# Package 'pointblank'

November 20, 2020

**Type** Package

**Version** 0.6.0

**Title** Validation of Local and Remote Data Tables

**Description** Validate data in data frames, 'tibble' objects, 'Spark'
'DataFrames', and database tables (e.g., 'PostgreSQL' and 'MySQL').
Validation pipelines can be made using easily-readable, consecutive
validation steps. Upon execution of the validation plan, several reporting
options are available. User-defined thresholds for failure rates allow for
the determination of appropriate reporting actions. Many other workflows are
available including an information management workflow, where the aim is to
record, collect, and generate useful information on data tables.

**License** MIT + file LICENSE

**URL** <https://github.com/rich-iannone/pointblank>

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

**Encoding** UTF-8

**LazyData** true

**ByteCompile** true

**RoxygenNote** 7.1.1

**Depends** R (>= 3.5.0)

**Imports** base64enc (>= 0.1-3), blastula (>= 0.3.1), cli (>= 2.0.2), DBI
(>= 1.1.0), digest (>= 0.6.25), dplyr (>= 1.0.0), dbplyr (>=
1.4.4), fs (>= 1.4.1), ggforce (>= 0.3.1), ggplot2 (>= 3.3.0),
glue (>= 1.3.2), gt (>= 0.2.1), htmltools (>= 0.4.0), log4r (>=
0.3.2), knitr (>= 1.28), rlang (>= 0.4.6), magrittr, scales (>=
1.1.1), testthat (>= 2.3.2), tibble (>= 3.0.0), tidyselect (>=
1.1.0), yaml (>= 2.2.1)

**Suggests** covr, duckdb, lubridate, RSQLite, RMariaDB, RMySQL,
RPostgres, readr, rmarkdown, sparklyr, dittodb, odbc

**NeedsCompilation** no

**Author** Richard Iannone [aut, cre] (<https://orcid.org/0000-0003-3925-190X>),
Mauricio Vargas [aut] (<https://orcid.org/0000-0003-1017-7574>)

**Maintainer** Richard Iannone `<riannone@me.com>`

**Repository** CRAN

**Date/Publication** 2020-11-20 07:30:10 UTC

# R **topics documented:**

---

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 functions oper-
ating directly on data (e.g., `mtcars %>% col_vals_lt("mpg",35)`). There are two helper func-
tions that are convenient when using validation functions directly on data (the `agent`-less work-
flow): `warn_on_fail()` and `stop_on_fail()`. These helpers either warn or stop (default fail-
ure threshold for each is set to 1), and, they do so with informative warning or error messages.
The `stop_on_fail()` helper is applied by default when using validation functions directly on data
(more information on this is provided in *Details*).

## Usage

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

warn_on_fail(warn_at = 1)

stop_on_fail(stop_at = 1)
```

## Arguments

warn_at, stop_at, notify_at

> The threshold number or fraction of test units that can provide a *fail* result be-
> fore 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 test units are found to *fail*, then the designated fail-
> ure 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 test
> 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 pro-
> vided 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 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. For example using `small_table %>% col_is_integer("date")`
will provide a detailed stop message by default, indicating the reason for the failure. If you were to
supply the `fns` for `stop` or `warn` manually then the stock functions would be overridden. Further-
more, if `actions` is NULL in this workflow (the default), **pointblank** will use a `stop_at` value of `1`
(providing a detailed, context-specific error message if there are any *fail* units). We can absolutely
suppress this automatic stopping behavior by at each validation step by setting `active = FALSE`. In
this interactive data case, there is no stock function given for `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 suffi-
cient). Instead, using the `end_fns` argument is a better choice since that scheme provides useful

data on the entire interrogation, allowing for finer control on side effects and reducing potential for duplicating any side effects.

**Function ID**

1-5

**See Also**

Other Planning and Prep: col_schema(), create_agent(), create_informant(), db_tbl(), scan_data(), validate_rmd()

**Examples**

```
# 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 = 0.5
  )

# Use the included `small_table` dataset
# for the validation example
small_table

# Validate that values in column
# `a` are always greater than `2` and
# apply the list of action levels (`al`)
agent <-
  create_agent(tbl = small_table) %>%
  col_vals_gt(vars(a), 2, actions = al) %>%
  interrogate()

# The report from the agent will show
# that the `warn` state has been entered
# for the first and only validation step;
# Let's look at the *tibble* version of the
# agent report (accessible through the use
# of the `get_agent_report()` function)
agent %>%
  get_agent_report(display_table = FALSE)

# In the context of using validation
# 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) and the `small_table`
# data will be returned
```

```
suppressWarnings(
  small_table %>%
    col_vals_gt(vars(a), 2, 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

7-4

## See Also

Other Post-interrogation: [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()
```

```
# 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 function, the `expect_col_exists()` expectation function, and the
`test_col_exists()` test function all check whether one or more columns exist in the target table.
The only requirement is specification of the column names. The validation function can be used
directly on a data table or with an *agent* object (technically, a `ptblank_agent` object) whereas the
expectation and test functions can only be used with a data table. The types of data tables that can be
used include data frames, tibbles, database tables (`tbl_dbi`), and Spark DataFrames (`tbl_spark`).
Each validation step or expectation will operate over a single test unit, which is whether the column
exists or not.

## Usage

```
col_exists(
  x,
  columns,
  actions = NULL,
  step_id = NULL,
  label = NULL,
  brief = NULL,
  active = TRUE
)

expect_col_exists(object, columns, threshold = 1)

test_col_exists(object, columns, threshold = 1)
```

## Arguments

| | |
|---|---|
| x | A data frame, tibble (`tbl_df` or `tbl_dbi`), Spark DataFrame (`tbl_spark`), or, an agent object of class ptblank_agent that is created with [create_agent()](create_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()](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()](action_levels()) helper function. |

step_id        One or more optional identifiers for the single or multiple validation steps gen-
               erated from calling a validation function. The use of step IDs serves to distin-
               guish validation steps from each other and provide an opportunity for supplying
               a more meaningful label compared to the step index. By default this is NULL,
               and **pointblank** will automatically generate the step ID value (based on the step
               index) in this case. One or more values can be provided, and the exact number
               of ID values should (1) match the number of validation steps that the validation
               function call will produce (influenced by the number of columns provided), (2)
               be an ID string not used in any previous validation step, and (3) be a vector with
               unique values.

label          An optional label for the validation step. This label appears in the agent report
               and for the best appearance it should be kept short.

brief          An optional, text-based description for the validation step. If nothing is provided
               here then an *autobrief* is generated by the agent, using the language provided
               in [create_agent()](#)'s lang argument (which defaults to "en" or English). The
               *autobrief* incorporates details of the validation step so it's often the preferred
               option in most cases (where a label might be better suited to succinctly describe
               the validation).

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.

object         A data frame, tibble (tbl_df or tbl_dbi), or Spark DataFrame (tbl_spark)
               that serves as the target table for the expectation function or the test function.

threshold      A simple failure threshold value for use with the expectation (expect_) and
               the test (test_) function variants. By default, this is set to 1 meaning that any
               single unit of failure in data validation results in an overall test failure. Whole
               numbers beyond 1 indicate that any failing units up to that absolute threshold
               value will result in a succeeding **testthat** test or evaluate to TRUE. Likewise,
               fractional values (between 0 and 1) act as a proportional failure threshold, where
               0.15 means that 15 percent of failing test units results in an overall test failure.

### 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 vali-
dation 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 of 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

For the validation function, the return value is either a ptblank_agent object or a table object (depending on whether an agent object or a table was passed to x). The expectation function invisibly returns its input but, in the context of testing data, the function is called primarily for its potential side-effects (e.g., signaling failure). The test function returns a logical value.

### Function ID

2-23

### See Also

Other validation 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_expr(), 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

```
# For all examples here, we'll use
# a simple table with two columns:
# `a` and `b`
tbl <-
  dplyr::tibble(
    a = c(5, 7, 6, 5, 8, 7),
    b = c(7, 1, 0, 0, 0, 3)
  )

# A: Using an `agent` with validation
#    functions and then `interrogate()`

# Validate that columns `a` and `b`
# exist in the `tbl` table; this
# makes two distinct validation
# steps since two columns were
# provided to `vars()`
agent <-
  create_agent(tbl) %>%
  col_exists(vars(a, b)) %>%
  interrogate()

# Determine if this validation
```

```
# had no failing test units (1)
all_passed(agent)

# Calling `agent` in the console
# prints the agent's report; but we
# can get a `gt_tbl` object directly
# with `get_agent_report(agent)`

# B: Using the validation function
#     directly on the data (no `agent`)

# This way of using validation functions
# acts as a data filter: data is passed
# through but should `stop()` if there
# is a single test unit failing; the
# behavior of side effects can be
# customized with the `actions` option
tbl %>% col_exists(vars(a, b))

# C: Using the expectation function

# With the `expect_*()` form, we need
# to be more exacting and provide one
# column at a time; this is primarily
# used in testthat tests
expect_col_exists(tbl, vars(a))
expect_col_exists(tbl, vars(b))

# D: Using the test function

# With the `test_*()` form, we should
# get a single logical value returned
# to us (even if there are multiple
# columns tested, as is the case below)
tbl %>% test_col_exists(vars(a, b))
```

---

col_is_character                *Do the columns contain character/string data?*

---

## Description

The col_is_character() validation function, the expect_col_is_character() expectation func-
tion, and the test_col_is_character() test function all check whether one or more columns in
a table 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. The validation function can be used directly on
a data table or with an *agent* object (technically, a ptblank_agent object) whereas the expectation
and test functions can only be used with a data table. The types of data tables that can be used
include data frames, tibbles, database tables (tbl_dbi), and Spark DataFrames (tbl_spark). Each
validation step or expectation will operate over a single test unit, which is whether the column is a
character-type column or not.

## Usage

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

expect_col_is_character(object, columns, threshold = 1)

test_col_is_character(object, columns, threshold = 1)
```

## Arguments

| | |
|---|---|
| x | A data frame, tibble (tbl_df or tbl_dbi), Spark DataFrame (tbl_spark), or, an agent object of class ptblank_agent that is created with [create_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. |
| step_id | One or more optional identifiers for the single or multiple validation steps generated from calling a validation function. The use of step IDs serves to distinguish validation steps from each other and provide an opportunity for supplying a more meaningful label compared to the step index. By default this is NULL, and **pointblank** will automatically generate the step ID value (based on the step index) in this case. One or more values can be provided, and the exact number of ID values should (1) match the number of validation steps that the validation function call will produce (influenced by the number of columns provided), (2) be an ID string not used in any previous validation step, and (3) be a vector with unique values. |
| label | An optional label for the validation step. This label appears in the agent report and for the best appearance it should be kept short. |
| brief | An optional, text-based description for the validation step. If nothing is provided here then an *autobrief* is generated by the agent, using the language provided in [create_agent()]()'s lang argument (which defaults to "en" or English). The *autobrief* incorporates details of the validation step so it's often the preferred option in most cases (where a label might be better suited to succinctly describe the validation). |
| 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.

object          A data frame, tibble (tbl_df or tbl_dbi), or Spark DataFrame (tbl_spark) that serves as the target table for the expectation function or the test function.

threshold       A simple failure threshold value for use with the expectation (expect_) and the test (test_) function variants. By default, this is set to 1 meaning that any single unit of failure in data validation results in an overall test failure. Whole numbers beyond 1 indicate that any failing units up to that absolute threshold value will result in a succeeding **testthat** test or evaluate to TRUE. Likewise, fractional values (between 0 and 1) act as a proportional failure threshold, where 0.15 means that 15 percent of failing test units results in an overall test failure.

### 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 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 of 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

For the validation function, the return value is either a ptblank_agent object or a table object (depending on whether an agent object or a table was passed to x). The expectation function invisibly returns its input but, in the context of testing data, the function is called primarily for its potential side-effects (e.g., signaling failure). The test function returns a logical value.

### Function ID

2-16

### See Also

Other validation 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_expr()](#), [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

```
# For all examples here, we'll use
# a simple table with a numeric column
# (`a`) and a character column (`b`)
tbl <-
  dplyr::tibble(
    a = c(5, 7, 6, 5, 8, 7),
    b = LETTERS[1:6]
  )

# A: Using an `agent` with validation
#    functions and then `interrogate()`

# Validate that column `b` has the
# `character` class
agent <-
  create_agent(tbl) %>%
  col_is_character(vars(b)) %>%
  interrogate()

# Determine if this validation
# had no failing test units (1)
all_passed(agent)

# Calling `agent` in the console
# prints the agent's report; but we
# can get a `gt_tbl` object directly
# with `get_agent_report(agent)`

# B: Using the validation function
#    directly on the data (no `agent`)

# This way of using validation functions
# acts as a data filter: data is passed
# through but should `stop()` if there
# is a single test unit failing; the
# behavior of side effects can be
# customized with the `actions` option
tbl %>% col_is_character(vars(b))

# C: Using the expectation function

# With the `expect_*()` form, we would
# typically perform one validation at a
# time; this is primarily used in
# testthat tests
expect_col_is_character(tbl, vars(b))

# D: Using the test function
```

```
# With the `test_*()` form, we should
# get a single logical value returned
# to us
tbl %>% test_col_is_character(vars(b))
```

---

col_is_date                        *Do the columns contain R* Date *objects?*

---

## Description

The col_is_date() validation function, the expect_col_is_date() expectation function, and
the test_col_is_date() test function all check whether one or more columns in a table 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. The validation function can be used directly on a data table
or with an *agent* object (technically, a ptblank_agent object) whereas the expectation and test
functions can only be used with a data table. The types of data tables that can be used include data
frames, tibbles, database tables (tbl_dbi), and Spark DataFrames (tbl_spark). Each validation
step or expectation will operate over a single test unit, which is whether the column is a Date-type
column or not.

## Usage

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

expect_col_is_date(object, columns, threshold = 1)

test_col_is_date(object, columns, threshold = 1)
```

## Arguments

| | |
|---|---|
| x | A data frame, tibble (tbl_df or tbl_dbi), Spark DataFrame (tbl_spark), or, an agent object of class ptblank_agent that is created with create_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. |

step_id         One or more optional identifiers for the single or multiple validation steps gen-
                erated from calling a validation function. The use of step IDs serves to distin-
                guish validation steps from each other and provide an opportunity for supplying
                a more meaningful label compared to the step index. By default this is NULL,
                and **pointblank** will automatically generate the step ID value (based on the step
                index) in this case. One or more values can be provided, and the exact number
                of ID values should (1) match the number of validation steps that the validation
                function call will produce (influenced by the number of columns provided), (2)
                be an ID string not used in any previous validation step, and (3) be a vector with
                unique values.

label           An optional label for the validation step. This label appears in the agent report
                and for the best appearance it should be kept short.

brief           An optional, text-based description for the validation step. If nothing is provided
                here then an *autobrief* is generated by the agent, using the language provided
                in [create_agent()](#)'s lang argument (which defaults to "en" or English). The
                *autobrief* incorporates details of the validation step so it's often the preferred
                option in most cases (where a label might be better suited to succinctly describe
                the validation).

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.

object          A data frame, tibble (tbl_df or tbl_dbi), or Spark DataFrame (tbl_spark)
                that serves as the target table for the expectation function or the test function.

threshold       A simple failure threshold value for use with the expectation (expect_) and
                the test (test_) function variants. By default, this is set to 1 meaning that any
                single unit of failure in data validation results in an overall test failure. Whole
                numbers beyond 1 indicate that any failing units up to that absolute threshold
                value will result in a succeeding **testthat** test or evaluate to TRUE. Likewise,
                fractional values (between 0 and 1) act as a proportional failure threshold, where
                0.15 means that 15 percent of failing test units results in an overall test failure.

### 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 vali-
dation 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 of 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

For the validation function, the return value is either a `ptblank_agent` object or a table object (depending on whether an agent object or a table was passed to x). The expectation function invisibly returns its input but, in the context of testing data, the function is called primarily for its potential side-effects (e.g., signaling failure). The test function returns a logical value.

