Package ‘tabshiftr’

January 31, 2023

Title Reshape Disorganised Messy Data
Version 0.4.1
Description Helps the user to build and register schema descriptions of disorganised (messy) tables. Disorganised tables are tables that are not in a topologically coherent form, where packages such as ‘tidyr’ could be used for reshaping. The schema description documents the arrangement of input tables and is used to reshape them into a standardised (tidy) output format.

URL https://github.com/luckinet/tabshiftr
BugReports https://github.com/luckinet/tabshiftr/issues
Depends R (>= 2.10)
Language en-gb
License GPL-3
Encoding UTF-8
LazyData true
Imports checkmate, rlang, tibble, dplyr, tidyr, magrittr, tidyselect, testthat, crayon, methods, purrr, stringr
RoxygenNote 7.2.3
Suggests knitr, rmarkdown, bookdown, readr, covr
VignetteBuilder knitr
NeedsCompilation no
Author Steffen Ehrmann [aut, cre] (<https://orcid.org/0000-0002-2958-0796>), Tsvetelina Tomova [ctb], Carsten Meyer [aut] (<https://orcid.org/0000-0003-3927-5856>), Abdulmaged Alhemiary [ctb], Amelie Haas [ctb], Annika Ertel [ctb], Arne Rümmler [ctb] (<https://orcid.org/0000-0001-8637-9071>), Caroline Busse [ctb]
Maintainer Steffen Ehrmann <steffen.ehrmann@posteo.de>
Repository CRAN
Date/Publication 2023-01-31 13:20:02 UTC
\textbf{R topics documented:}

\begin{itemize}
  \item \texttt{.eval_find} \hspace{1cm} \begin{tabular}{r} \begin{itemize}
\item \texttt{.eval_find} \hspace{1cm} \begin{tabular}{r} \begin{itemize}
\item Evaluate \texttt{.find} constructs
\item \texttt{.eval_find(input = NULL, col = NULL, row = NULL, clusters = NULL)}
\item \texttt{input} [character(1)]
\begin{itemize}
\item table to reorganise.
\end{itemize}
\item \texttt{col} [list(2)]
\begin{itemize}
\item the output of the respective \texttt{.find} construct used to match in columns.
\end{itemize}
\item \texttt{row} [list(2)]
\begin{itemize}
\item the output of the respective \texttt{.find} construct used to match in rows.
\end{itemize}
\end{itemize}
\end{itemize}
\end{itemize}
\end{itemize}
\end{itemize}
\end{itemize}
\end{itemize}
.eval_sum

clusters [list(7)]
the cluster slot of the schema.

Value
the columns or rows of the evaluated position

Description
Evaluate .sum constructs

Usage
.eval_sum(input = NULL, groups = NULL, data = NULL)

Arguments
input [character(1)]
table to reorganise.

clusters [list(7)]
the cluster slot of the schema.

Value
the columns or rows of the evaluated position

Description
Test for a valid table

Usage
.expect_valid_table(x = NULL,
  units = 1,
  variables = NULL,
  groups = FALSE,
  flags = FALSE
)
Arguments

x: a table to test.
units: the number of units in the output table (from 1 to 3)
variables: the variables that should be in the output table (either "harvested" or "production")
groups: whether or not groups are in the test table.
flags: whether or not flags are in the test table.

Value

Either an error message of the invalid expectations, or the output of the last successful expectation.

Description

Find the location of a variable not based on it’s columns/rows, but based on a regular expression or function

Usage

```r
.find(
  fun = NULL,
  pattern = NULL,
  col = NULL,
  row = NULL,
  invert = FALSE,
  relative = FALSE
)
```

Arguments

- **fun**: [character(1)]
  - function to identify columns or rows in the input table on the fly.
- **pattern**: [character(1)]
  - character string containing a regular expression to identify columns or rows in the input table on the fly.
- **col**: [integerish(1)]
  - optionally, in case this function should only be applied to certain columns, provides this here.
- **row**: [integerish(1)]
  - optionally, in case this function should only be applied to certain rows, provides this here.
invert [logical(1)]
whether or not the identified columns or rows should be inverted, i.e., all other columns or rows should be selected.

relative [logical(1)]
whether or not the values provided in col or row are relative to the cluster position(s) or whether they are absolute positions, i.e, refer to the overall table.

