometimes require extra care. These functions, like [dplyr::mutate\(\)](#) or [dplyr::summarise\(\)](#) are defined in the {dplyr} and {tidyr} packages. When using variants, like {dplyr} for **data.frame** objects, or {dbplyr} to work with external databases, successive commands in a pipeline are pooled together but not computed. One has to [dplyr::collect\(\)](#) the result to get its final form. Most of the tidy functions that have their "speedy" counterpart prefixed with "s" are listed with [list_tidy_functions\(\)](#). Their main usages are (excluding less used arguments, or those that are not compatibles with the speedy "s" counterpart functions):

- [group_by\(.data, ...\)](#)
- [ungroup\(.data\)](#)
- [rename\(.data, ...\)](#)
- [rename_with\(.data, .fn, .cols = everything\(\), ...\)](#)
- [filter\(.data, ...\)](#)
- [select\(.data, ...\)](#)
- [mutate\(.data, ..., .keep = "all"\)](#)

- `transmute(.data, ...)`
 - `summarise(.data, ...)`
 - `full_join(x, y, by = NULL, suffix = c(".x", ".y"), copy = FALSE, ...)`
 - `left_join(x, y, by = NULL, suffix = c(".x", ".y"), copy = FALSE, ...)`
 - `right_join(x, y, by = NULL, suffix = c(".x", ".y"), copy = FALSE, ...)`
 - `inner_join(x, y, by = NULL, suffix = c(".x", ".y"), copy = FALSE, ...)`
 - `bind_rows(..., .id = NULL)`
 - `bind_cols(..., .name_repair = c("unique", "universal", "check_unique", "minimal"))`
 - `arrange(.data, ..., .by_group = FALSE)`
 - `count(x, ..., wt = NULL, sort = FALSE, name = NULL)`
 - `tally(x, wt = NULL, sort = FALSE, name = NULL)`
 - `add_count(x, ..., wt = NULL, sort = FALSE, name = NULL)`
 - `add_tally(x, wt = NULL, sort = FALSE, name = NULL)`
 - `pull(.data, var = -1, name = NULL)`
 - `distinct(.data, ..., .keep_all = FALSE)`
 - `drop_na(data, ...)`
 - `replace_na(data, replace)`
 - `pivot_longer(data, cols, names_to = "name", values_to = "value")`
 - `pivot_wider(data, names_from = name, values_from = value)`
 - `uncount(data, weights, .remove = TRUE, .id = NULL)`
 - `unite(data, col, ..., sep = "_", remove = TRUE, na.rm = FALSE)`
 - `separate(data, col, into, sep = "[^[:alnum:]]+", remove = TRUE, convert = FALSE)`
 - `separate_rows(data, ..., sep = "[^[:alnum:]].]", convert = FALSE)`
 - `fill(data, ..., .direction = c("down", "up", "downup", "updown"))`
 - `extract(data, col, into, regex = "[[:alnum:]]+", remove = TRUE, convert = FALSE)`
- plus the functions defined here under.

Usage

```
list_tidy_functions()
```

```
filter_ungroup(.data, ...)
```

```
mutate_ungroup(.data, ..., .keep = "all")
```

```
transmute_ungroup(.data, ...)
```

Arguments

- | | |
|--------------------|--|
| <code>.data</code> | A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from <code>dbplyr</code> or <code>dtplyr</code>). See <code>dplyr::mutate()</code> for more details. |
| <code>...</code> | Arguments dependent to the context of the function and most of the time, not evaluated in a standard way (cf. the tidyverse approach). |
| <code>.keep</code> | Which columns to keep. The default is "all", possible values are "used", "unused", or "none" (see <code>dplyr::mutate()</code>). |

Value

See corresponding "non-t" function for the full help page with indication of the return values. `list_tidy_functions()` returns a list of all the tidy(verbose) functions that have their speedy "s" counterpart, see [speedy_functions](#).

Note

The help page here is very basic and it aims mainly to list all the tidy functions. For more complete help, see the {dplyr} or {tidyr} packages. From {dplyr}, the `dplyr::slice()` and `slice_xxx()` functions are not added yet because they are not available for {dbplyr}. Also `dplyr::anti_join()`, `dplyr::semi_join()` and `dplyr::nest_join()` are not implemented yet. From {dplyr}, the `dplyr::slice()` and `slice_xxx()` functions are not added yet because they are not available for {dbplyr}. From {tidyr} `tidyr::expand()`, `tidyr::chop()`, `tidyr::unchop()`, `tidyr::nest()`, `tidyr::unnest()`, `tidyr::unnest_longer()`, `tidyr::unnest_wider()`, `tidyr::hoist()`, `tidyr::pack()` and `tidyr::unpack()` are not implemented yet.

