

Package ‘interfacer’

August 1, 2024

Title Define and Enforce Contracts for Dataframes as Function Parameters

Version 0.2.3

Description A dataframe validation framework for package builders who use dataframes as function parameters. It performs checks on column names, coerces data-types, and checks grouping to make sure user inputs conform to a specification provided by the package author. It provides a mechanism for package authors to automatically document supported dataframe inputs and selectively dispatch to functions depending on the format of a dataframe much like S3 does for classes. It also contains some developer tools to make working with and documenting dataframe specifications easier. It helps package developers to improve their documentation and simplifies parameter validation where dataframes are used as function parameters.

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.3.1

Language en-GB

Imports dplyr, glue, magrittr, purrr, rlang, tibble, tidyselect, stringr, forcats, knitr, digest, roxygen2

Suggests rmarkdown, devtools, ggplot2, testthat (>= 3.0.0), fs, readr, usethis, whisker, clipr, tidyr, binom, spelling

VignetteBuilder knitr

Config/testthat/edition 3

Depends R (>= 2.10)

URL <https://bristol-vaccine-centre.github.io/interfacer/>,
<https://github.com/bristol-vaccine-centre/interfacer>

BugReports <https://github.com/bristol-vaccine-centre/interfacer/issues>

NeedsCompilation no

Author Robert Challen [aut, cre, cph]
(<https://orcid.org/0000-0002-5504-7768>)

Maintainer Robert Challen <rob.challen@bristol.ac.uk>

Repository CRAN

Date/Publication 2024-08-01 08:50:02 UTC

Contents

as.list.iface	3
format.iface	3
iclip	4
iconvert	5
idispach	6
idocument	7
iface	8
if_col_present	10
igroup_process	11
imapper	13
iproto	14
ireturn	14
is.iface	15
is_col_present	16
itest	16
ivalidate	17
knit_print.iface	18
print.iface	19
roxy_tag_parse.roxy_tag_iparam	19
roxy_tag_rd.roxy_tag_iparam	20
switch_pipeline	21
type.anything	22
type.character	22
type.complete	23
type.date	23
type.default	24
type.double	24
type.enum	25
type.factor	25
type.finite	26
type.group_unique	26
type.integer	27
type.in_range	27
type.logical	28
type.not_missing	28
type.numeric	29
type.of_type	29
type.positive_double	30
type.positive_integer	30
type.proportion	31
use_dataframe	31

<code>as.list.iface</code>	3
<code>use_iface</code>	32
Index	34

<code>as.list.iface</code>	<i>Cast an iface to a plain list.</i>
----------------------------	---------------------------------------

Description

Cast an iface to a plain list.

Usage

```
## S3 method for class 'iface'
as.list(x, ..., flatten = FALSE)
```

Arguments

- `x` object to be coerced or tested.
- `...` objects, possibly named.
- `flatten` get a list of lists representation instead of the dataframe column by column list.

Value

a list representation of the iface input.

Examples

```
my_iface = iface(
  col1 = integer + group_unique ~ "an integer column"
)

as.list(my_iface, flatten=TRUE)
```

<code>format.iface</code>	<i>Format an iface specification for printing</i>
---------------------------	---

Description

Format an iface specification for printing

Usage

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

Arguments

x an iface specification
 ... not used.

Value

a formatted string representation of an iface

Examples

```
my_iface = iface(
  col1 = integer + group_unique ~ "an integer column"
)

print(my_iface)
knitr::knit_print(my_iface)
```

 iclip

Create an iface specification from an example dataframe

Description

When developing with interfacr it is useful to be able to base a function input off a prototype that you are for example using as testing. This function generates an `interfacr::iface` specification for the supplied data frame and copies it to the clipboard so that it can be pasted into the package code you are working on.

Usage

```
iclip(df, df_name = deparse(substitute(df)))
```

Arguments

df a prototype dataframe
 df_name an optional name for the parameter (defaults to `i_<df name>`)

Details

If the dataframe contains one or more list columns with nested dataframes the nested dataframes are also defined using a second iface specification.

Value

nothing, populates clipboard

Examples

```
if (interactive()) iclip(iris)
```

iconvert	<i>Convert a dataframe to a format compatible with an interface specification</i>
----------	---

Description

This function is called by `ivalidate` and is not generally intended to be used directly by the end user. It may be helpful in debugging during package development to interactive test a `iface` spec. `iconvert` is an interactive version of `ivalidate`.