Details
This function is basically a wild-card for when columns or rows are not known ad-hoc, but have to be assigned on the fly. This can be very helpful when several tables contain the same variables, but the arrangement may be slightly different.

Value
the index values where the target was found.

How does this work
The first step in using any schema is validating it via the function validateSchema. This happens by default in reorganise, but can also be done manually, for example when debugging complicated schema descriptions.

In case that function encounters a schema that wants to find columns or rows on the fly via .find, it combines all cells of columns and all cells of rows into one character string and matches the regular expression or function on those. Columns/rows that have a match are returned as the respective column/row value.

Examples
# use regular expressions to find cell positions
(input <- tabs2shift$clusters_messy)

schema <- setCluster(id = "territories",
                   left = .find(pattern = "comm*"),
                   top = .find(pattern = "comm*")) %>%
setIDVar(name = "territories", columns = c(1, 1, 4), rows = c(2, 9, 9)) %>%
setIDVar(name = "year", columns = 4, rows = c(3:6), distinct = TRUE) %>%
setIDVar(name = "commodities", columns = c(1, 1, 4)) %>%
setObsVar(name = "harvested", columns = c(2, 2, 5)) %>%
setObsVar(name = "production", columns = c(3, 3, 6))

schema
validateSchema(schema = schema, input = input)

# use a function to find rows
(input <- tabs2shift$messy_rows)

schema <-
setFilter(rows = .find(fun = is.numeric, col = 1, invert = TRUE)) %>%
setIDVar(name = "territories", columns = 1) %>%
setIDVar(name = "year", columns = 2) %>%
setIDVar(name = "commodities", columns = 3) %>%
```r
setObsVar(name = "harvested", columns = 5) %>%
setObsVar(name = "production", columns = 6)
reorganise(schema = schema, input = input)
```

---

**.sum**

*Summarise groups of rows or columns*

**Description**

Summarise groups of rows or columns

**Usage**

```
.sum(..., character = NULL, numeric = NULL)
```

**Arguments**

- `...` [integerish(1)]
  columns or rows that shall be combined. If there are several items provided, they will be summarised into one group that is combined according to its type and the respective function provided in `character` or `numeric`.

- `character` [function(1)]
  function by which character columns or rows shall be combined.

- `numeric` [function(1)]
  function by which numeric columns or rows shall be combined.

**Details**

By default character values are summarised with the function `paste0(na.omit(x), collapse = "-/-")` and numeric values with the function `sum(x, na.rm = TRUE)`. To avoid un-intuitive behavior, it is wisest to explicitly specify how all exceptions, such as NA-values, shall be handled and thus to provide a new function.

**Value**

the index values where the target was found.
### .tidyVars

**Match variables**

**Description**

This function matches id and observed variables and reshapes them accordingly.

**Usage**

```
.tidyVars(ids = NULL, obs = NULL, clust = NULL, grp = NULL)
```

**Arguments**

- `ids` list of id variables
- `obs` list of observed variables
- `clust` list of cluster variables
- `grp` list of group variables

**Value**

a symmetric list of variables (all with the same dimensions)

---

### .updateFormat

**Update the formatting of a table**

**Description**

This function updates the format of a table by applying a schema description to it.

**Usage**

```
.updateFormat(input = NULL, schema = NULL)
```

**Arguments**

- `input` [character(1)] table to reorganise.
- `schema` [character(1)] the schema description of `input`. 
### getClusterVar
**Extract cluster variables**

**Description**

This function extracts the cluster variable from a table by applying a schema description to it.

**Usage**

```r
getClusterVar(schema = NULL, input = NULL)
```

**Arguments**

- `schema` [character(1)]
  - the (validated) schema description of `input`.
- `input` [character(1)]
  - table to reorganise.