### Function ID

2-20

### See Also

Other validation 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_expr()`, `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

```
# The `small_table` dataset in the
# package has a `date` column; the
# following examples will validate
# that that column is of the `Date`
# class

# A: Using an `agent` with validation
#    functions and then `interrogate()`

# Validate that the column `date` has
# the `Date` class
agent <-
  create_agent(small_table) %>%
  col_is_date(vars(date)) %>%
  interrogate()

# Determine if this validation
# had no failing test units (1)
all_passed(agent)

# Calling `agent` in the console
# prints the agent's report; but we
# can get a `gt_tbl` object directly
```

```
# with `get_agent_report(agent)`

# B: Using the validation function
#    directly on the data (no `agent`)

# This way of using validation functions
# acts as a data filter: data is passed
# through but should `stop()` if there
# is a single test unit failing; the
# behavior of side effects can be
# customized with the `actions` option
small_table %>%
  col_is_date(vars(date)) %>%
  dplyr::slice(1:5)

# C: Using the expectation function

# With the `expect_*()` form, we would
# typically perform one validation at a
# time; this is primarily used in
# testthat tests
expect_col_is_date(
  small_table, vars(date)
)

# D: Using the test function

# With the `test_*()` form, we should
# get a single logical value returned
# to us
small_table %>%
  test_col_is_date(vars(date))
```

---

col_is_factor                   *Do the columns contain R* factor *objects?*

---

## Description

The col_is_factor() validation function, the expect_col_is_factor() expectation function, and the test_col_is_factor() test function all check whether one or more columns in a table 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. The validation function can be used directly on a data table or with an *agent* object (technically, a ptblank_agent object) whereas the expectation and test functions can only be used with a data table. The types of data tables that can be used include data frames, tibbles, database tables (tbl_dbi), and Spark DataFrames (tbl_spark). Each validation step or expectation 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,
  step_id = NULL,
  label = NULL,
  brief = NULL,
  active = TRUE
)

expect_col_is_factor(object, columns, threshold = 1)

test_col_is_factor(object, columns, threshold = 1)
```

**Arguments**

| | |
|---|---|
| x | A data frame, tibble (tbl_df or tbl_dbi), Spark DataFrame (tbl_spark), or, an agent object of class ptblank_agent that is created with [create_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. |
| step_id | One or more optional identifiers for the single or multiple validation steps generated from calling a validation function. The use of step IDs serves to distinguish validation steps from each other and provide an opportunity for supplying a more meaningful label compared to the step index. By default this is NULL, and **pointblank** will automatically generate the step ID value (based on the step index) in this case. One or more values can be provided, and the exact number of ID values should (1) match the number of validation steps that the validation function call will produce (influenced by the number of columns provided), (2) be an ID string not used in any previous validation step, and (3) be a vector with unique values. |
| label | An optional label for the validation step. This label appears in the agent report and for the best appearance it should be kept short. |
| brief | An optional, text-based description for the validation step. If nothing is provided here then an *autobrief* is generated by the agent, using the language provided in [create_agent()]()'s lang argument (which defaults to "en" or English). The *autobrief* incorporates details of the validation step so it's often the preferred option in most cases (where a label might be better suited to succinctly describe the validation). |
| 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.

object          A data frame, tibble (tbl_df or tbl_dbi), or Spark DataFrame (tbl_spark) that serves as the target table for the expectation function or the test function.

threshold       A simple failure threshold value for use with the expectation (expect_) and the test (test_) function variants. By default, this is set to 1 meaning that any single unit of failure in data validation results in an overall test failure. Whole numbers beyond 1 indicate that any failing units up to that absolute threshold value will result in a succeeding **testthat** test or evaluate to TRUE. Likewise, fractional values (between 0 and 1) act as a proportional failure threshold, where 0.15 means that 15 percent of failing test units results in an overall test failure.

### 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 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 of 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

For the validation function, the return value is either a ptblank_agent object or a table object (depending on whether an agent object or a table was passed to x). The expectation function invisibly returns its input but, in the context of testing data, the function is called primarily for its potential side-effects (e.g., signaling failure). The test function returns a logical value.

### Function ID

2-22

### See Also

Other validation 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_expr()](#), [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**

```
# Let's modify the `f` column in the
# `small_table` dataset so that the
# values are factors instead of having
# the `character` class; the following
# examples will validate that the `f`
# column was successfully mutated and
# now consists of factors
tbl <-
  small_table %>%
  dplyr::mutate(f = factor(f))

# A: Using an `agent` with validation
#    functions and then `interrogate()`

# Validate that the column `f` in the
# `tbl` object is of the `factor` class
agent <-
  create_agent(tbl) %>%
  col_is_factor(vars(f)) %>%
  interrogate()

# Determine if this validation
# had no failing test units (1)
all_passed(agent)

# Calling `agent` in the console
# prints the agent's report; but we
# can get a `gt_tbl` object directly
# with `get_agent_report(agent)`

# B: Using the validation function
#    directly on the data (no `agent`)

# This way of using validation functions
# acts as a data filter: data is passed
# through but should `stop()` if there
# is a single test unit failing; the
# behavior of side effects can be
# customized with the `actions` option
tbl %>%
  col_is_factor(vars(f)) %>%
  dplyr::slice(1:5)

# C: Using the expectation function

# With the `expect_*()` form, we would
# typically perform one validation at a
# time; this is primarily used in
```

```
# testthat tests
expect_col_is_factor(tbl, vars(f))

# D: Using the test function

# With the `test_*()` form, we should
# get a single logical value returned
# to us
tbl %>% test_col_is_factor(vars(f))
```

---

col_is_integer *Do the columns contain integer values?*

---

## Description

The `col_is_integer()` validation function, the `expect_col_is_integer()` expectation function, and the `test_col_is_integer()` test function all check whether one or more columns in a table 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. The validation function can be used directly on a data table or with an *agent* object (technically, a ptblank_agent object) whereas the expectation and test functions can only be used with a data table. The types of data tables that can be used include data frames, tibbles, database tables (tbl_dbi), and Spark DataFrames (tbl_spark). Each validation step or expectation will operate over a single test unit, which is whether the column is an integer-type column or not.

## Usage

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

expect_col_is_integer(object, columns, threshold = 1)

test_col_is_integer(object, columns, threshold = 1)
```

## Arguments

x           A data frame, tibble (`tbl_df` or `tbl_dbi`), Spark DataFrame (`tbl_spark`), or, an agent object of class ptblank_agent that is created with [create_agent()](create_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.

step_id          One or more optional identifiers for the single or multiple validation steps gen-
                 erated from calling a validation function. The use of step IDs serves to distin-
                 guish validation steps from each other and provide an opportunity for supplying
                 a more meaningful label compared to the step index. By default this is NULL,
                 and **pointblank** will automatically generate the step ID value (based on the step
                 index) in this case. One or more values can be provided, and the exact number
                 of ID values should (1) match the number of validation steps that the validation
                 function call will produce (influenced by the number of `columns` provided), (2)
                 be an ID string not used in any previous validation step, and (3) be a vector with
                 unique values.

label            An optional label for the validation step. This label appears in the agent report
                 and for the best appearance it should be kept short.

brief            An optional, text-based description for the validation step. If nothing is provided
                 here then an *autobrief* is generated by the agent, using the language provided
                 in `create_agent()`'s `lang` argument (which defaults to "en" or English). The
                 *autobrief* incorporates details of the validation step so it's often the preferred
                 option in most cases (where a `label` might be better suited to succinctly describe
                 the validation).

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.

object           A data frame, tibble (`tbl_df` or `tbl_dbi`), or Spark DataFrame (`tbl_spark`)
                 that serves as the target table for the expectation function or the test function.

threshold        A simple failure threshold value for use with the expectation (`expect_`) and
                 the test (`test_`) function variants. By default, this is set to 1 meaning that any
                 single unit of failure in data validation results in an overall test failure. Whole
                 numbers beyond 1 indicate that any failing units up to that absolute threshold
                 value will result in a succeeding **testthat** test or evaluate to TRUE. Likewise,
                 fractional values (between 0 and 1) act as a proportional failure threshold, where
                 0.15 means that 15 percent of failing test units results in an overall test failure.

## 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 vali-
dation 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 of 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

For the validation function, the return value is either a `ptblank_agent` object or a table object (depending on whether an agent object or a table was passed to x). The expectation function invisibly returns its input but, in the context of testing data, the function is called primarily for its potential side-effects (e.g., signaling failure). The test function returns a logical value.

## Function ID

2-18

## See Also

Other validation 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_expr()`, `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

```
# For all examples here, we'll use
# a simple table with a character
# column (`a`) and a integer column
# (`b`)
tbl <-
  dplyr::tibble(
    a = letters[1:6],
    b = 2:7
  )

# A: Using an `agent` with validation
#    functions and then `interrogate()`

# Validate that column `b` has the
# `integer` class
agent <-
  create_agent(tbl) %>%
  col_is_integer(vars(b)) %>%
  interrogate()
```

```
# Determine if this validation
# had no failing test units (1)
all_passed(agent)

# Calling `agent` in the console
# prints the agent's report; but we
# can get a `gt_tbl` object directly
# with `get_agent_report(agent)`

# B: Using the validation function
#    directly on the data (no `agent`)

# This way of using validation functions
# acts as a data filter: data is passed
# through but should `stop()` if there
# is a single test unit failing; the
# behavior of side effects can be
# customized with the `actions` option
tbl %>% col_is_integer(vars(b))

# C: Using the expectation function

# With the `expect_*()` form, we would
# typically perform one validation at a
# time; this is primarily used in
# testthat tests
expect_col_is_integer(tbl, vars(b))

# D: Using the test function

# With the `test_*()` form, we should
# get a single logical value returned
# to us
tbl %>% test_col_is_integer(vars(b))
```

---

col_is_logical               *Do the columns contain logical values?*

---

### Description

The `col_is_logical()` validation function, the `expect_col_is_logical()` expectation function, and the `test_col_is_logical()` test function all check whether one or more columns in a table 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. The validation function can be used directly on a data table or with an *agent* object (technically, a `ptblank_agent` object) whereas the expectation and test functions can only be used with a data table. The types of data tables that can be used include data frames, tibbles, database tables (`tbl_dbi`), and Spark DataFrames (`tbl_spark`). Each validation step or expectation will operate over a single test unit, which is whether the column is an logical-type column or not.

**Usage**

```
col_is_logical(
  x,
  columns,
  actions = NULL,
  step_id = NULL,
  label = NULL,
  brief = NULL,
  active = TRUE
)

expect_col_is_logical(object, columns, threshold = 1)

test_col_is_logical(object, columns, threshold = 1)
```

**Arguments**

| | |
|---|---|
| x | A data frame, tibble (tbl_df or tbl_dbi), Spark DataFrame (tbl_spark), or, an agent object of class ptblank_agent that is created with create_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. |
| step_id | One or more optional identifiers for the single or multiple validation steps generated from calling a validation function. The use of step IDs serves to distinguish validation steps from each other and provide an opportunity for supplying a more meaningful label compared to the step index. By default this is NULL, and **pointblank** will automatically generate the step ID value (based on the step index) in this case. One or more values can be provided, and the exact number of ID values should (1) match the number of validation steps that the validation function call will produce (influenced by the number of columns provided), (2) be an ID string not used in any previous validation step, and (3) be a vector with unique values. |
| label | An optional label for the validation step. This label appears in the agent report and for the best appearance it should be kept short. |
| brief | An optional, text-based description for the validation step. If nothing is provided here then an *autobrief* is generated by the agent, using the language provided in create_agent()'s lang argument (which defaults to "en" or English). The *autobrief* incorporates details of the validation step so it's often the preferred option in most cases (where a label might be better suited to succinctly describe the validation). |
| 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. |
| object    | A data frame, tibble (tbl_df or tbl_dbi), or Spark DataFrame (tbl_spark) that serves as the target table for the expectation function or the test function. |
| threshold | A simple failure threshold value for use with the expectation (expect_) and the test (test_) function variants. By default, this is set to 1 meaning that any single unit of failure in data validation results in an overall test failure. Whole numbers beyond 1 indicate that any failing units up to that absolute threshold value will result in a succeeding **testthat** test or evaluate to TRUE. Likewise, fractional values (between 0 and 1) act as a proportional failure threshold, where 0.15 means that 15 percent of failing test units results in an overall test failure. |

### 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 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 of 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

For the validation function, the return value is either a ptblank_agent object or a table object (depending on whether an agent object or a table was passed to x). The expectation function invisibly returns its input but, in the context of testing data, the function is called primarily for its potential side-effects (e.g., signaling failure). The test function returns a logical value.

### Function ID

2-19

### See Also

Other validation 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_expr()](#), [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

```
# The `small_table` dataset in the
# package has an `e` column which has
# logical values; the following examples
# will validate that that column is of
# the `logical` class

# A: Using an `agent` with validation
#     functions and then `interrogate()`

# Validate that the column `e` has the
# `logical` class
agent <-
  create_agent(small_table) %>%
  col_is_logical(vars(e)) %>%
  interrogate()

# Determine if this validation
# had no failing test units (1)
all_passed(agent)

# Calling `agent` in the console
# prints the agent's report; but we
# can get a `gt_tbl` object directly
# with `get_agent_report(agent)`

# B: Using the validation function
#     directly on the data (no `agent`)

# This way of using validation functions
# acts as a data filter: data is passed
# through but should `stop()` if there
# is a single test unit failing; the
# behavior of side effects can be
# customized with the `actions` option
small_table %>%
  col_is_logical(vars(e)) %>%
  dplyr::slice(1:5)

# C: Using the expectation function

# With the `expect_*()` form, we would
# typically perform one validation at a
# time; this is primarily used in
# testthat tests
expect_col_is_logical(
  small_table, vars(e)
)
```

```
# D: Using the test function

# With the `test_*()` form, we should
# get a single logical value returned
# to us
small_table %>%
  test_col_is_logical(vars(e))
```

col_is_numeric                 *Do the columns contain numeric values?*

### Description

The `col_is_numeric()` validation function, the `expect_col_is_numeric()` expectation function, and the `test_col_is_numeric()` test function all check whether one or more columns in a table 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. The validation function can be used directly on a data table or with an *agent* object (technically, a ptblank_agent object) whereas the expectation and test functions can only be used with a data table. The types of data tables that can be used include data frames, tibbles, database tables (tbl_dbi), and Spark DataFrames (tbl_spark). Each validation step or expectation will operate over a single test unit, which is whether the column is a numeric-type column or not.

### Usage

```
col_is_numeric(
  x,
  columns,
  actions = NULL,
  step_id = NULL,
  label = NULL,
  brief = NULL,
  active = TRUE
)

expect_col_is_numeric(object, columns, threshold = 1)

test_col_is_numeric(object, columns, threshold = 1)
```

### Arguments

| | |
|---|---|
| x | A data frame, tibble (`tbl_df` or `tbl_dbi`), Spark DataFrame (`tbl_spark`), or, an agent object of class ptblank_agent that is created with [create_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.

step_id        One or more optional identifiers for the single or multiple validation steps generated from calling a validation function. The use of step IDs serves to distinguish validation steps from each other and provide an opportunity for supplying a more meaningful label compared to the step index. By default this is NULL, and **pointblank** will automatically generate the step ID value (based on the step index) in this case. One or more values can be provided, and the exact number of ID values should (1) match the number of validation steps that the validation function call will produce (influenced by the number of `columns` provided), (2) be an ID string not used in any previous validation step, and (3) be a vector with unique values.

label          An optional label for the validation step. This label appears in the agent report and for the best appearance it should be kept short.

brief          An optional, text-based description for the validation step. If nothing is provided here then an *autobrief* is generated by the agent, using the language provided in `create_agent()`'s `lang` argument (which defaults to "en" or English). The *autobrief* incorporates details of the validation step so it's often the preferred option in most cases (where a `label` might be better suited to succinctly describe the validation).

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.

object         A data frame, tibble (`tbl_df` or `tbl_dbi`), or Spark DataFrame (`tbl_spark`) that serves as the target table for the expectation function or the test function.

threshold      A simple failure threshold value for use with the expectation (`expect_`) and the test (`test_`) function variants. By default, this is set to 1 meaning that any single unit of failure in data validation results in an overall test failure. Whole numbers beyond 1 indicate that any failing units up to that absolute threshold value will result in a succeeding **testthat** test or evaluate to TRUE. Likewise, fractional values (between 0 and 1) act as a proportional failure threshold, where 0.15 means that 15 percent of failing test units results in an overall test failure.

### 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 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 of 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

For the validation function, the return value is either a `ptblank_agent` object or a table object (depending on whether an agent object or a table was passed to x). The expectation function invisibly returns its input but, in the context of testing data, the function is called primarily for its potential side-effects (e.g., signaling failure). The test function returns a logical value.

## Function ID

2-17

## See Also

Other validation 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_expr()`, `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

```
# The `small_table` dataset in the
# package has a `d` column that is
# known to be numeric; the following
# examples will validate that that
# column is indeed of the `numeric`
# class

# A: Using an `agent` with validation
#    functions and then `interrogate()`

# Validate that the column `d` has
# the `numeric` class
agent <-
  create_agent(small_table) %>%
  col_is_numeric(vars(d)) %>%
  interrogate()

# Determine if this validation
# had no failing test units (1)
```

```
all_passed(agent)

# Calling `agent` in the console
# prints the agent's report; but we
# can get a `gt_tbl` object directly
# with `get_agent_report(agent)`

# B: Using the validation function
#     directly on the data (no `agent`)

# This way of using validation functions
# acts as a data filter: data is passed
# through but should `stop()` if there
# is a single test unit failing; the
# behavior of side effects can be
# customized with the `actions` option
small_table %>%
  col_is_numeric(vars(d)) %>%
  dplyr::slice(1:5)

# C: Using the expectation function

# With the `expect_*()` form, we would
# typically perform one validation at a
# time; this is primarily used in
# testthat tests
expect_col_is_numeric(
  small_table, vars(d)
)

# D: Using the test function

# With the `test_*()` form, we should
# get a single logical value returned
# to us
small_table %>%
  test_col_is_numeric(vars(d))
```

---

col_is_posix                        *Do the columns contain* POSIXct *dates?*

---

### Description

The col_is_posix() validation function, the expect_col_is_posix() expectation function, and the test_col_is_posix() test function all check whether one or more columns in a table 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. The validation function can be used directly on a data table or with an *agent* object (technically, a ptblank_agent object) whereas the expectation and test functions can only be used with a data table. The types of data tables that can be used

include data frames, tibbles, database tables (`tbl_dbi`), and Spark DataFrames (`tbl_spark`). Each validation step or expectation will operate over a single test unit, which is whether the column is a `POSIXct`-type column or not.

## Usage

```
col_is_posix(
  x,
  columns,
  actions = NULL,
  step_id = NULL,
  label = NULL,
  brief = NULL,
  active = TRUE
)

expect_col_is_posix(object, columns, threshold = 1)

test_col_is_posix(object, columns, threshold = 1)
```

## Arguments

| | |
|---|---|
| x | A data frame, tibble (`tbl_df` or `tbl_dbi`), Spark DataFrame (`tbl_spark`), or, an agent object of class `ptblank_agent` that is created with [`create_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. |
| step_id | One or more optional identifiers for the single or multiple validation steps generated from calling a validation function. The use of step IDs serves to distinguish validation steps from each other and provide an opportunity for supplying a more meaningful label compared to the step index. By default this is `NULL`, and **pointblank** will automatically generate the step ID value (based on the step index) in this case. One or more values can be provided, and the exact number of ID values should (1) match the number of validation steps that the validation function call will produce (influenced by the number of `columns` provided), (2) be an ID string not used in any previous validation step, and (3) be a vector with unique values. |
| label | An optional label for the validation step. This label appears in the agent report and for the best appearance it should be kept short. |
| brief | An optional, text-based description for the validation step. If nothing is provided here then an *autobrief* is generated by the agent, using the language provided in [`create_agent()`]()'s `lang` argument (which defaults to `"en"` or English). The *autobrief* incorporates details of the validation step so it's often the preferred option in most cases (where a `label` might be better suited to succinctly describe the validation). |

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.

object         A data frame, tibble (tbl_df or tbl_dbi), or Spark DataFrame (tbl_spark)
               that serves as the target table for the expectation function or the test function.

threshold      A simple failure threshold value for use with the expectation (expect_) and
               the test (test_) function variants. By default, this is set to 1 meaning that any
               single unit of failure in data validation results in an overall test failure. Whole
               numbers beyond 1 indicate that any failing units up to that absolute threshold
               value will result in a succeeding **testthat** test or evaluate to TRUE. Likewise,
               fractional values (between 0 and 1) act as a proportional failure threshold, where
               0.15 means that 15 percent of failing test units results in an overall test failure.