Usage

```
iconvert(  
  df,  
  iface,  
  .imap = interfacer::imapper(),  
  .dname = "<unknown>",  
  .fname = "<unknown>",  
  .has_dots = TRUE,  
  .prune = FALSE,  
  .env = rlang::current_env()  
)
```

Arguments

<code>df</code>	the dataframe to convert
<code>iface</code>	the interface spec as an <code>iface</code>
<code>.imap</code>	an optional <code>imapper</code> mapping
<code>.dname</code>	the name of the parameter value (optional).
<code>.fname</code>	the name of the function (optional).
<code>.has_dots</code>	internal library use only. Changes the nature of the error message.
<code>.prune</code>	do you want to remove non matching columns?
<code>.env</code>	internal use only

Value

the input dataframe coerced to be conformant to the `iface` specification, or an informative error is thrown.

Examples

```
i_diamonds = iface(  
  color = enum(D,E,F,G,H,I,J,extra) ~ "the colour",  
  price = integer ~ "the price"  
)
```

```
iconvert(ggplot2::diamonds, i_diamonds, .prune = TRUE)
```

idispatch	<i>Dispatch to a named function based on the characteristics of a dataframe</i>
-----------	---

Description

This provides a dataframe analogy to S3 dispatch.

Usage

```
idispatch(x, ..., .default = NULL)
```

Arguments

x	a dataframe
...	a set of function name=interfacer::iface pairs
.default	a function to apply in the situation where none of the rules can be matched. The default results in an error being thrown.

Details

If multiple possible dataframe formats are possible for a function, each with different processing requirements, then the choice of function can be made based on matching the input dataframe to a set of `iface` specifications. The first matching `iface` specification determines which function is used for dispatch.

Value

the result of dispatching the dataframe to the first function that matches the rules in `...`. Matching is permissive in that the test is passed if a dataframe can be coerced to the `iface` specified format.

Examples

```
i1 = iface( col1 = integer ~ "An integer column" )
i2 = iface( col2 = integer ~ "A different integer column" )

# this is an example function that would typically be inside a package, and
# is exported from the package.
extract_mean = function(df, ...) {
  idispatch(df,
    extract_mean.i1 = i1,
    extract_mean.i2 = i2
  )
}
```

```

# this is expected to be an internal package function
# the naming convention here is based on S3 but it is not required
extract_mean.i1 = function(df = i1, ...) {
  message("using i1")
  # input validation is not required in functions that are being called using
  # `dispatch` as the validation occurs during dispatch.
  mean(df$col1)
}

extract_mean.i2 = function(df = i2, uplift = 1, ...) {
  message("using i2")
  mean(df$col2)+uplift
}

# this input matches `i1` and the `extract_mean` call is dispatched
# via `extract_mean.i1`
test = tibble::tibble( col2 = 1:10 )
extract_mean(test, uplift = 50)

# this input matches `i2` and the `extract_mean` call is dispatched
# via `extract_mean.i2`
test2 = tibble::tibble( col1 = 1:10 )
extract_mean(test2, uplift = 50)

# This input does not match any of the allowable input specifications and
# generates an error.
test3 = tibble::tibble( wrong_col = 1:10 )
try(extract_mean(test3, uplift = 50))

```

idocument

Document an interface contract for inserting into roxygen2

Description

This function is expected to be called within the documentation of a function as inline code in the parameter documentation of the function. It details the expected columns that the input dataframe should possess. This has mostly been superseded by the `@iparam <name> <description>` roxygen2 tag which does this automatically, however in some circumstances (particularly multiple dispatch) you may want to assemble dataframe documentation manually.

Usage

```
idocument(fn, param = NULL)
```

Arguments

fn	the function that you are documenting
param	the parameter you are documenting (optional. if missing defaults to the first argument of the function)

Value

a markdown snippet

Examples

```
#' @param df `r idocument(x, df)`
x = function(df = iface( col1 = integer ~ "an integer column" )) {}

cat(idocument(x, df))
```

iface

Construct an interface specification

Description

An `iface` specification defines the expected structure of a dataframe, in terms of the column names, column types, grouping structure and uniqueness constraints that the dataframe must conform to. A dataframe can be tested for conformance to an `iface` specification using `iconvert`.

