R topics documented:

action_levels ........................................ 3
allPassed ............................................ 5
col_exists ............................................ 6
col_is_character ..................................... 7
col_is_date ........................................... 9
col_is_factor ......................................... 11
col_is_integer ....................................... 13
col_is_logical ....................................... 15
col_is_numeric ..................................... 16
col_is_posix ......................................... 18
col_schema ........................................... 20
col_schema_match ................................... 22
col_vals_between ................................... 24
col_vals_equal ....................................... 27
col_vals_gt .......................................... 29
col_vals_gte ......................................... 32
col_vals_in_set ..................................... 34
col_vals_lt .......................................... 37
col_vals_lte ......................................... 39
col_vals_not_between ................................... 42
col_vals_not_equal .................................. 45
col_vals_not_in_set .................................. 47
col_vals_not_null .................................... 50
col_vals_null ....................................... 52
col_vals_regex ...................................... 55
conjointly ............................................. 57
create_agent ....................................... 60
email Blast .......................................... 61
email_preview ....................................... 64
get_agent_report .................................... 66
get_agent_x_list .................................... 68
get_data_extracts ................................... 70
get_sundered_data .................................. 72
interrogate .......................................... 73
rows_distinct ....................................... 74
rows_not_duplicated .................................. 77
scan_data ........................................... 78
small_table .......................................... 79
small_table_sqlite .................................. 80
stock_msg_body ..................................... 81
stock_msg_footer .................................... 82

Index 83
action_levels

Set action levels: failure thresholds and functions to invoke

Description

The action_levels() function works with the actions argument that is present in the create_agent() function and 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. If provided to create_agent() then the policies will be applied to every validation step, acting as a default for the validation as a whole. Calls of action_levels() could also be applied directly to any validation step and this will act as an override if set also in create_agent(). Usage of action_levels() is required to have any useful side effects (i.e., warnings, throwing errors) in the case of validation step functions operating directly on data.

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 failure states. If this a decimal value between 0 and 1 then it’s a proportional failure threshold (e.g., 0.15 indicates that if 15% percent of the validation units are found to fail, then the designated failure state is entered). Absolute values starting from 1 can be used instead, and this constitutes an absolute failure threshold (e.g., 10 means that if 10 of the validation units are found to fail, the failure state is entered).

fns

A named list of functions that is to be paired with the appropriate failure states. The syntax for this list involves using failure state names from the set of warn, stop, and notify. The functions corresponding to the failure 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 differently if using an agent or using validation step functions 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. Furthermore, if actions is NULL in this workflow, pointblank will use a warn_at value of 1 (providing a warning if there are any fail units). We can absolutely suppress this automatic warning behavior by at each validation step by setting active = FALSE. In this interactive data case there is no stock function given for the notify_at. The notify failure state is less commonly used in this workflow as it is in the agent-based one.
When using an *agent*, we often opt to not use any functions in *fns* as the *warn*, *stop*, and *notify* failure states will be reported on when using `create_agent_report()` (and, usually that's sufficient).

**Function ID**

1-3

**See Also**

Other Planning and Prep: `col_schema()`, `create_agent()`, `scan_data()`

**Examples**

```r
# Create a simple data frame with
# a column of numerical values
tbl <- dplyr::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) %>%
  interrogate()

# The report from the agent will show
# that the `warn` state has been entered
# for the first and only validation step.
# agent

# In the context of using validation
# stop functions directly on data, their
# use is commonly to trigger warnings
# and raise errors. The following will
# provide a warning (but that's suppressed
# here).
suppressWarnings(
  tbl %>%
  col_vals_gt(vars(a), 5, actions = al)
)
```
**all_passed**

---

Did all of the validations fully pass?

---

**Description**

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**

`all_passed(agent)`

**Arguments**

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

**Value**

A logical value.

**Function ID**

5-5

**See Also**

Other Post-interrogation: `get_agent_report()`, `get_agent_x_list()`, `get_data_extracts()`, `get_sundered_data()`

**Examples**

```r
# Create a simple table with  
# a column of numerical values  
tbl <-  
  dplyr::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, active = TRUE)
```

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 `c()` 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.
- `active`: A logical value indicating whether the validation step should be active. If the step function is working with an agent, `FALSE` will make the validation step inactive (still reporting its presence and keeping indexes for the steps unchanged). If the step function will be operating directly on data, then any step with `active = FALSE` will simply pass the data through with no validation whatsoever. The default for this is `TRUE`.

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

# Validate that columns `a` and `b`
# exist in the `tbl` table
agent <-
  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

```r
col_is_character(x, columns, actions = NULL, brief = NULL, active = TRUE)
```

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.
- **active**: A logical value indicating whether the validation step should be active. If the step function is working with an agent, `FALSE` will make the validation step inactive (still reporting its presence and keeping indexes for the steps unchanged). If the step function will be operating directly on data, then any step with `active = FALSE` will simply pass the data through with no validation whatsoever. The default for this is `TRUE`.

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

2-16
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_schema_match()`, `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
# Create a simple table with
# a column of `character` values
tbl <-
  dplyr::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)
```

---

**col_is_date**

*Do the columns contain R Date objects?*

Description

The `col_is_date()` validation step function checks whether one or more columns is of the R Date 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 Date-type column or not.

Usage

```r
col_is_date(x, columns, actions = NULL, brief = NULL, active = TRUE)
```

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.

active A logical value indicating whether the validation step should be active. If the step function is working with an agent, FALSE will make the validation step inactive (still reporting its presence and keeping indexes for the steps unchanged). If the step function will be operating directly on data, then any step with active = FALSE will simply pass the data through with no validation whatsoever. The default for this is TRUE.

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

2-20

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_schema_match(), 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

# Create a simple table with
# a column of 'Date' values
tbl <-
dplyr::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.

Usage

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

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.
- `active` A logical value indicating whether the validation step should be active. If the step function is working with an agent, FALSE will make the validation step inactive (still reporting its presence and keeping indexes for the steps unchanged). If the step function will be operating directly on data, then any step with active = FALSE will simply pass the data through with no validation whatsoever. The default for this is TRUE.
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 valida-
tion 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

2-22

See Also

Other Validation Step Functions: col_exists(), col_is_character(), col_is_date(), col_is_integer(),
col_is_logical(), col_is_numeric(), col_is_posix(), col_schema_match(), 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

# Create a simple table with
# a column of 'factor' values
tbl <-
dplyr::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**

