Package ‘pointblank’

January 10, 2020

Type Package

Version 0.3.0

Title Validation of Local and Remote Data Tables

Description Validate data in data frames, ‘tibble’ objects, and in database
tables (e.g., ‘PostgreSQL’ and ‘MySQL’). Validation pipelines can be made
using easily-readable, consecutive validation steps. Upon execution of the
validation plan, several reporting options are available. User-defined
thresholds for failure rates allow for the determination of appropriate
reporting actions.

License MIT + file LICENSE

URL https://github.com/rich-iannone/pointblank

BugReports https://github.com/rich-iannone/pointblank/issues

Encoding UTF-8

LazyData true

ByteCompile true

RoxygenNote 7.0.2

Depends R (>= 3.5.0)

Imports dplyr (>= 0.8.3), magrittr, rlang (>= 0.4.2), tidyselect (>=
0.2.5)

Suggests readr, testthat, covr

Enhances gt (>= 0.1.0)

Additional_repositories http://ddsjoberg.github.io/drat

NeedsCompilation no

Author Richard Iannone [aut, cre] (<https://orcid.org/0000-0003-3925-190X>)

Maintainer Richard Iannone <riannone@me.com>

Repository CRAN

Date/Publication 2020-01-10 10:50:10 UTC
R topics documented:

- **action_levels**: Set action levels for reacting to exceeding thresholds

**Description**

This helper function works with the `actions` argument that is present in every validation step function. With it, we can provide threshold `fail` levels for any combination of `warn`, `stop`, or `notify` states. We can react to any entrance of a state by supplying corresponding functions to the `fns` argument. They will undergo evaluation at the time when the matching state is entered.
**Usage**

```
action_levels(warn_at = NULL, stop_at = NULL, notify_at = NULL, fns = NULL)
```

**Arguments**

- **warn_at**, **stop_at**, **notify_at**
  The threshold number or fraction of validation units that can provide a *fail* result before entering the warn, stop, or notify states.

- **fns**
  A named list of functions that can be used with each action type. The syntax for this list involves using names from the set of warn, stop, and notify. The functions corresponding to the states are provided as formulas (e.g., `list(warn = ~ warning("Too many failures.")))`. A series of expressions for each named state can be used by enclosing the set of statements with `{ }`.

**Details**

The output of the `action_levels()` call in `actions` will be interpreted slightly different if using an `agent` or using a validation step function directly on a data table. For convenience, when working directly on data any values supplied to `warn_at` or `stop_at` will be automatically given a stock `warning()` or `stop()` function. If you were to supply those manually then the stock functions would be overridden. In the that interactive data case there is no automatic reaction function given for the notify state (as that state is less commonly used and should instead be intended for custom reporting functions).

When using an `agent`, we often opt to not use any functions in `fns` as the warn, stop, and notify states will be reported on when using `create_agent_report()` (and, usually that’s sufficient).

**Examples**

```r
library(dplyr)

# Create a simple data frame with
# a column of numerical values
tbl <- tibble(a = c(5, 7, 8, 5))

# Create an `action_levels()` list
# with fractional values for the
# `warn`, `stop`, and `notify` states
al <-
    action_levels(
        warn_at = 0.2,
        stop_at = 0.8,
        notify_at = 0.345
    )

# Validate that values in column
# `a` are always greater than 7 and
# apply the list of action levels
agent <-
    create_agent(tbl = tbl) %>%
    col_vals_gt(vars(a), 7, actions = al) %>%
```

```
Given an agent’s validation plan that had undergone interrogation via `interrogate()`, did every single validation step result in zero `fail` levels? Using the `all_passed()` function will let us know whether that’s `TRUE` or not.

**Usage**

```r
all_passed(agent)
```

**Arguments**

- `agent` An agent object of class `ptblank_agent`.

**Value**

A logical value.

**Function ID**

3-2

**See Also**

Other Interrogate and Get Info: `get_agent_report()`, `get_data_extracts()`, `interrogate()`
Examples

```r
library(dplyr)

# Create a simple table with
# a column of numerical values
tbl <- tibble(a = c(5, 7, 8, 5))

# Validate that values in column
# `a` are always greater than 4
agent <-
  create_agent(tbl = tbl) %>%
  col_vals_gt(vars(a), 4) %>%
  interrogate()

# Determine if these column
# validations have all passed
# by using `all_passed()`
all_passed(agent)
```

---

### col_exists

**Do one or more columns actually exist?**

#### Description

The `col_exists()` validation step function checks whether one or more columns exist in the target table. The only requirement is a specification of the column names. Each validation step will operate over a single test unit, which is whether the column exists or not.

#### Usage

```r
col_exists(x, columns, actions = NULL, brief = NULL)
```

#### Arguments

- **x**: A data frame, tibble, or an agent object of class `ptblank_agent`.
- **columns**: One or more columns from the table in focus. This can be provided as a vector of column names using `()` or bare column names enclosed in `vars()`.
- **actions**: A list containing threshold levels so that the validation step can react accordingly when exceeding the set levels. This is to be created with the `action_levels()` helper function.
- **brief**: An optional, text-based description for the validation step.
Details

If providing multiple column names, the result will be an expansion of validation steps to that number of column names (e.g., vars(col_a, col_b) will result in the entry of two validation steps). Aside from column names in quotes and in vars(), tidyselect helper functions are available for specifying columns. They are: starts_with(), ends_with(), contains(), matches(), and everything().

Often, we will want to specify actions for the validation. This argument, present in every validation step function, takes a specially-crafted list object that is best produced by the action_levels() function. Read that function’s documentation for the lowdown on how to create reactions to above-threshold failure levels in validation. The basic gist is that you’ll want at least a single threshold level (specified as either the fraction test units failed, or, an absolute value), often using the warn_at argument. Using action_levels(warn_at = 1) or action_levels(stop_at = 1) are good choices depending on the situation (the first produces a warning, the other stop()s).

Want to describe this validation step in some detail? Keep in mind that this is only useful if x is an agent. If that’s the case, brief the agent with some text that fits. Don’t worry if you don’t want to do it. The autobrief protocol is kicked in when brief = NULL and a simple brief will then be automatically generated.

Value

Either a ptblank_agent object or a table object, depending on what was passed to x.

Function ID

2-23

See Also

Other Validation Step Functions: col_is_character(), col_is_date(), col_is_factor(), col_is_integer(), col_is_logical(), col_is_numeric(), col_is posix(), col_vals_between(), col_vals_equal(), col_vals_gte(), col_vals_gt(), col_vals_in_set(), col_vals_lte(), col_vals_lt(), col_vals_not_between(), col_vals_not_equal(), col_vals_not_in_set(), col_vals_not_null(), col_vals_null(), col_vals_regex(), conjointly(), rows_distinct()
create_agent(tbl = tbl) %>%
col_exists(vars(a, b)) %>%
interrogate()

# Determine if these three validation
# steps passed by using `all_passed`
all_passed(agent)

col_is_character  Do the columns contain character/string data?

Description

The `col_is_character()` validation step function checks whether one or more columns is of the character type. Like many of the `col_is_*()`-type functions in `pointblank`, the only requirement is a specification of the column names. This function can be used directly on a data table or with an `agent` object (technically, a `ptblank_agent` object). Each validation step will operate over a single test unit, which is whether the column is a character-type column or not.

Usage

col_is_character(x, columns, actions = NULL, brief = NULL)

Arguments

x A data frame, tibble, or an agent object of class `ptblank_agent`.

columns The column (or a set of columns, provided as a character vector) to which this validation should be applied.

actions A list containing threshold levels so that the validation step can react accordingly when exceeding the set levels. This is to be created with the `action_levels()` helper function.

brief An optional, text-based description for the validation step.

Details

If providing multiple column names, the result will be an expansion of validation steps to that number of column names (e.g., `vars(col_a, col_b)` will result in the entry of two validation steps). Aside from column names in quotes and in `vars()`, `tidyselect` helper functions are available for specifying columns. They are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, and `everything()`.

Often, we will want to specify actions for the validation. This argument, present in every validation step function, takes a specially-crafted list object that is best produced by the `action_levels()` function. Read that function’s documentation for the lowdown on how to create reactions to above-threshold failure levels in validation. The basic gist is that you’ll want at least a single threshold level (specified as either the fraction test units failed, or, an absolute value), often using the `warn_at` argument. This is especially true when `x` is a table object because, otherwise, nothing happens. For
the `col_is_*()`-type functions, using `action_levels(warn_at = 1)` or `action_levels(stop_at = 1)` are good choices depending on the situation (the first produces a warning, the other stops).

Want to describe this validation step in some detail? Keep in mind that this is only useful if `x` is an agent. If that’s the case, brief the agent with some text that fits. Don’t worry if you don’t want to do it. The `autobrief` protocol is kicked in when `brief = NULL` and a simple brief will then be automatically generated.

**Value**

Either a `ptblank_agent` object or a table object, depending on what was passed to `x`.

**Function ID**

2-16

**See Also**

Other Validation Step Functions: `col_exists()`, `col_is_date()`, `col_is_factor()`, `col_is_integer()`, `col_is_logical()`, `col_is_numeric()`, `col_is_posix()`, `col_vals_between()`, `col_vals_equal()`, `col_vals_gte()`, `col_vals_gt()`, `col_vals_in_set()`, `col_vals_lte()`, `col_vals_lt()`, `col_vals_not_between()`, `col_vals_not_equal()`, `col_vals_not_in_set()`, `col_vals_not_null()`, `col_vals_null()`, `col_vals_regex()`, `conjointly()`, `rows_distinct()`

**Examples**

```r
library(dplyr)

# Create a simple table with
# a column of `character` values
tbl <- tibble(a = c("one", "two"))

# Validate that column `a` in the
# table is classed as `character`
agent <-
  create_agent(tbl = tbl) %>%
  col_is_character(vars(a)) %>%
  interrogate()

# Determine if these column
# validations have all passed
# by using `all_passed()`
all_passed(agent)
```
Description

The col_is_date() validation step function checks whether one or more columns is of the \texttt{R} \texttt{Date} type. Like many of the col_is_*()-type functions in \texttt{pointblank}, the only requirement is a specification of the column names. This function can be used directly on a data table or with an \textit{agent} object (technically, a \texttt{ptblank\_agent} object). Each validation step will operate over a single test unit, which is whether the column is a Date-type column or not.

Usage

col_is_date(x, columns, actions = NULL, brief = NULL)

Arguments

- \texttt{x} \hspace{1cm} A data frame, tibble, or an agent object of class \texttt{ptblank\_agent}.
- \texttt{columns} \hspace{1cm} The column (or a set of columns, provided as a character vector) to which this validation should be applied.
- \texttt{actions} \hspace{1cm} A list containing threshold levels so that the validation step can react accordingly when exceeding the set levels. This is to be created with the \texttt{action_levels()} helper function.
- \texttt{brief} \hspace{1cm} An optional, text-based description for the validation step.

Details

If providing multiple column names, the result will be an expansion of validation steps to that number of column names (e.g., \texttt{vars(col\_a, col\_b)} will result in the entry of two validation steps). Aside from column names in quotes and in \texttt{vars()}, \texttt{tidyselect} helper functions are available for specifying columns. They are: \texttt{starts\_with()}, \texttt{ends\_with()}, \texttt{contains()}, \texttt{matches()}, and \texttt{everything()}.

Often, we will want to specify actions for the validation. This argument, present in every validation step function, takes a specially-crafted list object that is best produced by the \texttt{action_levels()} function. Read that function’s documentation for the lowdown on how to create reactions to above-threshold failure levels in validation. The basic gist is that you’ll want at least a single threshold level (specified as either the fraction test units failed, or, an absolute value), often using the \texttt{warn\_at} argument. This is especially true when \texttt{x} is a table object because, otherwise, nothing happens. For the col_is_*()-type functions, using \texttt{action_levels(warn\_at = 1)} or \texttt{action_levels(stop\_at = 1)} are good choices depending on the situation (the first produces a warning, the other \texttt{stop}()s).

Want to describe this validation step in some detail? Keep in mind that this is only useful if \texttt{x} is an \textit{agent}. If that’s the case, \texttt{brief} the agent with some text that fits. Don’t worry if you don’t want to do it. The \texttt{autobrief} protocol is kicked in when \texttt{brief = NULL} and a simple brief will then be automatically generated.
col_is_factor

Value

Either a ptblank_agent object or a table object, depending on what was passed to x.

Function ID

2-20

See Also

Other Validation Step Functions: col_exists(), col_is_character(), col_is_factor(), col_is_integer(), col_is_logical(), col_is_numeric(), col_is_posix(), col_vals_between(), col_vals_equal(), col_vals_gte(), col_vals_gt(), col_vals_in_set(), col_vals_lte(), col_vals_lt(), col_vals_not_between(), col_vals_not_equal(), col_vals_not_in_set(), col_vals_not_null(), col_vals_null(), col_vals_regex(), conjointly(), rows_distinct()

Examples

library(dplyr)

# Create a simple table with
# a column of `Date` values
tbl <- tibble(a = as.Date("2017-08-15"))

# Validate that column `a` in the
# table is classed as `Date`
agent <-
  create_agent(tbl = tbl) %>%
  col_is_date(vars(a)) %>%
  interrogate()

# Determine if these column
# validations have all passed
# by using `all_passed()`
all_passed(agent)

col_is_factor  Do the columns contain R factor objects?

Description

The col_is_factor() validation step function checks whether one or more columns is of the factor type. Like many of the col_is_*()-type functions in pointblank, the only requirement is a specification of the column names. This function can be used directly on a data table or with an agent object (technically, a ptblank_agent object). Each validation step will operate over a single test unit, which is whether the column is a factor-type column or not.
col_is_factor

Usage

col_is_factor(x, columns, actions = NULL, brief = NULL)

Arguments

x  A data frame, tibble, or an agent object of class \texttt{ptblank\_agent}.
columns  The column (or a set of columns, provided as a character vector) to which this validation should be applied.
actions  A list containing threshold levels so that the validation step can react accordingly when exceeding the set levels. This is to be created with the \texttt{action\_levels()} helper function.
brief  An optional, text-based description for the validation step.

Details

If providing multiple column names, the result will be an expansion of validation steps to that number of column names (e.g., \texttt{vars(col\_a,col\_b)} will result in the entry of two validation steps). Aside from column names in quotes and in \texttt{vars()}, \texttt{tidyselect} helper functions are available for specifying columns. They are: \texttt{starts\_with()}, \texttt{ends\_with()}, \texttt{contains()}, \texttt{matches()}, and \texttt{everything()}.

Often, we will want to specify actions for the validation. This argument, present in every validation step function, takes a specially-crafted list object that is best produced by the \texttt{action\_levels()} function. Read that function’s documentation for the lowdown on how to create reactions to above-threshold failure levels in validation. The basic gist is that you’ll want at least a single threshold level (specified as either the fraction test units failed, or, an absolute value), often using the \texttt{warn\_at} argument. This is especially true when \texttt{x} is a table object because, otherwise, nothing happens. For the \texttt{col\_is\_\*()}-type functions, using \texttt{action\_levels(warn\_at = 1)} or \texttt{action\_levels(stop\_at = 1)} are good choices depending on the situation (the first produces a warning, the other \texttt{stop()}).

Want to describe this validation step in some detail? Keep in mind that this is only useful if \texttt{x} is an \textit{agent}. If that’s the case, \texttt{brief} the agent with some text that fits. Don’t worry if you don’t want to do it. The \texttt{autobrief} protocol is kicked in when \texttt{brief = NULL} and a simple brief will then be automatically generated.

Value

Either a \texttt{ptblank\_agent} object or a table object, depending on what was passed to \texttt{x}.

Function ID

2-22

See Also

Other Validation Step Functions: \texttt{col\_exists()}, \texttt{col\_is\_character()}, \texttt{col\_is\_date()}, \texttt{col\_is\_integer()}, \texttt{col\_is\_logical()}, \texttt{col\_is\_numeric()}, \texttt{col\_is\_posix()}, \texttt{col\_vals\_between()}, \texttt{col\_vals\_equal()}, \texttt{col\_vals\_gte()}, \texttt{col\_vals\_gt()}, \texttt{col\_vals\_in\_set()}, \texttt{col\_vals\_lte()}, \texttt{col\_vals\_lt()}, \texttt{col\_vals\_not\_between()}, \texttt{col\_vals\_not\_equal()}, \texttt{col\_vals\_not\_in\_set()}, \texttt{col\_vals\_not\_null()}, \texttt{col\_vals\_null()}, \texttt{col\_vals\_regex()}, \texttt{conjointly()}, \texttt{rows\_distinct()}
Examples

library(dplyr)

# Create a simple table with
# a column of 'factor' values
tbl <- tibble(a = factor(c("one", "two")))

# Validate that column 'a' in the
# table is classed as 'factor'
agent <-
  create_agent(tbl = tbl) %>%
  col_is_factor(vars(a)) %>%
  interrogate()

# Determine if these column
# validations have all passed
# by using 'all_passed()'
all_passed(agent)

---

**col_is_integer**  
*Do the columns contain integer values?*

Description

The `col_is_integer()` validation step function checks whether one or more columns is of the integer type. Like many of the `col_is_*()`-type functions in `pointblank`, the only requirement is a specification of the column names. This function can be used directly on a data table or with an agent object (technically, a `ptblank_agent` object). Each validation step will operate over a single test unit, which is whether the column is an integer-type column or not.

Usage