Usage

```
iface(..., .groups = NULL, .default = NULL)
```

Arguments

<code>...</code>	The specification of the interface (see details), or an unnamed <code>iface</code> object to extend, or both.
<code>.groups</code>	either <code>FALSE</code> for no grouping allowed or a formula of the form <code>~ var1 + var2 + ...</code> which defines what columns must be grouped in the dataframe (and in which order). If <code>NULL</code> (the default) then any grouping is permitted. If the formula contains a dot e.g. <code>~ . + var1 + var2</code> then the grouping must include <code>var1</code> and <code>var2</code> but other groups are also allowed.
<code>.default</code>	a default value to supply if there is nothing given in a function parameter using the <code>iface</code> as a formal. This is either <code>NULL</code> in which case there is no default, <code>TRUE</code> in which case the default is a zero row dataframe conforming to the specification, or a provided dataframe, which is checked to conform, and used as the default.

Details

An `iface` specification is designed to be used to define the type of a parameter in a function. This is done by using the `iface` specification as the default value of the parameter in the function definition. The definition can then be validated at runtime by a call to `ivalidate(...)` inside the function.

When developing a function output an `iface` specification may also be used in `ireturn(...)` to enforce that the output of a function is correct.

`iface` definitions can be printed and included in roxygen2 documentation and help us to document input dataframe parameters and dataframe return values in a standardised way by using the `@iparam roxygen2` tag.

iface specifications are defined in the form of a named list of formulae with the structure `column_name = type ~ "documentation"`.

type can be one of anything, character, complete, date, default, double, enum, factor, finite, group_unique, in_range, integer, logical, not_missing, numeric, of_type, positive_double, positive_integer, proportion (e.g. `enum(level1, level2, ...)`, `in_range(min, max)`) or alternatively anything that resolves to a function e.g. `as.ordered`.

If type is a function name, then the function must take a single vector parameter and return a single vector of the same size. The function must also return a zero length vector of an appropriate type if passed NULL.

type can also be a concatenation of rules separated by +, e.g. `integer + group_unique` for an integer that is unique within a group.

Value

the definition of an interface as a `iface` object

Examples

```
test_df = tibble::tibble(
  grp = c(rep("a", 10), rep("b", 10)),
  col1 = c(1:10, 1:10)
) %>% dplyr::group_by(grp)

my_iface = iface(
  col1 = integer + group_unique ~ "an integer column",
  .default = test_df
)

print(my_iface)

# the function x defines a formal `df` with default value of `my_iface`
# this default value is used to validate the structure of the user supplied
# value when the function is called.
x = function(df = my_iface, ...) {
  df = ivalidate(df, ...)
  return(df)
}

# this works
x(tibble::tibble(col1 = c(1,2,3)))

# this fails as x is of the wrong type
try(x(tibble::tibble(col1 = c("a", "b", "c"))))

# this fails as x has duplicates
try(x(tibble::tibble(col1 = c(1,2,3,3))))

# this gives the default value
x()
```

```

my_iface2 = iface(
  first_col = numeric ~ "column order example",
  my_iface,
  last_col = character ~ "another col", .groups = ~ first_col + col1
)
print(my_iface2)

my_iface3 = iface(
  col1 = integer + group_unique ~ "an integer column",
  .default = test_df_2
)
x = function(d = my_iface3) {ivalidate(d)}

# Doesn't work as test_df_2 hasn't been defined
try(x())

test_df_2 = tibble::tibble(
  grp = c(rep("a",10),rep("b",10)),
  col1 = c(1:10,1:10)
) %>% dplyr::group_by(grp)

# now it works as has been defined
x()

# it still works as default has been cached.
rm(test_df_2)
x()

```

if_col_present	<i>Execute a function or return a value if a column is present in a dataframe</i>
----------------	---

Description

The simple use case. For more complex behaviour see `switch_pipeline()`.

Usage

```
if_col_present(df, col, if_present, if_missing = ~.x)
```

Arguments

df	a dataframe
col	a column name
if_present	a purrr style function to execute on the dataframe if the column is present (or a plain value)
if_missing	a purrr style function to execute on the dataframe if the column is missing (or a plain value)

Value

either the value of `if_present/if_absent` or the result of calling `if_present/if_absent` as functions on `df`.

Examples

```
iris %>% if_col_present(Species, ~ .x %>% dplyr::rename(new = Species)) %>%
  colnames()

# in contrast to `purrr` absolute values are not interpreted as function names
iris %>% if_col_present(Species2, "Yes", "No")
```

 igroup_process

Handle unexpected additional grouping structure

Description

This function is designed to be used by a package author within an enclosing function. The enclosing function is assumed to take as input a dataframe and have an `iface` specified for that dataframe.