```r
col_is_integer(x, columns, actions = NULL, brief = NULL, active = TRUE)
```

**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.
- **active**: A logical value indicating whether the validation step should be active. If the step function is working with an agent, `FALSE` will make the validation step inactive (still reporting its presence and keeping indexes for the steps unchanged). If the step function will be operating directly on data, then any step with `active = FALSE` will simply pass the data through with no validation whatsoever. The default for this is `TRUE`.

**Details**

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.
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_exists(), col_is_character(), col_is_date(), col_is_factor(), col_is_logical(), col_is_numeric(), col_is_posix(), col_schema_match(), 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

# Create a simple table with a column of 'integer' values
tbl <-
dplyr::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

**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, active = TRUE)
```

**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.
- `active`: A logical value indicating whether the validation step should be active. If the step function is working with an agent, `FALSE` will make the validation step inactive (still reporting its presence and keeping indexes for the steps unchanged). If the step function will be operating directly on data, then any step with `active = FALSE` will simply pass the data through with no validation whatsoever. The default for this is `TRUE`.

**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**

2-19

**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_schema_match()`, `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
# Create a simple table with a column of 'logical' values
tbl <-
  dplyr::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, active = TRUE)
```

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.
- **active**: A logical value indicating whether the validation step should be active. If the step function is working with an agent, `FALSE` will make the validation step inactive (still reporting its presence and keeping indexes for the steps unchanged). If the step function will be operating directly on data, then any step with `active = FALSE` will simply pass the data through with no validation whatsoever. The default for this is `TRUE`.

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_schema_match()`, `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
# Create a simple table with a # column of `numeric` values
tbl <-
dplyr::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)) %>%
test()

# 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, active = TRUE)
```

**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.

active  A logical value indicating whether the validation step should be active. If the step function is working with an agent, FALSE will make the validation step inactive (still reporting its presence and keeping indexes for the steps unchanged). If the step function will be operating directly on data, then any step with `active = FALSE` will simply pass the data through with no validation whatsoever. The default for this is TRUE.

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_schema_match()`, `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
# Create a simple table with a
# column of `POSIXct` values
tbl <-
dplyr::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_schema**

*Generate a table column schema manually or with a reference table*

**Description**

A table column schema object, as can be created by `col_schema()`, is necessary when using the `col_schema_match()` validation step function (which checks whether the table object under study matches a known column schema). The `col_schema` object can be made by carefully supplying the column names and their types as a set of named arguments, or, we could provide a table object, which could be of the `data.frame`, `tbl_df`, or `tbl_dbi` varieties. There’s an additional option, which is just for validating the schema of a `tbl_dbi` object: we can validate the schema based on R column types (e.g., "numeric", "character", etc.), or, SQL column types (e.g., "double", "varchar", etc.). This is great if we want to validate table column schemas both on the server side and when tabular data is collected and loaded into R.

**Usage**

```r
col_schema(..., .tbl = NULL, .db_col_types = c("r", "sql"))
```

**Arguments**

*...* A set of named arguments where the names refer to column names and the values are one or more column types.
An option to use a table object to define the schema. If this is provided then any values provided to \ldots will be ignored.

Determines whether the column types refer to R column types ("r") or SQL column types ("sql").

Function ID

1-4

See Also

Other Planning and Prep: action_levels(), create_agent(), scan_data()

Examples

```r
# Create a simple table with two columns: one `integer` and the other `character`
tbl <-
dplyr::tibble(  
a = 1:5,  
b = letters[1:5]
)

# Create a column schema object that describes the columns and their types (in the expected order)
schema_obj <-
col_schema(  
a = "integer",  
b = "character"
)

# Validate that the schema object exactly defines the column names and column types of the `tbl` table
agent <-
create_agent(tbl = tbl) %>%
col_schema_match(schema_obj) %>%
interrogate()

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

# We can alternatively create a column schema object from a `tbl_df` object
schema_obj <-
col_schema(  
...  
```
.tbl = dplyr::tibble(
  a = integer(0),
  b = character(0)
)

# This should provide the same
# interrogation results as in the
# previous example
create_agent(tbl = tbl) %>%
  col_schema_match(schema_obj) %>%
  interrogate() %>%
  all_passed()

---

**col_schema_match**

*Do columns in the table (and their types) match a predefined schema?*

**Description**

The `col_schema_match()` validation step function works in conjunction with a `col_schema` object (generated through the `col_schema()` function) to determine whether the expected schema matches the target table. This validation step operates over a single test unit, which is whether the schema exactly matches that of the table. If the target table is a `tbl_sql` object, we can choose to validate the column schema that is based on R column types (e.g., "numeric", "character", etc.), or, SQL column types (e.g., "double", "varchar", etc.). That option is defined in the `col_schema()` function (with the `.db_col_types` argument).

**Usage**

`col_schema_match(x, schema, actions = NULL, brief = NULL, active = TRUE)`

**Arguments**

- **x**: A data frame, tibble, or an agent object of class `ptblank_agent`.
- **schema**: A table schema of type `col_schema` which can be generated using the `col_schema()` function.
- **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.
- **active**: A logical value indicating whether the validation step should be active. If the step function is working with an agent, `FALSE` will make the validation step inactive (still reporting its presence and keeping indexes for the steps unchanged). If the step function will be operating directly on data, then any step with `active = FALSE` will simply pass the data through with no validation whatsoever. The default for this is `TRUE`. 
Details

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()).