```
col_is_integer(x, columns, actions = NULL, brief = NULL)
```

Arguments

- `x`  
  A data frame, tibble, or an agent object of class `ptblank_agent`.

- `columns`  
  The column (or a set of columns, provided as a character vector) to which this validation should be applied.

- `actions`  
  A list containing threshold levels so that the validation step can react accordingly when exceeding the set levels. This is to be created with the `action_levels()` helper function.

- `brief`  
  An optional, text-based description for the validation step.
Details

If providing multiple column names, the result will be an expansion of validation steps to that number of column names (e.g., `vars(col_a, col_b)` will result in the entry of two validation steps). Aside from column names in quotes and in `vars()`, `tidyselect` helper functions are available for specifying columns. They are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, and `everything()`.

Often, we will want to specify actions for the validation. This argument, present in every validation step function, takes a specially-crafted list object that is最好 produced by the `action_levels()` function. Read that function’s documentation for the lowdown on how to create reactions to above-threshold failure levels in validation. The basic gist is that you’ll want at least a single threshold level (specified as either the fraction test units failed, or, an absolute value), often using the `warn_at` argument. This is especially true when x is a table object because, otherwise, nothing happens. For the `col_is_`()-type functions, using `action_levels(warn_at = 1)` or `action_levels(stop_at = 1)` are good choices depending on the situation (the first produces a warning, the other `stop()`s).

Want to describe this validation step in some detail? Keep in mind that this is only useful if x is an agent. If that’s the case, `brief` the agent with some text that fits. Don’t worry if you don’t want to do it. The `autobrief` protocol is kicked in when `brief = NULL` and a simple brief will then be automatically generated.

Value

Either a `ptblank_agent` object or a table object, depending on what was passed to x.

Function ID

2-18

See Also

Other Validation Step Functions: `col_checks()`, `col_is_character()`, `col_is_date()`, `col_is_factor()`, `col_is_logical()`, `col_is_numeric()`, `col_is POSIX()`, `col_vals_between()`, `col_vals_equal()`, `col_vals_gte()`, `col_vals_gt()`, `col_vals_in_set()`, `col_vals_lte()`, `col_vals_lt()`, `col_vals_not_between()`, `col_vals_not_equal()`, `col_vals_not_in_set()`, `col_vals_not_null()`, `col_vals_null()`, `col_vals_regex()`, `conjointly()`, `rows_distinct()`

Examples

```r
library(dplyr)

# Create a simple table with a # column of 'integer' values
tbl <- tibble(a = c(5L, 9L, 3L))

# Validate that column 'a' in the # table is classed as 'integer'
agent <-
  create_agent(tbl = tbl) %>%
  col_is_integer(vars(a)) %>%
  interrogate()
```
# Determine if these column validations have all passed 
# by using `all_passed()`
all_passed(agent)

col_is_logical  
*Do the columns contain logical values?*

**Description**

The `col_is_logical()` validation step function checks whether one or more columns is of the logical (TRUE/FALSE) type. Like many of the `col_is_*()`-type functions in pointblank, the only requirement is a specification of the column names. This function can be used directly on a data table or with an agent object (technically, a ptblank_agent object). Each validation step will operate over a single test unit, which is whether the column is an logical-type column or not.

**Usage**

```r
col_is_logical(x, columns, actions = NULL, brief = NULL)
```

**Arguments**

- `x`  
A data frame, tibble, or an agent object of class ptblank_agent.

- `columns`  
The column (or a set of columns, provided as a character vector) to which this validation should be applied.

- `actions`  
A list containing threshold levels so that the validation step can react accordingly when exceeding the set levels. This is to be created with the `action_levels()` helper function.

- `brief`  
An optional, text-based description for the validation step.

**Details**

If providing multiple column names, the result will be an expansion of validation steps to that number of column names (e.g., `vars(col_a, col_b)` will result in the entry of two validation steps). Aside from column names in quotes and in `vars()`, tidyselect helper functions are available for specifying columns. They are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, and `everything()`.

Often, we will want to specify actions for the validation. This argument, present in every validation step function, takes a specially-crafted list object that is best produced by the `action_levels()` function. Read that function’s documentation for the lowdown on how to create reactions to above-threshold failure levels in validation. The basic gist is that you’ll want at least a single threshold level (specified as either the fraction test units failed, or, an absolute value), often using the `warn_at` argument. This is especially true when `x` is a table object because, otherwise, nothing happens. For the `col_is_*()`-type functions, using `action_levels(warn_at = 1)` or `action_levels(stop_at = 1)` are good choices depending on the situation (the first produces a warning, the other `stop()`s).
Want to describe this validation step in some detail? Keep in mind that this is only useful if \( x \) is an *agent*. If that’s the case, brief the agent with some text that fits. Don’t worry if you don’t want to do it. The *autobrief* protocol is kicked in when \( \text{brief} = \text{NULL} \) and a simple brief will then be automatically generated.

**Value**

Either a `ptblank_agent` object or a table object, depending on what was passed to \( x \).

**Function ID**

2-19

**See Also**

Other Validation Step Functions: `col_exists()`, `col_is_character()`, `col_is_date()`, `col_is_factor()`, `col_is_integer()`, `col_is_numeric()`, `col_is_posix()`, `col_vals_between()`, `col_vals_equal()`, `col_vals_gte()`, `col_vals_gt()`, `col_vals_in_set()`, `col_vals_lte()`, `col_vals_lt()`, `col_vals_not_between()`, `col_vals_not_equal()`, `col_vals_not_in_set()`, `col_vals_not_null()`, `col_vals_null()`, `col_vals_regex()`, `conjointly()`, `rows_distinct()`

**Examples**

```r
library(dplyr)

# Create a simple table with a column of 'logical' values
tbl <- tibble(a = c(TRUE, FALSE))

# Validate that column 'a' in the table is classed as 'logical'
agent <-
  create_agent(tbl = tbl) %>%
  col_is_logical(vars(a)) %>%
  interrogate()

# Determine if this column validation has passed by using
# 'all_passed()'
all_passed(agent)
```

<table>
<thead>
<tr>
<th>col_is_numeric</th>
<th>Do the columns contain numeric values?</th>
</tr>
</thead>
</table>

```
Description

The `col_is_numeric()` validation step function checks whether one or more columns is of the numeric type. Like many of the `col_is_*()`-type functions in `pointblank`, the only requirement is a specification of the column names. This function can be used directly on a data table or with an `agent` object (technically, a `ptblank_agent` object). Each validation step will operate over a single test unit, which is whether the column is a numeric-type column or not.

Usage

```r
col_is_numeric(x, columns, actions = NULL, brief = NULL)
```

Arguments

- **x**: A data frame, tibble, or an agent object of class `ptblank_agent`.
- **columns**: The column (or a set of columns, provided as a character vector) to which this validation should be applied.
- **actions**: A list containing threshold levels so that the validation step can react accordingly when exceeding the set levels. This is to be created with the `action_levels()` helper function.
- **brief**: An optional, text-based description for the validation step.

Details

If providing multiple column names, the result will be an expansion of validation steps to that number of column names (e.g., `vars(col_a, col_b)` will result in the entry of two validation steps). Aside from column names in quotes and in `vars()`, `tidyselect` helper functions are available for specifying columns. They are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, and `everything()`.

Often, we will want to specify actions for the validation. This argument, present in every validation step function, takes a specially-crafted list object that is best produced by the `action_levels()` function. Read that function’s documentation for the lowdown on how to create reactions to above-threshold failure levels in validation. The basic gist is that you’ll want at least a single threshold level (specified as either the fraction test units failed, or, an absolute value), often using the `warn_at` argument. This is especially true when `x` is a table object because, otherwise, nothing happens. For the `col_is_*()`-type functions, using `action_levels(warn_at = 1)` or `action_levels(stop_at = 1)` are good choices depending on the situation (the first produces a warning, the other `stop()`s).

Want to describe this validation step in some detail? Keep in mind that this is only useful if `x` is an `agent`. If that’s the case, `brief` the agent with some text that fits. Don’t worry if you don’t want to do it. The `autobrief` protocol is kicked in when `brief = NULL` and a simple brief will then be automatically generated.

Value

Either a `ptblank_agent` object or a table object, depending on what was passed to `x`.

Function ID

2-17
See Also

Other Validation Step Functions: `col_exists()`, `col_is_character()`, `col_is_date()`, `col_is_factor()`, `col_is_integer()`, `col_is_logical()`, `col_is_posix()`, `col_vals_between()`, `col_vals_equal()`, `col_vals_gte()`, `col_vals_gt()`, `col_vals_in_set()`, `col_vals_lte()`, `col_vals_lt()`, `col_vals_not_between()`, `col_vals_not_equal()`, `col_vals_not_in_set()`, `col_vals_not_null()`, `col_vals_null()`, `col_vals_regex()`, `conjointly()`, `rows_distinct()`

Examples

```r
library(dplyr)

# Create a simple table with a column of 'numeric' values
tbl <- tibble(a = c(5.1, 2.9))

# Validate that column 'a' in the table is classed as 'numeric'
agent <- create_agent(tbl = tbl) %>%
          col_is_numeric(vars(a)) %>%
          interrogate()

# Determine if this column validation has passed by using `all_passed()`
all_passed(agent)
```

---

### col_is_posix

Do the columns contain POSIXct dates?

**Description**

The `col_is_posix()` validation step function checks whether one or more columns is of the R POSIXct date-time type. Like many of the `col_is_*()`-type functions in `pointblank`, the only requirement is a specification of the column names. This function can be used directly on a data table or with an `agent` object (technically, a `ptblank_agent` object). Each validation step will operate over a single test unit, which is whether the column is a POSIXct-type column or not.

**Usage**

```r
col_is_posix(x, columns, actions = NULL, brief = NULL)
```

**Arguments**

- `x`  
  A data frame, tibble, or an agent object of class `ptblank_agent`.
- `columns`  
  The column (or a set of columns, provided as a character vector) to which this validation should be applied.
actions
A list containing threshold levels so that the validation step can react accordingly when exceeding the set levels. This is to be created with the action_levels() helper function.

brief
An optional, text-based description for the validation step.

Details
If providing multiple column names, the result will be an expansion of validation steps to that number of column names (e.g., vars(col_a, col_b) will result in the entry of two validation steps). Aside from column names in quotes and in vars(), tidyselect helper functions are available for specifying columns. They are: starts_with(), ends_with(), contains(), matches(), and everything().