## 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 vali-
dation function, takes a specially-crafted list object that is best produced by the [action_levels()](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 of 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

For the validation function, the return value is either a ptblank_agent object or a table object (de-
pending on whether an agent object or a table was passed to x). The expectation function invisibly
returns its input but, in the context of testing data, the function is called primarily for its potential
side-effects (e.g., signaling failure). The test function returns a logical value.

## Function ID

2-18

**See Also**

Other validation 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_expr()`, `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**

```
# The `small_table` dataset in the
# package has a `date_time` column;
# the following examples will validate
# that that column is of the `POSIXct`
# and `POSIXt` classes

# A: Using an `agent` with validation
#    functions and then `interrogate()`

# Validate that the column `date_time`
# is indeed a date-time column
agent <-
  create_agent(small_table) %>%
  col_is_posix(vars(date_time)) %>%
  interrogate()

# Determine if this validation
# had no failing test units (1)
all_passed(agent)

# Calling `agent` in the console
# prints the agent's report; but we
# can get a `gt_tbl` object directly
# with `get_agent_report(agent)`

# B: Using the validation function
#    directly on the data (no `agent`)

# This way of using validation functions
# acts as a data filter: data is passed
# through but should `stop()` if there
# is a single test unit failing; the
# behavior of side effects can be
# customized with the `actions` option
small_table %>%
  col_is_posix(vars(date_time)) %>%
  dplyr::slice(1:5)

# C: Using the expectation function

# With the `expect_*()` form, we would
# typically perform one validation at a
# time; this is primarily used in
```

```
# testthat tests
expect_col_is_posix(
  small_table, vars(date_time)
)

# D: Using the test function

# With the `test_*()` form, we should
# get a single logical value returned
# to us
small_table %>%
  test_col_is_posix(vars(date_time))
```

---

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 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, tbl_dbi, or tbl_spark varieties. There's an additional
option, which is just for validating the schema of a tbl_dbi or tbl_spark object: we can validate
the schema based on R column types (e.g., ″numeric″, ″character″, etc.), SQL column types
(e.g., ″double″, ″varchar″, etc.), or Spark SQL column types (″DoubleType″, ″StringType″,
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

```
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. |
| .tbl | An option to use a table object to define the schema. If this is provided then any values provided to ... will be ignored. |
| .db_col_types | Determines whether the column types refer to R column types (″r″) or SQL column types (″sql″). |

### Function ID

1-7

**See Also**

Other Planning and Prep: `action_levels()`, `create_agent()`, `create_informant()`, `db_tbl()`, `scan_data()`, `validate_rmd()`

**Examples**

```
# 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
# `schema_obj` 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 function, the `expect_col_schema_match()` expectation function, and the `test_col_schema_match()` test function all work in conjunction with a `col_schema` object (generated through the [col_schema()](#) function) to determine whether the expected schema matches that of the target table. The validation function can be used directly on a data table or with an *agent* object (technically, a `ptblank_agent` object) whereas the expectation and test functions can only be used with a data table. The types of data tables that can be used include data frames, tibbles, database tables (`tbl_dbi`), and Spark DataFrames (`tbl_spark`).

## Usage

```
col_schema_match(
  x,
  schema,
  complete = TRUE,
  in_order = TRUE,
  is_exact = TRUE,
  actions = NULL,
  step_id = NULL,
  label = NULL,
  brief = NULL,
  active = TRUE
)

expect_col_schema_match(
  object,
  schema,
  complete = TRUE,
  in_order = TRUE,
  is_exact = TRUE,
  threshold = 1
)

test_col_schema_match(
  object,
  schema,
  complete = TRUE,
```

```
    in_order = TRUE,
    is_exact = TRUE,
    threshold = 1
)
```

## Arguments

| | |
|---|---|
| x | A data frame, tibble (`tbl_df` or `tbl_dbi`), Spark DataFrame (`tbl_spark`), or, an agent object of class ptblank_agent that is created with `create_agent()`. |
| schema | A table schema of type `col_schema` which can be generated using the `col_schema()` function. |
| complete | A requirement to account for all table columns in the provided schema. By default, this is TRUE and so that all column names in the target table must be present in the schema object. This restriction can be relaxed by using FALSE, where we can provide a subset of table columns in the schema. |
| in_order | A stringent requirement for enforcing the order of columns in the provided schema. By default, this is TRUE and the order of columns in both the schema and the target table must match. By setting to FALSE, this strict order requirement is removed. |
| is_exact | Determines whether the check for column types should be exact or even performed at all. For example, columns in R data frames may have multiple classes (e.g., a date-time column can have both the "POSIXct" and the "POSIXt" classes). If using is_exact == FALSE, the column type in the user-defined schema for a date-time value can be set as either "POSIXct" *or* "POSIXt" and pass validation (with this column, at least). This can be taken a step further and using NULL for a column type in the user-defined schema will skip the validation check of a column type. By default, is_exact is set to 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. |
| step_id | One or more optional identifiers for the single or multiple validation steps generated from calling a validation function. The use of step IDs serves to distinguish validation steps from each other and provide an opportunity for supplying a more meaningful label compared to the step index. By default this is NULL, and **pointblank** will automatically generate the step ID value (based on the step index) in this case. One or more values can be provided, and the exact number of ID values should (1) match the number of validation steps that the validation function call will produce (influenced by the number of columns provided), (2) be an ID string not used in any previous validation step, and (3) be a vector with unique values. |
| label | An optional label for the validation step. This label appears in the agent report and for the best appearance it should be kept short. |
| brief | An optional, text-based description for the validation step. If nothing is provided here then an *autobrief* is generated by the agent, using the language provided in `create_agent()`'s lang argument (which defaults to "en" or English). The *autobrief* incorporates details of the validation step so it's often the preferred |

option in most cases (where a `label` might be better suited to succinctly describe the validation).

`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`.

`object`     A data frame, tibble (`tbl_df` or `tbl_dbi`), or Spark DataFrame (`tbl_spark`) that serves as the target table for the expectation function or the test function.

`threshold`     A simple failure threshold value for use with the expectation (`expect_`) and the test (`test_`) function variants. By default, this is set to `1` meaning that any single unit of failure in data validation results in an overall test failure. Whole numbers beyond `1` indicate that any failing units up to that absolute threshold value will result in a succeeding **testthat** test or evaluate to `TRUE`. Likewise, fractional values (between `0` and `1`) act as a proportional failure threshold, where `0.15` means that 15 percent of failing test units results in an overall test failure.

## Details

The validation step or expectation operates over a single test unit, which is whether the schema matches that of the table (within the constraints enforced by the `complete`, `in_order`, and `is_exact` options). If the target table is a `tbl_dbi` or a `tbl_spark` object, we can choose to validate the column schema that is based on R column types (e.g., `"numeric"`, `"character"`, etc.), SQL column types (e.g., `"double"`, `"varchar"`, etc.), or Spark SQL types (e.g,. `"DoubleType"`, `"StringType"`, etc.). That option is defined in the [col_schema()](#) function (it is the `.db_col_types` argument).

There are options to make schema checking less stringent (by default, this validation operates with highest level of strictness). With the `complete` option set to `FALSE`, we can supply a `col_schema` object with a partial inclusion of columns. Using `in_order` set to `FALSE` means that there is no requirement for the columns defined in the `schema` object to be in the same order as in the target table. Finally, the `is_exact` option set to `FALSE` means that all column classes/types don't have to be provided for a particular column. It can even be `NULL`, skipping the check of the column type.

Often, we will want to specify `actions` for the validation. This argument, present in every validation 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 of 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

For the validation function, the return value is either a `ptblank_agent` object or a table object (depending on whether an agent object or a table was passed to `x`). The expectation function invisibly

returns its input but, in the context of testing data, the function is called primarily for its potential side-effects (e.g., signaling failure). The test function returns a logical value.

**Function ID**

2-24

**See Also**

Other validation 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_expr(), 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**

```
# For all examples here, we'll use
# a simple table with two columns:
# one `integer` (`a`) and the other
# `character` (`b`); the following
# examples will validate that the
# table columns abides match a schema
# object as created by `col_schema()`
tbl <-
  dplyr::tibble(
    a = 1:5,
    b = letters[1:5]
  )

tbl

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

# A: Using an `agent` with validation
#    functions and then `interrogate()`

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

```
    interrogate()

# Determine if this validation
# had no failing test units (there is
# a single test unit governed by
# whether there is a match)
all_passed(agent)

# Calling `agent` in the console
# prints the agent's report; but we
# can get a `gt_tbl` object directly
# with `get_agent_report(agent)`

# B: Using the validation function
#    directly on the data (no `agent`)

# This way of using validation functions
# acts as a data filter: data is passed
# through but should `stop()` if there
# is a single test unit failing; the
# behavior of side effects can be
# customized with the `actions` option
tbl %>% col_schema_match(schema_obj)

# C: Using the expectation function

# With the `expect_*()` form, we would
# typically perform one validation at a
# time; this is primarily used in
# testthat tests
expect_col_schema_match(tbl, schema_obj)

# D: Using the test function

# With the `test_*()` form, we should
# get a single logical value returned
# to us
tbl %>% test_col_schema_match(schema_obj)
```

---

col_vals_between    *Are column data between two specified values?*

---

## Description

The col_vals_between() validation function, the expect_col_vals_between() expectation function, and the test_col_vals_between() test function all check whether column values in a table 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 functions. The validation function can be used directly on a data table or with an *agent* object (technically, a ptblank_agent object) whereas the expectation and test functions can only be used with a data table. The types of data tables that can be used include data frames, tibbles, database tables (tbl_dbi), and Spark DataFrames (tbl_spark). Each validation step or expectation will operate over a single test unit, which is whether the column is an integer-type column or not. Each validation step or expectation will operate over the number of test units that is equal to the number of rows in the table (after any `preconditions` have been applied).

## Usage

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

expect_col_vals_between(
  object,
  columns,
  left,
  right,
  inclusive = c(TRUE, TRUE),
  na_pass = FALSE,
  preconditions = NULL,
  threshold = 1
)

test_col_vals_between(
  object,
  columns,
  left,
  right,
  inclusive = c(TRUE, TRUE),
  na_pass = FALSE,
  preconditions = NULL,
  threshold = 1
)
```

**Arguments**

| | |
|---|---|
| x | A data frame, tibble (`tbl_df` or `tbl_dbi`), Spark DataFrame (`tbl_spark`), or, an agent object of class ptblank_agent that is created with `create_agent()`. |
| columns | The column (or a set of columns, provided as a character vector) to which this validation should be applied. |
| left | The lower bound for the range. The validation includes this bound value (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 considered as passing test units? 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 . serves as the input data table to be transformed (e.g., ~ . %>% dplyr::mutate(col = col + 10). |
| 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. |
| step_id | One or more optional identifiers for the single or multiple validation steps generated from calling a validation function. The use of step IDs serves to distinguish validation steps from each other and provide an opportunity for supplying a more meaningful label compared to the step index. By default this is NULL, and **pointblank** will automatically generate the step ID value (based on the step index) in this case. One or more values can be provided, and the exact number of ID values should (1) match the number of validation steps that the validation function call will produce (influenced by the number of columns provided), (2) be an ID string not used in any previous validation step, and (3) be a vector with unique values. |
| label | An optional label for the validation step. This label appears in the agent report and for the best appearance it should be kept short. |
| brief | An optional, text-based description for the validation step. If nothing is provided here then an *autobrief* is generated by the agent, using the language provided in `create_agent()`'s lang argument (which defaults to "en" or English). The *autobrief* incorporates details of the validation step so it's often the preferred option in most cases (where a label might be better suited to succinctly describe the validation). |
| 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. |

object        A data frame, tibble (`tbl_df` or `tbl_dbi`), or Spark DataFrame (`tbl_spark`)
              that serves as the target table for the expectation function or the test function.

threshold     A simple failure threshold value for use with the expectation (`expect_`) and
              the test (`test_`) function variants. By default, this is set to 1 meaning that any
              single unit of failure in data validation results in an overall test failure. Whole
              numbers beyond 1 indicate that any failing units up to that absolute threshold
              value will result in a succeeding **testthat** test or evaluate to `TRUE`. Likewise,
              fractional values (between 0 and 1) act as a proportional failure threshold, where
              0.15 means that 15 percent of failing test units results in an overall test failure.

## 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 function supports special handling of `NA` values. The `na_pass` argument will deter-
mine 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 `NA`s encountered will accumulate failing test units.

Having table `preconditions` means **pointblank** will mutate the table just before interrogation.
Such a table mutation is isolated in scope to the validation step(s) produced by the validation func-
tion call. Using **dplyr** code is suggested here since the statements can be translated to SQL if
necessary. The code is most easily supplied as a one-sided **R** formula (using a leading ~). In
the formula representation, the . serves as the input data table to be transformed (e.g., `~ . %>%`
`dplyr::mutate(col_a = col_b + 10)`). Alternatively, a function could instead be supplied (e.g.,
`function(x) dplyr::mutate(x, col_a = col_b + 10)`).

Often, we will want to specify `actions` for the validation. This argument, present in every vali-
dation 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 of test units failed, or, an absolute value), often using the `warn_at` ar-
gument. 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

For the validation function, the return value is either a `ptblank_agent` object or a table object (de-
pending on whether an agent object or a table was passed to x). The expectation function invisibly
returns its input but, in the context of testing data, the function is called primarily for its potential
side-effects (e.g., signaling failure). The test function returns a logical value.

**Function ID**

2-7

**See Also**

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

Other validation 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_expr()`, `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**

```
# The `small_table` dataset in the
# package has a column of numeric
# values in `c` (there are a few NAs
# in that column); the following
# examples will validate the values
# in that numeric column

# A: Using an `agent` with validation
#    functions and then `interrogate()`

# Validate that values in column `c`
# are all between `1` and `9`; because
# there are NA values, we'll choose to
# let those pass validation by setting
# `na_pass = TRUE`
agent <-
  create_agent(small_table) %>%
  col_vals_between(
    vars(c), 1, 9, na_pass = TRUE
  ) %>%
  interrogate()

# Determine if this validation
# had no failing test units (there
# are 13 test units, one for each row)
all_passed(agent)

# Calling `agent` in the console
# prints the agent's report; but we
# can get a `gt_tbl` object directly
# with `get_agent_report(agent)`

# B: Using the validation function
#    directly on the data (no `agent`)

# This way of using validation functions
# acts as a data filter: data is passed
```

```
# through but should `stop()` if there
# is a single test unit failing; the
# behavior of side effects can be
# customized with the `actions` option
small_table %>%
  col_vals_between(
    vars(c), 1, 9, na_pass = TRUE
  ) %>%
  dplyr::pull(c)

# C: Using the expectation function

# With the `expect_*()` form, we would
# typically perform one validation at a
# time; this is primarily used in
# testthat tests
expect_col_vals_between(
  small_table, vars(c), 1, 9,
  na_pass = TRUE
)

# D: Using the test function

# With the `test_*()` form, we should
# get a single logical value returned
# to us
small_table %>%
  test_col_vals_between(
    vars(c), 1, 9,
    na_pass = TRUE
  )

# An additional note on the bounds for
# this function: they are inclusive by
# default (i.e., values of exactly 1
# and 9 will pass); we can modify the
# inclusiveness of the upper and lower
# bounds with the `inclusive` option,
# which is a length-2 logical vector

# Testing with the upper bound being
# non-inclusive, we get `FALSE` since
# two values are `9` and they now fall
# outside of the upper (or right) bound
small_table %>%
  test_col_vals_between(
    vars(c), 1, 9,
    inclusive = c(TRUE, FALSE),
    na_pass = TRUE
  )
```