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

2-24

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()`, `rows_distinct()`

Examples

```r
# Create a simple table with
# two columns: one 'integer' and
# the other 'character'
tbl <-
  dplyr::tibble(
    a = 1:5,
    b = letters[1:5]
  )

# Create a column schema object
# that describes the columns and
# their types (in the expected
# order)
schema_obj <-
  col_schema(
    a = "integer",
    b = "character"
  )

# Validate that the schema object
# 'col_schema_x' exactly defines
# the column names and column types
# of the 'tbl_x' table
```
agent <-
  create_agent(tbl = tbl) %>%
  col_schema_match(schema_obj) %>%
  interrogate()

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

---

**col_vals_between**

*Are 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 bounds. The bounds can be specified as single, literal values or as column names given in `vars()`. 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**

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

**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.
col_vals_between

left  
The lower bound for the range. The validation includes this bound value (if the
first element in inclusive is TRUE) in addition to values greater than left. This
can be a single value or a compatible column given in vars().

right 
The upper bound for the range. The validation includes this bound value (if the
second element in inclusive is TRUE) in addition to values lower than right.
This can be a single value or a compatible column given in vars().

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 valida-
tion. 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.

active  
A logical value indicating whether the validation step should be active. If the
step function is working with an agent, FALSE will make the validation step in-
active (still reporting its presence and keeping indexes for the steps unchanged).
If the step function will be operating directly on data, then any step with active
= FALSE will simply pass the data through with no validation whatsoever. The
default for this is TRUE.

Details

If providing multiple column names to columns, 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 valida-
tion 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**

2-7

**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_schema_match()`, `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
# Create a simple table with
# a column of numerical values
tbl <-
dplyr::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)
```
Description

The `col_vals_equal()` validation step function checks whether column values (in any number of specified columns) are equal to a specified value. The value can be specified as a single, literal value or as a column name given in `vars()`. 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, # 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 = 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.
  active = TRUE)
```

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.
active

A logical value indicating whether the validation step should be active. If the step function is working with an agent, FALSE will make the validation step inactive (still reporting its presence and keeping indexes for the steps unchanged). If the step function will be operating directly on data, then any step with active = FALSE will simply pass the data through with no validation whatsoever. The default for this is TRUE.

Details

If providing multiple column names to columns, 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 pointblank_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_schema_match()`, `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
# Create a simple table with two columns of numerical values
tbl <-
dplyr::tibble(
  a = c(1, 1, 1, 2, 2, 2),
  b = c(5, 5, 5, 3, 6, 3)
)

# Validate that values in column 'b' are equal to 5 when values in column 'a' are equal to 1
agent <-
  create_agent(tbl = tbl) %>%
  col_vals_equal(vars(b), 5,
                    preconditions =
                    ~ tbl %>% dplyr::filter(a == 1)
               ) %>%
  interrogate()

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

---

**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`). The value can be specified as a single, literal value or as a column name given in `vars()`. 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_gt(
  x,
  columns,
  value,
  na_pass = FALSE,
  preconditions = NULL,
  actions = NULL,
  brief = NULL,
  active = TRUE
)

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.
active A logical value indicating whether the validation step should be active. If the step function is working with an agent, FALSE will make the validation step inactive (still reporting its presence and keeping indexes for the steps unchanged). If the step function will be operating directly on data, then any step with active = FALSE will simply pass the data through with no validation whatsoever. The default for this is TRUE.

Details

If providing multiple column names to columns, 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_schema_match(), 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(), conjoinedy(), rows_distinct()

Examples

# Create a simple table with a
# column of numerical values
tbl <-
dplyr::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_vals_gte Are column data greater than or equal to a specified value?

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). The value can be specified as a single, literal value or as a column name given in vars(). 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_gte(
  x,
  columns,
  value,
  na_pass = FALSE,
  preconditions = NULL,
  actions = NULL,
  brief = NULL,
  active = TRUE
)

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.
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.

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.

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

A logical value indicating whether the validation step should be active. If the step function is working with an agent, FALSE will make the validation step inactive (still reporting its presence and keeping indexes for the steps unchanged). If the step function will be operating directly on data, then any step with active = FALSE will simply pass the data through with no validation whatsoever. The default for this is TRUE.

If providing multiple column names to columns, 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

2-5

See Also

The analogous function with a left-open bound: `col_vals_gt()`.

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_schema_match()`, `col_vals_between()`, `col_vals_equal()`, `col_vals_gte()`, `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
# Create a simple table with a column of numerical values
tbl <-
  dplyr::tibble(a = c(5, 7, 8, 5))

# Validate that values in column `a` are always greater than or equal to 5
agent <-
  create_agent(tbl = tbl) %>%
  col_vals_gte(vars(a), 5) %>%
  interrogate()

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

### col_vals_in_set

**Are column data part of a specified 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).
col_vals_in_set

Usage

col_vals_in_set(
  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
set,  # validation should be applied.
set,  # A vector of numeric or string-based elements, where column values found within
preconditions = NULL,  # this set will be considered as passing.
actions = NULL,  # expressions used for mutating the input table before proceeding with the valida-
brief = NULL,  # tion. This is ideally as a one-sided R formula using a leading ~. In the formula
active = TRUE  # 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
Arguments
ultimately returned.

x  # A list containing threshold levels so that the validation step can react accordingly
columns  # when exceeding the set levels. This is to be created with the action_levels() helper function.
set  # An optional, text-based description for the validation step.
preconditions  # A logical value indicating whether the validation step should be active. If the
actions  # step function is working with an agent, FALSE will make the validation step in-
brief  # active (still reporting its presence and keeping indexes for the steps unchanged).
active  # If the step function will be operating directly on data, then any step with active

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**

2-9

**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_schema_match()`, `col_vals_between()`, `col_vals_equal()`, `col_vals_gte()`, `col_vals_gt()`, `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
# Create a simple table with 2 columns: one with numerical values, the other with strings
tbl <-
  dplyr::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
```

36
col_vals_in_set
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 column data less than a specified 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`). The value can be specified as a single, literal value or as a column name given in `vars()`. 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,  # 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 = 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,
  actions = NULL,
  brief = NULL,
  active = TRUE
)
```

**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.

active A logical value indicating whether the validation step should be active. If the step function is working with an agent, FALSE will make the validation step inactive (still reporting its presence and keeping indexes for the steps unchanged). If the step function will be operating directly on data, then any step with active = FALSE will simply pass the data through with no validation whatsoever. The default for this is TRUE.