**Value**

a list per cluster with values of the cluster variable

**Examples**

```r
input <- tabs2shift$clusters_nested
schema <- setCluster(id = "sublevel",
  group = "territories", member = c(1, 1, 2),
  left = 1, top = c(3, 8, 15)) %>%
  setIDVar(name = "territories", columns = 1, rows = c(2, 14)) %>%
  setIDVar(name = "sublevel", columns = 1, rows = c(3, 8, 15)) %>%
  setIDVar(name = "year", columns = 7) %>%
  setIDVar(name = "commodities", columns = 2) %>%
  setObsVar(name = "harvested", columns = 5) %>%
  setObsVar(name = "production", columns = 6)

validateSchema(schema = schema, input = input) %>%
getClusterVar(input = input)
```

---

### getData
**Extract summarised data**

**Description**

This function extracts data from a table that are summarised by applying a schema description to it.
**getGroupVar**

**Usage**

```r
getData(schema = NULL, input = NULL)
```

**Arguments**

- `schema` [character(1)]
  - the (validated) schema description of `input`.
- `input` [character(1)]
  - table to reorganise.

**Value**

a table where columns and rows are summarised

**Examples**

```r
input <- tabs2shift$clusters_nested
schema <- setCluster(id = "sublevel",
  group = "territories", member = c(1, 1, 2),
  left = 1, top = c(3, 8, 15)) %>%
setIDVar(name = "territories", columns = 1, rows = c(2, 14)) %>%
setIDVar(name = "sublevel", columns = 1, rows = c(3, 8, 15)) %>%
setIDVar(name = "year", columns = 7) %>%
setIDVar(name = "commodities", columns = 2) %>%
setObsVar(name = "harvested", columns = 5) %>%
setObsVar(name = "production", columns = 6)

validateSchema(schema = schema, input = input) %>%
data(input = input)
```

---

**Description**

This function extracts the cluster grouping variable from a table by applying a schema description to it.

**Usage**

```r
getGroupVar(schema = NULL, input = NULL)
```

**Arguments**

- `schema` [character(1)]
  - the schema description of `input`.
- `input` [character(1)]
  - table to reorganise.
**Value**

a list per cluster with values of the grouping variable

**Examples**

```r
input <- tabs2shift$clusters_nested
schema <- setCluster(id = "sublevel",
  group = "territories", member = c(1, 1, 2),
  left = 1, top = c(3, 8, 15)) %>%
setIDVar(name = "territories", columns = 1, rows = c(2, 14)) %>%
setIDVar(name = "sublevel", columns = 1, rows = c(3, 7, 15)) %>%
setIDVar(name = "year", columns = 7) %>%
setIDVar(name = "commodities", columns = 2) %>%
setObsVar(name = "harvested", columns = 5) %>%
setObsVar(name = "production", columns = 6)

validateSchema(schema = schema, input = input) %>%
getGroupVar(input = input)
```

---

**getIDVars**

*Extract identifying variables*

**Description**

This function extracts the identifying variables from a table by applying a schema description to it.

**Usage**

```r
getIDVars(schema = NULL, input = NULL)
```

**Arguments**

- `schema`: [character(1)]
  the (validated) schema description of `input`.
- `input`: [character(1)]
  table to reorganise.

**Value**

a list per cluster with values of the identifying variables

**Examples**

```r
input <- tabs2shift$clusters_nested
schema <- setCluster(id = "sublevel",
  group = "territories", member = c(1, 1, 2),
  left = 1, top = c(3, 8, 15)) %>%
setIDVar(name = "territories", columns = 1, rows = c(2, 14)) %>%
setIDVar(name = "sublevel", columns = 1, rows = c(3, 7, 15)) %>%
setIDVar(name = "year", columns = 7) %>%
setIDVar(name = "commodities", columns = 2) %>%
setObsVar(name = "harvested", columns = 5) %>%
setObsVar(name = "production", columns = 6)

validateSchema(schema = schema, input = input) %>%
getGroupVar(input = input)
```
getObsVars

getObsVars(name = "sublevel", columns = 1, rows = c(3, 8, 15))
getObsVars(name = "year", columns = 7)
getObsVars(name = "commodities", columns = 2)
getObsVars(name = "harvested", columns = 5)
getObsVars(name = "production", columns = 6)

validateSchema(schema = schema, input = input)
getIDVars(input = input)

getObsVars

Extract observed variables

Description

This function extracts the observed variables from a table by applying a schema description to it.