Often, we will want to specify actions for the validation. This argument, present in every validation step function, takes a specially-crafted list object that is best produced by the action_levels() function. Read that function’s documentation for the lowdown on how to create reactions to above-threshold failure levels in validation. The basic gist is that you’ll want at least a single threshold level (specified as either the fraction test units failed, or, an absolute value), often using the warn_at argument. This is especially true when x is a table object because, otherwise, nothing happens. For the col_is_*()-type functions, using action_levels(warn_at = 1) or action_levels(stop_at = 1) are good choices depending on the situation (the first produces a warning, the other stop()s).

Want to describe this validation step in some detail? Keep in mind that this is only useful if x is an agent. If that’s the case, brief the agent with some text that fits. Don’t worry if you don’t want to do it. The autobrief protocol is kicked in when brief = NULL and a simple brief will then be automatically generated.

Verification step where a table column is expected to consist entirely of R POSIXct dates.

Value
Either a ptblank_agent object or a table object, depending on what was passed to x.

Function ID
2-18

See Also
Other Validation Step Functions: col_exists(), col_is_character(), col_is_date(), col_is_factor(), col_is_integer(), col_is_logical(), col_is_numeric(), col_vals_between(), col_vals_equal(), col_vals_gte(), col_vals_gt(), col_vals_in_set(), col_vals_lte(), col_vals_lt(), col_vals_not_between(), col_vals_not_equal(), col_vals_not_in_set(), col_vals_not_null(), col_vals_null(), col_vals_regex(), conjointly(), rows_distinct()

Examples
library(dplyr)

# Create a simple table with a
# column of `POSIXct` values
tbl <-
tibble(
  a = as.POSIXct(
    strptime(
      "2011-03-27 01:30:00",
      "%Y-%m-%d %H:%M:%S")
  )
)

# Validate that column `a` in the
# table is classed as `POSIXct`
agent <-
  create_agent(tbl = tbl) %>%
  col_is_posix(vars(a)) %>%
  interrogate()

# Determine if this column
# validation has passed by
# using `all_passed()`
all_passed(agent)

---

col_vals_between

Are numerical column data between two specified values?

Description

The `col_vals_between()` validation step function checks whether column values (in any number
of specified columns) fall within a range. The range specified with three arguments: left, right, and inclusive. The left and right values specify the lower and upper numeric bounds. The inclusive argument, as a vector of two logical values relating to left and right, states whether each bound is inclusive or not. The default is `c(TRUE, TRUE)`, where both endpoints are inclusive (i.e., [left, right]). For partially-unbounded versions of this function, we can use the `col_vals_lt()`, `col_vals_lte()`, `col_vals_gt()`, or `col_vals_gte()` validation step functions. This function can be used directly on a data table or with an `agent` object (technically, a `ptblank_agent` object). Each validation step will operate over the number of test units that is equal to the number of rows in the table (after any preconditions have been applied).

Usage

```
col_vals_between(
  x,
  columns,
  left,
  right,
  inclusive = c(TRUE, TRUE),
  na_pass = FALSE,
  preconditions = NULL,
  actions = NULL,
  brief = NULL
)
```
Arguments

x
A data frame, tibble, or an agent object of class ptblank_agent.

columns
The column (or a set of columns, provided as a character vector) to which this validation should be applied.

left
The lower bound for the range. The validation includes this bound value in addition to values greater than left.

right
The upper bound for the range. The validation includes this bound value in addition to values lower than right.

inclusive
A two-element logical value that indicates whether the left and right bounds should be inclusive. By default, both bounds are inclusive.

na_pass
Should any encountered NA values be allowed to pass a validation unit? This is by default FALSE. Set to TRUE to give NAs a pass.

preconditions
expressions used for mutating the input table before proceeding with the validation. This is ideally as a one-sided R formula using a leading ~. In the formula representation, the tbl serves as the input data table to be transformed (e.g., ~ tbl %>% dplyr::mutate(col = col + 10). A series of expressions can be used by enclosing the set of statements with { } but note that the tbl object must be ultimately returned.

actions
A list containing threshold levels so that the validation step can react accordingly when exceeding the set levels. This is to be created with the action_levels() helper function.

brief
An optional, text-based description for the validation step.

Details

If providing multiple column names, the result will be an expansion of validation steps to that number of column names (e.g., vars(col_a,col_b) will result in the entry of two validation steps). Aside from column names in quotes and in vars(), tidyselect helper functions are available for specifying columns. They are: starts_with(), ends_with(), contains(), matches(), and everything().

This validation step function supports special handling of NA values. The na_pass argument will determine whether an NA value appearing in a test unit should be counted as a pass or a fail. The default of na_pass = FALSE means that any NAs encountered will accumulate failing test units.

Having table preconditions means pointblank will mutate the table just before interrogation. It’s isolated to the validation steps produced by this validation step function. Using dplyr code is suggested here since the statements can be translated to SQL if necessary. The code is to be supplied as a one-sided R formula (using a leading ~). In the formula representation, the obligatory tbl variable will serve as the input data table to be transformed (e.g., ~ tbl %>% dplyr::mutate(col_a = col_b + 10). A series of expressions can be used by enclosing the set of statements with { } but note that the tbl variable must be ultimately returned.

Often, we will want to specify actions for the validation. This argument, present in every validation step function, takes a specially-crafted list object that is best produced by the action_levels() function. Read that function’s documentation for the lowdown on how to create reactions to above-threshold failure levels in validation. The basic gist is that you’ll want at least a single threshold level
col_vals_between

(specified as either the fraction test units failed, or, an absolute value), often using the warn_at argument. This is especially true when \( x \) is a table object because, otherwise, nothing happens. For the col_vals_*()-type functions, using action_levels(warn_at = 0.25) or action_levels(stop_at = 0.25) are good choices depending on the situation (the first produces a warning when a quarter of the total test units fails, the other stop()s at the same threshold level).

Want to describe this validation step in some detail? Keep in mind that this is only useful if \( x \) is an agent. If that’s the case, brief the agent with some text that fits. Don’t worry if you don’t want to do it. The autobrief protocol is kicked in when brief = NULL and a simple brief will then be automatically generated.

Value

Either a ptblank_agent object or a table object, depending on what was passed to \( x \).

Function ID

2-7

See Also

The analogue to this function: col_vals_not_between().

Other Validation Step Functions: col_exists(), col_is_character(), col_is_date(), col_is_factor(), col_is_integer(), col_is_logical(), col_is_numeric(), col_is POSIX(), col_vals_equal(), col_vals_gte(), col_vals_gt(), col_vals_in_set(), col_vals_lte(), col_vals_lt(), col_vals_not_between(), col_vals_not_equal(), col_vals_not_in_set(), col_vals_not_null(), col_vals_null(), col_vals_regex(), conjointly(), rows_distinct()

Examples

library(dplyr)

# Create a simple table with
# a column of numerical values
tbl <- tibble(a = c(5.6, 8.2, 7.8))

# Validate that values in
# column ‘a’ are all between
# 1 and 9
agent <-
  create_agent(tbl = tbl) %>%
  col_vals_between(vars(a), 1, 9) %>%
  interrogate()

# Determine if this column
# validation has passed by using
# ‘all_passed()’
all_passed(agent)
col_vals_equal

Are numerical column data equal to a specific value?

Description

The `col_vals_equal()` validation step function checks whether column values (in any number of specified columns) are equal to a specified value. This function can be used directly on a data table or with an `agent` object (technically, a `ptblank_agent` object). Each validation step will operate over the number of test units that is equal to the number of rows in the table (after any preconditions have been applied).

Usage

```r
col_vals_equal(
  x, 
  columns, 
  value, 
  na_pass = FALSE, 
  preconditions = NULL, 
  actions = NULL, 
  brief = NULL
)
```

Arguments

- **x**: A data frame, tibble, or an agent object of class `ptblank_agent`.
- **columns**: The column (or a set of columns, provided as a character vector) to which this validation should be applied.
- **value**: A numeric value used to test for equality.
- **na_pass**: Should any encountered NA values be allowed to pass a validation unit? This is by default `FALSE`. Set to `TRUE` to give NAs a pass.
- **preconditions**: Expressions used for mutating the input table before proceeding with the validation. This is ideally as a one-sided R formula using a leading `~`. In the formula representation, the `tbl` serves as the input data table to be transformed (e.g., `~ tbl %>% dplyr::mutate(col = col + 10)`). A series of expressions can be used by enclosing the set of statements with `{ }` but note that the `tbl` object must be ultimately returned.
- **actions**: A list containing threshold levels so that the validation step can react accordingly when exceeding the set levels. This is to be created with the `action_levels()` helper function.
- **brief**: An optional, text-based description for the validation step.
Details

If providing multiple column names, the result will be an expansion of validation steps to that number of column names (e.g., vars(col_a,col_b) will result in the entry of two validation steps). Aside from column names in quotes and in vars(), tidyselect helper functions are available for specifying columns. They are: starts_with(), ends_with(), contains(), matches(), and everything().

This validation step function supports special handling of NA values. The na_pass argument will determine whether an NA value appearing in a test unit should be counted as a pass or a fail. The default of na_pass = FALSE means that any NAs encountered will accumulate failing test units.

Having table preconditions means pointblank will mutate the table just before interrogation. It's isolated to the validation steps produced by this validation step function. Using dplyr code is suggested here since the statements can be translated to SQL if necessary. The code is to be supplied as a one-sided R formula (using a leading ~). In the formula representation, the obligatory tbl variable will serve as the input data table to be transformed (e.g., ~ tbl %>% dplyr::mutate(col_a = col_b + 10). A series of expressions can be used by enclosing the set of statements with { } but note that the tbl variable must be ultimately returned.

Often, we will want to specify actions for the validation. This argument, present in every validation step function, takes a specially-crafted list object that is best produced by the action_levels() function. Read that function’s documentation for the lowdown on how to create reactions to above-threshold failure levels in validation. The basic gist is that you'll want at least a single threshold level (specified as either the fraction test units failed, or, an absolute value), often using the warn_at argument. This is especially true when x is a table object because, otherwise, nothing happens. For the col_vals_*()-type functions, using action_levels(warn_at = 0.25) or action_levels(stop_at = 0.25) are good choices depending on the situation (the first produces a warning when a quarter of the total test units fails, the other stop()s at the same threshold level).

Want to describe this validation step in some detail? Keep in mind that this is only useful if x is an agent. If that's the case, brief the agent with some text that fits. Don’t worry if you don’t want to do it. The autobrief protocol is kicked in when brief = NULL and a simple brief will then be automatically generated.

Value

Either a ptblank_agent object or a table object, depending on what was passed to x.

Function ID

2-3

See Also

The analogue to this function: col_vals_not_equal().

Other Validation Step Functions: col_exists(), col_is_character(), col_is_date(), col_is_factor(), col_is_integer(), col_is_logical(), col_is_numeric(), col_is_posix(), col_vals_between(), col_vals_gte(), col_vals_gt(), col_vals_in_set(), col_vals_lte(), col_vals_lt(), col_vals_not_between(), col_vals_not_equal(), col_vals_not_in_set(), col_vals_not_null(), col_vals_null(), col_vals_regex(), conjointly(), rows_distinct()
Examples

```r
library(dplyr)

tbl <-
tibble(
  a = c(1, 1, 1, 2, 2, 2),
  b = c(5, 5, 5, 3, 6, 3)
)

agent <-
create_agent(tbl = tbl) %>%
col_vals_equal(vars(b), 5,
  preconditions = ~ tbl %>% dplyr::filter(a == 1)
) %>%
interrogate()

all_passed(agent)
```

### col_vals_gt

**Are numerical column data greater than a specific value?**

**Description**

The `col_vals_gt()` validation step function checks whether column values (in any number of specified columns) are greater than a specified value (the exact comparison used in this function is `col_val > value`). This function can be used directly on a data table or with an `agent` object (technically, a `ptblank_agent` object). Each validation step will operate over the number of test units that is equal to the number of rows in the table (after any preconditions have been applied).

**Usage**

```r
col_vals_gt(
  x,
  columns,
  value,
  na_pass = FALSE,
  preconditions = NULL,
  actions = NULL,
  brief = NULL
)
```
Arguments

x  A data frame, tibble, or an agent object of class ptblank_agent.
columns  The column (or a set of columns, provided as a character vector) to which this validation should be applied.
value  A numeric value used for this test. Any column values >value are considered passing.
na_pass  Should any encountered NA values be allowed to pass a validation unit? This is by default FALSE. Set to TRUE to give NAs a pass.
preconditions  expressions used for mutating the input table before proceeding with the validation. This is ideally as a one-sided R formula using a leading ~. In the formula representation, the tbl serves as the input data table to be transformed (e.g., ~ tbl %>% dplyr::mutate(col = col + 10)). A series of expressions can be used by enclosing the set of statements with { } but note that the tbl object must be ultimately returned.
actions  A list containing threshold levels so that the validation step can react accordingly when exceeding the set levels. This is to be created with the action_levels() helper function.
brief  An optional, text-based description for the validation step.

Details

If providing multiple column names, the result will be an expansion of validation steps to that number of column names (e.g., vars(col_a, col_b) will result in the entry of two validation steps). Aside from column names in quotes and in vars(), tidyselect helper functions are available for specifying columns. They are: starts_with(), ends_with(), contains(), matches(), and everything().

This validation step function supports special handling of NA values. The na_pass argument will determine whether an NA value appearing in a test unit should be counted as a pass or a fail. The default of na_pass = FALSE means that any NAs encountered will accumulate failing test units.

Having table preconditions means pointblank will mutate the table just before interrogation. It’s isolated to the validation steps produced by this validation step function. Using dplyr code is suggested here since the statements can be translated to SQL if necessary. The code is to be supplied as a one-sided R formula (using a leading ~). In the formula representation, the obligatory tbl variable will serve as the input data table to be transformed (e.g., ~ tbl %>% dplyr::mutate(col_a = col_b + 10)). A series of expressions can be used by enclosing the set of statements with { } but note that the tbl variable must be ultimately returned.