See Also

`collapse::num_vars()` to easily keep only numeric columns from a data frame, `collapse::fscale()` for scaling and centering matrix-like objects and data frames.

Examples

```
# TODO...
```

Index

- * **create 'subsettable' functions**
 - subsettable, 39
- * **labeling objects**
 - labelise, 24
- * **utilities**
 - labelise, 24
 - subsettable, 39
- .DollarNames.subsettable_type
 - (subsettable), 39
- \$.subsettable_type(subsettable), 39
- \$.subsettable_type2(subsettable), 39
- \$.subsettable_which(subsettable), 39
- %*f%(fstat_functions), 18
- %+f%(fstat_functions), 18
- %-+f%(fstat_functions), 18
- %->%(alt_assign), 5
- %-f%(fstat_functions), 18
- %-modf%(fstat_functions), 18
- %/*100f%(fstat_functions), 18
- %/f%(fstat_functions), 18
- %<-%(alt_assign), 5
- %modf%(fstat_functions), 18
- %replace_fillf%(fstat_functions), 18
- %replacef%(fstat_functions), 18
- aka, 3
- alt_assign, 5
- args_type(subsettable), 39
- as.matrix.tbl_df(as_dtx), 7
- as_dtbl(as_dtx), 7
- as_dtf(as_dtx), 7
- as_dtf(), 12
- as_dtrm(as_dtx), 7
- as_dtt(as_dtx), 7
- as_dtx, 7
- as_dtx(), 7, 9
- as_matrix(as_dtx), 7
- as_matrix(), 8
- base::as.matrix(), 8
- base::gettext(), 20–22
- base::gettextf(), 22
- base::ngettext(), 20
- base::units(), 25
- check_lang(gettext_), 20
- cl(labelise), 24
- cli::cli_abort(), 29, 37, 38
- collapse::fast-statistical-functions,
20
- collapse::fgroup_by(), 18
- collapse::fmean(), 18
- collapse::fnobs(), 20
- collapse::frename(), 32
- collapse::fscale(), 43
- collapse::fselect(), 32
- collapse::fsummarise(), 32
- collapse::num_vars(), 43
- collapse::TRA(), 19
- collect.default(alt_assign), 5
- collect_dtbl(collect_dtx), 9
- collect_dtf(collect_dtx), 9
- collect_dtrm(collect_dtx), 9
- collect_dtt(collect_dtx), 9
- collect_dtx, 9
- collect_dtx(), 9, 10, 12
- data_dot_mechanism, 10, 28
- default_dtx(as_dtx), 7
- default_dtx(), 5
- dplyr::anti_join(), 36, 43
- dplyr::bind_cols(), 35
- dplyr::collect(), 5, 9, 10, 41
- dplyr::group_by(), 7, 18, 23
- dplyr::mutate(), 32, 35, 41, 42
- dplyr::nest_join(), 36, 43
- dplyr::rename(), 32
- dplyr::select(), 32
- dplyr::semi_join(), 36, 43
- dplyr::slice(), 36, 43

dplyr::summarise(), 32, 36, 41
 dplyr::ungroup(), 7
 dtbl(dtx), 11
 dtbl(), 3, 12
 dtbl_rows(dtx_rows), 13
 dtbl_rows(), 14
 dtf(dtx), 11
 dtf(), 3, 12
 dtf_rows(dtx_rows), 13
 dtf_rows(), 14
 dtrm(dtx), 11
 dtrm(), 12
 dtrm_rows(dtx_rows), 13
 dtrm_rows(), 14
 dtt(dtx), 11
 dtt(), 3, 12
 dtt_rows(dtx_rows), 13
 dtt_rows(), 14
 dtx, 11
 dtx(), 3, 13
 dtx_rows, 13
 dtx_rows(), 12, 14

 expr_funs, 14

 f_, 15
 filter_ungroup(tidy_functions), 41
 fn(fstat_functions), 18
 fn(), 19, 20, 36
 fna(fstat_functions), 18
 fna(), 19, 20
 formula_masking, 16
 formula_masking(), 18
 formula_select, 17
 formula_select(), 17
 fstat_functions, 18