Usage

```
igroup_process(df = NULL, fn, ...)
```

Arguments

<code>df</code>	a dataframe from an enclosing function in which the grouping may or may not have been correctly supplied.
<code>fn</code>	a function to call with the correctly grouped dataframe as specified by the <code>iface</code> in the enclosing function.
<code>...</code>	passed onto <code>iconvert</code> this could be used to supply <code>.prune</code> parameters. triple dot parameters in the enclosing function will be separately handled and automatically passed to <code>fn</code> so in general should not be passed to <code>igroup_process</code> as an intermediary although it probably won't hurt. This behaviour is similar to <code>NextMethod</code> in S3 method dispatch.

Details

This function detects when the grouping of the input has additional groups over and above those in the specification and intercepts them, regrouping the dataframe and applying `fn` group-wise using an equivalent of a `dplyr::group_modify`. The parameters provided to the enclosing function will be passed to `fn` and they should have compatible method signatures.

Value

the result of calling `fn(df, ...)` on each unexpected group

Examples

```

# This specification requires that the dataframe is grouped only by the color
# column
i_diamond_price = interfacer::iface(
  color = enum(`D`,`E`,`F`,`G`,`H`,`I`,`J`, .ordered=TRUE) ~ "the color column",
  price = integer ~ "the price column",
  .groups = ~ color
)

# An example function which would be exported in a package
ex_mean = function(df = i_diamond_price, extra_param = ".") {

  # When called with a dataframe with extra groups `igroup_process` will
  # regroup the dataframe according to the structure
  # defined for `i_diamond_price` and apply the inner function to each group
  # after first calling `invalidate` on each group.

  igroup_process(df,

    # the real work of this function is provided as an anonymous inner
    # function (but can be any other function e.g. package private function)
    # or a purrr style lambda.

    function(df, extra_param) {
      message(extra_param, appendLF = FALSE)
      return(df %>% dplyr::summarise(mean_price = mean(price)))
    }

  )
}

# The correctly grouped dataframe. The `ex_mean` function calculates the mean
# price for each `color` group.
ggplot2::diamonds %>%
  dplyr::group_by(color) %>%
  ex_mean(extra_param = "without additional groups...") %>%
  dplyr::glimpse()

# If an additionally grouped dataframe is provided by the user. The `ex_mean`
# function calculates the mean price for each `cut`, `clarity`, and `color`
# combination.

ggplot2::diamonds %>%
  dplyr::group_by(cut, color, clarity) %>%
  ex_mean() %>%
  dplyr::glimpse()

# The output of this is actually grouped by cut then clarity as
# color is consumed by the igroup_dispatch summarise.

```

imapper	<i>Specify mappings that can make dataframes compatible with an iface specification</i>
---------	---

Description

When a function uses `ivalidate` internally to check a dataframe conforms to the input it can attempt to rescue an incorrectly formatted dataframe. This is a pretty advanced idea and is not generally recommended.

Usage

```
imapper(...)
```

Arguments

... a set of `dplyr::mutate()` specifications that when applied to a dataframe will rename or otherwise fix missing columns

Details

This function is expected to be used only in the context of a `.imap = mapper(...)` parameter to an `ivalidate` call to make sure that certain columns are present or are a set value. Anything provided here will overwrite existing dataframe columns and its use is likely to make function behaviour obtuse. It may be deprecated in the future. The ... input expressions should almost certainly check for the values already existing before overwriting them.

If you are considering using this for replacing missing values check using the `default(...)` iface type definition instead.

Value

a set of mappings

Examples

```
x = function(df = iface(col1 = integer ~ "an integer column" ), ...) {
  df = ivalidate(df,...)
}
input=tibble::tibble(col2 = c(1,2,3))
# This fails because col1 is missing
try(x(input))
# This fixes it for this input
x(input, .imap=imapper(col1 = col2))
```

iproto	<i>Generate a zero length dataframe conforming to an iface specification</i>
--------	--

Description

This function is used internally for default values for a dataframe parameter. It generates a zero length dataframe that conforms to a iface specification, in terms of column names, data types and groupings. Such a dataframe is not guaranteed to be fully conformant to the iface specification if, for example, completeness constraints are applied.

Usage

```
iproto(iface)
```

Arguments

iface the specification

Value

a dataframe conforming to iface

Examples

```
i = interfacer::iface(
  col1 = integer ~ "A number",
  col2 = character ~ "A string"
)

iproto(i)
```

ireturn	<i>Check a return parameter from a function</i>
---------	---

Description

This is intended to be used within a function to check the validity of a data frame being returned from a function against an ispec which is provided.