---

col_vals_equal        *Are column data equal to a specified value?*

---

### Description

The `col_vals_equal()` validation function, the `expect_col_vals_equal()` expectation function, and the `test_col_vals_equal()` test function all check whether column values in a table are equal to a specified `value`. The `value` can be specified as a single, literal value or as a column name given in `vars()`. The validation function can be used directly on a data table or with an *agent* object (technically, a `ptblank_agent` object) whereas the expectation and test functions can only be used with a data table. The types of data tables that can be used include data frames, tibbles, database tables (`tbl_dbi`), and Spark DataFrames (`tbl_spark`). Each validation step or expectation 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_equal(
  x,
  columns,
  value,
  na_pass = FALSE,
  preconditions = NULL,
  actions = NULL,
  step_id = NULL,
  label = NULL,
  brief = NULL,
  active = TRUE
)

expect_col_vals_equal(
  object,
  columns,
  value,
  na_pass = FALSE,
  preconditions = NULL,
  threshold = 1
)

test_col_vals_equal(
  object,
  columns,
  value,
  na_pass = FALSE,
  preconditions = NULL,
  threshold = 1
)
```

**Arguments**

| | |
|---|---|
| x | A data frame, tibble (`tbl_df` or `tbl_dbi`), Spark DataFrame (`tbl_spark`), or, an agent object of class ptblank_agent that is created with `create_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 considered as passing test units? 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 . serves as the input data table to be transformed (e.g., ~ . %>% dplyr::mutate(col = col + 10). |
| 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. |
| step_id | One or more optional identifiers for the single or multiple validation steps generated from calling a validation function. The use of step IDs serves to distinguish validation steps from each other and provide an opportunity for supplying a more meaningful label compared to the step index. By default this is `NULL`, and **pointblank** will automatically generate the step ID value (based on the step index) in this case. One or more values can be provided, and the exact number of ID values should (1) match the number of validation steps that the validation function call will produce (influenced by the number of columns provided), (2) be an ID string not used in any previous validation step, and (3) be a vector with unique values. |
| label | An optional label for the validation step. This label appears in the agent report and for the best appearance it should be kept short. |
| brief | An optional, text-based description for the validation step. If nothing is provided here then an *autobrief* is generated by the agent, using the language provided in `create_agent()`'s lang argument (which defaults to "en" or English). The *autobrief* incorporates details of the validation step so it's often the preferred option in most cases (where a label might be better suited to succinctly describe the validation). |
| 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`. |
| object | A data frame, tibble (`tbl_df` or `tbl_dbi`), or Spark DataFrame (`tbl_spark`) that serves as the target table for the expectation function or the test function. |
| threshold | A simple failure threshold value for use with the expectation (expect_) and the test (test_) function variants. By default, this is set to 1 meaning that any single unit of failure in data validation results in an overall test failure. Whole numbers beyond 1 indicate that any failing units up to that absolute threshold |

value will result in a succeeding **testthat** test or evaluate to TRUE. Likewise, fractional values (between 0 and 1) act as a proportional failure threshold, where 0.15 means that 15 percent of failing test units results in an overall test failure.

### 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 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. Such a table mutation is isolated in scope to the validation step(s) produced by the validation function call. Using **dplyr** code is suggested here since the statements can be translated to SQL if necessary. The code is most easily supplied as a one-sided **R** formula (using a leading ~). In the formula representation, the . serves as the input data table to be transformed (e.g., ~ . %>% dplyr::mutate(col_a = col_b + 10)). Alternatively, a function could instead be supplied (e.g., function(x) dplyr::mutate(x, col_a = col_b + 10)).

Often, we will want to specify actions for the validation. This argument, present in every validation 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 of 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

For the validation function, the return value is either a ptblank_agent object or a table object (depending on whether an agent object or a table was passed to x). The expectation function invisibly returns its input but, in the context of testing data, the function is called primarily for its potential side-effects (e.g., signaling failure). The test function returns a logical value.

### Function ID

2-3

**See Also**

The analogue to this function: col_vals_not_equal().

Other validation 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_expr(), 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**

```
# For all of the examples here, we'll
# use a simple table with three numeric
# columns (`a`, `b`, and `c`) and three
# character columns (`d`, `e`, and `f`)
tbl <-
  dplyr::tibble(
    a = c(5, 5, 5, 5, 5, 5),
    b = c(1, 1, 1, 2, 2, 2),
    c = c(1, 1, 1, 2, 2, 2),
    d = LETTERS[c(1:3, 5:7)],
    e = LETTERS[c(1:6)],
    f = LETTERS[c(1:6)]
  )

tbl

# A: Using an `agent` with validation
#    functions and then `interrogate()`

# Validate that values in column `a`
# are all equal to the value of `5`
agent <-
  create_agent(tbl) %>%
  col_vals_equal(vars(a), 5) %>%
  interrogate()

# Determine if this validation
# had no failing test units (there
# are 6 test units, one for each row)
all_passed(agent)

# Calling `agent` in the console
# prints the agent's report; but we
# can get a `gt_tbl` object directly
# with `get_agent_report(agent)`

# B: Using the validation function
#    directly on the data (no `agent`)

# This way of using validation functions
# acts as a data filter: data is passed
```

```
# through but should `stop()` if there
# is a single test unit failing; the
# behavior of side effects can be
# customized with the `actions` option
tbl %>%
  col_vals_equal(vars(a), 5) %>%
  dplyr::pull(a)

# C: Using the expectation function

# With the `expect_*()` form, we would
# typically perform one validation at a
# time; this is primarily used in
# testthat tests
expect_col_vals_equal(tbl, vars(a), 5)

# D: Using the test function

# With the `test_*()` form, we should
# get a single logical value returned
# to us
test_col_vals_equal(tbl, vars(a), 5)
```

---

col_vals_expr                     *Do column data agree with a predicate expression?*

---

### Description

The `col_vals_expr()` validation function checks for whether column values in a table match a user-defined predicate expression. The validation function can be used directly on a data table or with an *agent* object (technically, a ptblank_agent object) whereas the expectation and test functions can only be used with a data table. The types of data tables that can be used include data frames, tibbles, database tables (tbl_dbi), and Spark DataFrames (tbl_spark). Each validation step or expectation 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_expr(
  x,
  expr,
  preconditions = NULL,
  actions = NULL,
  step_id = NULL,
  label = NULL,
  brief = NULL,
  active = TRUE
)
```

```
expect_col_vals_expr(object, expr, preconditions = NULL, threshold = 1)

test_col_vals_expr(object, expr, preconditions = NULL, threshold = 1)
```

## Arguments

| | |
|---|---|
| x | A data frame, tibble (`tbl_df` or `tbl_dbi`), Spark DataFrame (`tbl_spark`), or, an agent object of class ptblank_agent that is created with [create_agent()](). |
| expr | An expression to use for this test. This can either be in the form of a call made with the expr() function or as a one-sided **R** formula (using a leading ~). |
| 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 . serves as the input data table to be transformed (e.g., ~ . %>% dplyr::mutate(col = col + 10). |
| 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. |
| step_id | One or more optional identifiers for the single or multiple validation steps generated from calling a validation function. The use of step IDs serves to distinguish validation steps from each other and provide an opportunity for supplying a more meaningful label compared to the step index. By default this is NULL, and **pointblank** will automatically generate the step ID value (based on the step index) in this case. One or more values can be provided, and the exact number of ID values should (1) match the number of validation steps that the validation function call will produce (influenced by the number of columns provided), (2) be an ID string not used in any previous validation step, and (3) be a vector with unique values. |
| label | An optional label for the validation step. This label appears in the agent report and for the best appearance it should be kept short. |
| brief | An optional, text-based description for the validation step. If nothing is provided here then an *autobrief* is generated by the agent, using the language provided in [create_agent()]()'s lang argument (which defaults to "en" or English). The *autobrief* incorporates details of the validation step so it's often the preferred option in most cases (where a label might be better suited to succinctly describe the validation). |
| 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. |
| object | A data frame, tibble (`tbl_df` or `tbl_dbi`), or Spark DataFrame (`tbl_spark`) that serves as the target table for the expectation function or the test function. |
| threshold | A simple failure threshold value for use with the expectation (expect_) and the test (test_) function variants. By default, this is set to 1 meaning that any |

single unit of failure in data validation results in an overall test failure. Whole numbers beyond 1 indicate that any failing units up to that absolute threshold value will result in a succeeding **testthat** test or evaluate to TRUE. Likewise, fractional values (between 0 and 1) act as a proportional failure threshold, where 0.15 means that 15 percent of failing test units results in an overall test failure.

### Details

Having table preconditions means **pointblank** will mutate the table just before interrogation. Such a table mutation is isolated in scope to the validation step(s) produced by the validation function call. Using **dplyr** code is suggested here since the statements can be translated to SQL if necessary. The code is most easily supplied as a one-sided **R** formula (using a leading ~). In the formula representation, the . serves as the input data table to be transformed (e.g., ~ . %>% dplyr::mutate(col_a = col_b + 10)). Alternatively, a function could instead be supplied (e.g., function(x) dplyr::mutate(x,col_a = col_b + 10)).

Often, we will want to specify actions for the validation. This argument, present in every validation function, takes a specially-crafted list object that is best produced by the [action_levels()](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 of 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

For the validation function, the return value is either a ptblank_agent object or a table object (depending on whether an agent object or a table was passed to x). The expectation function invisibly returns its input but, in the context of testing data, the function is called primarily for its potential side-effects (e.g., signaling failure). The test function returns a logical value.

### Function ID

2-25

### See Also

Other validation functions: [col_exists()](col_exists()), [col_is_character()](col_is_character()), [col_is_date()](col_is_date()), [col_is_factor()](col_is_factor()), [col_is_integer()](col_is_integer()), [col_is_logical()](col_is_logical()), [col_is_numeric()](col_is_numeric()), [col_is_posix()](col_is_posix()), [col_schema_match()](col_schema_match()), [col_vals_between()](col_vals_between()), [col_vals_equal()](col_vals_equal()), [col_vals_gte()](col_vals_gte()), [col_vals_gt()](col_vals_gt()), [col_vals_in_set()](col_vals_in_set()), [col_vals_lte()](col_vals_lte()), [col_vals_lt()](col_vals_lt()), [col_vals_not_between()](col_vals_not_between()), [col_vals_not_equal()](col_vals_not_equal()), [col_vals_not_in_set()](col_vals_not_in_set()), [col_vals_not_null()](col_vals_not_null()), [col_vals_null()](col_vals_null()), [col_vals_regex()](col_vals_regex()), [conjointly()](conjointly()), [rows_distinct()](rows_distinct())

## Examples

```
# For all of the examples here, we'll
# use a simple table with three numeric
# columns (`a`, `b`, and `c`) and three
# character columns (`d`, `e`, and `f`)
tbl <-
  dplyr::tibble(
    a = c(1, 2, 1, 7, 8, 6),
    b = c(0, 0, 0, 1, 1, 1),
    c = c(0.5, 0.3, 0.8, 1.4, 1.9, 1.2),
  )

tbl

# A: Using an `agent` with validation
#    functions and then `interrogate()`

# Validate that values in column `a`
# are integer-like by using the R modulo
# operator and expecting `0`
agent <-
  create_agent(tbl) %>%
  col_vals_expr(expr(a %% 1 == 0)) %>%
  interrogate()

# Determine if this validation
# had no failing test units (there
# are 6 test units, one for each row)
all_passed(agent)

# Calling `agent` in the console
# prints the agent's report; but we
# can get a `gt_tbl` object directly
# with `get_agent_report(agent)`

# B: Using the validation function
#    directly on the data (no `agent`)

# This way of using validation functions
# acts as a data filter: data is passed
# through but should `stop()` if there
# is a single test unit failing; the
# behavior of side effects can be
# customized with the `actions` option
tbl %>%
  col_vals_expr(expr(a %% 1 == 0)) %>%
  dplyr::pull(a)

# C: Using the expectation function

# With the `expect_*()` form, we would
# typically perform one validation at a
```

```
# time; this is primarily used in
# testthat tests
expect_col_vals_expr(tbl, ~ a %% 1 == 0)

# D: Using the test function

# With the `test_*()` form, we should
# get a single logical value returned
# to us
test_col_vals_expr(tbl, ~ a %% 1 == 0)

# Variations

# We can do more complex things by
# taking advantage of the `case_when()`
# and `between()` functions (available
# for use in the pointblank package)
tbl %>%
  test_col_vals_expr(~ case_when(
    b == 0 ~ a %>% between(0, 5) & c < 1,
    b == 1 ~ a > 5 & c >= 1
  ))

# If you only want to test a subset of
# rows, then the `case_when()` statement
# doesn't need to be exhaustive; any
# rows that don't fall into the cases
# will be pruned (giving us less test
# units overall)
tbl %>%
  test_col_vals_expr(~ case_when(
    b == 1 ~ a > 5 & c >= 1
  ))
```

---

col_vals_gt                      *Are column data greater than a specified value?*

---

### Description

The col_vals_gt() validation function, the expect_col_vals_gt() expectation function, and
the test_col_vals_gt() test function all check whether column values in a table 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(). The validation
function can be used directly on a data table or with an *agent* object (technically, a ptblank_agent
object) whereas the expectation and test functions can only be used with a data table. The types
of data tables that can be used include data frames, tibbles, database tables (tbl_dbi), and Spark
DataFrames (tbl_spark). Each validation step or expectation 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,
  step_id = NULL,
  label = NULL,
  brief = NULL,
  active = TRUE
)

expect_col_vals_gt(
  object,
  columns,
  value,
  na_pass = FALSE,
  preconditions = NULL,
  threshold = 1
)

test_col_vals_gt(
  object,
  columns,
  value,
  na_pass = FALSE,
  preconditions = NULL,
  threshold = 1
)
```

**Arguments**

| | |
|---|---|
| x | A data frame, tibble (`tbl_df` or `tbl_dbi`), Spark DataFrame (`tbl_spark`), or, an agent object of class `ptblank_agent` that is created with [`create_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 considered as passing test units? 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 `.` serves as the input data table to be transformed (e.g., `~ . %>% dplyr::mutate(col = col + 10)`. |

| | |
|---|---|
| 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. |
| step_id | One or more optional identifiers for the single or multiple validation steps generated from calling a validation function. The use of step IDs serves to distinguish validation steps from each other and provide an opportunity for supplying a more meaningful label compared to the step index. By default this is NULL, and **pointblank** will automatically generate the step ID value (based on the step index) in this case. One or more values can be provided, and the exact number of ID values should (1) match the number of validation steps that the validation function call will produce (influenced by the number of columns provided), (2) be an ID string not used in any previous validation step, and (3) be a vector with unique values. |
| label | An optional label for the validation step. This label appears in the agent report and for the best appearance it should be kept short. |
| brief | An optional, text-based description for the validation step. If nothing is provided here then an *autobrief* is generated by the agent, using the language provided in `create_agent()`'s lang argument (which defaults to "en" or English). The *autobrief* incorporates details of the validation step so it's often the preferred option in most cases (where a `label` might be better suited to succinctly describe the validation). |
| 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. |
| object | A data frame, tibble (`tbl_df` or `tbl_dbi`), or Spark DataFrame (`tbl_spark`) that serves as the target table for the expectation function or the test function. |
| threshold | A simple failure threshold value for use with the expectation (expect_) and the test (test_) function variants. By default, this is set to 1 meaning that any single unit of failure in data validation results in an overall test failure. Whole numbers beyond 1 indicate that any failing units up to that absolute threshold value will result in a succeeding **testthat** test or evaluate to TRUE. Likewise, fractional values (between 0 and 1) act as a proportional failure threshold, where 0.15 means that 15 percent of failing test units results in an overall test failure. |

### 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 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. Such a table mutation is isolated in scope to the validation step(s) produced by the validation function call. Using **dplyr** code is suggested here since the statements can be translated to SQL if necessary. The code is most easily supplied as a one-sided **R** formula (using a leading ~). In the formula representation, the . serves as the input data table to be transformed (e.g., ~ . %>% dplyr::mutate(col_a = col_b + 10)). Alternatively, a function could instead be supplied (e.g., function(x) dplyr::mutate(x,col_a = col_b + 10)).

Often, we will want to specify actions for the validation. This argument, present in every validation 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 of 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

For the validation function, the return value is either a ptblank_agent object or a table object (depending on whether an agent object or a table was passed to x). The expectation function invisibly returns its input but, in the context of testing data, the function is called primarily for its potential side-effects (e.g., signaling failure). The test function returns a logical value.

## Function ID

2-6

## See Also

The analogous function with a left-closed bound: [col_vals_gte()](#).

Other validation 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_expr()](#), [col_vals_gte()](#), [col_vals_in_set()](#), [col_vals_lte()](#), [col_vals_lt()](#), [col_vals_not_between()](#), [col_vals_not_equal()](#), [col_vals_not_in_set()](#), [col_vals_not_null()](#), [col_vals_null()](#), [col_vals_regex()](#), [conjointly()](#), [rows_distinct()](#)

## Examples

```
# For all of the examples here, we'll
# use a simple table with three numeric
# columns (`a`, `b`, and `c`) and three
# character columns (`d`, `e`, and `f`)
tbl <-
  dplyr::tibble(
```

```
    a = c(5, 5, 5, 5, 5, 5),
    b = c(1, 1, 1, 2, 2, 2),
    c = c(1, 1, 1, 2, 3, 4),
    d = LETTERS[a],
    e = LETTERS[b],
    f = LETTERS[c]
  )

tbl

# A: Using an `agent` with validation
#    functions and then `interrogate()`

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

# Determine if this validation
# had no failing test units (there
# are 6 test units, one for each row)
all_passed(agent)

# Calling `agent` in the console
# prints the agent's report; but we
# can get a `gt_tbl` object directly
# with `get_agent_report(agent)`

# B: Using the validation function
#    directly on the data (no `agent`)

# This way of using validation functions
# acts as a data filter: data is passed
# through but should `stop()` if there
# is a single test unit failing; the
# behavior of side effects can be
# customized with the `actions` option
tbl %>%
  col_vals_gt(vars(a), 4) %>%
  dplyr::pull(a)

# C: Using the expectation function

# With the `expect_*()` form, we would
# typically perform one validation at a
# time; this is primarily used in
# testthat tests
expect_col_vals_gt(tbl, vars(a), 4)

# D: Using the test function
```

```
# With the `test_*()` form, we should
# get a single logical value returned
# to us
test_col_vals_gt(tbl, vars(a), 4)
```

---

col_vals_gte                    *Are column data greater than or equal to a specified value?*

---

### Description

The `col_vals_gte()` validation function, the `expect_col_vals_gte()` expectation function, and the `test_col_vals_gte()` test function all check whether column values in a table 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()`. The validation step function can be used directly on a data table or with an *agent* object (technically, a `ptblank_agent` object) whereas the expectation and test functions can only be used with a data table. The types of data tables that can be used include data frames, tibbles, database tables (`tbl_dbi`), and Spark DataFrames (`tbl_spark`). Each validation step or expectation 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,
  step_id = NULL,
  label = NULL,
  brief = NULL,
  active = TRUE
)

expect_col_vals_gte(
  object,
  columns,
  value,
  na_pass = FALSE,
  preconditions = NULL,
  threshold = 1
)

test_col_vals_gte(
```

```
  object,
  columns,
  value,
  na_pass = FALSE,
  preconditions = NULL,
  threshold = 1
)
```

## Arguments

| | |
|---|---|
| x | A data frame, tibble (`tbl_df` or `tbl_dbi`), Spark DataFrame (`tbl_spark`), or, an agent object of class `ptblank_agent` that is created with [`create_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 considered as passing test units? 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 . serves as the input data table to be transformed (e.g., ~ . %>% dplyr::mutate(col = col + 10). |
| 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. |
| step_id | One or more optional identifiers for the single or multiple validation steps generated from calling a validation function. The use of step IDs serves to distinguish validation steps from each other and provide an opportunity for supplying a more meaningful label compared to the step index. By default this is NULL, and **pointblank** will automatically generate the step ID value (based on the step index) in this case. One or more values can be provided, and the exact number of ID values should (1) match the number of validation steps that the validation function call will produce (influenced by the number of `columns` provided), (2) be an ID string not used in any previous validation step, and (3) be a vector with unique values. |
| label | An optional label for the validation step. This label appears in the agent report and for the best appearance it should be kept short. |
| brief | An optional, text-based description for the validation step. If nothing is provided here then an *autobrief* is generated by the agent, using the language provided in [`create_agent()`](#)'s lang argument (which defaults to "en" or English). The *autobrief* incorporates details of the validation step so it's often the preferred option in most cases (where a `label` might be better suited to succinctly describe the validation). |
| 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. |
| object | A data frame, tibble (`tbl_df` or `tbl_dbi`), or Spark DataFrame (`tbl_spark`) that serves as the target table for the expectation function or the test function. |
| threshold | A simple failure threshold value for use with the expectation (`expect_`) and the test (`test_`) function variants. By default, this is set to 1 meaning that any single unit of failure in data validation results in an overall test failure. Whole numbers beyond 1 indicate that any failing units up to that absolute threshold value will result in a succeeding **testthat** test or evaluate to TRUE. Likewise, fractional values (between 0 and 1) act as a proportional failure threshold, where 0.15 means that 15 percent of failing test units results in an overall test failure. |

## 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 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. Such a table mutation is isolated in scope to the validation step(s) produced by the validation function call. Using **dplyr** code is suggested here since the statements can be translated to SQL if necessary. The code is most easily supplied as a one-sided **R** formula (using a leading ~). In the formula representation, the . serves as the input data table to be transformed (e.g., ~ . %>% dplyr::mutate(col_a = col_b + 10)). Alternatively, a function could instead be supplied (e.g., function(x) dplyr::mutate(x,col_a = col_b + 10)).