Usage

getObsVars(schema = NULL, input = NULL)

Arguments

- **schema** [character(1)]
  the (validated) schema description of input.
- **input** [character(1)]
  table to reorganise.

Value

a list per cluster with values of the observed variables

Examples

input <- tabs2shift$clusters_nested
schema <- setCluster(id = "sublevel",
  group = "territories", member = c(1, 1, 2),
  left = 1, top = c(3, 8, 15))
setIDVar(name = "territories", columns = 1, rows = c(2, 14))
setIDVar(name = "sublevel", columns = 1, rows = c(3, 8, 15))
setIDVar(name = "year", columns = 7)
setIDVar(name = "commodities", columns = 2)
setObsVar(name = "harvested", columns = 5)
setObsVar(name = "production", columns = 6)

validateSchema(schema = schema, input = input)
getObsVars(input = input)
**reorganise**  
*Reorganise a table*

**Description**

This function takes a disorganised messy table and rearranges columns and rows into a tidy table based on a schema description.

**Usage**

```
reorganise(input = NULL, schema = NULL)
```

**Arguments**

- **input**
  - `data.frame()`
  - table to reorganise.
- **schema**
  - `symbol()`
  - the schema description of `input`.

**Value**

A (tidy) table which is the result of reorganising `input` based on `schema`.

**Examples**

```r
# a rather disorganised table with messy clusters and a distinct variable
(input <- tabs2shift$clusters_messy)

# put together schema description by ...
# ... identifying cluster positions
(schema <- setCluster(id = "territories", left = c(1, 1, 4), top = c(1, 8, 8))

# ... specifying the cluster ID as id variable (obligatory)
(schema <- schema %>%
  setIDVar(name = "territories", columns = c(1, 1, 4), rows = c(2, 9, 9)))

# ... specifying the distinct variable (explicit position)
(schema <- schema %>%
  setIDVar(name = "year", columns = 4, rows = c(3:6), distinct = TRUE))

# ... specifying a tidy variable (by giving the column values)
(schema <- schema %>%
  setIDVar(name = "commodities", columns = c(1, 1, 4)))

# ... identifying the (tidy) observed variables
(schema <- schema %>%
  setObsVar(name = "harvested", columns = c(2, 2, 5)) %>%
  setObsVar(name = "production", columns = c(3, 3, 6)))
```
schema-class

# get the tidy output
reorganise(input, schema)

---

schema-class

## The schema class (S4) and its methods

### Description

A schema stores the information of where which information is stored in a table of data.

### Slots

- **cluster** [list(1)]
  - description of clusters in the table.
- **format** [list(1)]
  - description of the table format
- **variables** [named list(.)]
  - description of identifying and observed variables.

### Setting up schema descriptions

This section outlines the currently recommended strategy for setting up schema descriptions. For example tables and the respective schemas, see the vignette.

1. **Variables**: Clarify which are the identifying variables and which are the observed variables. Make sure not to mistake a listed observed variable as identifying variable.
2. **Clusters**: Determine whether there are clusters and if so, find the origin (top left cell) of each cluster and provide the required information in `setCluster(top = ..., left = ...)`. It is advised to treat a table that contains meta-data in the top rows as cluster, as this is often the case with implicit variables. All variables need to be specified in each cluster (in case clusters are all organised in the same arrangement), or `relative = TRUE` can be used. Data may be organised into clusters a) whenever a set of variables occurs more than once in the same table, nested into another variable, or b) when the data are organised into separate spreadsheets or files according to one of the variables (depending on the context, these issues can also be solved differently). In both cases the variable responsible for clustering (the cluster ID) can be either an identifying variable, or a categorical observed variable:
   - in case the cluster ID is an identifying variable, provide its name in `setCluster(id = ...)` and specify it as an identifying variable (`setIDVar`)
   - in case it is a observed variable, provide simply `setCluster(..., id = "observed")`.
3. **Meta-data**: Provide potentially information about the format (`setFormat`).
4. **Identifying variables**: Determine the following:
   - is the variable available at all? This is particularly important when the data are split up into tables that are in spreadsheets or files. Often the variable that splits up the data (and thus identifies the clusters) is not explicitly available in the table anymore. In such a case, provide the value in `setIDVar(..., value = ...)`. 