Usage

```
ireturn(df, iface, .prune = FALSE)
```

Arguments

df	a dataframe - if missing then the first parameter of the calling function is assumed to be a dataframe.
iface	the interface specification that df should conform to.
.prune	get rid of excess columns that are not in the spec.

Value

a dataframe based on df with validity checks passed, data-types coerced, and correct grouping applied to conform to iface

Examples

```
input = iface(col_in = integer ~ "an integer column" )
output = iface(col_out = integer ~ "an integer column" )

x = function(df = input, ...) {
  df = ivalidate(...)
  tmp = df %>% dplyr::rename(col_out = col_in)
  ireturn(tmp, output)
}
x(tibble::tibble(col_in = c(1,2,3)))
output
```

is.iface

Check if an object is an interface specification

Description

Check if an object is an interface specification

Usage

```
is.iface(x, ...)
```

Arguments

x	the object to check
...	ignored

Value

a boolean.

<code>is_col_present</code>	<i>Check for existence of a set of columns in a dataframe</i>
-----------------------------	---

Description

Check for existence of a set of columns in a dataframe

Usage

```
is_col_present(df, ...)
```

Arguments

<code>df</code>	a dataframe to test
<code>...</code>	the column names (unquoted)

Value

TRUE if the columns are all there, false otherwise

Examples

```
is_col_present(iris, Species, Petal.Width)
```

<code>itest</code>	<i>Test dataframe conformance to an interface specification.</i>
--------------------	--

Description

`ivalidate` throws errors deliberately however sometimes dealing with invalid input may be desirable. `itest` is generally designed to be used within a function which specifies the expected input using `iface`, and allows the function to test if its given input is conformant to the interface.

Usage

```
itest(df = NULL, iface = NULL, .imap = imapper())
```

Arguments

<code>df</code>	a dataframe to test. If missing the first parameter of the calling function is assumed to be the dataframe to test.
<code>iface</code>	an interface specification produced by <code>iface()</code> . If missing this will be inferred from the current function signature.
<code>.imap</code>	an optional mapping specification produced by <code>imapper()</code>

Value

TRUE if the dataframe is conformant, FALSE otherwise

Examples

```
if (rlang::is_installed("ggplot2")) {
  i_diamonds = iface(
    color = enum(D,E,F,G,H,I,J,extra) ~ "the colour",
    price = integer ~ "the price"
  )

  # Ad hoc testing
  itest(ggplot2::diamonds, i_diamonds)

  # Use within function:
  x = function(df = i_diamonds) {
    if(itest()) message("PASS!")
  }

  x(ggplot2::diamonds)
}
```

ivalidate	<i>Perform interface checks on dataframe inputs using enclosing function formal parameter definitions</i>
-----------	---

Description

`ivalidate(...)` is intended to be used within a function to check the validity of a data frame parameter (usually the first parameter) against an `ispec` which is given as a default value of a formal parameter.

Usage

```
ivalidate(df = NULL, ..., .imap = imapper(), .prune = FALSE, .default = NULL)
```

Arguments

<code>df</code>	a dataframe - if missing then the first parameter of the calling function is assumed to be a dataframe.
<code>...</code>	not used but <code>ivalidate</code> should be included in call to inherit <code>.imap</code> from the caller function.
<code>.imap</code>	a set of mappings as an <code>imapper</code> object.
<code>.prune</code>	get rid of excess columns that are not in the spec.
<code>.default</code>	a default dataframe conforming to the specification. This overrides any defaults defined in the interface specification

Value

a dataframe based on df with validity checks passed and . imap mappings applied if present

Examples

```
x = function(df = iface(col1 = integer ~ "an integer column" ), ...) {
  df = invalidate(...)
  return(df)
}
input=tibble::tibble(col1 = c(1,2,3))
x(input)

# This fails because col1 is not coercable to integer
input2=tibble::tibble(col1 = c(1.5,2,3))
try(x(input2))
```

knit_print.iface	<i>Format an iface specification for printing</i>
------------------	---

Description

Format an iface specification for printing

Usage

```
knit_print.iface(x, ...)
```

Arguments

x	an iface specification
...	not used.