Often, we will want to specify `actions` for the validation. This argument, present in every validation 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 of 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

For the validation function, the return value is either a `ptblank_agent` object or a table object (depending on whether an agent object or a table was passed to x). The expectation function invisibly

returns its input but, in the context of testing data, the function is called primarily for its potential side-effects (e.g., signaling failure). The test function returns a logical value.

**Function ID**

2-5

**See Also**

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

Other validation 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_expr(), 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**

```
# For all of the examples here, we'll
# use a simple table with three numeric
# columns (`a`, `b`, and `c`) and three
# character columns (`d`, `e`, and `f`)
tbl <-
  dplyr::tibble(
      a = c(5, 5, 5, 5, 5, 5),
      b = c(1, 1, 1, 2, 2, 2),
      c = c(1, 1, 1, 2, 3, 4),
      d = LETTERS[a],
      e = LETTERS[b],
      f = LETTERS[c]
  )

tbl

# A: Using an `agent` with validation
#    functions and then `interrogate()`

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

# Determine if this validation
# had no failing test units (there
# are 6 test units, one for each row)
all_passed(agent)

# Calling `agent` in the console
# prints the agent's report; but we
```

```
# can get a `gt_tbl` object directly
# with `get_agent_report(agent)`

# B: Using the validation function
#    directly on the data (no `agent`)

# This way of using validation functions
# acts as a data filter: data is passed
# through but should `stop()` if there
# is a single test unit failing; the
# behavior of side effects can be
# customized with the `actions` option
tbl %>%
  col_vals_gte(vars(a), 5) %>%
  dplyr::pull(a)

# C: Using the expectation function

# With the `expect_*()` form, we would
# typically perform one validation at a
# time; this is primarily used in
# testthat tests
expect_col_vals_gte(tbl, vars(a), 5)

# D: Using the test function

# With the `test_*()` form, we should
# get a single logical value returned
# to us
test_col_vals_gte(tbl, vars(a), 5)
```

---

col_vals_in_set             *Are column data part of a specified set of values?*

---

## Description

The `col_vals_in_set()` validation function, the `expect_col_vals_in_set()` expectation func-
tion, and the `test_col_vals_in_set()` test function all check whether column values in a table
are part of a specified set of values. The validation step function can be used directly on a data ta-
ble or with an *agent* object (technically, a `ptblank_agent` object) whereas the expectation and test
functions can only be used with a data table. The types of data tables that can be used include data
frames, tibbles, database tables (`tbl_dbi`), and Spark DataFrames (`tbl_spark`). Each validation
step or expectation 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_in_set(
```

```
  x,
  columns,
  set,
  preconditions = NULL,
  actions = NULL,
  step_id = NULL,
  label = NULL,
  brief = NULL,
  active = TRUE
)

expect_col_vals_in_set(
  object,
  columns,
  set,
  preconditions = NULL,
  threshold = 1
)

test_col_vals_in_set(object, columns, set, preconditions = NULL, threshold = 1)
```

## Arguments

| | |
|---|---|
| x | A data frame, tibble (`tbl_df` or `tbl_dbi`), Spark DataFrame (`tbl_spark`), or, an agent object of class ptblank_agent that is created with [create_agent()](create_agent()). |
| columns | The column (or a set of columns, provided as a character vector) to which this validation should be applied. |
| set | A vector of numeric or string-based elements, where column values found within this `set` will be considered as passing. |
| preconditions | expressions used for mutating the input table before proceeding with the validation. This is ideally as a one-sided R formula using a leading ~. In the formula representation, the . serves as the input data table to be transformed (e.g., ~ . %>% dplyr::mutate(col = col + 10). |
| 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()](action_levels()) helper function. |
| step_id | One or more optional identifiers for the single or multiple validation steps generated from calling a validation function. The use of step IDs serves to distinguish validation steps from each other and provide an opportunity for supplying a more meaningful label compared to the step index. By default this is NULL, and **pointblank** will automatically generate the step ID value (based on the step index) in this case. One or more values can be provided, and the exact number of ID values should (1) match the number of validation steps that the validation function call will produce (influenced by the number of `columns` provided), (2) be an ID string not used in any previous validation step, and (3) be a vector with unique values. |
| label | An optional label for the validation step. This label appears in the agent report and for the best appearance it should be kept short. |

brief            An optional, text-based description for the validation step. If nothing is provided
                 here then an *autobrief* is generated by the agent, using the language provided
                 in [`create_agent()`](#)'s lang argument (which defaults to "en" or English). The
                 *autobrief* incorporates details of the validation step so it's often the preferred
                 option in most cases (where a label might be better suited to succinctly describe
                 the validation).

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.

object           A data frame, tibble (tbl_df or tbl_dbi), or Spark DataFrame (tbl_spark)
                 that serves as the target table for the expectation function or the test function.

threshold        A simple failure threshold value for use with the expectation (expect_) and
                 the test (test_) function variants. By default, this is set to 1 meaning that any
                 single unit of failure in data validation results in an overall test failure. Whole
                 numbers beyond 1 indicate that any failing units up to that absolute threshold
                 value will result in a succeeding **testthat** test or evaluate to TRUE. Likewise,
                 fractional values (between 0 and 1) act as a proportional failure threshold, where
                 0.15 means that 15 percent of failing test units results in an overall test failure.

## 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.
Such a table mutation is isolated in scope to the validation step(s) produced by the validation func-
tion call. Using **dplyr** code is suggested here since the statements can be translated to SQL if
necessary. The code is most easily supplied as a one-sided **R** formula (using a leading ~). In
the formula representation, the . serves as the input data table to be transformed (e.g., ~ . %>%
dplyr::mutate(col_a = col_b + 10)). Alternatively, a function could instead be supplied (e.g.,
function(x) dplyr::mutate(x,col_a = col_b + 10)).

Often, we will want to specify actions for the validation. This argument, present in every vali-
dation 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 of test units failed, or, an absolute value), often using the warn_at ar-
gument. 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

For the validation function, the return value is either a ptblank_agent object or a table object (depending on whether an agent object or a table was passed to x). The expectation function invisibly returns its input but, in the context of testing data, the function is called primarily for its potential side-effects (e.g., signaling failure). The test function returns a logical value.

### Function ID

2-9

### See Also

The analogue to this function: col_vals_not_in_set().

Other validation 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_expr(), 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

```
# The `small_table` dataset in the
# package will be used to validate that
# column values are part of a given set

# A: Using an `agent` with validation
#    functions and then `interrogate()`

# Validate that values in column `f`
# are all part of the set of values
# containing `low`, `mid`, and `high`
agent <-
  create_agent(small_table) %>%
  col_vals_in_set(
    vars(f), c("low", "mid", "high")
  ) %>%
  interrogate()

# Determine if this validation
# had no failing test units (there
# are 13 test units, one for each row)
all_passed(agent)

# Calling `agent` in the console
# prints the agent's report; but we
# can get a `gt_tbl` object directly
# with `get_agent_report(agent)`
```

```
# B: Using the validation function
#    directly on the data (no `agent`)

# This way of using validation functions
# acts as a data filter: data is passed
# through but should `stop()` if there
# is a single test unit failing; the
# behavior of side effects can be
# customized with the `actions` option
small_table %>%
  col_vals_in_set(
    vars(f), c("low", "mid", "high")
  ) %>%
  dplyr::pull(f) %>%
  unique()

# C: Using the expectation function

# With the `expect_*()` form, we would
# typically perform one validation at a
# time; this is primarily used in
# testthat tests
expect_col_vals_in_set(
  small_table,
  vars(f), c("low", "mid", "high")
)

# D: Using the test function

# With the `test_*()` form, we should
# get a single logical value returned
# to us
small_table %>%
  test_col_vals_in_set(
    vars(f), c("low", "mid", "high")
  )
```

---

col_vals_lt                    *Are column data less than a specified value?*

---

### Description

The col_vals_lt() validation function, the expect_col_vals_lt() expectation function, and
the test_col_vals_lt() test function all check whether column values in a table 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(). The validation
function can be used directly on a data table or with an *agent* object (technically, a ptblank_agent
object) whereas the expectation and test functions can only be used with a data table. The types

of data tables that can be used include data frames, tibbles, database tables (`tbl_dbi`), and Spark DataFrames (`tbl_spark`). Each validation step or expectation 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_lt(
  x,
  columns,
  value,
  na_pass = FALSE,
  preconditions = NULL,
  actions = NULL,
  step_id = NULL,
  label = NULL,
  brief = NULL,
  active = TRUE
)

expect_col_vals_lt(
  object,
  columns,
  value,
  na_pass = FALSE,
  preconditions = NULL,
  threshold = 1
)

test_col_vals_lt(
  object,
  columns,
  value,
  na_pass = FALSE,
  preconditions = NULL,
  threshold = 1
)
```

## Arguments

| | |
|---|---|
| x | A data frame, tibble (`tbl_df` or `tbl_dbi`), Spark DataFrame (`tbl_spark`), or, an agent object of class ptblank_agent that is created with [create_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 considered as passing test units? 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 . serves as the input data table to be transformed (e.g., ~ .
                 %>% dplyr::mutate(col = col + 10).

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.

step_id          One or more optional identifiers for the single or multiple validation steps gen-
                 erated from calling a validation function. The use of step IDs serves to distin-
                 guish validation steps from each other and provide an opportunity for supplying
                 a more meaningful label compared to the step index. By default this is NULL,
                 and **pointblank** will automatically generate the step ID value (based on the step
                 index) in this case. One or more values can be provided, and the exact number
                 of ID values should (1) match the number of validation steps that the validation
                 function call will produce (influenced by the number of columns provided), (2)
                 be an ID string not used in any previous validation step, and (3) be a vector with
                 unique values.

label            An optional label for the validation step. This label appears in the agent report
                 and for the best appearance it should be kept short.

brief            An optional, text-based description for the validation step. If nothing is provided
                 here then an *autobrief* is generated by the agent, using the language provided
                 in create_agent()'s lang argument (which defaults to "en" or English). The
                 *autobrief* incorporates details of the validation step so it's often the preferred
                 option in most cases (where a label might be better suited to succinctly describe
                 the validation).

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.

object           A data frame, tibble (tbl_df or tbl_dbi), or Spark DataFrame (tbl_spark)
                 that serves as the target table for the expectation function or the test function.

threshold        A simple failure threshold value for use with the expectation (expect_) and
                 the test (test_) function variants. By default, this is set to 1 meaning that any
                 single unit of failure in data validation results in an overall test failure. Whole
                 numbers beyond 1 indicate that any failing units up to that absolute threshold
                 value will result in a succeeding **testthat** test or evaluate to TRUE. Likewise,
                 fractional values (between 0 and 1) act as a proportional failure threshold, where
                 0.15 means that 15 percent of failing test units results in an overall test failure.

### 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 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. Such a table mutation is isolated in scope to the validation step(s) produced by the validation function call. Using **dplyr** code is suggested here since the statements can be translated to SQL if necessary. The code is most easily supplied as a one-sided **R** formula (using a leading ~). In the formula representation, the . serves as the input data table to be transformed (e.g., `~ . %>% dplyr::mutate(col_a = col_b + 10)`). Alternatively, a function could instead be supplied (e.g., `function(x) dplyr::mutate(x,col_a = col_b + 10)`).

Often, we will want to specify `actions` for the validation. This argument, present in every validation 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 of 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

For the validation function, the return value is either a `ptblank_agent` object or a table object (depending on whether an agent object or a table was passed to x). The expectation function invisibly returns its input but, in the context of testing data, the function is called primarily for its potential side-effects (e.g., signaling failure). The test function returns a logical value.

### Function ID

2-1

### See Also

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

Other validation 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_expr()](#), [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**

```
# For all of the examples here, we'll
# use a simple table with three numeric
# columns (`a`, `b`, and `c`) and three
# character columns (`d`, `e`, and `f`)
tbl <-
  dplyr::tibble(
      a = c(5, 5, 5, 5, 5, 5),
      b = c(1, 1, 1, 2, 2, 2),
      c = c(1, 1, 1, 2, 3, 4),
      d = LETTERS[a],
      e = LETTERS[b],
      f = LETTERS[c]
  )

tbl

# A: Using an `agent` with validation
#    functions and then `interrogate()`

# Validate that values in column `c`
# are all less than the value of `5`
agent <-
  create_agent(tbl) %>%
  col_vals_lt(vars(c), 5) %>%
  interrogate()

# Determine if this validation
# had no failing test units (there
# are 6 test units, one for each row)
all_passed(agent)

# Calling `agent` in the console
# prints the agent's report; but we
# can get a `gt_tbl` object directly
# with `get_agent_report(agent)`

# B: Using the validation function
#    directly on the data (no `agent`)

# This way of using validation functions
# acts as a data filter: data is passed
# through but should `stop()` if there
# is a single test unit failing; the
# behavior of side effects can be
# customized with the `actions` option
tbl %>%
  col_vals_lt(vars(c), 5) %>%
  dplyr::pull(c)

# C: Using the expectation function
```

```
# With the `expect_*()` form, we would
# typically perform one validation at a
# time; this is primarily used in
# testthat tests
expect_col_vals_lt(tbl, vars(c), 5)

# D: Using the test function

# With the `test_*()` form, we should
# get a single logical value returned
# to us
test_col_vals_lt(tbl, vars(c), 5)
```

---

col_vals_lte                   *Are column data less than or equal to a specified value?*

---

#### Description

The col_vals_lte() validation function, the expect_col_vals_lte() expectation function, and
the test_col_vals_lte() test function all check whether column values in a table 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(). The
validation step function can be used directly on a data table or with an *agent* object (technically, a
ptblank_agent object) whereas the expectation and test functions can only be used with a data ta-
ble. The types of data tables that can be used include data frames, tibbles, database tables (tbl_dbi),
and Spark DataFrames (tbl_spark). Each validation step or expectation will operate over the num-
ber 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,
  step_id = NULL,
  label = NULL,
  brief = NULL,
  active = TRUE
)

expect_col_vals_lte(
  object,
  columns,
```

```
  value,
  na_pass = FALSE,
  preconditions = NULL,
  threshold = 1
)

test_col_vals_lte(
  object,
  columns,
  value,
  na_pass = FALSE,
  preconditions = NULL,
  threshold = 1
)
```

## Arguments

| | |
|---|---|
| x | A data frame, tibble (tbl_df or tbl_dbi), Spark DataFrame (tbl_spark), or, an agent object of class ptblank_agent that is created with create_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 considered as passing test units? 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 . serves as the input data table to be transformed (e.g., ~ . %>% dplyr::mutate(col = col + 10). |
| 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. |
| step_id | One or more optional identifiers for the single or multiple validation steps gen-erated from calling a validation function. The use of step IDs serves to distin-guish validation steps from each other and provide an opportunity for supplying a more meaningful label compared to the step index. By default this is NULL, and **pointblank** will automatically generate the step ID value (based on the step index) in this case. One or more values can be provided, and the exact number of ID values should (1) match the number of validation steps that the validation function call will produce (influenced by the number of columns provided), (2) be an ID string not used in any previous validation step, and (3) be a vector with unique values. |
| label | An optional label for the validation step. This label appears in the agent report and for the best appearance it should be kept short. |
| brief | An optional, text-based description for the validation step. If nothing is provided here then an *autobrief* is generated by the agent, using the language provided |

in [`create_agent()`](#)'s lang argument (which defaults to "en" or English). The *autobrief* incorporates details of the validation step so it's often the preferred option in most cases (where a label might be better suited to succinctly describe the validation).

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.

object          A data frame, tibble (tbl_df or tbl_dbi), or Spark DataFrame (tbl_spark) that serves as the target table for the expectation function or the test function.

threshold       A simple failure threshold value for use with the expectation (expect_) and the test (test_) function variants. By default, this is set to 1 meaning that any single unit of failure in data validation results in an overall test failure. Whole numbers beyond 1 indicate that any failing units up to that absolute threshold value will result in a succeeding **testthat** test or evaluate to TRUE. Likewise, fractional values (between 0 and 1) act as a proportional failure threshold, where 0.15 means that 15 percent of failing test units results in an overall test failure.

## 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 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. Such a table mutation is isolated in scope to the validation step(s) produced by the validation function call. Using **dplyr** code is suggested here since the statements can be translated to SQL if necessary. The code is most easily supplied as a one-sided **R** formula (using a leading ~). In the formula representation, the . serves as the input data table to be transformed (e.g., ~ . %>% dplyr::mutate(col_a = col_b + 10)). Alternatively, a function could instead be supplied (e.g., function(x) dplyr::mutate(x, col_a = col_b + 10)).

Often, we will want to specify actions for the validation. This argument, present in every validation 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 of 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

For the validation function, the return value is either a ptblank_agent object or a table object (depending on whether an agent object or a table was passed to x). The expectation function invisibly returns its input but, in the context of testing data, the function is called primarily for its potential side-effects (e.g., signaling failure). The test function returns a logical value.

## Function ID

2-2

## See Also

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

Other validation 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_expr(), 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