Details

If providing multiple column names to columns, 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_schema_match()`, `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**

```r
# Create a simple table with a column of numerical values
tbl <-
  dplyr::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)
```

---

**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`). The value can be specified as a single, literal value or as a column name given in `vars()`. 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_lte(
  x,
  columns,
  value,
  na_pass = FALSE,
  preconditions = NULL,
  actions = NULL,
  brief = NULL,
  active = TRUE
)

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.

active A logical value indicating whether the validation step should be active. If the step function is working with an agent, FALSE will make the validation step inactive (still reporting its presence and keeping indexes for the steps unchanged). If the step function will be operating directly on data, then any step with active = FALSE will simply pass the data through with no validation whatsoever. The default for this is TRUE.

Details

If providing multiple column names to columns, 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

2-2

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_schema_match(), 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

# Create a simple table with a
# column of numerical values
tbl <-
dplyr::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 column data not between two specified values?

**Description**

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 bounds. The bounds can be specified as single, literal values or as column names given in `vars()`. 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**

```r
col_vals_not_between(
  x,  # required
  columns,  # required
  left,  # required
  right,  # required
  inclusive = c(TRUE, TRUE),
  na_pass = FALSE,
  preconditions = NULL,
  actions = 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 >= left and <= 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.
- **active**: A logical value indicating whether the validation step should be active. If the step function is working with an agent, FALSE will make the validation step inactive (still reporting its presence and keeping indexes for the steps unchanged). If the step function will be operating directly on data, then any step with active = FALSE will simply pass the data through with no validation whatsoever. The default for this is TRUE.

Details

If providing multiple column names to columns, 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

2-8

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_schema_match()`, `col_vals_between()`, `col_vals_equal()`, `col_vals_gte()`, `col_vals_gt()`, `col_vals_in_set()`, `col_vals_lte()`, `col_vals_lte()`, `col_vals_not_equal()`, `col_vals_not_in_set()`, `col_vals_not_null()`, `col_vals_not_null()`, `col_vals_regex()`, `conjointly()`., `rows_distinct()`

Examples

```r
# Create a simple table with a
column of numerical values
tbl <-
dplyr::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 column data not equal to a specified 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, 
  columns, 
  value, 
  na_pass = FALSE, 
  preconditions = NULL, 
  actions = NULL, 
  brief = NULL, 
  active = TRUE
)
```

#### 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 non-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.
active  A logical value indicating whether the validation step should be active. If the step function is working with an agent, FALSE will make the validation step inactive (still reporting its presence and keeping indexes for the steps unchanged). If the step function will be operating directly on data, then any step with active = FALSE will simply pass the data through with no validation whatsoever. The default for this is TRUE.

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 pointblank_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_schema_match()`, `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
# Create a simple table with two
# columns of numerical values
tbl <-
  dplyr::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)
```

---

**col_vals_not_in_set**

*Are data not part of a specified set of values?*

**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).

---

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_schema_match()`, `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
# Create a simple table with two
# columns of numerical values
tbl <-
  dplyr::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)
```

---

**col_vals_not_in_set**

*Are data not part of a specified set of values?*

**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

\[
col_vals_not_in_set(
  x,
  columns,
  set,
  preconditions = NULL,
  actions = NULL,
  brief = NULL,
  active = TRUE
)
\]

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.
- **active**: A logical value indicating whether the validation step should be active. If the step function is working with an agent, FALSE will make the validation step inactive (still reporting its presence and keeping indexes for the steps unchanged). If the step function will be operating directly on data, then any step with active = FALSE will simply pass the data through with no validation whatsoever. The default for this is TRUE.

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

2-10

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_schema_match(), 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()
# 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)

---

col_vals_not_null Are column data not NULL/NA?

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, active = TRUE
)
```

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.
col_vals_not_null

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.

active  A logical value indicating whether the validation step should be active. If the step function is working with an agent, FALSE will make the validation step inactive (still reporting its presence and keeping indexes for the steps unchanged). If the step function will be operating directly on data, then any step with active = FALSE will simply pass the data through with no validation whatsoever. The default for this is TRUE.

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_schema_match()`, `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

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

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

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

---

### col_vals_null

<table>
<thead>
<tr>
<th>Are column data NULL/NA?</th>
</tr>
</thead>
</table>

**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).
col_vals_null

Usage

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

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.
- **active**: A logical value indicating whether the validation step should be active. If the step function is working with an agent, FALSE will make the validation step inactive (still reporting its presence and keeping indexes for the steps unchanged). If the step function will be operating directly on data, then any step with `active = FALSE` will simply pass the data through with no validation whatsoever. The default for this is TRUE.

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**

2-11

**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_schema_match()`, `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
# Create a simple table with two
# columns of numerical values
tbl <-
  dplyr::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)
```
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,
  active = TRUE
)
```

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**: Null. 
- **brief**: Null.
- **active**: TRUE.
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.

active A logical value indicating whether the validation step should be active. If the step function is working with an agent, `FALSE` will make the validation step inactive (still reporting its presence and keeping indexes for the steps unchanged). If the step function will be operating directly on data, then any step with `active = FALSE` will simply pass the data through with no validation whatsoever. The default for this is `TRUE`.