Often, we will want to specify actions for the validation. This argument, present in every validation step function, takes a specially-crafted list object that is best produced by the action_levels() function. Read that function’s documentation for the lowdown on how to create reactions to above-threshold failure levels in validation. The basic gist is that you’ll want at least a single threshold level (specified as either the fraction test units failed, or, an absolute value), often using the warn_at argument. This is especially true when x is a table object because, otherwise, nothing happens. For the col_vals_*()-type functions, using action_levels(warn_at = 0.25) or action_levels(stop_at = 0.25) are good choices depending on the situation (the first produces a warning when a quarter of the total test units fails, the other stop()s at the same threshold level).
Want to describe this validation step in some detail? Keep in mind that this is only useful if x is an agent. If that’s the case, brief the agent with some text that fits. Don’t worry if you don’t want to do it. The autobrief protocol is kicked in when brief = NULL and a simple brief will then be automatically generated.

Value

Either a ptblank_agent object or a table object, depending on what was passed to x.

Function ID

2-6

See Also

The analogous function with a left-closed bound: col_vals_gte().
Other Validation Step Functions: col_exists(), col_is_character(), col_is_date(), col_is_factor(), col_is_integer(), col_is_logical(), col_is_numeric(), col_is_posix(), col_vals_between(), col_vals_equal(), col_vals_gte(), col_vals_in_set(), col_vals_lte(), col_vals_lt(), col_vals_not_between(), col_vals_not_equal(), col_vals_not_in_set(), col_vals_not_null(), col_vals_null(), col_vals_regex(), conjointly(), rows_distinct()

Examples

library(dplyr)

# Create a simple table with a
# column of numerical values
tbl <- tibble(a = c(5, 7, 8, 5))

# Validate that values in column
# 'a' are always greater than 4
agent <-
  create_agent(tbl = tbl) %>%
  col_vals_gt(vars(a), 4) %>%
  interrogate()

# Determine if these column
# validations have all passed
# by using `all_passed()`
all_passed(agent)
Description

The `col_vals_gte()` validation step function checks whether column values (in any number of specified columns) are greater than or equal to a specified value (the exact comparison used in this function is `col_val >= value`). This function can be used directly on a data table or with an `agent` object (technically, a `ptblank_agent` object). Each validation step will operate over the number of test units that is equal to the number of rows in the table (after any preconditions have been applied).

Usage

```r
col_vals_gte(
  x,
  columns,
  value,
  na_pass = FALSE,
  preconditions = NULL,
  actions = NULL,
  brief = NULL
)
```

Arguments

- **x**: A data frame, tibble, or an agent object of class `ptblank_agent`.
- **columns**: The column (or a set of columns, provided as a character vector) to which this validation should be applied.
- **value**: A numeric value used for this test. Any column values >= value are considered passing.
- **na_pass**: Should any encountered NA values be allowed to pass a validation unit? This is by default `FALSE`. Set to `TRUE` to give NAs a pass.
- **preconditions**: expressions used for mutating the input table before proceeding with the validation. This is ideally as a one-sided R formula using a leading ~. In the formula representation, the tbl serves as the input data table to be transformed (e.g., `~ tbl %>% dplyr::mutate(col = col + 10)`). A series of expressions can be used by enclosing the set of statements with `{ }` but note that the tbl object must be ultimately returned.
- **actions**: A list containing threshold levels so that the validation step can react accordingly when exceeding the set levels. This is to be created with the `action_levels()` helper function.
- **brief**: An optional, text-based description for the validation step.

Details

If providing multiple column names, the result will be an expansion of validation steps to that number of column names (e.g., `vars(col_a, col_b)` will result in the entry of two validation steps). Aside from column names in quotes and in `vars()`, `tidyselect` helper functions are available for specifying columns. They are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, and `everything()`.
This validation step function supports special handling of NA values. The \texttt{na\_pass} argument will determine whether an NA value appearing in a test unit should be counted as a \textit{pass} or a \textit{fail}. The default of \texttt{na\_pass = FALSE} means that any NAs encountered will accumulate failing test units.

Having table preconditions means \texttt{pointblank} will mutate the table just before interrogation. It’s isolated to the validation steps produced by this validation step function. Using \texttt{dplyr} code is suggested here since the statements can be translated to SQL if necessary. The code is to be supplied as a one-sided R formula (using a leading \texttt{~}). In the formula representation, the obligatory \texttt{tbl} variable will serve as the input data table to be transformed (e.g., \texttt{~ tbl %>% dplyr::mutate(col\_a = col\_b + 10)}). A series of expressions can be used by enclosing the set of statements with \{ \} but note that the \texttt{tbl} variable must be ultimately returned.

Often, we will want to specify actions for the validation. This argument, present in every validation step function, takes a specially-crafted list object that is best produced by the \texttt{action\_levels()} function. Read that function’s documentation for the lowdown on how to create reactions to above-threshold failure levels in validation. The basic gist is that you’ll want at least a single threshold level (specified as either the fraction test units failed, or, an absolute value), often using the \texttt{warn\_at} argument. This is especially true when \texttt{x} is a table object because, otherwise, nothing happens. For the \texttt{col\_vals\_*()-type functions}, using \texttt{action\_levels(warn\_at = 0.25)} or \texttt{action\_levels(stop\_at = 0.25)} are good choices depending on the situation (the first produces a warning when a quarter of the total test units fails, the other \texttt{stop()}s at the same threshold level).

Want to describe this validation step in some detail? Keep in mind that this is only useful if \texttt{x} is an \textit{agent}. If that’s the case, \texttt{brief} the agent with some text that fits. Don’t worry if you don’t want to do it. The \texttt{autobrief} protocol is kicked in when \texttt{brief = NULL} and a simple brief will then be automatically generated.

**Value**

Either a \texttt{ptblank\_agent} object or a table object, depending on what was passed to \texttt{x}.

**Function ID**

2-5

**See Also**

The analogous function with a left-open bound: \texttt{col\_vals\_gt()}.

Other Validation Step Functions: \texttt{col\_exists()}, \texttt{col\_is\_character()}, \texttt{col\_is\_date()}, \texttt{col\_is\_factor()}, \texttt{col\_is\_integer()}, \texttt{col\_is\_logical()}, \texttt{col\_is\_numeric()}, \texttt{col\_is\_posix()}, \texttt{col\_vals\_between()}, \texttt{col\_vals\_equal()}, \texttt{col\_vals\_gt()}, \texttt{col\_vals\_in\_set()}, \texttt{col\_vals\_lte()}, \texttt{col\_vals\_lt()}, \texttt{col\_vals\_not\_between()}, \texttt{col\_vals\_not\_equal()}, \texttt{col\_vals\_not\_in\_set()}, \texttt{col\_vals\_not\_null()}, \texttt{col\_vals\_null()}, \texttt{col\_vals\_regex()}, \texttt{conjointly()}, \texttt{rows\_distinct()}

**Examples**

```r
library(dplyr)

# Create a simple table with a
# column of numerical values
tbl <- tibble(a = c(5, 7, 8, 5))
```
col_vals_in_set

Are column data part of a specific set of values?

Description

The `col_vals_in_set()` validation step function checks whether column values (in any number of specified columns) are part of a specified set of values. This function can be used directly on a data table or with an `agent` object (technically, a `ptblank_agent` object). Each validation step will operate over the number of test units that is equal to the number of rows in the table (after any preconditions have been applied).

Usage

```r
col_vals_in_set(
  x,
  columns,
  set,
  preconditions = NULL,
  actions = NULL,
  brief = NULL
)
```

Arguments

- `x`: A data frame, tibble, or an agent object of class `ptblank_agent`.
- `columns`: The column (or a set of columns, provided as a character vector) to which this validation should be applied.
- `set`: A vector of numeric or string-based elements, where column values found within this set will be considered as passing.
- `preconditions`: expressions used for mutating the input table before proceeding with the validation. This is ideally as a one-sided R formula using a leading `~`. In the formula representation, the `tbl` serves as the input data table to be transformed (e.g., `~ tbl %>% dplyr::mutate(col = col + 10)`). A series of expressions can be used by enclosing the set of statements with `{ }` but note that the `tbl` object must be ultimately returned.
actions A list containing threshold levels so that the validation step can react accordingly when exceeding the set levels. This is to be created with the `action_levels()` helper function.

brief An optional, text-based description for the validation step.

Details

If providing multiple column names, the result will be an expansion of validation steps to that number of column names (e.g., `vars(col_a,col_b)` will result in the entry of two validation steps). Aside from column names in quotes and in `vars()`, `tidyselect` helper functions are available for specifying columns. They are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, and `everything()`.

Having table preconditions means `pointblank` will mutate the table just before interrogation. It’s isolated to the validation steps produced by this validation step function. Using `dplyr` code is suggested here since the statements can be translated to SQL if necessary. The code is to be supplied as a one-sided R formula (using a leading ~). In the formula representation, the obligatory `tbl` variable will serve as the input data table to be transformed (e.g., ~ `tbl` %>% `mutate(col_a = col_b + 10)`). A series of expressions can be used by enclosing the set of statements with `{ }` but note that the `tbl` variable must be ultimately returned.

Often, we will want to specify actions for the validation. This argument, present in every validation step function, takes a specially-crafted list object that is best produced by the `action_levels()` function. Read that function’s documentation for the lowdown on how to create reactions to above-threshold failure levels in validation. The basic gist is that you’ll want at least a single threshold level (specified as either the fraction test units failed, or, an absolute value), often using the `warn_at` argument. This is especially true when `x` is a table object because, otherwise, nothing happens. For the `col_vals_*()`-type functions, using `action_levels(warn_at = 0.25)` or `action_levels(stop_at = 0.25)` are good choices depending on the situation (the first produces a warning when a quarter of the total test units fails, the other stops at the same threshold level).

Want to describe this validation step in some detail? Keep in mind that this is only useful if `x` is an agent. If that’s the case, `brief` the agent with some text that fits. Don’t worry if you don’t want to do it. The `autobrief` protocol is kicked in when `brief = NULL` and a simple brief will then be automatically generated.

Value

Either a `ptblank_agent` object or a table object, depending on what was passed to `x`.

Function ID

2-9

See Also

The analogue to this function: `col_vals_not_in_set()`.

Other Validation Step Functions: `col_exists()`, `col_is_character()`, `col_is_date()`, `col_is_factor()`, `col_is_integer()`, `col_is_logical()`, `col_is_numeric()`, `col_is_posix()`, `col_vals_between()`, `col_vals_equal()`, `col_vals_gte()`, `col_vals_gt()`, `col_vals_lte()`, `col_vals_lt()`, `col_vals_not_between()`.
Examples

library(dplyr)

# Create a simple table with 2
# columns: one with numerical
# values, the other with strings
tbl <-
tibble(
    a = c(1, 2, 3, 4),
    b = rep(c("one", "two"), 2)
)

# Validate that all numerical values
# in column `a` belong to a numerical
# set, and, create an analogous
# validation check for column `b` with
# a set of string values
agent <-
    create_agent(tbl = tbl) %>%
    col_vals_in_set(vars(a), 1:4) %>%
    col_vals_in_set(vars(b), c("one", "two")) %>%
    interrogate()

# Determine if these column
# validations have all passed
# by using `all_passed()`
all_passed(agent)

---

**col_vals_lt**

Are numerical column data less than a specific value?

Description

The `col_vals_lt()` validation step function checks whether column values (in any number of specified columns) are less than a specified value (the exact comparison used in this function is `col_val < value`). This function can be used directly on a data table or with an agent object (technically, a ptblank_agent object). Each validation step will operate over the number of test units that is equal to the number of rows in the table (after any preconditions have been applied).