```
# For all of the examples here, we'll
# use a simple table with three numeric
# columns (`a`, `b`, and `c`) and three
# character columns (`d`, `e`, and `f`)
tbl <-
  dplyr::tibble(
      a = c(5, 5, 5, 5, 5, 5),
      b = c(1, 1, 1, 2, 2, 2),
      c = c(1, 1, 1, 2, 3, 4),
      d = LETTERS[a],
      e = LETTERS[b],
      f = LETTERS[c]
  )

tbl

# A: Using an `agent` with validation
#    functions and then `interrogate()`

# Validate that values in column `c`
# are all less than or equal to the
# value of `4`
agent <-
  create_agent(tbl) %>%
```

```
    col_vals_lte(vars(c), 4) %>%
    interrogate()

# Determine if this validation
# had no failing test units (there
# are 6 test units, one for each row)
all_passed(agent)

# Calling `agent` in the console
# prints the agent's report; but we
# can get a `gt_tbl` object directly
# with `get_agent_report(agent)`

# B: Using the validation function
#    directly on the data (no `agent`)

# This way of using validation functions
# acts as a data filter: data is passed
# through but should `stop()` if there
# is a single test unit failing; the
# behavior of side effects can be
# customized with the `actions` option
tbl %>%
  col_vals_lte(vars(c), 4) %>%
  dplyr::pull(c)

# C: Using the expectation function

# With the `expect_*()` form, we would
# typically perform one validation at a
# time; this is primarily used in
# testthat tests
expect_col_vals_lte(tbl, vars(c), 4)

# D: Using the test function

# With the `test_*()` form, we should
# get a single logical value returned
# to us
test_col_vals_lte(tbl, vars(c), 4)
```

---

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

---

**Description**

The col_vals_not_between() validation function, the expect_col_vals_not_between() expectation function, and the test_col_vals_not_between() test function all check whether column values in a table *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 functions. The validation func-
tion can be used directly on a data table or with an *agent* object (technically, a ptblank_agent
object) whereas the expectation and test functions can only be used with a data table. The types
of data tables that can be used include data frames, tibbles, database tables (tbl_dbi), and Spark
DataFrames (tbl_spark). Each validation step or expectation 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_between(
  x,
  columns,
  left,
  right,
  inclusive = c(TRUE, TRUE),
  na_pass = FALSE,
  preconditions = NULL,
  actions = NULL,
  step_id = NULL,
  label = NULL,
  brief = NULL,
  active = TRUE
)

expect_col_vals_not_between(
  object,
  columns,
  left,
  right,
  inclusive = c(TRUE, TRUE),
  na_pass = FALSE,
  preconditions = NULL,
  threshold = 1
)

test_col_vals_not_between(
  object,
  columns,
  left,
  right,
  inclusive = c(TRUE, TRUE),
  na_pass = FALSE,
  preconditions = NULL,
  threshold = 1
```

)

## Arguments

| | |
|---|---|
| x | A data frame, tibble (`tbl_df` or `tbl_dbi`), Spark DataFrame (`tbl_spark`), or, an agent object of class ptblank_agent that is created with [create_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 uppers 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 considered as passing test units? 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 . serves as the input data table to be transformed (e.g., ~ . %>% dplyr::mutate(col = col + 10). |
| 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. |
| step_id | One or more optional identifiers for the single or multiple validation steps generated from calling a validation function. The use of step IDs serves to distinguish validation steps from each other and provide an opportunity for supplying a more meaningful label compared to the step index. By default this is `NULL`, and **pointblank** will automatically generate the step ID value (based on the step index) in this case. One or more values can be provided, and the exact number of ID values should (1) match the number of validation steps that the validation function call will produce (influenced by the number of `columns` provided), (2) be an ID string not used in any previous validation step, and (3) be a vector with unique values. |
| label | An optional label for the validation step. This label appears in the agent report and for the best appearance it should be kept short. |
| brief | An optional, text-based description for the validation step. If nothing is provided here then an *autobrief* is generated by the agent, using the language provided in [create_agent()]()'s lang argument (which defaults to "en" or English). The *autobrief* incorporates details of the validation step so it's often the preferred option in most cases (where a label might be better suited to succinctly describe the validation). |
| 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`. |

object              A data frame, tibble (`tbl_df` or `tbl_dbi`), or Spark DataFrame (`tbl_spark`)
                    that serves as the target table for the expectation function or the test function.

threshold           A simple failure threshold value for use with the expectation (`expect_`) and
                    the test (`test_`) function variants. By default, this is set to 1 meaning that any
                    single unit of failure in data validation results in an overall test failure. Whole
                    numbers beyond 1 indicate that any failing units up to that absolute threshold
                    value will result in a succeeding **testthat** test or evaluate to `TRUE`. Likewise,
                    fractional values (between `0` and `1`) act as a proportional failure threshold, where
                    `0.15` means that 15 percent of failing test units results in an overall test failure.

## 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 function supports special handling of `NA` values. The `na_pass` argument will deter-
mine 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.
Such a table mutation is isolated in scope to the validation step(s) produced by the validation func-
tion call. Using **dplyr** code is suggested here since the statements can be translated to SQL if
necessary. The code is most easily supplied as a one-sided **R** formula (using a leading ~). In
the formula representation, the `.` serves as the input data table to be transformed (e.g., `~ . %>%`
`dplyr::mutate(col_a = col_b + 10)`). Alternatively, a function could instead be supplied (e.g.,
`function(x) dplyr::mutate(x, col_a = col_b + 10)`).

Often, we will want to specify `actions` for the validation. This argument, present in every vali-
dation 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 of test units failed, or, an absolute value), often using the `warn_at` ar-
gument. 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

For the validation function, the return value is either a `ptblank_agent` object or a table object (de-
pending on whether an agent object or a table was passed to x). The expectation function invisibly
returns its input but, in the context of testing data, the function is called primarily for its potential
side-effects (e.g., signaling failure). The test function returns a logical value.

**Function ID**

2-8

**See Also**

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

Other validation 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_expr()`, `col_vals_gte()`, `col_vals_gt()`, `col_vals_in_set()`, `col_vals_lte()`, `col_vals_lt()`, `col_vals_not_equal()`, `col_vals_not_in_set()`, `col_vals_not_null()`, `col_vals_null()`, `col_vals_regex()`, `conjointly()`, `rows_distinct()`

**Examples**

```
# The `small_table` dataset in the
# package has a column of numeric
# values in `c` (there are a few NAs
# in that column); the following
# examples will validate the values
# in that numeric column

# A: Using an `agent` with validation
#    functions and then `interrogate()`

# Validate that values in column `c`
# are all between `10` and `20`; because
# there are NA values, we'll choose to
# let those pass validation by setting
# `na_pass = TRUE`
agent <-
  create_agent(small_table) %>%
  col_vals_not_between(
    vars(c), 10, 20, na_pass = TRUE
  ) %>%
  interrogate()

# Determine if this validation
# had no failing test units (there
# are 13 test units, one for each row)
all_passed(agent)

# Calling `agent` in the console
# prints the agent's report; but we
# can get a `gt_tbl` object directly
# with `get_agent_report(agent)`

# B: Using the validation function
#    directly on the data (no `agent`)

# This way of using validation functions
# acts as a data filter: data is passed
```

```
# through but should `stop()` if there
# is a single test unit failing; the
# behavior of side effects can be
# customized with the `actions` option
small_table %>%
  col_vals_not_between(
    vars(c), 10, 20, na_pass = TRUE
  ) %>%
  dplyr::pull(c)

# C: Using the expectation function

# With the `expect_*()` form, we would
# typically perform one validation at a
# time; this is primarily used in
# testthat tests
expect_col_vals_not_between(
  small_table, vars(c), 10, 20,
  na_pass = TRUE
)

# D: Using the test function

# With the `test_*()` form, we should
# get a single logical value returned
# to us
small_table %>%
  test_col_vals_not_between(
    vars(c), 10, 20,
    na_pass = TRUE
  )

# An additional note on the bounds for
# this function: they are inclusive by
# default; we can modify the
# inclusiveness of the upper and lower
# bounds with the `inclusive` option,
# which is a length-2 logical vector

# In changing the lower bound to be
# `9` and making it non-inclusive, we
# get `TRUE` since although two values
# are `9` and they fall outside of the
# lower (or left) bound (and any values
# 'not between' count as passing test
# units)
small_table %>%
  test_col_vals_not_between(
    vars(c), 9, 20,
    inclusive = c(FALSE, TRUE),
    na_pass = TRUE
  )
```

---

col_vals_not_equal *Are column data not equal to a specified value?*

---

## Description

The `col_vals_not_equal()` validation function, the `expect_col_vals_not_equal()` expecta-
tion function, and the `test_col_vals_not_equal()` test function all check whether column values
in a table *are not* equal to a specified `value`. The validation step function can be used directly on
a data table or with an *agent* object (technically, a `ptblank_agent` object) whereas the expectation
and test functions can only be used with a data table. The types of data tables that can be used
include data frames, tibbles, database tables (`tbl_dbi`), and Spark DataFrames (`tbl_spark`). Each
validation step or expectation 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_equal(
  x,
  columns,
  value,
  na_pass = FALSE,
  preconditions = NULL,
  actions = NULL,
  step_id = NULL,
  label = NULL,
  brief = NULL,
  active = TRUE
)

expect_col_vals_not_equal(
  object,
  columns,
  value,
  na_pass = FALSE,
  preconditions = NULL,
  threshold = 1
)

test_col_vals_not_equal(
  object,
  columns,
  value,
  na_pass = FALSE,
  preconditions = NULL,
  threshold = 1
)
```

## Arguments

| | |
|---|---|
| x | A data frame, tibble (tbl_df or tbl_dbi), Spark DataFrame (tbl_spark), or, an agent object of class ptblank_agent that is created with create_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 considered as passing test units? 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 . serves as the input data table to be transformed (e.g., ~ . %>% dplyr::mutate(col = col + 10). |
| 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. |
| step_id | One or more optional identifiers for the single or multiple validation steps generated from calling a validation function. The use of step IDs serves to distinguish validation steps from each other and provide an opportunity for supplying a more meaningful label compared to the step index. By default this is NULL, and **pointblank** will automatically generate the step ID value (based on the step index) in this case. One or more values can be provided, and the exact number of ID values should (1) match the number of validation steps that the validation function call will produce (influenced by the number of columns provided), (2) be an ID string not used in any previous validation step, and (3) be a vector with unique values. |
| label | An optional label for the validation step. This label appears in the agent report and for the best appearance it should be kept short. |
| brief | An optional, text-based description for the validation step. If nothing is provided here then an *autobrief* is generated by the agent, using the language provided in create_agent()'s lang argument (which defaults to "en" or English). The *autobrief* incorporates details of the validation step so it's often the preferred option in most cases (where a label might be better suited to succinctly describe the validation). |
| 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. |
| object | A data frame, tibble (tbl_df or tbl_dbi), or Spark DataFrame (tbl_spark) that serves as the target table for the expectation function or the test function. |
| threshold | A simple failure threshold value for use with the expectation (expect_) and the test (test_) function variants. By default, this is set to 1 meaning that any single unit of failure in data validation results in an overall test failure. Whole numbers beyond 1 indicate that any failing units up to that absolute threshold |

value will result in a succeeding **testthat** test or evaluate to TRUE. Likewise, fractional values (between 0 and 1) act as a proportional failure threshold, where 0.15 means that 15 percent of failing test units results in an overall test failure.

### 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 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. Such a table mutation is isolated in scope to the validation step(s) produced by the validation function call. Using **dplyr** code is suggested here since the statements can be translated to SQL if necessary. The code is most easily supplied as a one-sided **R** formula (using a leading ~). In the formula representation, the . serves as the input data table to be transformed (e.g., ~ . %>% dplyr::mutate(col_a = col_b + 10)). Alternatively, a function could instead be supplied (e.g., function(x) dplyr::mutate(x,col_a = col_b + 10)).

Often, we will want to specify actions for the validation. This argument, present in every validation 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 of 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

For the validation function, the return value is either a ptblank_agent object or a table object (depending on whether an agent object or a table was passed to x). The expectation function invisibly returns its input but, in the context of testing data, the function is called primarily for its potential side-effects (e.g., signaling failure). The test function returns a logical value.

### Function ID

2-4

**See Also**

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

Other validation 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_expr()`, `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**

```
# For all of the examples here, we'll
# use a simple table with three numeric
# columns (`a`, `b`, and `c`) and three
# character columns (`d`, `e`, and `f`)
tbl <-
  dplyr::tibble(
    a = c(5, 5, 5, 5, 5, 5),
    b = c(1, 1, 1, 2, 2, 2),
    c = c(1, 1, 1, 2, 2, 2),
    d = LETTERS[c(1:3, 5:7)],
    e = LETTERS[c(1:6)],
    f = LETTERS[c(1:6)]
  )

tbl

# A: Using an `agent` with validation
#    functions and then `interrogate()`

# Validate that values in column `a`
# are all *not* equal to the value
# of `6`
agent <-
  create_agent(tbl) %>%
  col_vals_not_equal(vars(a), 6) %>%
  interrogate()

# Determine if this validation
# had no failing test units (there
# are 6 test units, one for each row)
all_passed(agent)

# Calling `agent` in the console
# prints the agent's report; but we
# can get a `gt_tbl` object directly
# with `get_agent_report(agent)`

# B: Using the validation function
#    directly on the data (no `agent`)

# This way of using validation functions
```

```
# acts as a data filter: data is passed
# through but should `stop()` if there
# is a single test unit failing; the
# behavior of side effects can be
# customized with the `actions` option
tbl %>%
  col_vals_not_equal(vars(a), 6) %>%
  dplyr::pull(a)

# C: Using the expectation function

# With the `expect_*()` form, we would
# typically perform one validation at a
# time; this is primarily used in
# testthat tests
expect_col_vals_not_equal(tbl, vars(a), 6)

# D: Using the test function

# With the `test_*()` form, we should
# get a single logical value returned
# to us
test_col_vals_not_equal(tbl, vars(a), 6)
```

---

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

---

### Description

The col_vals_not_in_set() validation function, the expect_col_vals_not_in_set() expectation function, and the test_col_vals_not_in_set() test function all check whether column values in a table *are not part* of a specified set of values. The validation function can be used directly on a data table or with an *agent* object (technically, a ptblank_agent object) whereas the expectation and test functions can only be used with a data table. The types of data tables that can be used include data frames, tibbles, database tables (tbl_dbi), and Spark DataFrames (tbl_spark). Each validation step or expectation 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,
  step_id = NULL,
  label = NULL,
```

```
    brief = NULL,
    active = TRUE
)

expect_col_vals_not_in_set(
    object,
    columns,
    set,
    preconditions = NULL,
    threshold = 1
)

test_col_vals_not_in_set(
    object,
    columns,
    set,
    preconditions = NULL,
    threshold = 1
)
```

## Arguments

| | |
|---|---|
| x | A data frame, tibble (tbl_df or tbl_dbi), Spark DataFrame (tbl_spark), or, an agent object of class ptblank_agent that is created with [create_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 . serves as the input data table to be transformed (e.g., ~ . %>% dplyr::mutate(col = col + 10). |
| 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. |
| step_id | One or more optional identifiers for the single or multiple validation steps generated from calling a validation function. The use of step IDs serves to distinguish validation steps from each other and provide an opportunity for supplying a more meaningful label compared to the step index. By default this is NULL, and **pointblank** will automatically generate the step ID value (based on the step index) in this case. One or more values can be provided, and the exact number of ID values should (1) match the number of validation steps that the validation function call will produce (influenced by the number of columns provided), (2) be an ID string not used in any previous validation step, and (3) be a vector with unique values. |
| label | An optional label for the validation step. This label appears in the agent report and for the best appearance it should be kept short. |

brief       An optional, text-based description for the validation step. If nothing is provided here then an *autobrief* is generated by the agent, using the language provided in [`create_agent()`](#)'s lang argument (which defaults to "en" or English). The *autobrief* incorporates details of the validation step so it's often the preferred option in most cases (where a `label` might be better suited to succinctly describe the validation).

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`.

object       A data frame, tibble (`tbl_df` or `tbl_dbi`), or Spark DataFrame (`tbl_spark`) that serves as the target table for the expectation function or the test function.

threshold       A simple failure threshold value for use with the expectation (`expect_`) and the test (`test_`) function variants. By default, this is set to `1` meaning that any single unit of failure in data validation results in an overall test failure. Whole numbers beyond `1` indicate that any failing units up to that absolute threshold value will result in a succeeding **testthat** test or evaluate to `TRUE`. Likewise, fractional values (between `0` and `1`) act as a proportional failure threshold, where `0.15` means that 15 percent of failing test units results in an overall test failure.