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 interpolation. 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`. 

conjointly

Function ID
2-13

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_schema_match()`, `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()`, `conjointly()`, `rows_distinct()`

Examples

```r
# Create a simple table with a
dataframe containing strings
tbl <-
dplyr::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>conjointly</th>
<th>Perform multiple rowwise validations for joint validity</th>
</tr>
</thead>
</table>

Description

The `conjointly()` validation step function checks whether the same test units all pass multiple validations with `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).
conjointly(
  x,
  ...,
  .list = list2(...),
  preconditions = NULL,
  actions = NULL,
  brief = NULL,
  active = TRUE
)

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.

active A logical value indicating whether the validation step should be active. If the step function is working with an agent, FALSE will make the validation step inactive (still reporting its presence and keeping indexes for the steps unchanged). If the step function will be operating directly on data, then any step with active = FALSE will simply pass the data through with no validation whatsoever. The default for this is TRUE.

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...)} \n
conjointly
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.

### 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_schema_match()`, `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

```r
# Create a simple table with three columns of numerical values
tbl <-
  dplyr::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))) %>%
  interrogate()
```
Create a pointblank agent object

Description

Creates an agent object, which is used in a data quality reporting workflow. The overall aim of this workflow is to generate some reporting on the state of data quality for a target table. We can supply as many validation step functions as the user wishes to write to get validation coverage on that table. We supply a single table to create_agent(), which becomes the sole focus of the data quality analysis. After application of one or more validation step functions, we need to use the interrogate() function to complete the process; the validation step functions, when called on an agent, are merely instructions up to that point. The agent then has information on how the interrogation went. Reporting of the interrogation can be performed with the get_agent_report() function. If we just need to know whether all validations completely passed (i.e., all steps had no failing test units), the all_passed() function should be used.

Usage

create_agent(
  tbl,
  name = NULL,
  actions = NULL,
  end_fns = NULL,
  embed_report = FALSE,
  reporting_lang = NULL
)

Arguments

tbl The input table. 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.

actions A list containing threshold levels so that all validation steps can react accordingly when exceeding the set levels. This is to be created with the action_levels() helper function. Should an action levels list be used for specific validation step, any default set here will be overridden.

derm_fns A list of functions that should be performed at the end of an interrogation.

embed_report An option to embed a gt-based validation report into the ptblank_agent object. If FALSE (the default) then the table object will be not generated and available with the agent upon returning from the interrogation.

reporting_lang The language to use for automatic creation of briefs (short descriptions for each validation step). By default, NULL will create English ("en") text. Other options include French ("fr"), German ("de"), Italian ("it"), and Spanish ("es").
Value

A ptblank_agent object.

Function ID

1-2

See Also

Other Planning and Prep: action_levels(), col_schema(), scan_data()

Examples

```r
# Create a simple table with a
# column of numerical values
tbl <-
  dplyr::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
agent <-
  agent %>%
    col_vals_gt(vars(a), 4) %>%
    interrogate()
```

Description

The email_blast() function is useful for sending an email message that explains the result of a pointblank validation. It is powered by the blastula and glue packages. This function should be invoked as part of the end_fns argument of create_agent(). It’s also possible to invoke email_blast() as part of the fns argument of the action_levels() function (to possibly send an email message at one or more steps).
email_blast

Usage

email_blast(
  x,
  to,
  from,
  credentials = NULL,
  msg_subject = NULL,
  msg_header = NULL,
  msg_body = stock_msg_body(),
  msg_footer = stock_msg_footer(),
  send_condition = ~TRUE %in% x$notify
)

Arguments

x A reference to list object prepared by the agent. It’s only available in an internal evaluation context.
to, from The email addresses for the recipients and the sender.
credentials A credentials list object that is produced by either of the blastula::creds(), blastula::creds_anonymous(), blastula::creds_key(), or blastula::creds_file() functions. Please refer to the blastula documentation for details on each of these helper functions.
msg_subject The subject line of the email message.
msg_header, msg_body, msg_footer Content for the header, body, and footer components of the HTML email message.
send_condition An expression that should evaluate to a logical vector of length 1. If TRUE then the email will be sent, if FALSE then that won’t happen. The expression can use x-list variables (e.g., x$notify, x$type, etc.) and all of those variables can be viewed using the get_agent_x_list() function. The default expression is ~TRUE %in% x$notify, which results in TRUE if there are any TRUE values in the x$notify logical vector (i.e., any validation step results in a ‘notify’ condition).

Details

To better get a handle on emailing with email_blast(), the analogous email_preview() can be used with a pointblank agent object or the output obtained from using the get_agent_x_list() function.

Function ID

3-1

See Also

Other Emailing: email_preview(), stock_msg_body(), stock_msg_footer()
Examples

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

# Create an action_levels() list with absolute values for the warn, and notify states (with thresholds of 1 and 2 'fail' units)
al <-
  action_levels(
    warn_at = 1,
    notify_at = 2
  )

# Validate that values in column `a` from `tbl` are always > 5 and that `b` values are always < 5;
# first, apply the `actions_levels()` directive to `actions` and set up an `email_blast()` as one of the `end_fns` (by default, the email will be sent if there is a single `notify` state across all validation steps)
# agent <-
# create_agent(
#   tbl = tbl,
#   actions = al,
#   end_fns = list(
#     ~ email_blast(
#       x,
#       to = "joe_public@example.com",
#       from = "pb_notif@example.com",
#       msg_subject = "Table Validation",
#       credentials = blastula::creds_key(
#         id = "gmail"
#       ),
#     )
#   )
# )
# )
# col_vals_gt(vars(a), 5) %>%
# col_vals_lt(vars(b), 5) %>%
# interrogate()

# This example was intentionally not run because email credentials aren't available and the `to`
# and `from` email addresses are nonexistent; to look at the email message before sending anything of the like, we can use the `email_preview()` function

e-mail_object <-
  create_agent(
    tbl = tbl,
    actions = al
  ) %>%
  col_vals_gt(vars(a), 5) %>%
  col_vals_lt(vars(b), 5) %>%
  interrogate() %>%
  email_preview()

email_preview  Get a preview of an email before actually sending that email

Description

The `email_preview()` function provides a preview of an email that would normally be produced and sent through the `email_blast()` function. The `x` that we need for this is the agent x-list that is produced by the `get_agent_x_list()` function. Or, we can supply an agent object. In both cases, the email message will appear in the Viewer and a `blastula` `email_message` object will be returned.

Usage

email_preview(
  x,
  msg_header = NULL,
  msg_body = stock_msg_body(),
  msg_footer = stock_msg_footer()
)

Arguments

x A pointblank agent or an agent x-list. The x-list object can be created with the `get_agent_x_list()` function. It is recommended that the `i = NULL` and `generate_report = TRUE` so that the agent report is available within the email preview.

msg_header, msg_body, msg_footer
  Content for the header, body, and footer components of the HTML email message.

Value

A `blastula` `email_message` object.
Function ID

3-2

See Also

Other Emailing: email_blast(), stock_msg_body(), stock_msg_footer()

Examples

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

# Create an `action_levels()` list
# with absolute values for the `warn`, and `notify` states (with thresholds of 1 and 2 'fail' units)
al <-
  action_levels(
    warn_at = 1,
    notify_at = 2
  )

# In a workflow that involves an `agent` object, we can set up a series of `end_fns` and have report emailing with `email_blast()` but, first, we can look at the email message object beforehand by using the `email_preview()` function on an `agent` object
email_object <-
  create_agent(
    tbl = tbl,
    actions = al
  ) %>%
  col_vals_gt(vars(a), 5) %>%
  col_vals_lt(vars(b), 5) %>%
  interrogate() %>%
  email_preview()

# The `email_preview()` function can also be used on an agent x-list to get the same email message object
email_object <-
  create_agent(
    tbl = tbl,
    actions = al
  )
```