Value

a formatted string representation of an iface

Examples

```
my_iface = iface(
  col1 = integer + group_unique ~ "an integer column"
)

print(my_iface)
knitr::knit_print(my_iface)
```

print.iface	<i>Format an iface specification for printing</i>
-------------	---

Description

Format an iface specification for printing

Usage

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

Arguments

x	an iface specification
...	not used.

Value

a formatted string representation of an iface

Examples

```
my_iface = iface(
  col1 = integer + group_unique ~ "an integer column"
)

print(my_iface)
knitr::knit_print(my_iface)
```

roxy_tag_parse.roxy_tag_iparam	<i>Parser for @iparam tags</i>
--------------------------------	--------------------------------

Description

The @iparam <name> <description> tag can be used in roxygen2 documentation of a function to describe a dataframe parameter. The function must be using `interfacer::iface` to define the input dataframe parameter format. The @iparam tag will then generate documentation about the type of dataframe the function is expecting.

Usage

```
## S3 method for class 'roxy_tag_iparam'
roxy_tag_parse(x)
```

Arguments

x A tag

Value

a roxy_tag object with the val field set to the parsed value

Examples

```
# This provides support to `roxygen2` and only gets executed in the context
# of `devtools::document()`. There is no interactive use of this function.
```

```
roxy_tag_rd.roxy_tag_iparam
                          Support for @iparam tags
```

Description

The @iparam <name> <description> tag can be used in roxygen2 documentation of a function to describe a dataframe parameter. The function must be using `interfacer::iface` to define the input dataframe parameter format. The @iparam tag will then generate documentation about the type of dataframe the function is expecting.

Usage

```
## S3 method for class 'roxy_tag_iparam'
roxy_tag_rd(x, base_path, env)
```

Arguments

x The tag
base_path Path to package root directory.
env Environment in which to evaluate code (if needed)

Value

an `roxygen2::rd_section` (see roxygen2 documentation)

Examples

```
# An example function definition:
fn_definition <- "
#' This is a title
#'
#' This is the description.
#'
#' @md
#' @iparam df the input
```

```

#' @export
f <- function(df = interfacer::iface(
  id = integer ~ \"an integer `ID`\",
  test = logical ~ \"the test result\"
)) {
  invalidate(df)
}
"

# For this example we manually parse the function specification in `fn_definition`
# creating a .Rd block - normally this is done by `roxygen2` which then
# writes this to an .Rd file. This function is not intended to be used
# outside of a call to `devtools::document`.

tmp = roxygen2::parse_text(fn_definition)
print(tmp)

```

switch_pipeline

Branch a dplyr pipeline based on a set of conditions

Description

Branch a dplyr pipeline based on a set of conditions

Usage

```
switch_pipeline(.x, ...)
```

Arguments

.x	a dataframe
...	a list of formulae of the type predicate ~ purrr function using .x as the single parameter

Value

the result of applying purrr function to .x in the case where predicate evaluates to true. Both predicate and function can refer to the pipeline dataframe using .x

Examples

```

iris %>% switch_pipeline(
  is_col_present(.x, Species) ~ .x %>% dplyr::rename(new = Species)
) %>% dplyr::glimpse()

```

type.anything	<i>Coerce to an unspecified type</i>
---------------	--------------------------------------

Description

Coerce to an unspecified type

Usage

```
type.anything(x)
```

Arguments

x	any vector
---	------------

Value

the input (unless x is NULL in which case a character())

type.character	<i>Coerce to a character.</i>
----------------	-------------------------------

Description

Coerce to a character.

Usage

```
type.character()
```

Value

the input as a character.

type.complete	<i>Coerce to a complete set of values.</i>
---------------	--

Description

This test checks either for factors that all factor levels are present in the input, or for numerics if the sequence from minimum to maximum by the smallest difference are not all (approximately) present. Empty values are ignored.

Usage

```
type.complete(x)
```

Arguments

x any vector, factor or numeric

Value

the input or error if not complete

type.date	<i>Coerce to a Date.</i>
-----------	--------------------------

Description

Coerce to a Date.

Usage

```
type.date(x, ...)
```

Arguments

x an object to be converted.
... further arguments to be passed from or to other methods.

Value

the input as a date vector, error if this would involve data loss.

type.default	<i>Set a default value for a column</i>
--------------	---

Description

Any NA values will be replaced by this value. N.b. default values must be provided before any other rules if the validation is not to fail.

Usage

```
type.default(value)
```

Arguments

value a length one item of the correct type.