Usage

```r
col_vals_lt(
    x,
    columns,
)```
value,
na_pass = FALSE,
preconditions = NULL,
actions = NULL,
brief = NULL
)

Arguments

x A data frame, tibble, or an agent object of class ptblank_agent.
columns The column (or a set of columns, provided as a character vector) to which this validation should be applied.
value A numeric value used for this test. Any column values < value are considered passing.
na_pass Should any encountered NA values be allowed to pass a validation unit? This is by default FALSE. Set to TRUE to give NAs a pass.
preconditions expressions used for mutating the input table before proceeding with the validation. This is ideally as a one-sided R formula using a leading ~. In the formula representation, the tbl serves as the input data table to be transformed (e.g., ~ tbl %>% dplyr::mutate(col = col + 10)). A series of expressions can be used by enclosing the set of statements with { } but note that the tbl object must be ultimately returned.
actions A list containing threshold levels so that the validation step can react accordingly when exceeding the set levels. This is to be created with the action_levels() helper function.
brief An optional, text-based description for the validation step.

Details

If providing multiple column names, the result will be an expansion of validation steps to that number of column names (e.g., vars(col_a, col_b) will result in the entry of two validation steps). Aside from column names in quotes and in vars(), tidyselect helper functions are available for specifying columns. They are: starts_with(), ends_with(), contains(), matches(), and everything().

This validation step function supports special handling of NA values. The na_pass argument will determine whether an NA value appearing in a test unit should be counted as a pass or a fail. The default of na_pass = FALSE means that any NAs encountered will accumulate failing test units.

Having table preconditions means pointblank will mutate the table just before interrogation. It’s isolated to the validation steps produced by this validation step function. Using dplyr code is suggested here since the statements can be translated to SQL if necessary. The code is to be supplied as a one-sided R formula (using a leading ~). In the formula representation, the obligatory tbl variable will serve as the input data table to be transformed (e.g., ~ tbl %>% dplyr::mutate(col_a = col_b + 10)). A series of expressions can be used by enclosing the set of statements with { } but note that the tbl variable must be ultimately returned.

Often, we will want to specify actions for the validation. This argument, present in every validation step function, takes a specially-crafted list object that is best produced by the action_levels()
function. Read that function’s documentation for the lowdown on how to create reactions to above-threshold failure levels in validation. The basic gist is that you’ll want at least a single threshold level (specified as either the fraction test units failed, or, an absolute value), often using the warn_at argument. This is especially true when x is a table object because, otherwise, nothing happens. For the col_vals_*()-type functions, using action_levels(warn_at = 0.25) or action_levels(stop_at = 0.25) are good choices depending on the situation (the first produces a warning when a quarter of the total test units fails, the other stop()s at the same threshold level).

Want to describe this validation step in some detail? Keep in mind that this is only useful if x is an agent. If that’s the case, brief the agent with some text that fits. Don’t worry if you don’t want to do it. The autobrief protocol is kicked in when brief = NULL and a simple brief will then be automatically generated.

Value

Either a ptblank_agent object or a table object, depending on what was passed to x.

Function ID

2-1

See Also

The analogous function with a right-closed bound: col_vals_lte().

Other Validation Step Functions: col_exists(), col_is_character(), col_is_date(), col_is_factor(), col_is_integer(), col_is_logical(), col_is_numeric(), col_is_posix(), col_vals_between(), col_vals_equal(), col_vals_gte(), col_vals_gt(), col_vals_in_set(), col_vals_lte(), col_vals_not_between(), col_vals_not_equal(), col_vals_not_in_set(), col_vals_not_null(), col_vals_null(), col_vals_regex(), conjointly(), rows_distinct

Examples

library(dplyr)

# Create a simple table with a
# column of numerical values
tbl <- tibble(a = c(5, 4, 1, 2))

# Validate that values in
# column 'a' are always less
# than 6
agent <-
  create_agent(tbl = tbl) %>%
  col_vals_lt(vars(a), 6) %>%
  interrogate()

# Determine if this column
# validation has passed by using
# 'all_passed()'
all_passed(agent)
Are numerical column data less than or equal to a specific value?

Description

The `col_vals_lte()` validation step function checks whether column values (in any number of specified columns) are less than or equal to a specified value (the exact comparison used in this function is `col_val <= value`). This function can be used directly on a data table or with an agent object (technically, a ptblank_agent object). Each validation step will operate over the number of test units that is equal to the number of rows in the table (after any preconditions have been applied).

Usage

```r
col_vals_lte(
  x,
  columns,
  value,
  na_pass = FALSE,
  preconditions = NULL,
  actions = NULL,
  brief = NULL
)
```

Arguments

- `x`: A data frame, tibble, or an agent object of class ptblank_agent.
- `columns`: The column (or a set of columns, provided as a character vector) to which this validation should be applied.
- `value`: A numeric value used for this test. Any column values <= value are considered passing.
- `na_pass`: Should any encountered NA values be allowed to pass a validation unit? This is by default FALSE. Set to TRUE to give NAs a pass.
- `preconditions`: expressions used for mutating the input table before proceeding with the validation. This is ideally as a one-sided R formula using a leading ~. In the formula representation, the tbl serves as the input data table to be transformed (e.g., `~ tbl %>% dplyr::mutate(col = col + 10`). A series of expressions can be used by enclosing the set of statements with `{ }` but note that the tbl object must be ultimately returned.
- `actions`: A list containing threshold levels so that the validation step can react accordingly when exceeding the set levels. This is to be created with the `action_levels()` helper function.
- `brief`: An optional, text-based description for the validation step.
Details

If providing multiple column names, the result will be an expansion of validation steps to that number of column names (e.g., `vars(col_a, col_b)` will result in the entry of two validation steps). Aside from column names in quotes and in `vars()`, `tidyselect` helper functions are available for specifying columns. They are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, and `everything()`.

This validation step function supports special handling of `NA` values. The `na_pass` argument will determine whether an NA value appearing in a test unit should be counted as a pass or a fail. The default of `na_pass = FALSE` means that any NAs encountered will accumulate failing test units.

Having table preconditions means `pointblank` will mutate the table just before interrogation. It’s isolated to the validation steps produced by this validation step function. Using `dplyr` code is suggested here since the statements can be translated to SQL if necessary. The code is to be supplied as a one-sided R formula (using a leading `~`). In the formula representation, the obligatory `tbl` variable will serve as the input data table to be transformed (e.g., `~ tbl %>% dplyr::mutate(col_a = col_b + 10)`). A series of expressions can be used by enclosing the set of statements with `{ }` but note that the `tbl` variable must be ultimately returned.

Often, we will want to specify actions for the validation. This argument, present in every validation step function, takes a specially-crafted list object that is best produced by the `action_levels()` function. Read that function’s documentation for the lowdown on how to create reactions to above-threshold failure levels in validation. The basic gist is that you’ll want at least a single threshold level (specified as either the fraction test units failed, or, an absolute value), often using the `warn_at` argument. This is especially true when `x` is a table object because, otherwise, nothing happens. For the `col_vals_*()`-type functions, using `action_levels(warn_at = 0.25)` or `action_levels(stop_at = 0.25)` are good choices depending on the situation (the first produces a warning when a quarter of the total test units fails, the other stops at the same threshold level).

Want to describe this validation step in some detail? Keep in mind that this is only useful if `x` is an agent. If that’s the case, `brief` the agent with some text that fits. Don’t worry if you don’t want to do it. The `autobrief` protocol is kicked in when `brief = NULL` and a simple brief will then be automatically generated.

Value

Either a `ptblank_agent` object or a table object, depending on what was passed to `x`.

Function ID

2-2

See Also

The analogous function with a right-open bound: `col_vals_lt()`.

Other Validation Step Functions: `col_exists()`, `col_is_character()`, `col_is_date()`, `col_is_factor()`, `col_is_integer()`, `col_is_logical()`, `col_is_numeric()`, `col_is posix()`, `col_vals_between()`, `col_vals_equal()`, `col_vals_gte()`, `col_vals_gt()`, `col_vals_in_set()`, `col_vals_lt()`, `col_vals_not_between()`, `col_vals_not_equal()`, `col_vals_not_in_set()`, `col_vals_not_null()`, `col_vals_null()`, `col_vals_regex()`, `conjointly()`, `rows_distinct()`
**Examples**

```r
library(dplyr)

# Create a simple table with a
# column of numerical values
.tbl <-
  tibble(
    a = c(5, 4, 1, 2),
    b = c(3, 2, 5, 6)
  )

# Validate that the sum of
# values across columns `a`
# and `b` are always less
# than or equal to 10
agent <-
  create_agent(tbl = tbl) %>%
  col_vals_lte(vars(a_b), 10,
    preconditions = ~ {
      tbl %>% dplyr::mutate(a_b = a + b)
    })
  %>%
  interrogate()

# Determine if this column
# validation has passed by using
# `all_passed`
all_passed(agent)
```

---

**col_vals_not_between**  
*Are numerical column data not between two specified values?*

**Description**

Verification step where column data should not be between two values.

**Usage**

```r
col_vals_not_between(
  x,
  columns,
  left,
  right,
  inclusive = c(TRUE, TRUE),
  na_pass = FALSE,
  preconditions = NULL,
  actions = NULL,
  brief = NULL
)
```
**Arguments**

- `x` A data frame, tibble, or an agent object of class `ptblank_agent`.
- `columns` The column (or a set of columns, provided as a character vector) to which this validation should be applied.
- `left`, `right` The lower and upper bounds for the range. The validation Any values $\geq$ left and $\leq$ right will be considered as failing.
- `inclusive` A two-element logical value that indicates whether the left and right bounds should be inclusive. By default, both bounds are inclusive.
- `na_pass` Should any encountered NA values be allowed to pass a validation unit? This is by default FALSE. Set to TRUE to give NAs a pass.
- `preconditions` expressions used for mutating the input table before proceeding with the validation. This is ideally as a one-sided R formula using a leading ~. In the formula representation, the tbl serves as the input data table to be transformed (e.g., ~ tbl %>% dplyr::mutate(col = col + 10). A series of expressions can be used by enclosing the set of statements with { } but note that the tbl object must be ultimately returned.
- `actions` A list containing threshold levels so that the validation step can react accordingly when exceeding the set levels. This is to be created with the `action_levels()` helper function.
- `brief` An optional, text-based description for the validation step.

**Details**

The `col_vals_not_between()` validation step function checks whether column values (in any number of specified columns) do not fall within a range. The range specified with three arguments: `left`, `right`, and `inclusive`. The `left` and `right` values specify the lower and upper numeric bounds. The `inclusive` argument, as a vector of two logical values relating to `left` and `right`, states whether each bound is inclusive or not. The default is `c(TRUE, TRUE)`, where both endpoints are inclusive (i.e., `[left, right]`). For partially-unbounded versions of this function, we can use the `col_vals_lt()`, `col_vals_lte()`, `col_vals_gt()`, or `col_vals_gte()` validation step functions. This function can be used directly on a data table or with an `agent` object (technically, a `ptblank_agent` object). Each validation step will operate over the number of test units that is equal to the number of rows in the table (after any `preconditions` have been applied).

If providing multiple column names, the result will be an expansion of validation steps to that number of column names (e.g., `vars(col_a, col_b)` will result in the entry of two validation steps). Aside from column names in quotes and in `vars()`, `tidyselect` helper functions are available for specifying columns. They are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, and `everything()`.

This validation step function supports special handling of NA values. The `na_pass` argument will determine whether an NA value appearing in a test unit should be counted as a pass or a fail. The default of `na_pass = FALSE` means that any NAs encountered will accumulate failing test units.

Having table preconditions means `pointblank` will mutate the table just before interrogation. It’s isolated to the validation steps produced by this validation step function. Using `dplyr` code is suggested here since the statements can be translated to SQL if necessary. The code is to be supplied as a one-sided R formula (using a leading ~). In the formula representation, the obligatory tbl
variable will serve as the input data table to be transformed (e.g., ~ tbl %>% dplyr::mutate(col_a = col_b + 10). A series of expressions can be used by enclosing the set of statements with { } but note that the tbl variable must be ultimately returned.

Often, we will want to specify actions for the validation. This argument, present in every validation step function, takes a specially-crafted list object that is best produced by the action_levels() function. Read that function’s documentation for the lowdown on how to create reactions to above-threshold failure levels in validation. The basic gist is that you’ll want at least a single threshold level (specified as either the fraction test units failed, or, an absolute value), often using the warn_at argument. This is especially true when x is a table object because, otherwise, nothing happens. For the col_vals_*()-type functions, using action_levels(warn_at = 0.25) or action_levels(stop_at = 0.25) are good choices depending on the situation (the first produces a warning when a quarter of the total test units fails, the other stop()s at the same threshold level).

Want to describe this validation step in some detail? Keep in mind that this is only useful if x is an agent. If that’s the case, brief the agent with some text that fits. Don’t worry if you don’t want to do it. The autobrief protocol is kicked in when brief = NULL and a simple brief will then be automatically generated.

Value

Either a ptblank_agent object or a table object, depending on what was passed to x.

Function ID

2-8

See Also

The analogue to this function: col_vals_between().

Other Validation Step Functions: col_exists(), col_is_character(), col_is_date(), col_is_factor(),
col_is_integer(), col_is_logical(), col_is_numeric(), col_is POSIX(), col_vals_between(),
col_vals_equal(), col_vals_gte(), col_vals_gt(), col_vals_in_set(), col_vals_lte(),
col_vals_lt(), col_vals_not_equal(), col_vals_not_in_set(), col_vals_not_null(), col_vals_null(),
col_vals_regex(), jointly(), rows_distinct()

Examples

library(dplyr)

# Create a simple table with a
# column of numerical values
tbl <- tibble(a = c(5.6, 7.8, 3.4))

# Validate that none of the values
# in column `a` are between 9 and 10,
# or, between 0 and 2
agent <-
create_agent(tbl = tbl) %>%
col_vals_not_between(vars(a), 9, 10) %>%
col_vals_not_between(vars(a), 0, 2) %>%
interrogate()
# Determine if these column validations have all passed by
# using `all_passed()`
all_passed(agent)

## col_vals_not_equal

Are numerical column data not equal to a specific value?

### Description

The `col_vals_not_equal()` validation step function checks whether column values (in any number of specified columns) are not equal to a specified value. This function can be used directly on a data table or with an `agent` object (technically, a `ptblank_agent` object). Each validation step will operate over the number of test units that is equal to the number of rows in the table (after any preconditions have been applied).

### Usage