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
- `values`: the values used in the validation step, where applicable; for a `conjointly()` validation step, this is a listing of all sub-validations
- `precon`: indicates whether any there are any preconditions to apply before interrogation and, if so, the number of statements used
- `active`: a logical value that indicates whether a validation step is set to 'active' during an interrogation
- `eval`: a character value that denotes the result of each validation step functions' evaluation during 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`, which is the default) then a `gt` table will be displayed with the same information.
Usage

get_agent_report(
  agent,
  arrange_by = c("i", "severity"),
  keep = c("all", "fail_states"),
  display_table = TRUE,
  ...
)

Arguments

agent An agent object of class ptblank_agent.
arrange_by A choice to arrange the report table rows by the validation step number ("i", the default), or, to arrange in descending order by severity of the failure state (with "severity").
keep An option to keep "all" of the report’s table rows (the default), or, keep only those rows that reflect one or more "fail_states".
display_table Should a display table be generated? If TRUE (the default), 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

5-1

See Also

Other Post-interrogation: all_passed(), get_agent_x_list(), get_data_extracts(), get_sundered_data()

Examples

# Create a simple table with a column of numerical values
tbl <-
  dplyr::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

get_agent_report(agent, arrange_by = c("i", "severity"), keep = c("all", "fail_states"), display_table = TRUE, ...)

)
```r
# agent by using `get_agent_report()
agent %>%
  get_agent_report(display_table = FALSE)
```

---

**get_agent_x_list**  
*Get the agent’s x-list*

---

**Description**

The agent’s **x-list** is a record of information that the agent possesses at any given time. The **x-list** will contain the most complete information after an interrogation has taken place (before then, the data largely reflects the validation plan). The **x-list** can be constrained to a particular validation step (by supplying the step number to the `i` argument), or, we can get the information for all validation steps by leaving `i` unspecified. The **x-list** is indeed an R list object that contains a veritable cornucopia of information.

**Usage**

```r
get_agent_x_list(agent, i = NULL)
```

**Arguments**

- `agent`: An agent object of class `ptblank_agent`.
- `i`: The validation step number, which is assigned to each validation step in the order of invocation. If `NULL` (the default), the **x-list** will provide information for all validation steps. If a valid step number is provided then **x-list** will have information pertaining only to that step.

**Details**

For an **x-list** obtained with `i` specified for a validation step, the following components are available:

- `time`: the time at which the validation may have been performed (POSIXct [0 or 1])
- `name`: the (optional) name given to the validation (chr [1])
- `tbl_name`: the name of the table object, if available (chr [1])
- `tbl_src`: the type of table used in the validation (chr [1])
- `tbl_src_details`: if the table is a database table, this provides further details for the DB table (chr [1])
- `tbl`: the table object itself
- `col_names`: the table’s column names (chr [ncol(tbl)])
- `col_types`: the table’s column types (chr [ncol(tbl)])
- `i`: the table’s column types (chr [1])
- `type`: the type of validation, value is validation step function name (chr [1])
- `columns`: the columns specified for the validation step function (chr [variable length])
• values: the values specified for the validation step function (mixed types [variable length])
• briefs: the brief for the validation step in the specified reporting_lang (chr [1])
• eval_error, eval_warning: indicates whether the evaluation of the step function, during interrogation, resulted in an error or a warning (lgl [1])
• capture_stack: a list of captured errors or warnings during step-function evaluation at interrogation time (list [1])
• n: the number of test units for the validation step (num [1])
• n_passed, n_failed: the number of passing and failing test units for the validation step (num [1])
• f_passed: the fraction of passing test units for the validation step, n_passed / n (num [1])
• f_failed: the fraction of failing test units for the validation step, n_failed / n (num [1])
• warn, stop, notify: a logical value indicating whether the level of failing test units caused the corresponding conditions to be entered (lgl [1])
• reporting_lang: the two-letter language code that indicates which language should be used for all briefs, the agent report, and the reporting generated by the scan_data() function (chr [1])

If i is unspecified (i.e., not constrained to a specific validation step) then certain length-one components in the x-list will be expanded to the total number of validation steps (these are: i, type, columns, values, briefs, eval_error, eval_warning, capture_stack, n, n_passed, n_failed, f_passed, f_failed, warn, stop, and notify). The x-list will also have additional components when i is NULL, which are:

• report_object: a gt table object, which is also presented as the default print method for a ptblank_agent
• email_object: a blastula email_message object with a default set of components
• report_html: the HTML source for the report_object, provided as a length-one character vector
• report_html_small: the HTML source for a narrower, more condensed version of report_object, provided as a length-one character vector; The HTML has inlined styles, making it more suitable for email message bodies

Value
A list object.