## 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. Such a table mutation is isolated in scope to the validation step(s) produced by the validation function call. Using **dplyr** code is suggested here since the statements can be translated to SQL if necessary. The code is most easily supplied as a one-sided **R** formula (using a leading `~`). In the formula representation, the `.` serves as the input data table to be transformed (e.g., `~ . %>% dplyr::mutate(col_a = col_b + 10)`). Alternatively, a function could instead be supplied (e.g., `function(x) dplyr::mutate(x, col_a = col_b + 10)`).

Often, we will want to specify `actions` for the validation. This argument, present in every validation 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 of 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

For the validation function, the return value is either a ptblank_agent object or a table object (depending on whether an agent object or a table was passed to x). The expectation function invisibly returns its input but, in the context of testing data, the function is called primarily for its potential side-effects (e.g., signaling failure). The test function returns a logical value.

## Function ID

2-10

## See Also

The analogue to this function: col_vals_in_set().

Other validation 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_expr(), col_vals_gte(), col_vals_gt(), col_vals_in_set(), col_vals_lte(), col_vals_lt(), col_vals_not_between(), col_vals_not_equal(), col_vals_not_null(), col_vals_null(), col_vals_regex(), conjointly(), rows_distinct()

## Examples

```
# The `small_table` dataset in the
# package will be used to validate that
# column values are part of a given set

# A: Using an `agent` with validation
#    functions and then `interrogate()`

# Validate that values in column `f`
# contain none of the values `lows`,
# `mids`, and `highs`
agent <-
  create_agent(small_table) %>%
  col_vals_not_in_set(
    vars(f), c("lows", "mids", "highs")
  ) %>%
  interrogate()

# Determine if this validation
# had no failing test units (there
# are 13 test units, one for each row)
all_passed(agent)

# Calling `agent` in the console
# prints the agent's report; but we
# can get a `gt_tbl` object directly
# with `get_agent_report(agent)`
```

```
# B: Using the validation function
#    directly on the data (no `agent`)

# This way of using validation functions
# acts as a data filter: data is passed
# through but should `stop()` if there
# is a single test unit failing; the
# behavior of side effects can be
# customized with the `actions` option
small_table %>%
  col_vals_not_in_set(
    vars(f), c("lows", "mids", "highs")
  ) %>%
  dplyr::pull(f) %>%
  unique()

# C: Using the expectation function

# With the `expect_*()` form, we would
# typically perform one validation at a
# time; this is primarily used in
# testthat tests
expect_col_vals_not_in_set(
  small_table,
  vars(f), c("lows", "mids", "highs")
)

# D: Using the test function

# With the `test_*()` form, we should
# get a single logical value returned
# to us
small_table %>%
  test_col_vals_not_in_set(
    vars(f), c("lows", "mids", "highs")
  )
```

---

col_vals_not_null          *Are column data not* NULL/NA*?*

---

### Description

The col_vals_not_null() validation function, the expect_col_vals_not_null() expectation
function, and the test_col_vals_not_null() test function all check whether column values in a
table *are not* NA values or, in the database context, *not* NULL values. The validation function can be
used directly on a data table or with an *agent* object (technically, a ptblank_agent object) whereas
the expectation and test functions can only be used with a data table. The types of data tables
that can be used include data frames, tibbles, database tables (tbl_dbi), and Spark DataFrames

(tbl_spark). Each validation step or expectation 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_null(
  x,
  columns,
  preconditions = NULL,
  actions = NULL,
  step_id = NULL,
  label = NULL,
  brief = NULL,
  active = TRUE
)

expect_col_vals_not_null(object, columns, preconditions = NULL, threshold = 1)

test_col_vals_not_null(object, columns, preconditions = NULL, threshold = 1)
```

## Arguments

x                 A data frame, tibble (tbl_df or tbl_dbi), Spark DataFrame (tbl_spark), or, an agent object of class ptblank_agent that is created with [create_agent()](create_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 . serves as the input data table to be transformed (e.g., ~ . %>% dplyr::mutate(col = col + 10).

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()](action_levels()) helper function.

step_id           One or more optional identifiers for the single or multiple validation steps generated from calling a validation function. The use of step IDs serves to distinguish validation steps from each other and provide an opportunity for supplying a more meaningful label compared to the step index. By default this is NULL, and **pointblank** will automatically generate the step ID value (based on the step index) in this case. One or more values can be provided, and the exact number of ID values should (1) match the number of validation steps that the validation function call will produce (influenced by the number of columns provided), (2) be an ID string not used in any previous validation step, and (3) be a vector with unique values.

label             An optional label for the validation step. This label appears in the agent report and for the best appearance it should be kept short.

brief             An optional, text-based description for the validation step. If nothing is provided here then an *autobrief* is generated by the agent, using the language provided

in [`create_agent()`](#)'s lang argument (which defaults to `"en"` or English). The *autobrief* incorporates details of the validation step so it's often the preferred option in most cases (where a `label` might be better suited to succinctly describe the validation).

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`.

object    A data frame, tibble (`tbl_df` or `tbl_dbi`), or Spark DataFrame (`tbl_spark`) that serves as the target table for the expectation function or the test function.

threshold    A simple failure threshold value for use with the expectation (`expect_`) and the test (`test_`) function variants. By default, this is set to `1` meaning that any single unit of failure in data validation results in an overall test failure. Whole numbers beyond `1` indicate that any failing units up to that absolute threshold value will result in a succeeding **testthat** test or evaluate to `TRUE`. Likewise, fractional values (between `0` and `1`) act as a proportional failure threshold, where `0.15` means that 15 percent of failing test units results in an overall test failure.

## 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. Such a table mutation is isolated in scope to the validation step(s) produced by the validation function call. Using **dplyr** code is suggested here since the statements can be translated to SQL if necessary. The code is most easily supplied as a one-sided **R** formula (using a leading ~). In the formula representation, the `.` serves as the input data table to be transformed (e.g., `~ . %>% dplyr::mutate(col_a = col_b + 10)`). Alternatively, a function could instead be supplied (e.g., `function(x) dplyr::mutate(x,col_a = col_b + 10)`).

Often, we will want to specify `actions` for the validation. This argument, present in every validation 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 of 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**

For the validation function, the return value is either a ptblank_agent object or a table object (depending on whether an agent object or a table was passed to x). The expectation function invisibly returns its input but, in the context of testing data, the function is called primarily for its potential side-effects (e.g., signaling failure). The test function returns a logical value.

**Function ID**

2-12

**See Also**

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

Other validation 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_expr()`, `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**

```
# For all examples here, we'll use
# a simple table with four columns:
# `a`, `b`, `c`, and `d`
tbl <-
  dplyr::tibble(
    a = c( 5,  7,  6,  5,  8),
    b = c( 7,  1,  0,  0,  0),
    c = c(NA, NA, NA, NA, NA),
    d = c(35, 23, NA, NA, NA)
  )

tbl

# A: Using an `agent` with validation
#    functions and then `interrogate()`

# Validate that all values in column
# `b` are *not* NA (they would be
# non-NULL in a database context, which
# isn't the case here)
agent <-
  create_agent(tbl) %>%
  col_vals_not_null(vars(b)) %>%
  interrogate()

# Determine if this validation
# had no failing test units (there
# are 5 test units, one for each row)
all_passed(agent)
```

```
# Calling `agent` in the console
# prints the agent's report; but we
# can get a `gt_tbl` object directly
# with `get_agent_report(agent)`

# B: Using the validation function
#     directly on the data (no `agent`)

# This way of using validation functions
# acts as a data filter: data is passed
# through but should `stop()` if there
# is a single test unit failing; the
# behavior of side effects can be
# customized with the `actions` option
tbl %>%
  col_vals_not_null(vars(b)) %>%
  dplyr::pull(b)

# C: Using the expectation function

# With the `expect_*()` form, we would
# typically perform one validation at a
# time; this is primarily used in
# testthat tests
expect_col_vals_not_null(tbl, vars(b))

# D: Using the test function

# With the `test_*()` form, we should
# get a single logical value returned
# to us
tbl %>% test_col_vals_not_null(vars(b))
```

---

col_vals_null                 *Are column data* NULL/NA*?*

---

## Description

The col_vals_null() validation function, the expect_col_vals_null() expectation function, and the test_col_vals_null() test function all check whether column values in a table are NA values or, in the database context, NULL values. The validation function can be used directly on a data table or with an *agent* object (technically, a ptblank_agent object) whereas the expectation and test functions can only be used with a data table. The types of data tables that can be used include data frames, tibbles, database tables (tbl_dbi), and Spark DataFrames (tbl_spark). Each validation step or expectation will operate over the number of test units that is equal to the number of rows in the table (after any preconditions have been applied).

**Usage**

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

expect_col_vals_null(object, columns, preconditions = NULL, threshold = 1)

test_col_vals_null(object, columns, preconditions = NULL, threshold = 1)
```

**Arguments**

| | |
|---|---|
| x | A data frame, tibble (tbl_df or tbl_dbi), Spark DataFrame (tbl_spark), or, an agent object of class ptblank_agent that is created with create_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 . serves as the input data table to be transformed (e.g., ~ . %>% dplyr::mutate(col = col + 10). |
| 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. |
| step_id | One or more optional identifiers for the single or multiple validation steps generated from calling a validation function. The use of step IDs serves to distinguish validation steps from each other and provide an opportunity for supplying a more meaningful label compared to the step index. By default this is NULL, and **pointblank** will automatically generate the step ID value (based on the step index) in this case. One or more values can be provided, and the exact number of ID values should (1) match the number of validation steps that the validation function call will produce (influenced by the number of columns provided), (2) be an ID string not used in any previous validation step, and (3) be a vector with unique values. |
| label | An optional label for the validation step. This label appears in the agent report and for the best appearance it should be kept short. |
| brief | An optional, text-based description for the validation step. If nothing is provided here then an *autobrief* is generated by the agent, using the language provided in create_agent()'s lang argument (which defaults to "en" or English). The *autobrief* incorporates details of the validation step so it's often the preferred option in most cases (where a label might be better suited to succinctly describe the validation). |

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.

object
: A data frame, tibble (tbl_df or tbl_dbi), or Spark DataFrame (tbl_spark) that serves as the target table for the expectation function or the test function.

threshold
: A simple failure threshold value for use with the expectation (expect_) and the test (test_) function variants. By default, this is set to 1 meaning that any single unit of failure in data validation results in an overall test failure. Whole numbers beyond 1 indicate that any failing units up to that absolute threshold value will result in a succeeding **testthat** test or evaluate to TRUE. Likewise, fractional values (between 0 and 1) act as a proportional failure threshold, where 0.15 means that 15 percent of failing test units results in an overall test failure.

### 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. Such a table mutation is isolated in scope to the validation step(s) produced by the validation function call. Using **dplyr** code is suggested here since the statements can be translated to SQL if necessary. The code is most easily supplied as a one-sided **R** formula (using a leading ~). In the formula representation, the . serves as the input data table to be transformed (e.g., ~ . %>% dplyr::mutate(col_a = col_b + 10)). Alternatively, a function could instead be supplied (e.g., function(x) dplyr::mutate(x,col_a = col_b + 10)).

Often, we will want to specify actions for the validation. This argument, present in every validation 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 of 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

For the validation function, the return value is either a ptblank_agent object or a table object (depending on whether an agent object or a table was passed to x). The expectation function invisibly

returns its input but, in the context of testing data, the function is called primarily for its potential side-effects (e.g., signaling failure). The test function returns a logical value.

## Function ID

2-11

## See Also

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

Other validation 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_expr()`, `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

```
# For all examples here, we'll use
# a simple table with four columns:
# `a`, `b`, `c`, and `d`
tbl <-
  dplyr::tibble(
    a = c( 5,  7,  6,  5,  8),
    b = c( 7,  1,  0,  0,  0),
    c = c(NA, NA, NA, NA, NA),
    d = c(35, 23, NA, NA, NA)
  )

tbl

# A: Using an `agent` with validation
#    functions and then `interrogate()`

# Validate that all values in column
# `c` are NA (they would be NULL in a
# database context, which isn't the
# case here)
agent <-
  create_agent(tbl) %>%
  col_vals_null(vars(c)) %>%
  interrogate()

# Determine if this validation
# had no failing test units (there
# are 5 test units, one for each row)
all_passed(agent)

# Calling `agent` in the console
# prints the agent's report; but we
# can get a `gt_tbl` object directly
# with `get_agent_report(agent)`
```

```
# B: Using the validation function
#   directly on the data (no `agent`)

# This way of using validation functions
# acts as a data filter: data is passed
# through but should `stop()` if there
# is a single test unit failing; the
# behavior of side effects can be
# customized with the `actions` option
tbl %>%
  col_vals_null(vars(c)) %>%
  dplyr::pull(c)

# C: Using the expectation function

# With the `expect_*()` form, we would
# typically perform one validation at a
# time; this is primarily used in
# testthat tests
expect_col_vals_null(tbl, vars(c))

# D: Using the test function

# With the `test_*()` form, we should
# get a single logical value returned
# to us
tbl %>% test_col_vals_null(vars(c))
```

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

### Description

The col_vals_regex() validation function, the expect_col_vals_regex() expectation function,
and the test_col_vals_regex() test function all check whether column values in a table corre-
spond to a regex matching expression. The validation step function can be used directly on a data
table or with an *agent* object (technically, a ptblank_agent object) whereas the expectation and
test functions can only be used with a data table. The types of data tables that can be used include
data frames, tibbles, database tables (tbl_dbi), and Spark DataFrames (tbl_spark). Each valida-
tion step or expectation 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_regex(
  x,
  columns,
```

```
  regex,
  na_pass = FALSE,
  preconditions = NULL,
  actions = NULL,
  step_id = NULL,
  label = NULL,
  brief = NULL,
  active = TRUE
)

expect_col_vals_regex(
  object,
  columns,
  regex,
  na_pass = FALSE,
  preconditions = NULL,
  threshold = 1
)

test_col_vals_regex(
  object,
  columns,
  regex,
  na_pass = FALSE,
  preconditions = NULL,
  threshold = 1
)
```

## Arguments

| | |
|---|---|
| x | A data frame, tibble (`tbl_df` or `tbl_dbi`), Spark DataFrame (`tbl_spark`), or, an agent object of class `ptblank_agent` that is created with [`create_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 considered as passing test units? 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 . serves as the input data table to be transformed (e.g., ~ . %>% dplyr::mutate(col = col + 10). |
| 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. |
| step_id | One or more optional identifiers for the single or multiple validation steps generated from calling a validation function. The use of step IDs serves to distinguish validation steps from each other and provide an opportunity for supplying |

a more meaningful label compared to the step index. By default this is NULL, and **pointblank** will automatically generate the step ID value (based on the step index) in this case. One or more values can be provided, and the exact number of ID values should (1) match the number of validation steps that the validation function call will produce (influenced by the number of columns provided), (2) be an ID string not used in any previous validation step, and (3) be a vector with unique values.

label              An optional label for the validation step. This label appears in the agent report and for the best appearance it should be kept short.

brief              An optional, text-based description for the validation step. If nothing is provided here then an *autobrief* is generated by the agent, using the language provided in [create_agent()](#)'s lang argument (which defaults to "en" or English). The *autobrief* incorporates details of the validation step so it's often the preferred option in most cases (where a label might be better suited to succinctly describe the validation).

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.

object             A data frame, tibble (tbl_df or tbl_dbi), or Spark DataFrame (tbl_spark) that serves as the target table for the expectation function or the test function.

threshold          A simple failure threshold value for use with the expectation (expect_) and the test (test_) function variants. By default, this is set to 1 meaning that any single unit of failure in data validation results in an overall test failure. Whole numbers beyond 1 indicate that any failing units up to that absolute threshold value will result in a succeeding **testthat** test or evaluate to TRUE. Likewise, fractional values (between 0 and 1) act as a proportional failure threshold, where 0.15 means that 15 percent of failing test units results in an overall test failure.

### 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 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. Such a table mutation is isolated in scope to the validation step(s) produced by the validation function call. Using **dplyr** code is suggested here since the statements can be translated to SQL if necessary. The code is most easily supplied as a one-sided **R** formula (using a leading ~). In the formula representation, the . serves as the input data table to be transformed (e.g., ~ . %>%

dplyr::mutate(col_a = col_b + 10)). Alternatively, a function could instead be supplied (e.g., function(x) dplyr::mutate(x,col_a = col_b + 10)).

Often, we will want to specify actions for the validation. This argument, present in every validation 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 of 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

For the validation function, the return value is either a ptblank_agent object or a table object (depending on whether an agent object or a table was passed to x). The expectation function invisibly returns its input but, in the context of testing data, the function is called primarily for its potential side-effects (e.g., signaling failure). The test function returns a logical value.

## Function ID

2-13

## See Also

Other validation 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_expr(), 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

```
# The `small_table` dataset in the
# package has a character-based `b`
# column with values that adhere to
# a very particular pattern; the
# following examples will validate
# that that column abides by a regex
# pattern
small_table

# This is the regex pattern that will
# be used throughout
pattern <- "[0-9]-[a-z]{3}-[0-9]{3}"
```

```
# A: Using an `agent` with validation
#    functions and then `interrogate()`

# Validate that all values in column
# `b` match the regex `pattern`
agent <-
  create_agent(small_table) %>%
  col_vals_regex(vars(b), pattern) %>%
  interrogate()

# Determine if this validation
# had no failing test units (there
# are 13 test units, one for each row)
all_passed(agent)

# Calling `agent` in the console
# prints the agent's report; but we
# can get a `gt_tbl` object directly
# with `get_agent_report(agent)`

# B: Using the validation function
#    directly on the data (no `agent`)

# This way of using validation functions
# acts as a data filter: data is passed
# through but should `stop()` if there
# is a single test unit failing; the
# behavior of side effects can be
# customized with the `actions` option
small_table %>%
  col_vals_regex(vars(b), pattern) %>%
  dplyr::slice(1:5)