```r
col_vals_not_equal(
  x,  # A data frame, tibble, or an agent object of class ptblank_agent.
  columns,  # The column (or a set of columns, provided as a character vector) to which this validation should be applied.
  value,  # a numeric value used to test for non-equality.
  na_pass = FALSE,  # Should any encountered NA values be allowed to pass a validation unit? This is by default FALSE. Set to TRUE to give NAs a pass.
  preconditions = NULL,  # expressions used for mutating the input table before proceeding with the validation. This is ideally as a one-sided R formula using a leading ~. In the formula representation, the `tbl` serves as the input data table to be transformed (e.g., `~ tbl %>% dplyr::mutate(col = col + 10)`). A series of expressions can be used by enclosing the set of statements with `{ }` but note that the `tbl` object must be ultimately returned.
  actions = NULL,  # A list containing threshold levels so that the validation step can react accordingly when exceeding the set levels. This is to be created with the `action_levels()` helper function.
  brief = NULL)  # An optional, text-based description for the validation step.
```
Details

If providing multiple column names, the result will be an expansion of validation steps to that number of column names (e.g., vars(col_a, col_b) will result in the entry of two validation steps). Aside from column names in quotes and in `vars()`, `tidyselect` helper functions are available for specifying columns. They are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, and `everything()`.

This validation step function supports special handling of NA values. The `na_pass` argument will determine whether an NA value appearing in a test unit should be counted as a pass or a fail. The default of `na_pass = FALSE` means that any NAs encountered will accumulate failing test units.

Having table preconditions means `pointblank` will mutate the table just before interrogation. It’s isolated to the validation steps produced by this validation step function. Using `dplyr` code is suggested here since the statements can be translated to SQL if necessary. The code is to be supplied as a one-sided `R` formula (using a leading `~`). In the formula representation, the obligatory `tbl` variable will serve as the input data table to be transformed (e.g., `~ tbl %>% dplyr::mutate(col_a = col_b + 10)`). A series of expressions can be used by enclosing the set of statements with `{ }` but note that the `tbl` variable must be ultimately returned.

Often, we will want to specify actions for the validation. This argument, present in every validation step function, takes a specially-crafted list object that is best produced by the `action_levels()` function. Read that function’s documentation for the lowdown on how to create reactions to above-threshold failure levels in validation. The basic gist is that you’ll want at least a single threshold level (specified as either the fraction test units failed, or, an absolute value), often using the `warn_at` argument. This is especially true when `x` is a table object because, otherwise, nothing happens. For the `col_vals_*()-type functions, using `action_levels(warn_at = 0.25)` or `action_levels(stop_at = 0.25)` are good choices depending on the situation (the first produces a warning when a quarter of the total test units fails, the other `stop()`s at the same threshold level).

Want to describe this validation step in some detail? Keep in mind that this is only useful if `x` is an `agent`. If that’s the case, `brief` the agent with some text that fits. Don’t worry if you don’t want to do it. The `autobrief` protocol is kicked in when `brief = NULL` and a simple brief will then be automatically generated.

Value

Either a `ptblank_agent` object or a table object, depending on what was passed to `x`.

Function ID

2-4

See Also

The analogue to this function: `col_vals_equal()`.

Other Validation Step Functions: `col_exists()`, `col_is_character()`, `col_is_date()`, `col_is_factor()`, `col_is_integer()`, `col_is_logical()`, `col_is_numeric()`, `col_is_posix()`, `col_vals_between()`, `col_vals_equal()`, `col_vals_gte()`, `col_vals_gt()`, `col_vals_in_set()`, `col_vals_lte()`, `col_vals_lt()`, `col_vals_not_between()`, `col_vals_not_in_set()`, `col_vals_not_null()`, `col_vals_null()`, `col_vals_regex()`, `conjointly()`, `rows_distinct()`
Examples

```r
library(dplyr)

# Create a simple table with two columns of numerical values
tbl <-
tibble(
  a = c(1, 1, 1, 2, 2, 2),
  b = c(5, 5, 5, 3, 6, 3)
)

# Validate that values in column 'b' are not equal to 5 when values in column 'a' are equal to 2
agent <-
  create_agent(tbl = tbl) %>%
  col_vals_not_equal(vars(b), 5,
    preconditions = ~ tbl %>% dplyr::filter(a == 2)
  ) %>%
  interrogate()

# Determine if this column validation has passed by using `all_passed`
# `all_passed(agent)
```

---

**Description**

The `col_vals_not_in_set()` validation step function checks whether column values (in any number of specified columns) are not part of a specified set of values. This function can be used directly on a data table or with an `agent` object (technically, a `ptblank_agent` object). Each validation step will operate over the number of test units that is equal to the number of rows in the table (after any preconditions have been applied).

**Usage**

```r
col_vals_not_in_set(
  x,
  columns,
  set,
  preconditions = NULL,
  actions = NULL,
  brief = NULL
)
```
Arguments

**x**  
A data frame, tibble, or an agent object of class `ptblank_agent`.

**columns**  
The column (or a set of columns, provided as a character vector) to which this validation should be applied.

**set**  
A vector of numeric or string-based elements, where column values found within this set will be considered as failing.

**preconditions**  
Expressions used for mutating the input table before proceeding with the validation. This is ideally as a one-sided R formula using a leading `~`. In the formula representation, the `tbl` serves as the input data table to be transformed (e.g., `~ tbl %>% dplyr::mutate(col = col + 10)`). A series of expressions can be used by enclosing the set of statements with `{ }` but note that the `tbl` object must be ultimately returned.

**actions**  
A list containing threshold levels so that the validation step can react accordingly when exceeding the set levels. This is to be created with the `action_levels()` helper function.

**brief**  
An optional, text-based description for the validation step.

Details

If providing multiple column names, the result will be an expansion of validation steps to that number of column names (e.g., `vars(col_a,col_b)` will result in the entry of two validation steps). Aside from column names in quotes and in `vars()`, `tidyselect` helper functions are available for specifying columns. They are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, and `everything()`.

Having table preconditions means `pointblank` will mutate the table just before interrogation. It’s isolated to the validation steps produced by this validation step function. Using `dplyr` code is suggested here since the statements can be translated to SQL if necessary. The code is to be supplied as a one-sided R formula (using a leading `~`). In the formula representation, the obligatory `tbl` variable will serve as the input data table to be transformed (e.g., `~ tbl %>% dplyr::mutate(col_a = col_b + 10)`). A series of expressions can be used by enclosing the set of statements with `{ }` but note that the `tbl` variable must be ultimately returned.

Often, we will want to specify actions for the validation. This argument, present in every validation step function, takes a specially-crafted list object that is best produced by the `action_levels()` function. Read that function’s documentation for the lowdown on how to create reactions to above-threshold failure levels in validation. The basic gist is that you’ll want at least a single threshold level (specified as either the fraction test units failed, or, an absolute value), often using the `warn_at` argument. This is especially true when `x` is a table object because, otherwise, nothing happens. For the `col_vals_*()`-type functions, using `action_levels(warn_at = 0.25)` or `action_levels(stop_at = 0.25)` are good choices depending on the situation (the first produces a warning when a quarter of the total test units fails, the other stops at the same threshold level).

Want to describe this validation step in some detail? Keep in mind that this is only useful if `x` is an agent. If that’s the case, `brief` the agent with some text that fits. Don’t worry if you don’t want to do it. The `autobrief` protocol is kicked in when `brief = NULL` and a simple `brief` will then be automatically generated.
**Value**

Either a `ptblank_agent` object or a table object, depending on what was passed to `x`.

**Function ID**

2-10

**See Also**

The analogue to this function: `col_vals_in_set()`.

Other Validation Step Functions: `col_exists()`, `col_is_character()`, `col_is_date()`, `col_is_factor()`, `col_is_integer()`, `col_is_logical()`, `col_is_numeric()`, `col_is_posix()`, `col_vals_between()`, `col_vals_equal()`, `col_vals_gte()`, `col_vals_gt()`, `col_vals_in_set()`, `col_vals_lte()`, `col_vals_lt()`, `col_vals_not_between()`, `col_vals_not_equal()`, `col_vals_not_null()`, `col_vals_null()`, `col_vals_regex()`, `conjointly()`, `rows_distinct()`.

**Examples**

```r
library(dplyr)

# Create a simple table with 2 columns: one with numerical values, the other with strings
tbl <-
  tibble(
    a = c(1, 2, 3, 4),
    b = rep(c("one", "two"), 2)
  )

# Validate that all numerical values in column 'a' do not belong to a specified numerical set, and create an analogous validation check for column 'b' within a set of string values
agent <-
  create_agent(tbl = tbl) %>%
  col_vals_not_in_set(vars(a), 7:10) %>%
  col_vals_not_in_set(vars(b), c("seven", "eight")) %>%
  interrogate()

# Determine if these column validations have all passed
# by using 'all_passed()'
all_passed(agent)
```
Description

The `col_vals_not_null()` validation step function checks whether column values (in any number of specified columns) are not NA values or, in the database context, not NULL values. This function can be used directly on a data table or with an agent object (technically, a `ptblank_agent` object). Each validation step will operate over the number of test units that is equal to the number of rows in the table (after any preconditions have been applied).

Usage

```r
col_vals_not_null(
  x,  
  columns, 
  preconditions = NULL, 
  actions = NULL, 
  brief = NULL
)
```

Arguments

- `x`: A data frame, tibble, or an agent object of class `ptblank_agent`.
- `columns`: The column (or a set of columns, provided as a character vector) to which this validation should be applied.
- `preconditions`: expressions used for mutating the input table before proceeding with the validation. This is ideally as a one-sided R formula using a leading ~. In the formula representation, the tbl serves as the input data table to be transformed (e.g., ~ tbl %>% dplyr::mutate(col = col + 10). A series of expressions can be used by enclosing the set of statements with {} but note that the tbl object must be ultimately returned.
- `actions`: A list containing threshold levels so that the validation step can react accordingly when exceeding the set levels. This is to be created with the `action_levels()` helper function.
- `brief`: An optional, text-based description for the validation step.

Details

If providing multiple column names, the result will be an expansion of validation steps to that number of column names (e.g., `vars(col_a,col_b)` will result in the entry of two validation steps). Aside from column names in quotes and in `vars()`, `tidyselect` helper functions are available for specifying columns. They are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, and `everything()`.

Having table preconditions means `pointblank` will mutate the table just before interrogation. It's isolated to the validation steps produced by this validation step function. Using `dplyr` code is
suggested here since the statements can be translated to SQL if necessary. The code is to be supplied as a one-sided R formula (using a leading ~). In the formula representation, the obligatory tbl variable will serve as the input data table to be transformed (e.g., ~ tbl %>% dplyr::mutate(col_a = col_b + 10). A series of expressions can be used by enclosing the set of statements with { } but note that the tbl variable must be ultimately returned.

Often, we will want to specify actions for the validation. This argument, present in every validation step function, takes a specially-crafted list object that is best produced by the `action_levels()` function. Read that function’s documentation for the lowdown on how to create reactions to above-threshold failure levels in validation. The basic gist is that you’ll want at least a single threshold level (specified as either the fraction test units failed, or, an absolute value), often using the `warn_at` argument. This is especially true when x is a table object because, otherwise, nothing happens. For the `col_vals_*()`-type functions, using `action_levels(warn_at = 0.25)` or `action_levels(stop_at = 0.25)` are good choices depending on the situation (the first produces a warning when a quarter of the total test units fails, the other `stop()`s at the same threshold level).

Want to describe this validation step in some detail? Keep in mind that this is only useful if x is an agent. If that’s the case, brief the agent with some text that fits. Don’t worry if you don’t want to do it. The `autobrief` protocol is kicked in when `brief = NULL` and a simple brief will then be automatically generated.

**Value**

Either a `ptblank_agent` object or a table object, depending on what was passed to x.

**Function ID**

2-12

**See Also**

The analogue to this function: `col_vals_null()`.

Other Validation Step Functions: `col_exists()`, `col_is_character()`, `col_is_date()`, `col_is_factor()`, `col_is_integer()`, `col_is_logical()`, `col_is_numeric()`, `col_is_posix()`, `col_vals_between()`, `col_vals_equal()`, `col_vals_gte()`, `col_vals_gt()`, `col_vals_in_set()`, `col_vals_lte()`, `col_vals_lt()`, `col_vals_not_between()`, `col_vals_not_equal()`, `col_vals_not_in_set()`, `col_vals_null()`, `col_vals_regex()`, `conjointly()`, `rows_distinct()`

**Examples**

```r
col_vals_not_null
library(dplyr)
# Create a simple table with two
# columns of numerical values
tbl <-
tibble(
  a = c(1, 2, NA, NA),
  b = c(2, 2, 5, 5)
)

# Validate that all values in
# col_vals_null

## Description

The col_vals_null() validation step function checks whether column values (in any number of specified columns) are NA values or, in the database context, NULL values. This function can be used directly on a data table or with an agent object (technically, a ptblank_agent object). Each validation step will operate over the number of test units that is equal to the number of rows in the table (after any preconditions have been applied).

## Usage

```
col_vals_null(x, columns, preconditions = NULL, actions = NULL, brief = NULL)
```

## Arguments

- **x**
  A data frame, tibble, or an agent object of class ptblank_agent.

- **columns**
  The column (or a set of columns, provided as a character vector) to which this validation should be applied.

- **preconditions**
  expressions used for mutating the input table before proceeding with the validation. This is ideally as a one-sided R formula using a leading ~. In the formula representation, the tbl serves as the input data table to be transformed (e.g., ~ tbl %>% dplyr::mutate(col = col + 10). A series of expressions can be used by enclosing the set of statements with { } but note that the tbl object must be ultimately returned.

- **actions**
  A list containing threshold levels so that the validation step can react accordingly when exceeding the set levels. This is to be created with the action_levels() helper function.

- **brief**
  An optional, text-based description for the validation step.
Details

If providing multiple column names, the result will be an expansion of validation steps to that number of column names (e.g., `vars(col_a,col_b)` will result in the entry of two validation steps). Aside from column names in quotes and in `vars()`, `tidyselect` helper functions are available for specifying columns. They are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, and `everything()`.

Having table preconditions means `pointblank` will mutate the table just before interrogation. It’s isolated to the validation steps produced by this validation step function. Using `dplyr` code is suggested here since the statements can be translated to SQL if necessary. The code is to be supplied as a one-sided R formula (using a leading ~). In the formula representation, the obligatory tbl variable will serve as the input data table to be transformed (e.g., `~ tbl %>% dplyr::mutate(col_a = col_b + 10)`). A series of expressions can be used by enclosing the set of statements with { } but note that the tbl variable must be ultimately returned.

Often, we will want to specify actions for the validation. This argument, present in every validation step function, takes a specially-crafted list object that is best produced by the `action_levels()` function. Read that function’s documentation for the lowdown on how to create reactions to above-threshold failure levels in validation. The basic gist is that you’ll want at least a single threshold level (specified as either the fraction test units failed, or, an absolute value), often using the `warn_at` argument. This is especially true when x is a table object because, otherwise, nothing happens. For the `col_vals_*()`-type functions, using `action_levels(warn_at = 0.25)` or `action_levels(stop_at = 0.25)` are good choices depending on the situation (the first produces a warning when a quarter of the total test units fails, the other stops at the same threshold level).