---
• all columns in which the variable values sit.
• in case the variable is in several columns, determine additionally the row in which its values sit. In this case, the values will look like they are part of a header.
• in case the variable must be split off of another column, provide a regular expression that results in the target subset via `setIDVar(..., split = ...)`. 
• in case the variable is distinct from the main table, provide the explicit (non-relative) position and set `setIDVar(..., distinct = TRUE)`.

5. *Observed variable*: Determine the following:
• all columns in which the values of the variable sit.
• the unit and conversion factor.
• in case the variable is not tidy, go through the following cases one after the other:
  – in case the variable is nested in a wide identifying variable, determine in addition to the columns in which the values sit also the rows in which the *variable name* sits.
  – in case the names of the variable are given as a value of an identifying variable, give the column name as `setObsVar(..., key = ...)`, together with the name of the respective observed variable (as it appears in the table) in values.
  – in case the name of the variable is the ID of clusters, specify `setObsVar(..., key = "cluster", value = ...)`, where values has the cluster number the variable refers to.

---

**schema_default**

Default template of a schema description

**Description**

Default template of a schema description

**Usage**

`schema_default`

**Format**

The object of class schema describes at which position in a table which information can be found. It contains the four slots clusters, format, filter and variables.

The default schema description contains all slots and fields that are required by default and identifying and observed variables are added to it into the variables slot.
**setCluster**

**Set where the clusters are**

**Description**

There is hardly any limit to how data can be arranged in a spreadsheet, apart from the apparent organisation into a lattice of cells. However, it is often the case that data are gathered into topologically coherent chunks. Those chunks are what is called 'cluster' in tabshiftr.

**Usage**

```r
setCluster(
    schema = NULL,
    id = NULL,
    group = NULL,
    member = NULL,
    left = NULL,
    top = NULL,
    width = NULL,
    height = NULL
)
```

**Arguments**

- **schema**
  In case this information is added to an already existing schema, provide that schema here (overwrites previous information).

- **id**
  When data are clustered, it is typically the case that the data are segregated according to a categorical variables of interest. In such cases, this variable needs to be registered as cluster ID.

- **group**
  When clusters themselves are clustered, they are typically nested into another categorical variable, which needs to be registered as group ID.

- **member**
  For each cluster, specify here to which group it belongs. Clusters are enumerated from left to right and from top to bottom.

- **left**
  The horizontal cell value of the top-left cell of each cluster. This can also be a vector of values in case there are several clusters.

- **top**
  The vertical cell values of the top-left cell of each cluster. This can also be a vector of values in case there are several clusters.

- **width**
  The width of each cluster in cells. This can also be a vector of values in case there are several clusters.
The height of each cluster in cells. This can also be a vector of values in case there are several clusters.

Details

Please also take a look at the currently suggested strategy to set up a schema description.

Value

An object of class `schema`.

See Also

Other functions to describe table arrangement: `setFilter()`, `setFormat()`, `setGroups()`, `setIDVar()`, `setObsVar()`

Examples

# please check the vignette for examples
**setFormat**

Set the specific format of a table

**Description**

Any table makes some assumptions about the data, but they are mostly not explicitly recorded in the commonly available table format. This concerns, for example, the symbol(s) that signal "not available" values or the symbol that is used as decimal sign.