 get_language(gettext_), 20
 get_sciviews_lang(gettext_), 20
 get_section(section), 31
 get_type(subsettable), 39
 gettext(), 37, 38
 gettext_, 20
 gettextf_(gettext_), 20

 is_dtbl(is_dtx), 23
 is_dtf(is_dtx), 23
 is_dtrm(is_dtx), 23
 is_dtt(is_dtx), 23

 is_dtx, 23
 is_dtx(), 12

 label(), 25
 labelise, 24
 labelize(labelise), 24
 list_fstat_functions(fstat_functions), 18
 list_fstat_functions(), 19
 list_speedy_functions(speedy_functions), 32
 list_tidy_functions(tidy_functions), 41
 list_tidy_functions(), 41, 43
 list_types(subsettable), 39

 mean(), 18
 mutate_ungroup(tidy_functions), 41

 name_function_type(subsettable), 39
 ngettext_(gettext_), 20

 object_info(stop_), 37

 prepare_data_dot, 26
 prepare_data_dot(), 11
 prepare_data_dot2(prepare_data_dot), 26
 print.aka(aka), 3
 print.section(section), 31
 process_error_msg, 29

 recall_with_data_dot(prepare_data_dot), 26
 recall_with_data_dot2(prepare_data_dot), 26
 retarget, 30
 rlang::f_lhs(), 17
 rlang::f_rhs(), 17
 rlang::is_formula(), 17

 sadd_count(speedy_functions), 32
 sadd_tally(speedy_functions), 32
 sarrange(speedy_functions), 32
 sbind_cols(speedy_functions), 32
 sbind_rows(speedy_functions), 32
 scount(speedy_functions), 32
 sdistinct(speedy_functions), 32
 sdrop_na(speedy_functions), 32
 section, 31
 set_language(gettext_), 20
 set_sciviews_lang(gettext_), 20

sextract (speedy_functions), 32
 sfill (speedy_functions), 32
 sfilter (speedy_functions), 32
 sfilter_ungroup (speedy_functions), 32
 sfull_join (speedy_functions), 32
 sgroup_by (speedy_functions), 32
 sgroup_by(), 18
 sinner_join (speedy_functions), 32
 sleft_join (speedy_functions), 32
 smutate (speedy_functions), 32
 smutate(), 32
 smutate_ungroup (speedy_functions), 32
 speedy_functions, 32, 43
 spivot_longer (speedy_functions), 32
 spivot_wider (speedy_functions), 32
 spull (speedy_functions), 32
 srename (speedy_functions), 32
 srename_with (speedy_functions), 32
 sreplace_na (speedy_functions), 32
 sright_join (speedy_functions), 32
 sselect (speedy_functions), 32
 sseparate (speedy_functions), 32
 sseparate_rows (speedy_functions), 32
 ssummarise (speedy_functions), 32
 ssummarise(), 36
 stally (speedy_functions), 32
 stats::ave(), 18
 stop(), 38
 stop_, 37
 stop_top_call (stop_), 37
 str.aka (aka), 3
 str.section (section), 31
 stransmute (speedy_functions), 32
 stransmute_ungroup (speedy_functions), 32
 subsetting, 39
 suncount (speedy_functions), 32
 sungroup (speedy_functions), 32
 sunite (speedy_functions), 32
 svBase (svBase-package), 3
 svBase-package, 3
 svTidy::select_(), 32, 41
 sys.call(), 28
 test_gettext_lang (gettext_), 20
 tibble::tibble(), 12
 tidy_functions, 41
 tidyr::chop(), 36, 43
 tidyr::expand(), 36, 43
 tidyr::hoist(), 36, 43
 tidyr::nest(), 36, 43
 tidyr::pack(), 36, 43
 tidyr::pivot_longer(), 36
 tidyr::unchop(), 36, 43
 tidyr::unnest(), 36, 43
 tidyr::unnest_longer(), 36, 43
 tidyr::unnest_wider(), 36, 43
 tidyr::unpack(), 36, 43
 tidyselect::eval_select(), 17
 tools::update_pkg_po(), 22
 transmute_ungroup (tidy_functions), 41
 unlabelise (labelise), 24
 unlabelize (labelise), 24
 warning(), 37, 38
 warning_ (stop_), 37
 zeallot::operator, 5