Value

a validation function that switches NAs for default values

type.double	<i>Coerce to a double.</i>
-------------	----------------------------

Description

Coerce to a double.

Usage

```
type.double()
```

Value

the input as a double, error if this would involve data loss.

type.enum	<i>Define a conformance rule to match a factor with specific levels.</i>
-----------	--

Description

Define a conformance rule to match a factor with specific levels.

Usage

```
type.enum(..., .drop = FALSE, .ordered = FALSE)
```

Arguments

...	the levels (no quotes, backticks if required)
.drop	should levels present in the data and not specified cause an error (FALSE the default) or be silently dropped to NA values (TRUE).
.ordered	must the factor be ordered

Value

a function that can check and convert input into the factor with specified levels. This will re-level factors with matching levels but in a different order.

Examples

```
f = type.enum(one,two,three)
f(c("three","two","one"))
f(factor(rep(1:3,5), labels = c("one","two","three")))
```

type.factor	<i>Coerce to a factor.</i>
-------------	----------------------------

Description

Coerce to a factor.

Usage

```
type.factor(x)
```

Arguments

x	any vector
---	------------

Value

the input as a factor, error if this would involve data loss.

type.finite	<i>Check for non-finite values</i>
-------------	------------------------------------

Description

Any non finite values will cause failure of validation.

Usage

```
type.finite(x)
```

Arguments

x any vector that can be coerced to numeric

Value

the input coerced to a numeric value, or an error if any non-finite values detected

type.group_unique	<i>Coerce to a unique value.</i>
-------------------	----------------------------------

Description

Coerce to a unique value.

Usage

```
type.group_unique(x)
```

Arguments

x any vector

Value

the input or error if any of x is not unique.

type.integer	<i>Coerce to integer</i>
--------------	--------------------------

Description

Coerce to integer

Usage

```
type.integer(x)
```

Arguments

x any vector

Value

the input as an integer, error if this would involve data loss.

type.in_range	<i>Define a conformance rule to confirm that a numeric is in a set range</i>
---------------	--

Description

This is anticipated to be part of a iface rule e.g.

Usage

```
type.in_range(min, max, include.min = TRUE, include.max = TRUE)
```

Arguments

min	the lower limit
max	the upper limit
include.min	is lower limit open (default TRUE)
include.max	is upper limit open (default TRUE)

Details

```
iface(test_col = integer + in_range(-10,10) ~ "An integer from -10 to 10")
```

Value

a function which checks the values and returns them if OK or throws an error if not

Examples

```

type.in_range(0,10,TRUE,TRUE)(0:10)
try(type.in_range(0,10,TRUE,FALSE)(0:10))
try(type.in_range(0,10,FALSE)(0:10))
type.in_range(0,10,FALSE,TRUE)(1:10)
type.in_range(0,10,TRUE,FALSE)(0:9)
type.in_range(0,Inf,FALSE,FALSE)(1:9)
try(type.in_range(0,10)(1:99))

```

type.logical	<i>Coerce to a logical</i>
--------------	----------------------------

Description

Coerce to a logical

Usage

```
type.logical(x)
```

Arguments

x any vector

Value

the input as a logical, error if this would involve data loss.

type.not_missing	<i>Check for missing values</i>
------------------	---------------------------------

Description

Any NA values will cause failure of validation.

Usage

```
type.not_missing(x)
```

Arguments

x any vector, factor or numeric

Value

the input if no missing values detected, otherwise an error

type.numeric	<i>Coerce to a numeric.</i>
--------------	-----------------------------

Description

Coerce to a numeric.

Usage

```
type.numeric()
```

Value

the input as a numeric, error if this would involve data loss.

type.of_type	<i>Check for a given class</i>
--------------	--------------------------------

Description

Any values of the wrong class will cause failure of validation. This is particularly useful for custom vectors of for list types (e.g. `list(of_type(1m))`)

Usage

```
type.of_type(type, .not_null = FALSE)
```

Arguments

type	the class of the type we are checking as a symbol
.not_null	are NULL values allowed (for list column entries only)

Value

a function that can check the input is of the correct type.

`type.positive_double` *Coerce to a positive double.*

Description

Coerce to a positive double.

Usage

```
type.positive_double(x)
```

Arguments

`x` object to be coerced or tested.

Value

the input as a positive double, error if this would involve data loss.

`type.positive_integer` *Coerce to a positive integer.*

Description

Coerce to a positive integer.