**Usage**

```r
setFormat(
    schema = NULL,
    decimal = NULL,
    thousand = NULL,
    na_values = NULL,
    flags = NULL
)
```

**Arguments**

- **schema**
  - [schema(1)]
  - In case this information is added to an already existing schema, provide that schema here (overwrites previous information).
The symbols that should be interpreted as decimal separator.

thousand [character(1)]  
The symbols that should be interpreted as thousand separator.

na_values [character(.)]  
The symbols that should be interpreted as NA.

flags [data.frame(2)]  
The typically character based flags that should be shaved off of observed variables to make them identifiable as numeric values. This must be a data.frame with two columns with names flag and value.

Details  
Please also take a look at the currently suggested strategy to set up a schema description.

Value  
An object of class schema.

See Also  
Other functions to describe table arrangement: setCluster(), setFilter(), setGroups(), setIDVar(), setObsVar()

Examples  
# please check the vignette for examples

---

### setGroups

**Set Groups**

**Description**

This function allows to set groups for rows, columns or clusters that shall be summarised.

**Usage**

setGroups(schema = NULL, rows = NULL, columns = NULL)

**Arguments**

- **schema** [schema(1)]  
  In case this information is added to an already existing schema, provide that schema here (overwrites previous information).

- **rows** [list(3)]  
  the output of .sum indicating the rows and a function according to which those rows should be summarised.
setIDVar

columns \[\text{list(3)}\]
the output of .sum indicating the columns and a function according to which those columns should be summarised.

Value
An object of class schema.

See Also
Other functions to describe table arrangement: setCluster(), setFilter(), setFormat(), setIDVar(), setObsVar()

Examples
# please check the vignette for examples

---

setIDVar

Set an identifying variable

Description
Identifying variables are those variables that describe the (qualitative) properties that make each observation (as described by the observed variables) unique.

Usage

```r
setIDVar(
  schema = NULL,
  name = NULL,
  value = NULL,
  columns = NULL,
  rows = NULL,
  split = NULL,
  merge = NULL,
  distinct = FALSE
)
```

Arguments

- `schema` [schema(1)]
  In case this information is added to an already existing schema, provide that schema here (overwrites previous information).
- `name` [character(1)]
  Name of the new identifying variable.
- `value` [character(1)]
  In case the variable is an implicit variable (i.e., which is not in the origin table), specify it here.
setObsVar

columns [integerish(.)]
The column(s) in which the values of the new variable are recorded.

rows [integerish(.)]
In case the variable is in several columns, specify here additionally the row in which the names are recorded.

split [character(1)]
In case the variable is part of a compound value, this should be a regular expression that splits the respective value off of that compound value. See extract on how to set up the regular expression.

merge [character(1)]
In case a variable is made up of several columns, this should be the character string that would connect the two columns (e.g., an empty space " ").

distinct [logical(1)]
whether or not the variable is distinct from a cluster. This is the case when the variable is not systematically available for all clusters and thus needs to be registered separately from clusters.

Details

Please also take a look at the currently suggested strategy to set up a schema description.

Value

An object of class schema.

See Also

Other functions to describe table arrangement: setCluster(), setFilter(), setFormat(), setGroups(), setObsVar()

Examples

# please check the vignette for examples

---

setObsVar

Set an observed variable

Description

Observed variables are those variables that contain the (quantitative) observed/measured values of each unique unit (as described by the identifying variables). There may be several of them and in a tidy table they’d be recorded as separate columns.
**setObsVar**

**Usage**

```r
setObsVar(
    schema = NULL,
    name = NULL,
    columns = NULL,
    top = NULL,
    distinct = FALSE,
    unit = NULL,
    factor = 1,
    key = NULL,
    value = NULL
)
```

**Arguments**

- **schema**
  - `[schema(1)]`
  - In case this information is added to an already existing schema, provide that schema here (overwrites previous information).

- **name**
  - `[character(1)]`
  - Name of the new measured variable.

- **columns**
  - `[integerish(.)]`
  - The column(s) in which the values of the new variable are recorded.

- **top**
  - `[integerish(.)]`
  - In case the variable is nested in a wide identifying variable, specify here additionally the topmost row in which the variable `name` sits.

- **distinct**
  - `[logical(1)]`
  - Whether or not the variable is distinct from a cluster. This is the case when the variable is recorded somewhere `on the side` and thus not explicitly included in all clusters.