Want to describe this validation step in some detail? Keep in mind that this is only useful if x is an agent. If that’s the case, `brief` the agent with some text that fits. Don’t worry if you don’t want to do it. The `autobrief` protocol is kicked in when `brief = NULL` and a simple brief will then be automatically generated.

Value

Either a `ptblank_agent` object or a table object, depending on what was passed to x.

Function ID

2-11

See Also

The analogue to this function: `col_vals_not_null()`.

Other Validation Step Functions: `col_exists()`, `col_is_character()`, `col_is_date()`, `col_is_factor()`, `col_is_integer()`, `col_is_logical()`, `col_is_numeric()`, `col_is_posix()`, `col_vals_between()`, `col_vals_equal()`, `col_vals_gte()`, `col_vals_gt()`, `col_vals_in_set()`, `col_vals_lte()`, `col_vals_lt()`, `col_vals_not_between()`, `col_vals_not_equal()`, `col_vals_not_in_set()`, `col_vals_not_null()`, `col_vals_regex()`, `conjointly()`, `rows_distinct()`

Examples

```r
library(dplyr)
```
# Create a simple table with two columns of numerical values
tbl <-
  tibble(
    a = c(1, 2, NA, NA),
    b = c(2, 2, 5, 5)
  )

# Validate that all values in column `a` are NULL when values in column `b` are equal to 5
agent <-
  create_agent(tbl = tbl) %>%
  col_vals_null(vars(a),
    preconditions = ~ tbl %>% dplyr::filter(b >= 5)
  ) %>%
  interrogate()

# Determine if these column validations have all passed
# by using `all_passed()`
all_passed(agent)

---

**col_vals_regex**

*Do strings in column data match a regex pattern?*

**Description**

The `col_vals_regex()` validation step function checks whether column values (in any number of specified columns) should correspond to a regex matching expression. This function can be used directly on a data table or with an `agent` object (technically, a `ptblank_agent` object). Each validation step will operate over the number of test units that is equal to the number of rows in the table (after any preconditions have been applied).

**Usage**

```r
col_vals_regex(
  x,
  columns,
  regex,
  na_pass = FALSE,
  preconditions = NULL,
  actions = NULL,
  brief = NULL
)
```
Arguments

- **x**: A data frame, tibble, or an agent object of class `ptblank_agent`.
- **columns**: The column (or a set of columns, provided as a character vector) to which this validation should be applied.
- **regex**: A regex pattern to test for matching strings.
- **na_pass**: Should any encountered NA values be allowed to pass a validation unit? This is by default FALSE. Set to TRUE to give NAs a pass.
- **preconditions**: expressions used for mutating the input table before proceeding with the validation. This is ideally as a one-sided R formula using a leading ~. In the formula representation, the tbl serves as the input data table to be transformed (e.g., `~ tbl %>% dplyr::mutate(col = col + 10)`). A series of expressions can be used by enclosing the set of statements with `{ }` but note that the tbl object must be ultimately returned.
- **actions**: A list containing threshold levels so that the validation step can react accordingly when exceeding the set levels. This is to be created with the `action_levels()` helper function.
- **brief**: An optional, text-based description for the validation step.

Details

If providing multiple column names, the result will be an expansion of validation steps to that number of column names (e.g., `vars(col_a, col_b)` will result in the entry of two validation steps). Aside from column names in quotes and in `vars()`, `tidyselect` helper functions are available for specifying columns. They are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, and `everything()`.

This validation step function supports special handling of NA values. The `na_pass` argument will determine whether an NA value appearing in a test unit should be counted as a pass or a fail. The default of `na_pass = FALSE` means that any NAs encountered will accumulate failing test units.

Having table preconditions means `pointblank` will mutate the table just before interrogation. It’s isolated to the validation steps produced by this validation step function. Using `dplyr` code is suggested here since the statements can be translated to SQL if necessary. The code is to be supplied as a one-sided R formula (using a leading ~). In the formula representation, the obligatory tbl variable will serve as the input data table to be transformed (e.g., `~ tbl %>% dplyr::mutate(col_a = col_b + 10)`). A series of expressions can be used by enclosing the set of statements with `{ }` but note that the tbl variable must be ultimately returned.

Often, we will want to specify actions for the validation. This argument, present in every validation step function, takes a specially-crafted list object that is best produced by the `action_levels()` function. Read that function’s documentation for the lowdown on how to create reactions to above-threshold failure levels in validation. The basic gist is that you’ll want at least a single threshold level (specified as either the fraction test units failed, or, an absolute value), often using the `warn_at` argument. This is especially true when `x` is a table object because, otherwise, nothing happens. For the `col_vals_*`()-type functions, using `action_levels(warn_at = 0.25)` or `action_levels(stop_at = 0.25)` are good choices depending on the situation (the first produces a warning when a quarter of the total test units fails, the other stops at the same threshold level).

Want to describe this validation step in some detail? Keep in mind that this is only useful if `x` is an agent. If that’s the case, `brief` the agent with some text that fits. Don’t worry if you don’t want
to do it. The *autobrief* protocol is kicked in when \texttt{brief = NULL} and a simple brief will then be automatically generated.

**Value**

Either a \texttt{ptblank\_agent} object or a table object, depending on what was passed to \texttt{x}.

**Function ID**

2-13

**See Also**

Other Validation Step Functions: \texttt{col\_exists()}, \texttt{col\_is\_character()}, \texttt{col\_is\_date()}, \texttt{col\_is\_factor()}, \texttt{col\_is\_integer()}, \texttt{col\_is\_logical()}, \texttt{col\_is\_numeric()}, \texttt{col\_is\_posix()}, \texttt{col\_vals\_between()}, \texttt{col\_vals\_equal()}, \texttt{col\_vals\_gte()}, \texttt{col\_vals\_gt()}, \texttt{col\_vals\_in\_set()}, \texttt{col\_vals\_lte()}, \texttt{col\_vals\_lt()}, \texttt{col\_vals\_not\_between()}, \texttt{col\_vals\_not\_equal()}, \texttt{col\_vals\_not\_in\_set()}, \texttt{col\_vals\_not\_null()}, \texttt{col\_vals\_null()}, \texttt{conjointly()}, \texttt{rows\_distinct()}

**Examples**

```r
library(dplyr)

# Create a simple table with a
# column containing strings
tbl <- tibble(a = c("s_0131", "s_0231"))

# Validate that all string values in
# column `a` match a regex statement
agent <-
  create_agent(tbl = tbl) %>%
  col_vals_regex(vars(a), "^s_[0-9]{4}$") %>%
  interrogate()

# Determine if these column
# validations have all passed
# by using `all\_passed()`
all_passed(agent)
```

---

<table>
<thead>
<tr>
<th>jointly</th>
<th><strong>Perform multiple rowwise validations for joint validity</strong></th>
</tr>
</thead>
</table>

**Description**

The \texttt{conjointly()} validation step function checks whether the same test units all pass multiple validations with \texttt{col\_vals\_}\*()-type functions. Because of the imposed constraint on the allowed validation step functions, all test units are rows of the table (after any common preconditions have been applied). This validation step function (internally composed of multiple steps) ultimately
performs a rowwise test of whether all sub-validations reported a `pass` for the same test units. In practice, an example of a joint validation is testing whether values for column `a` are greater than a specific value while values for column `b` lie within a specified range. The validation step functions to be part of the conjoint validation are to be supplied as one-sided R formulas (using a leading `~`, and having a `.` stand in as the data object). This function can be used directly on a data table or with an `agent` object (technically, a `ptblank_agent` object).

Usage

```r
conjointly(
  x,
  ...
)
```

Arguments

- `x` A data frame, tibble, or an agent object of class `ptblank_agent`.
- `...` a collection one-sided formulas that consist of validation step functions that validate row units. Specifically, these functions should be those with the naming pattern `col_vals_*()`. An example of this is `~ col_vals_gte(., vars(a), 5.5), ~ col_vals_not_null(., vars(b))`.
- `.list` Allows for the use of a list as an input alternative to `...`.
- `preconditions` expressions used for mutating the input table before proceeding with the validation. This is ideally as a one-sided R formula using a leading `~`. In the formula representation, the `tbl` serves as the input data table to be transformed (e.g., `~ tbl %>% dplyr::mutate(col = col + 10)`). A series of expressions can be used by enclosing the set of statements with `{ }` but note that the `tbl` object must be ultimately returned.
- `actions` A list containing threshold levels so that the validation step can react accordingly when exceeding the set levels. This is to be created with the `action_levels()` helper function.
- `brief` An optional, text-based description for the validation step.

Details

If providing multiple column names in any of the supplied validation step functions, the result will be an expansion of sub-validation steps to that number of column names. Aside from column names in quotes and in `vars()`, `tidyselect` helper functions are available for specifying columns. They are: `starts_with()`, `ends_with()`, `contains()`, `matches()`, and `everything()`.

Having table `preconditions` means `pointblank` will mutate the table just before interrogation. It’s isolated to the validation steps produced by this validation step function. Using `dplyr` code is suggested here since the statements can be translated to SQL if necessary. The code is to be supplied as a one-sided R formula (using a leading `~`). In the formula representation, the obligatory `tbl` variable will serve as the input data table to be transformed (e.g., `~ tbl %>% dplyr::mutate(col_a...`
= col_b + 10). A series of expressions can be used by enclosing the set of statements with { } but note that the tbl variable must be ultimately returned.

Often, we will want to specify actions for the validation. This argument, present in every validation step function, takes a specially-crafted list object that is best produced by the `action_levels()` function. Read that function’s documentation for the lowdown on how to create reactions to above-threshold failure levels in validation. The basic gist is that you’ll want at least a single threshold level (specified as either the fraction test units failed, or, an absolute value), often using the warn_at argument. This is especially true when `x` is a table object because, otherwise, nothing happens. For the `col_vals_*`-type functions, using `action_levels(warn_at = 0.25)` or `action_levels(stop_at = 0.25)` are good choices depending on the situation (the first produces a warning when a quarter of the total test units fails, the other stops at the same threshold level).

Want to describe this validation step in some detail? Keep in mind that this is only useful if `x` is an `agent`. If that’s the case, brief the agent with some text that fits. Don’t worry if you don’t want to do it. The `autobrief` protocol is kicked in when `brief = NULL` and a simple brief will then be automatically generated.

**Function ID**

2-14

**See Also**

Other Validation Step Functions: `col_exists()`, `col_is_character()`, `col_is_date()`, `col_is_factor()`, `col_is_integer()`, `col_is_logical()`, `col_is_numeric()`, `col_is_posix()`, `col_vals_between()`, `col_vals_equal()`, `col_vals_gte()`, `col_vals_gt()`, `col_vals_in_set()`, `col_vals_lte()`, `col_vals_lt()`, `col_vals_not_between()`, `col_vals_not_equal()`, `col_vals_not_in_set()`, `col_vals_not_null()`, `col_vals_null()`, `col_vals_regex()`, `rows_distinct()`

**Examples**

library(dplyr)

# Create a simple table with three
# columns of numerical values

tbl <-
  tibble(
    a = c(5, 7, 6, 5, 8, 7),
    b = c(3, 4, 6, 8, 9, 11),
    c = c(2, 6, 8, NA, 3, 8)
  )

# Validate that values in column
# 'a' are always greater than 4
agent <-
  create_agent(tbl = tbl) %>%
  conjointly(
   - col_vals_gt(., vars(a), 6),
   - col_vals_lt(., vars(b), 10),
   - col_vals_not_null(., vars(c))
  ) %>%
create_agent

interrogate()

create_agent  Create a pointblank agent object

Description
Creates an agent object.

Usage
create_agent(tbl, name = NULL)

Arguments
- **tbl**: The input table that will be the focus of the validation. This can be a data frame, a tibble, or a tbl_dbi object.
- **name**: An optional name for the validation plan that the agent will eventually carry out during the interrogation process. If no value is provided, a name will be generated based on the current system time.

Value
A `ptblank_agent` object.

Function ID
1-1

Examples
library(dplyr)

# Create a simple table with a
column of numerical values
tbl <- tibble(a = c(5, 7, 8, 7))

# Create a pointblank 'agent' object
agent <- create_agent(tbl = tbl)

# Then, as with any 'ptblank_agent'
# object, we can add validation steps
# to the validation plan and then
# eventually use 'interrogate()'
# to perform the validations; here,
# with a single validation step, we
# expect that values in column 'a'
# are always greater than 4
get_agent_report

agent <-
  agent %>%
  col_vals_gt(vars(a), 4) %>%
  interrogate()

# Get a tibble-based report from the
# agent by using `get_agent_report()`
agent %>%
  get_agent_report(display_table = FALSE)

---

get_agent_report  Get a simple report from an agent

Description

We can get the essential information from an agent by using the get_agent_report() function. The amount of fields with intel is different depending on whether or not the agent performed an interrogation (with interrogate()). The tibble that is returned has the following columns:

- i: the validation step number
- type: the validation type, which mirrors the name of the validation step function
- columns: the names of the columns used in the validation step
- value: the numeric value used in the validation step, where applicable
- set: the set values used in the validation step; for a jointly() validation step, this is a listing of all sub-validations
- regex: the regex used for a col_vals_regex() validation step
- preconds: a logical value indicating whether any preconditions where applied before interrogation
- units: the total number of validation units for the validation step
- n_pass: the number of validation units that received a pass
- f_pass: the fraction of validation units that received a pass
- W: a logical value stating whether the warn state was entered
- S: a logical value stating whether the stop state was entered
- N: a logical value stating whether the notify state was entered
- extract: a logical value that indicates whether a data extract is available for the validation step

If the gt package is installed (and if display_table = TRUE) then a gt table will be displayed with the same information.