Function ID
5-2

See Also
Other Post-interrogation: all_passed(), get_agent_report(), get_data_extracts(), get_sundered_data()
Examples

# Create a simple data frame with
# a column of numerical values
tbl <- dplyr::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
  )

# Create an agent (giving it the
# `tbl` and the `al` objects),
# supply two validation step
# functions, then interrogate
agent <-
  create_agent(
    tbl = tbl,
    actions = al
  ) %>%
  col_vals_gt(vars(a), 7) %>%
  col_is_numeric(vars(a)) %>%
  interrogate()

# Get the agent x-list
x <- get_agent_x_list(agent)

# Print the x-list object `x`
x

# Get the `f_passed` component
# of the x-list
x$f_passed

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).

Usage

get_data_extracts(agent, i = NULL)
Arguments

agent An agent object of class `ptblank_agent`. It should have had `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. If `NULL` (the default), all data extract tables will be provided in a list object.

Value

A list of tables if `i` is not provided, or, a standalone table if `i` is given.

Function ID

5-3

See Also

Other Post-interrogation: `all_passed()`, `get_agent_report()`, `get_agent_x_list()`, `get_sundered_data()`

Examples

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

# Create 2 simple validation steps
# that test whether values within
# column `a`
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)
```
get_sundered_data

Sunder the data, splitting it into 'pass' and 'fail' pieces

Description

Validation of the data is one thing but, sometimes, you want to use the best part of the input dataset for something else. The `get_sundered_data()` function works with an agent object that has intel (i.e., post `interrogate()`) and gets either the 'pass' data piece (rows with no failing units across all row-based validation step functions), or, the 'fail' data piece (rows with at least one failing unit across the same series of validations). There are some caveats, only those validation steps with no preconditions are considered. And, the validation steps used for this must, again, be of the row-based variety (e.g., the `col_vals_*()` functions, and `conjointly()`).

Usage

```r
get_sundered_data(agent, type = "pass", id_cols = NULL)
```

Arguments

- `agent`: An agent object of class `ptblank_agent`. It should have had `interrogate()` called on it, such that the validation steps were actually carried out.
- `type`: The desired piece of data resulting from the splitting. Options are "pass" (the default) and "fail".
- `id_cols`: An optional specification of one or more identifying columns. When taken together, we can count on this single column or grouping of columns to distinguish rows.

Value

A list of table objects if `type` is NULL, or, a table object piece if a type is given.

Function ID

5-4

See Also

Other Post-interrogation: `all_passed()`, `get_agent_report()`, `get_agent_x_list()`, `get_data_extracts()`

Examples

```r
# Create a series of three validation
# steps focus on test row values for
# the 'small_table' tibble object;
# `interrogate()` immediately
agent <-
  create_agent(tbl = small_table) %>%
col_vals_gt(vars(d), 100) %>%
get_sundered_data(agent, type = "pass", id_cols = NULL)
```
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 \( \text{sample\_limit} \) 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 \( \text{sample\_frac} \) option.

Value
A `ptblank_agent` object.

Function ID
4-1

Examples

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

# Validate that values in column `a` from `tbl` 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()
```

---

**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
```r
rows_distinct(
  x,
  columns = NULL,
  preconditions = NULL,
  actions = NULL,
  brief = NULL,
  active = TRUE
)
```

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.
- `active` A logical value indicating whether the validation step should be active. If the step function is working with an agent, `FALSE` will make the validation step inactive (still reporting its presence and keeping indexes for the steps unchanged). If the step function will be operating directly on data, then any step with `active = FALSE` will simply pass the data through with no validation whatsoever. The default for this is `TRUE`.

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 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**

2-15

**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_schema_match()`, `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
# Create a simple table with three columns of numerical values
tbl <-
dplyr::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 <-
```

```
rows_not_duplicated

```
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,
  active = TRUE
)

**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.
- **active**
  A logical value indicating whether the validation step should be active. If the step function is working with an agent, `FALSE` will make the validation step inactive (still reporting its presence and keeping indexes for the steps unchanged).
If the step function will be operating directly on data, then any step with `active = FALSE` will simply pass the data through with no validation whatsoever. The default for this is `TRUE`.

**Value**

A `ptblank_agent` object.

---

**Description**

Generates an HTML report that scours the input table data. Before calling up an `agent` to validate the data, it's a good idea to understand the data with some level of precision. Make this the initial step of a well-balanced *data quality reporting* workflow. The reporting output contains several sections to make everything more digestible, and these are:

- **Overview** Table dimensions, duplicate row count, column types, and reproducibility information
- **Variables** A summary for each table variable and further statistics and summaries depending on the variable type
- **Interactions** A matrix plot that shows interactions between variables
- **Correlations** A set of correlation matrix plots for numerical variables
- **Missing Values** A summary figure that shows the degree of missingness across variables
- **Sample** A table that provides the head and tail rows of the dataset

The output HTML report is viewable in the RStudio Viewer and can also be integrated in R Markdown HTML reports. If you need the output HTML as a string, it’s possible to get that by using `as.character()` (e.g., `scan_data(tbl = iris) %>% as.character()`). The resulting HTML string is a complete HTML document where Bootstrap and jQuery are embedded within.

**Usage**

```r
scan_data(
  tbl,
  sections = c("overview", "variables", "interactions", "correlations", "missing", "sample"),
  navbar = TRUE,
  reporting_lang = NULL
)
```
Arguments

- **tbl**: The input table. This can be a data frame, a tibble, or a tbl_dbi object.
- **sections**: The sections to include in the finalized Table Scan report. A character vector with section names is required here. The sections in their default order are: "overview", "variables", "interactions", "correlations", "missing", and "sample". This vector can be comprised of less elements and the order can be changed to suit the desired layout of the report.
- **navbar**: Should there be a navigation bar anchored to the top of the report page? By default this is `TRUE`.
- **reporting_lang**: The language to use for label text in the report. By default, `NULL` will create English ("en") text. Other options include French ("fr"), German ("de"), Italian ("it"), and Spanish ("es").

Function ID

1-1

See Also

Other Planning and Prep: `action_levels()`, `col_schema()`, `create_agent()`

Examples