- **unit**
  - `[character(1)]`
  - the unit of this variable.

- **factor**
  - `[numeric(1)]`
  - the factor that needs to be multiplied with the values to convert to `unit`, defaults to 1. For instance, if values are recorded in acres, but shall be recorded in hectare, the factor would be 0.40468.

- **key**
  - `[integerish(1)]`
  - If the variable is recorded (together with other variables) so that the variable names are listed in one column and the respective values are listed in another column, give here the number of the column that contains the variable names. Can alternatively be "cluster", in case observed variables are the cluster ID.

- **value**
  - `[character(1)]`
  - If the variable is recorded (together with other variables) so that the variable names are listed in one column and the respective values are listed in another column, give here the level in the names column that refer to the values of this variable.
Details
Please also take a look at the currently suggested strategy to set up a schema description.

Value
An object of class schema.

See Also
Other functions to describe table arrangement: setCluster(), setFilter(), setFormat(), setGroups(), setIDVar()

Examples
# please check the vignette for examples

---

show, schema-method \hspace{1cm} \textit{Print the schema}

Description
Print the schema

Usage
## S4 method for signature 'schema'
show(object)

Arguments

object \hspace{1cm} [schema]
the schema to print.

---

tabs2shift \hspace{1cm} \textit{List of table types}

Description
List of table types

Usage
tabs2shift

Format
The object of class list contains 20 different types of tables that are used throughout the unit-tests and examples/vignette.
validateSchema

Check and update schema descriptions

Description

This function takes a raw schema description and updates values that were only given as wildcard or implied values. It is automatically called by reorganise, but can also be used in concert with the getters to debug a schema.

Usage

validateSchema(schema = NULL, input = NULL)

Arguments

schema [symbol(1)]
the schema description.

input [data.frame(1)]
an input for which to check a schema description.

Details

The core idea of a schema description is that it can be written in a very generic way, as long as it describes sufficiently where in a table what variable can be found. A very generic way can be via using the function .find to identify the initially unknown cell-locations of a variable on-the-fly, for example when it is merely known that a variable must be in the table, but not where it is.

validateSchema matches a schema with an input table and inserts the accordingly evaluated positions (of clusters, filters and variables), adapts some of the meta-data and ensures formal consistency of the schema.

Value

An updated schema description

Examples

# build a schema for an already tidy table
(tidyTab <- tabs2shift$tidy)

schema <-
  setIDVar(name = "territories", col = 1) %>%
  setIDVar(name = "year", col = .find(pattern = "period")) %>%
  setIDVar(name = "commodities", col = 3) %>%
  setObsVar(name = "harvested", col = 5) %>%
  setObsVar(name = "production", col = 6)

# before ...
schema
# ... after
validateSchema(schema = schema, input = tidyTab)
Index

* datasets
  schema_default, 14
  tabs2shift, 22
* functions to describe table arrangement
  setCluster, 15
  setFilter, 16
  setFormat, 17
  setGroups, 18
  setIDVar, 19
  setObsVar, 20
  .eval_find, 2
  .eval_sum, 3
  .expect_valid_table, 3
  .find, 4, 23
  .sum, 6, 18, 19
  .tidyVars, 7
  .updateFormat, 7

clusters, 13

extract, 20

format, 13

getClusterVar, 8
getData, 8
getGroupVar, 9
getIDVars, 10
getObsVars, 11

identifying, 13

Logic, 16

observed, 13

reorganise, 3, 5, 12

schema, 16–20, 22
schema (schema-class), 13
schema description, 16, 18, 20, 22

schema-class, 13
schema_default, 14
setCluster, 13, 15, 17–20, 22
setFilter, 16, 16, 18–20, 22
setFormat, 13, 16, 17, 17, 19, 20, 22
setGroups, 16–18, 18, 20, 22
setIDVar, 13, 14, 16–19, 19, 22
setObsVar, 14, 16–20, 20
show, schema-method, 22

tabs2shift, 22

validateSchema, 5, 23