Usage

```
type.positive_integer(x)
```

Arguments

`x` any vector

Value

the input as a positive integer, error if this would involve data loss.

type.proportion	<i>Coerce to a number between 0 and 1</i>
-----------------	---

Description

Coerce to a number between 0 and 1

Usage

```
type.proportion(x)
```

Arguments

x object to be coerced or tested.

Value

the input as a number from 0 to 1, error if this would involve data loss.

use_dataframe	<i>Use a dataframe in a package including structure based documentation</i>
---------------	---

Description

Using the interfacier framework you can document data during development. This provides the basic documentation framework for a dataset based on a dataframe in the correct format into the right place.

Usage

```
use_dataframe(
  df,
  name = deparse(substitute(df)),
  output = "R/data.R",
  pkg = "."
)
```

Arguments

df the data frame to use

name the name of the variable you wish to use (defaults to whatever the function is called with)

output where to write data documentation code (defaults to R/data.R)

pkg the package (defaults to current)

Details

If this is your only use case for `interfacer` then you will not need to import `interfacer` in your package, as none of the generated code will depend on it.

Value

nothing, used for side effects.

Examples

```
# example code
if (interactive()) {
  # This is not run as it is designed for interactive use only and will
  # write to the userspace after checking that is what the user wants.
  use_dataframe(iris)
}
```

use_iface

Generate interfacer code for a dataframe

Description

Generating and documenting an `iface` for a given dataframe would be time consuming and annoying if you could not do it automatically. In this case as you interactively develop a package using a test dataframe, the structure of which can be explicitly documented and made into a specific contract within the package. This supports development using test dataframes as a prototype for function ensuring future user input conforms to the same expectations as the test data.

Usage

```
use_iface(
  df,
  name = deparse(substitute(df)),
  output = "R/interfaces.R",
  use_as_default = FALSE,
  pkg = "."
)
```

Arguments

<code>df</code>	the data frame to use
<code>name</code>	the name of the variable you wish to use (defaults to whatever the dataframe was called)
<code>output</code>	where within the current package to write data documentation code (defaults to <code>R/interfaces.R</code>)

`use_as_default` if this is set to true the current dataframe is saved as package data and the `interfacer::iface` specification is created referring to the package copy of the current dataframe as the default value.

`pkg` the package (defaults to current)

Value

nothing, used for side effects.

Examples

```
# example code
if (interactive()) {
  # This is not run as it is designed for interactive use only and will
  # write to the userspace after checking that is what the user wants.
  use_iface(iris)
}
```

Index

* document

- iclip, 4
- idocument, 7
- roxy_tag_parse.roxy_tag_iparam, 19
- roxy_tag_rd.roxy_tag_iparam, 20
- use_dataframe, 31
- use_iface, 32

* interface

- iconvert, 5
- idispach, 6
- iface, 8
- igroup_process, 11
- imapper, 13
- iproto, 14
- ireturn, 14
- is_iface, 15
- itest, 16
- invalidate, 17

* rules

- type.anything, 22
- type.character, 22
- type.complete, 23
- type.date, 23
- type.default, 24
- type.double, 24
- type.enum, 25
- type.factor, 25
- type.finite, 26
- type.group_unique, 26
- type.in_range, 27
- type.integer, 27
- type.logical, 28
- type.not_missing, 28
- type.numeric, 29
- type.of_type, 29
- type.positive_double, 30
- type.positive_integer, 30
- type.proportion, 31

as.list_iface, 3

format_iface, 3

- iclip, 4
- iconvert, 5
- idispach, 6
- idocument, 7
- if_col_present, 10
- iface, 8
- igroup_process, 11
- imapper, 13
- iproto, 14
- ireturn, 14
- is_iface, 15
- is_col_present, 16
- itest, 16
- invalidate, 17

knit_print_iface, 18

print_iface, 19

- roxy_tag_parse.roxy_tag_iparam, 19
- roxy_tag_rd.roxy_tag_iparam, 20

switch_pipeline, 21

- type.anything, 22
- type.character, 22
- type.complete, 23
- type.date, 23
- type.default, 24
- type.double, 24
- type.enum, 25
- type.factor, 25
- type.finite, 26
- type.group_unique, 26
- type.in_range, 27
- type.integer, 27
- type.logical, 28
- type.not_missing, 28
- type.numeric, 29

`type.of_type`, [29](#)
`type.positive_double`, [30](#)
`type.positive_integer`, [30](#)
`type.proportion`, [31](#)

`use_dataframe`, [31](#)
`use_iface`, [32](#)