```r
# Get an HTML report that describes all of
# the data in the 'dplyr::storms' dataset
# scan_data(tbl = dplyr::storms)
```

small_table

A small table that is 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

6-1

See Also

Other Datasets: `small_table_sqlite()`

Examples

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

---

**small_table_sqlite**  A SQLite version of the small_table dataset

Description

The `small_table_sqlite()` function creates a SQLite, tbl_dbi version of the small_table dataset. A requirement is the availability of the DBI and RSQLite packages. These packages can be installed by using `install.packages("DBI")` and `install.packages("RSQLite")`.

Usage

`small_table_sqlite()`

Function ID

6-2
stock_msg_body

See Also

Other Datasets: small_table

Examples

# Use `small_table_sqlite()` to
# create a SQLite version of the
# `small_table` table
#
# small_table_sqlite <- small_table_sqlite()

stock_msg_body

Provide simple email message body components: body

Description

The `stock_msg_body()` function simply provides some stock text for an email message sent via `email_blast()` or previewed through `email_preview()`.

Usage

stock_msg_body()

Value

Text suitable for the msg_body arguments of `email_blast()` and `email_preview()`.

Function ID

3-3

See Also

Other Emailing: `email_blast()`, `email_preview()`, `stock_msg_footer()`
stock_msg_footer

stock_msg_footer  Provide simple email message body components: footer

Description
The stock_msg_footer() functions simply provide some stock text for an email message sent via email_blast() or previewed through email_preview().

Usage
stock_msg_footer()

Value
Text suitable for the msg_footer argument of email_blast() and email_preview().

Function ID
3-4

See Also
Other Emailing: email_blast(), email_preview(), stock_msg_body()
Index

*Topic **datasets**
  - small_table, 79

**action_levels**, 3, 21, 61, 79
**action_levels()**, 6, 8, 10–13, 15, 17, 19, 22, 23, 25, 27, 28, 30, 31, 33, 35, 36, 38, 40, 41, 43–46, 48, 49, 51, 53, 54, 56, 58–61, 75–77
**all_passed**, 5, 67, 69, 71, 72
**all_passed()**, 60, 73

**blastula::creds**, 62
**blastula::creds_anonymous**, 62
**blastula::creds_file**, 62
**blastula::creds_key**, 62

**col_exists**, 6, 9, 10, 12, 14, 16, 18, 19, 23, 26, 29, 31, 34, 36, 39, 41, 44, 47, 49, 52, 54, 57, 59, 76
**col_is_character**, 7, 7, 10, 12, 14, 16, 18, 19, 23, 26, 29, 31, 34, 36, 39, 41, 44, 47, 49, 52, 54, 57, 59, 76
**col_is_date**, 7, 9, 9, 12, 14, 16, 18, 19, 23, 26, 29, 31, 34, 36, 39, 41, 44, 47, 49, 52, 54, 57, 59, 76
**col_is_factor**, 7, 9, 10, 11, 14, 16, 18, 19, 23, 26, 29, 31, 34, 36, 39, 41, 44, 47, 49, 52, 54, 57, 59, 76
**col_is_integer**, 7, 9, 10, 12, 13, 16, 18, 19, 23, 26, 29, 31, 34, 36, 39, 41, 44, 47, 49, 52, 54, 57, 59, 76
**col_is_logical**, 7, 9, 10, 12, 14, 15, 18, 19, 23, 26, 29, 31, 34, 36, 39, 41, 44, 47, 49, 52, 54, 57, 59, 76
**col_is_numeric**, 7, 9, 10, 12, 14, 16, 19, 23, 26, 29, 31, 34, 36, 39, 41, 44, 47, 49, 52, 54, 57, 59, 76
**col_is_posix**, 7, 9, 10, 12, 14, 16, 18, 19, 23, 26, 29, 31, 34, 36, 39, 41, 44, 47, 49, 52, 54, 57, 59, 76

**col_schema**, 4, 20, 61, 79
**col_schema()**, 22
**col_schema_match**, 7, 9, 10, 12, 14, 16, 18, 19, 22, 26, 29, 31, 34, 36, 39, 41, 44, 47, 49, 52, 54, 57, 59, 76
**col_schema_match()**, 20
**col_vals_between**, 7, 9, 10, 12, 14, 16, 18, 19, 23, 24, 29, 31, 34, 36, 39, 41, 44, 47, 49, 52, 54, 57, 59, 76
**col_vals_between()**, 44
**col_vals_equal**, 7, 9, 10, 12, 14, 16, 18, 19, 23, 26, 27, 31, 34, 36, 39, 41, 44, 47, 49, 52, 54, 57, 59, 76
**col_vals_equal()**, 47
**col_vals_gt**, 7, 9, 10, 12, 14, 16, 18, 19, 23, 26, 29, 34, 36, 39, 41, 44, 47, 49, 52, 54, 57, 59, 76
**col_vals_gt()**, 24, 34, 42
**col_vals_gte**, 7, 9, 10, 12, 14, 16, 18, 19, 23, 26, 29, 31, 34, 36, 39, 41, 44, 47, 49, 52, 54, 57, 59, 76
**col_vals_gte()**, 24, 31, 42
**col_vals_in_set**, 7, 9, 10, 12, 14, 16, 18, 19, 23, 26, 29, 31, 34, 36, 39, 41, 44, 47, 49, 52, 54, 57, 59, 76
**col_vals_in_set()**, 49
**col_vals_lt**, 7, 9, 10, 12, 14, 16, 18, 19, 23, 26, 29, 31, 34, 36, 37, 41, 44, 47, 49, 52, 54, 57, 59, 76
**col_vals_lt()**, 24, 41, 42
**col_vals_lte**, 7, 9, 10, 12, 14, 16, 18, 19, 23, 26, 29, 31, 34, 36, 39, 41, 44, 47, 49, 52, 54, 57, 59, 76
**col_vals_lte()**, 24, 39, 42
**col_vals_not_between**, 7, 9, 10, 12, 14, 16, 18, 19, 23, 26, 29, 31, 34, 36, 39, 41, 42, 47, 49, 52, 54, 57, 59, 76
**col_vals_not_between()**, 26
**col_vals_not_equal**, 7, 9, 10, 12, 14, 16, 18,