Usage

get_agent_report(agent, display_table = TRUE)
**get_data_extracts**

Collect data extracts from a validation step

**Description**

Get data that didn’t pass a validation step. The amount of data available in a particular extract depends on both the fraction of validation units that didn’t pass a validation step and the level of sampling or explicit collection from that set of units (this is defined within the `interrogate()` call).

**Arguments**

- **agent**: An agent object of class `ptblank_agent`.
- **display_table**: Should a display table be generated? If TRUE, and if the `gt` package is installed, a display table for the report will be shown in the Viewer. If FALSE, or if `gt` is not available, then a tibble will be returned.

**Value**

A tibble.

**Function ID**

3-3

**See Also**

Other Interrogate and Get Info: `all_passed()`, `get_data_extracts()`, `interrogate()`

**Examples**

```r
library(dplyr)

# Create a simple table with a
# column of numerical values
tbl <- tibble(a = c(5, 7, 8, 5))

# Validate that values in column
# `a` are always greater than 4
agent <-
  create_agent(tbl = tbl) %>%
  col_vals_gt(vars(a), 4) %>%
  interrogate()

# Get a tibble-based report from the
# agent by using `get_agent_report`
agent %>%
  get_agent_report(display_table = FALSE)
```
get_data_extracts

Usage

get_data_extracts(agent, i = NULL)

Arguments

agent  An agent object of class ptblank_agent. It should have had \texttt{interrogate()} called on it, such that the validation steps were carried out and any sample rows from non-passing validations could potentially be available in the object.

i  The validation step number, which is assigned to each validation step in the order of definition.

Value

A list of tibbles if \( i \) is not provided, or, a tibble if \( i \) is given.

Function ID

3-4

See Also

Other Interrogate and Get Info: \texttt{all_passed()}, \texttt{get_agent_report()}, \texttt{interrogate()}

Examples

library(dplyr)

# Create a simple table with a column of numerical values
tbl <- tibble(a = c(5, 7, 8, 5))

# Create 2 simple validation steps that test whether values within column `a`'s agent <-
create_agent(tbl = tbl) %>%
col_vals_between(vars(a), 4, 6) %>%
col_vals_lte(vars(a), 7) %>%
interrogate(
  extract_failed = TRUE,
  get_first_n = 10
)

# Get row sample data for those rows in `tbl` that did not pass the first validation step (`col_vals_between`) agent %>% get_data_extracts(i = 1)
interrogate

Given an agent that has a validation plan, perform an interrogation

Description

When the agent has all the information on what to do (i.e., a validation plan which is a series of validation steps), the interrogation process can occur according its plan. After that, the agent will have gathered intel, and we can use functions like `get_agent_report()` and `all_passed()` to understand how the interrogation went down.

Usage

```r
interrogate(
  agent,
  extract_failed = TRUE,
  get_first_n = NULL,
  sample_n = NULL,
  sample_frac = NULL,
  sample_limit = 5000
)
```

Arguments

- **agent**: An agent object of class `ptblank_agent`.
- **extract_failed**: An option to collect rows that didn’t pass a particular validation step. The default is `TRUE` and further options allow for fine control of how these rows are collected.
- **get_first_n**: If the option to collect non-passing rows is chosen, there is the option here to collect the first `n` rows here. Supply the number of rows to extract from the top of the non-passing rows table (the ordering of data from the original table is retained).
- **sample_n**: If the option to collect non-passing rows is chosen, this option allows for the sampling of `n` rows. Supply the number of rows to sample from the non-passing rows table. If `n` is greater than the number of non-passing rows, then all the rows will be returned.
- **sample_frac**: If the option to collect non-passing rows is chosen, this option allows for the sampling of a fraction of those rows. Provide a number in the range of 0 and 1. The number of rows to return may be extremely large (and this is especially when querying remote databases), however, the `sample_limit` option will apply a hard limit to the returned rows.
- **sample_limit**: A value that limits the possible number of rows returned when sampling non-passing rows using the `sample_frac` option.

Value

A `ptblank_agent` object.
Function ID
3-1

See Also
Other Interrogate and Get Info: all_passed(), get_agent_report(), get_data_extracts()

Examples

library(dplyr)

# Create a simple table with two
tables with numerical values
tbl <-
tibble(
a = c(5, 7, 6, 5, 8, 7),
b = c(7, 1, 0, 0, 0, 3)
)

# Validate that values in column
table's values are always > 5,
# using `interrogate()` carries out
# the validation plan and completes
# the whole process
agent <-
create_agent(tbl = tbl) %>%
col_vals_gt(vars(a), 5) %>%
interrogate()

# Get a tibble-based report from the
# agent by using `get_agent_report()`
agent %>%
get_agent_report(display_table = FALSE)

rows_distinct Verify that row data are distinct

Description
The rows_distinct() validation step function checks whether row values (optionally constrained
to a selection of specified columns) are, when taken as a complete unit, distinct from all other units
in the table. This function can be used directly on a data table or with an agent object (technically, a
ptblank_agent object). This validation step will operate over the number of test units that is equal
to the number of rows in the table (after any preconditions have been applied).
Usage

```
rows_distinct(
  x,
  columns = NULL,
  preconditions = NULL,
  actions = NULL,
  brief = NULL
)
```

Arguments

- **x**: A data frame, tibble, or an agent object of class `ptblank_agent`.
- **columns**: The column (or a set of columns, provided as a character vector) to which this validation should be applied.
- **preconditions**: Expressions used for mutating the input table before proceeding with the validation. This is ideally as a one-sided R formula using a leading `~`. In the formula representation, the `tbl` serves as the input data table to be transformed (e.g., `~ tbl %>% dplyr::mutate(col = col + 10)`). A series of expressions can be used by enclosing the set of statements with `{ }` but note that the `tbl` object must be ultimately returned.
- **actions**: A list containing threshold levels so that the validation step can react accordingly when exceeding the set levels. This is to be created with the `action_levels()` helper function.
- **brief**: An optional, text-based description for the validation step.

Details

We can specify the constraining column names in quotes, in `vars()`, and with the following `tidyselect` helper functions: `starts_with()`, `ends_with()`, `contains()`, `matches()`, and `everything()`.

Having table preconditions means `pointblank` will mutate the table just before interrogation. It’s isolated to the validation steps produced by this validation step function. Using `dplyr` code is suggested here since the statements can be translated to SQL if necessary. The code is to be supplied as a one-sided R formula (using a leading `~`). In the formula representation, the obligatory `tbl` variable will serve as the input data table to be transformed (e.g., `~ tbl %>% dplyr::mutate(col_a = col_b + 10)`). A series of expressions can be used by enclosing the set of statements with `{ }` but note that the `tbl` variable must be ultimately returned.

Often, we will want to specify actions for the validation. This argument, present in every validation step function, takes a specially-crafted list object that is best produced by the `action_levels()` function. Read that function’s documentation for the lowdown on how to create reactions to above-threshold failure levels in validation. The basic gist is that you’ll want at least a single threshold level (specified as either the fraction test units failed, or, an absolute value), often using the `warn_at` argument. This is especially true when `x` is a table object because, otherwise, nothing happens. For the `col_vals_*()`-type functions, using `action_levels(warn_at = 0.25)` or `action_levels(stop_at = 0.25)` are good choices depending on the situation (the first produces a warning when a quarter of the total test units fails, the other `stop()`s at the same threshold level).

Want to describe this validation step in some detail? Keep in mind that this is only useful if `x` is an `agent`. If that’s the case, `brief` the agent with some text that fits. Don’t worry if you don’t want
to do it. The *autobrief* protocol is kicked in when `brief = NULL` and a simple brief will then be automatically generated.

**Value**

Either a `ptblank_agent` object or a table object, depending on what was passed to `x`.

**Function ID**

2-15

**See Also**

Other Validation Step Functions: `col_exists()`, `col_is_character()`, `col_is_date()`, `col_is_factor()`, `col_is_integer()`, `col_is_logical()`, `col_is_numeric()`, `col_is_posix()`, `col_vals_between()`, `col_vals_equal()`, `col_vals_gte()`, `col_vals_gt()`, `col_vals_in_set()`, `col_vals_lte()`, `col_vals_lt()`, `col_vals_not_between()`, `col_vals_not_equal()`, `col_vals_not_in_set()`, `col_vals_not_null()`, `col_vals_null()`, `col_vals_regex()`, `conjointly()`

**Examples**

```r
library(dplyr)

# Create a simple table with three columns of numerical values
tbl <-
  tibble(
    a = c(5, 7, 6, 5, 8, 7),
    b = c(7, 1, 0, 0, 8, 3),
    c = c(1, 1, 1, 3, 3, 3)
  )

# Validate that when considering only data in columns 'a' and 'b', there are no duplicate rows (i.e., all rows are distinct)
agent <-
  create_agent(tbl = tbl) %>%
  rows_distinct(vars(a, b)) %>%
  interrogate()

# Determine if these column validations have all passed
# by using `all_passed()`
all_passed(agent)
```
rows_not_duplicated  Verify that row data are not duplicated (deprecated)

Description

Verify that row data are not duplicated (deprecated)

Usage

rows_not_duplicated(
  x,
  columns = NULL,
  preconditions = NULL,
  brief = NULL,
  actions = NULL
)

Arguments

x  An agent object of class ptblank_agent.

columns The column (or a set of columns, provided as a character vector) to which this validation should be applied.

preconditions expressions used for mutating the input table before proceeding with the validation. This is ideally as a one-sided R formula using a leading ~. In the formula representation, the tbl serves as the input data table to be transformed (e.g., ~ tbl %>% dplyr::mutate(col = col + 10). A series of expressions can be used by enclosing the set of statements with {} but note that the tbl object must be ultimately returned.

brief An optional, text-based description for the validation step.

actions A list containing threshold levels so that the validation step can react accordingly when exceeding the set levels. This is to be created with the action_levels() helper function.

Value

A ptblank_agent object.
small_table

A small table that useful for testing

Description

This is a small table with a few different types of columns. It’s probably just useful when testing the functions from pointblank. Rows 9 and 10 are exact duplicates. The c column contains two NA values.

Usage

small_table

Format

A tibble with 13 rows and 8 variables:

- **date_time** A date-time column (of the POSIXct class) with dates that correspond exactly to those in the date column. Time values are somewhat randomized but all 'seconds' values are 00.
- **date** A Date column with dates from 2016-01-04 to 2016-01-30.
- **a** An integer column with values ranging from 1 to 8.
- **b** A character column with values that adhere to a common pattern.
- **c** An integer column with values ranging from 2 to 9. Contains two NA values.
- **d** A numeric column with values ranging from 108 to 10000.
- **e** A logical column.
- **f** A character column with "low", "mid", and "high" values.

Function ID

4-1

Examples

```r
library(dplyr)

# Here is a glimpse at the data
# available in `small_table`
dplyr::glimpse(small_table)
```
Index

+Topic **datasets**
  - small_table, 62

action_levels, 2
action_levels(), 5–7, 9, 11–14, 16, 18, 20,
  22, 23, 25, 27, 28, 30, 32, 34, 35,
  37–40, 42, 44–47, 49, 51, 52, 59, 61
all_passed, 4, 55, 56, 58
all_passed(), 57

col_exists, 5, 8, 10, 11, 13, 15, 17, 18, 21,
  23, 26, 30, 33, 35, 38, 40, 43, 45,
  47, 50, 52, 60

col_is_character, 6, 7, 10, 11, 13, 15, 17,
  18, 21, 23, 26, 28, 30, 33, 35, 38, 40,
  43, 45, 47, 50, 52, 60

col_is_date, 6, 8, 9, 11, 13, 15, 17, 18, 21,
  23, 26, 30, 33, 35, 38, 40, 43, 45,
  47, 50, 52, 60

col_is_factor, 6, 8, 10, 10, 13, 15, 17, 18,
  21, 23, 26, 30, 33, 35, 38, 40, 43,
  45, 47, 50, 52, 60

col_is_integer, 6, 8, 10, 11, 12, 15, 17, 18,
  21, 23, 26, 28, 30, 33, 35, 38, 40, 43,
  45, 47, 50, 52, 60

col_is_logical, 6, 8, 10, 11, 13, 14, 17, 18,
  21, 23, 26, 30, 33, 35, 38, 40, 43,
  45, 47, 50, 52, 60

col_is_numeric, 6, 8, 10, 11, 13, 15, 18,
  21, 23, 26, 30, 33, 35, 38, 40, 43,
  45, 47, 50, 52, 60

col_is_posix, 6, 8, 10, 11, 13, 15, 17, 21,
  23, 26, 28, 30, 33, 35, 38, 40, 43, 45,
  47, 50, 52, 60

col_vals_between, 6, 8, 10, 11, 13, 15, 17,
  18, 19, 23, 26, 28, 30, 33, 35, 38, 40,
  43, 45, 47, 50, 52, 60

col_vals_between(), 38

col_vals_equal, 6, 8, 10, 11, 13, 15, 17, 18,
  21, 22, 26, 28, 30, 33, 35, 38, 40, 43,
INDEX

21, 23, 26, 28, 31, 33, 35, 38, 40, 43, 45, 46, 50, 52, 60

col_vals_null(), 45

col_vals_regex, 6, 8, 10, 11, 13, 15, 17, 18, 21, 23, 26, 28, 31, 33, 35, 38, 40, 43, 45, 47, 48, 52, 60

conjointly, 6, 8, 10, 11, 13, 15, 17, 18, 21, 23, 26, 28, 31, 33, 35, 38, 40, 43, 45, 47, 50, 50, 60

create_agent, 53

get_agent_report, 4, 54, 56, 58
get_agent_report(), 57

generate_data_extracts, 4, 55, 55, 58

interrogate, 4, 55, 56, 57
interrogate(), 55, 56

rows_distinct, 6, 8, 10, 11, 13, 15, 17, 18, 21, 23, 26, 28, 31, 33, 35, 38, 40, 43, 45, 47, 50, 52, 58

rows_not_duplicated, 61

small_table, 62

vars(), 5