# C: Using the expectation function

# With the `expect_*()` form, we would
# typically perform one validation at a
# time; this is primarily used in
# testthat tests
expect_col_vals_regex(
  small_table,
  vars(b), pattern
)

# D: Using the test function

# With the `test_*()` form, we should
# get a single logical value returned
# to us
small_table %>%
  test_col_vals_regex(
    vars(b), pattern
  )
```

---

conjointly                    *Perform multiple rowwise validations for joint validity*

---

**Description**

The conjointly() validation function, the expect_conjointly() expectation function, and the test_conjointly() test function all check whether test units at each index (typically each row) all pass multiple validations with col_vals_*()-type functions. Because of the imposed constraint on the allowed validation functions, all test units are rows of the table (after any common preconditions have been applied). Each of the functions (composed with multiple validation function calls) ultimately perform 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 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). The validation function can be used directly on a data table or with an *agent* object (technically, a ptblank_agent object) whereas the expectation and test functions can only be used with a data table.

**Usage**

```
conjointly(
  x,
  ...,
  .list = list2(...),
  preconditions = NULL,
  actions = NULL,
  step_id = NULL,
  label = NULL,
  brief = NULL,
  active = TRUE
)

expect_conjointly(
  object,
  ...,
  .list = list2(...),
  preconditions = NULL,
  threshold = 1
)

test_conjointly(
  object,
  ...,
  .list = list2(...),
  preconditions = NULL,
```

```
    threshold = 1
)
```

## Arguments

| | |
|---|---|
| x | A data frame, tibble (tbl_df or tbl_dbi), Spark DataFrame (tbl_spark), or, an agent object of class ptblank_agent that is created with create_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 . serves as the input data table to be transformed (e.g., ~ . %>% dplyr::mutate(col = col + 10). |
| 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. |
| step_id | One or more optional identifiers for the single or multiple validation steps generated from calling a validation function. The use of step IDs serves to distinguish validation steps from each other and provide an opportunity for supplying a more meaningful label compared to the step index. By default this is NULL, and **pointblank** will automatically generate the step ID value (based on the step index) in this case. One or more values can be provided, and the exact number of ID values should (1) match the number of validation steps that the validation function call will produce (influenced by the number of columns provided), (2) be an ID string not used in any previous validation step, and (3) be a vector with unique values. |
| label | An optional label for the validation step. This label appears in the agent report and for the best appearance it should be kept short. |
| brief | An optional, text-based description for the validation step. If nothing is provided here then an *autobrief* is generated by the agent, using the language provided in create_agent()'s lang argument (which defaults to "en" or English). The *autobrief* incorporates details of the validation step so it's often the preferred option in most cases (where a label might be better suited to succinctly describe the validation). |
| 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. |
| object | A data frame, tibble (tbl_df or tbl_dbi), or Spark DataFrame (tbl_spark) that serves as the target table for the expectation function or the test function. |
| threshold | A simple failure threshold value for use with the expectation (expect_) and the test (test_) function variants. By default, this is set to 1 meaning that any |

single unit of failure in data validation results in an overall test failure. Whole numbers beyond 1 indicate that any failing units up to that absolute threshold value will result in a succeeding **testthat** test or evaluate to TRUE. Likewise, fractional values (between 0 and 1) act as a proportional failure threshold, where 0.15 means that 15 percent of failing test units results in an overall test failure.

## 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. Such a table mutation is isolated in scope to the validation step(s) produced by the validation function call. Using **dplyr** code is suggested here since the statements can be translated to SQL if necessary. The code is most easily supplied as a one-sided **R** formula (using a leading ~). In the formula representation, the . serves as the input data table to be transformed (e.g., ~ . %>% dplyr::mutate(col_a = col_b + 10)). Alternatively, a function could instead be supplied (e.g., function(x) dplyr::mutate(x,col_a = col_b + 10)).

Often, we will want to specify actions for the validation. This argument, present in every validation 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 of 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

For the validation function, the return value is either a ptblank_agent object or a table object (depending on whether an agent object or a table was passed to x). The expectation function invisibly returns its input but, in the context of testing data, the function is called primarily for its potential side-effects (e.g., signaling failure). The test function returns a logical value.

## Function ID

2-14

## See Also

Other validation 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_expr(), 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**

```
# For all examples here, we'll use
# a simple table with three numeric
# columns (`a`, `b`, and `c`); this is
# a very basic table but it'll be more
# useful when explaining things later
tbl <-
  dplyr::tibble(
    a = c(5, 2, 6),
    b = c(3, 4, 6),
    c = c(9, 8, 7)
  )

tbl

# A: Using an `agent` with validation
#    functions and then `interrogate()`

# Validate a number of things on a
# row-by-row basis using validation
# functions of the `col_vals*` type
# (all have the same number of test
# units): (1) values in `a` are less
# than `4`, (2) values in `c` are
# greater than the adjacent values in
# `a`, and (3) there aren't any NA
# values in `b`
agent <-
  create_agent(tbl = tbl) %>%
  conjointly(
    ~ col_vals_lt(., vars(a), 8),
    ~ col_vals_gt(., vars(c), vars(a)),
    ~ col_vals_not_null(., vars(b))
    ) %>%
  interrogate()

# Determine if this validation
# had no failing test units (there
# are 3 test units, one for each row)
all_passed(agent)

# Calling `agent` in the console
# prints the agent's report; but we
# can get a `gt_tbl` object directly
# with `get_agent_report(agent)`

# What's going on? Think of there being
# three parallel validations, each
```

```
# producing a column of `TRUE` or `FALSE`
# values (`pass` or `fail`) and line them
# up side-by-side, any rows with any
# `FALSE` values results in a conjoint
# `fail` test unit

# B: Using the validation function
#    directly on the data (no `agent`)

# This way of using validation functions
# acts as a data filter: data is passed
# through but should `stop()` if there
# is a single test unit failing; the
# behavior of side effects can be
# customized with the `actions` option
tbl %>%
  conjointly(
    ~ col_vals_lt(., vars(a), 8),
    ~ col_vals_gt(., vars(c), vars(a)),
    ~ col_vals_not_null(., vars(b))
  )

# C: Using the expectation function

# With the `expect_*()` form, we would
# typically perform one validation at a
# time; this is primarily used in
# testthat tests
expect_conjointly(
  tbl,
  ~ col_vals_lt(., vars(a), 8),
  ~ col_vals_gt(., vars(c), vars(a)),
  ~ col_vals_not_null(., vars(b))
)

# D: Using the test function

# With the `test_*()` form, we should
# get a single logical value returned
# to us
tbl %>%
  test_conjointly(
    ~ col_vals_lt(., vars(a), 8),
    ~ col_vals_gt(., vars(c), vars(a)),
    ~ col_vals_not_null(., vars(b))
  )
```

---

create_agent                    *Create a* **pointblank** agent *object*

---

**Description**

The create_agent() function creates an *agent* object, which is used in a *data quality reporting* workflow. The overall aim of this workflow is to generate useful reporting information for assessing the level of data quality for the target table. We can supply as many validation functions as the user wishes to write, thereby increasing the level of validation coverage for that table. The *agent* assigned by the create_agent() call takes validation functions, which expand to validation steps (each one is numbered). This process is known as developing a *validation plan*.

The validation functions, when called on an *agent*, are merely instructions up to the point the [interrogate()](interrogate()) function is called. That kicks off the process of the *agent* acting on the *validation plan* and getting results for each step. Once the interrogation process is complete, we can say that the *agent* has intel. Calling the *agent* itself will result in a reporting table. This reporting of the interrogation can also be accessed with the [get_agent_report()](get_agent_report()) function, where there are more reporting options.

**Usage**

```
create_agent(
  tbl = NULL,
  read_fn = NULL,
  tbl_name = NULL,
  label = NULL,
  actions = NULL,
  end_fns = NULL,
  embed_report = FALSE,
  lang = NULL,
  locale = NULL
)
```

**Arguments**

| | |
|---|---|
| tbl | The input table. This can be a data frame, a tibble, a tbl_dbi object, or a tbl_spark object. Alternatively, a function can be used to read in the input data table with the read_fn argument (in which case, tbl can be NULL). |
| read_fn | A function that's used for reading in the data. Even if a tbl is provided, this function will be invoked to obtain the data (i.e., the read_fn takes priority). There are two ways to specify a read_fn: (1) using a function (e.g., function() { <table reading code> }) or, (2) with an R formula expression (e.g., ~ { <table reading code> }). |
| tbl_name | A optional name to assign to the input table object. If no value is provided, a name will be generated based on whatever information is available. This table name will be displayed in the header area of the agent report generated by printing the *agent* or calling [get_agent_report()](get_agent_report()). |
| label | An optional label for the validation plan. If no value is provided, a label will be generated based on the current system time. Markdown can be used here to make the label more visually appealing (it will appear in the header area of the agent report). |

actions                A option to include a list with 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 a
                       specific validation step, the default set specified here will be overridden.

end_fns                A list of function calls that should be performed at the end of an interroga-
                       tion. Each function call should be in the form of a one-sided R formula ex-
                       pression, so overall this construction should be used: end_fns = list(~ <R state-
                       ments>, ~ <R statements>, ...). An example of a function that can be sensibly
                       used here is [email_blast()](#), where an email of the validation report is gener-
                       ated and sent based on sending condition.

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.

lang                   The language to use for automatic creation of briefs (short descriptions for each
                       validation step) and for the *agent report* (a summary table that provides the val-
                       idation plan and the results from the interrogation. By default, NULL will create
                       English ("en") text. Other options include French ("fr"), German ("de"), Ital-
                       ian ("it"), Spanish ("es"), Portuguese, ("pt"), Chinese ("zh"), and Russian
                       ("ru").

locale                 An optional locale ID to use for formatting values in the *agent report* sum-
                       mary table according the locale's rules. Examples include "en_US" for English
                       (United States) and "fr_FR" for French (France); more simply, this can be a
                       language identifier without a country designation, like "es" for Spanish (Spain,
                       same as "es_ES").

## Details

A very detailed list object, known as the x-list, can be obtained by using the [get_agent_x_list()](#)
function on the *agent*. This font of information can be taken as a whole, or, broken down by the
step number (with the i argument).

Sometimes it is useful to see which rows were the failing ones. By using the [get_data_extracts()](#)
function on the *agent*, we either get a list of tibbles (for those steps that have data extracts) or one
tibble if the validation step is specified with the i argument.

If we just need to know whether all validations completely passed (i.e., all steps had no failing test
units), the [all_passed()](#) function could be used on the *agent*. However, in practice, it's not often
the case that all data validation steps are free from any failing units.

## Value

A ptblank_agent object.

## Figures

## Function ID

1-2

**See Also**

Other Planning and Prep: action_levels(), col_schema(), create_informant(), db_tbl(), scan_data(), validate_rmd()

**Examples**

```
# Let's walk through a data quality
# analysis of an extremely small table;
# it's actually called `small_table` and
# we can find it as a dataset in this
# package
small_table

# We ought to think about what's
# tolerable in terms of data quality so
# let's designate proportional failure
# thresholds to the `warn`, `stop`, and
# `notify` states using `action_levels()`
al <-
  action_levels(
      warn_at = 0.10,
      stop_at = 0.25,
    notify_at = 0.35
  )

# Now create a pointblank `agent` object
# and give it the `al` object (which
# serves as a default for all validation
# steps which can be overridden); the
# static thresholds provided by `al` will
# make the reporting a bit more useful
agent <-
  create_agent(
    read_fn = ~ small_table,
    label = "An example.",
    actions = al
  )

# Then, as with any `agent` object, we
# can add steps to the validation plan by
# using as many validation functions as we
# want; then, we use `interrogate()` to
# physically perform the validations and
# gather intel
agent <-
  agent %>%
  col_exists(vars(date, date_time)) %>%
  col_vals_regex(
    vars(b), "[0-9]-[a-z]{3}-[0-9]{3}"
  ) %>%
  rows_distinct() %>%
  col_vals_gt(vars(d), 100) %>%
```

```
    col_vals_lte(vars(c), 5) %>%
    col_vals_equal(
      vars(d), vars(d),
      na_pass = TRUE
    ) %>%
    col_vals_between(
      vars(c),
      left = vars(a), right = vars(d),
      na_pass = TRUE
    ) %>%
    interrogate()

# Calling `agent` in the console
# prints the agent's report; but we
# can get a `gt_tbl` object directly
# with `get_agent_report(agent)`
report <- get_agent_report(agent)
class(report)

# What can you do with the report?
# Print it from an R Markdown code
# chunk, use it in a **blastula** email,
# put it in a webpage, or further
# modify it with the **gt** package

# From the report we know that Step
# 4 had two test units (rows, really)
# that failed; we can see those rows
# with `get_data_extracts()`
agent %>% get_data_extracts(i = 4)

# We can get an x-list for the whole
# validation (8 steps), or, just for
# the 4th step with `get_agent_x_list()`
xl_step_4 <-
  agent %>% get_agent_x_list(i = 4)

# And then we can peruse the different
# parts of the list; let's get the
# fraction of test units that failed
xl_step_4$f_failed

# Just printing the x-list will tell
# us what's available therein
xl_step_4

# An x-list not specific to any step
# will have way more information and a
# slightly different structure; see
# `help(get_agent_x_list)` for more info
# get_agent_x_list(agent)
```

---

create_informant          *Create a* **pointblank** informant *object*

---

## Description

The `create_informant()` function creates an *informant* object, which is used in an *information management* workflow. The overall aim of this workflow is to record, collect, and generate useful information on data tables. We can supply as information that is useful for describing a particular data table. The *informant* object created by the `create_informant()` function takes information-focused functions (the info_*() series of functions).

## Usage

```
create_informant(
  tbl = NULL,
  read_fn = NULL,
  agent = NULL,
  tbl_name = NULL,
  label = NULL,
  lang = NULL,
  locale = NULL
)
```

## Arguments

| | |
|---|---|
| `tbl` | The input table. This can be a data frame, a tibble, a `tbl_dbi` object, or a `tbl_spark` object. Alternatively, a function can be used to read in the input data table with the `read_fn` argument (in which case, `tbl` can be `NULL`). |
| `read_fn` | A function that's used for reading in the data. Even if a `tbl` is provided, this function will be invoked to obtain the data (i.e., the `read_fn` takes priority). There are two ways to specify a `read_fn`: (1) using a function (e.g., function() { <table reading code> }) or, (2) with an R formula expression. |
| `agent` | A pointblank *agent* object. This object can be used instead of supplying a table in `tbl` or a table-reading function in `read_fn`. |
| `tbl_name` | A optional name to assign to the input table object. If no value is provided, a name will be generated based on whatever information is available. |
| `label` | An optional label for the information report. If no value is provided, a label will be generated based on the current system time. Markdown can be used here to make the label more visually appealing (it will appear in the header area of the information report). |
| `lang` | The language to use for the information report (a summary table that provides all of the available information for the table. By default, `NULL` will create English (`"en"`) text. Other options include French (`"fr"`), German (`"de"`), Italian (`"it"`), Spanish (`"es"`), Portuguese, (`"pt"`), Chinese (`"zh"`), and Russian (`"ru"`). |

locale          An optional locale ID to use for formatting values in the information report
                according the locale's rules. Examples include ″en_US″ for English (United
                States) and ″fr_FR″ for French (France); more simply, this can be a language
                identifier without a country designation, like "es" for Spanish (Spain, same as
                ″es_ES″).

## Value

A `ptblank_informant` object.

## Figures

## Function ID

1-3

## See Also

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

## Examples

```
# Let's walk through how we can
# generate some useful information for a
# really small table; it's actually
# called `small_table` and we can find
# it as a dataset in this package
small_table

# Create a pointblank `informant`
# object with `create_informant()`
# and the `small_table` dataset
informant <-
  create_informant(
    read_fn = ~small_table,
    tbl_name = ″small_table″,
    label = ″An example.″
  )

# This function creates some information
# without any extra help by profiling
# the supplied table object; it adds
# the sections: (1) 'table' and
# (2) 'columns' and we can print the
# object to see the information report

# Alternatively, we can get the same report
# by using `get_informant_report()`
report <- get_informant_report(informant)
```

```
class(report)
```

---

db_tbl                     *Get a table from a database*

---

## Description

If your target table is in a database, the db_tbl() function is a handy way of accessing it. This function simplifies the process of getting a tbl_dbi object, which usually involves a combination of building a connection to a database and using the dplyr::tbl() function with the connection and the table name (or a reference to a table in a schema). A better option is to use this function as the read_fn parameter in [create_agent()](#) and [create_informant()](#). This can be done by using a leading ~ (e.g,. read_fn = ~db_tbl(...)).

The username and password are supplied though environment variables. If desired, these can be supplied directly by enclosing those values in I().

## Usage

```
db_tbl(db, dbname, table, user, password, host = NULL, port = NULL)
```

## Arguments

| | |
|---|---|
| db | Either an appropriate driver function (e.g., RPostgres::Postgres()) or a short-hand name for the database type. Valid names are: ″postgresql″, ″postgres″, or ″pgsql″ (PostgreSQL, using the RPostgres::Postgres() driver function); ″mysql″ (MySQL, using RMySQL::MySQL()); ″maria″ or ″mariadb″ (MariaDB, using RMariaDB::MariaDB()); ″duckdb″ (DuckDB, using duckdb::duckdb()); and ″sqlite″ (SQLite, using RSQLite::SQLite()). |
| dbname | The database name. |
| table | The name of the table, or, a reference to a table in a schema (two-element vector with the names of schema and table). Alternatively, this can be supplied as a data table to copy into an in-memory database connection. This only works if: (1) the db is either ″sqlite″ or ″duckdb″, (2) the dbname was chosen as ″:memory:″, and (3) the data_tbl is a data frame or a tibble object. |
| user, password | The environment variables used to access the username and password for the database. |
| host, port | The database host and optional port number. |

## Value

A tbl_dbi object.

## Function ID

1-6

**See Also**

Other Planning and Prep: action_levels(), col_schema(), create_agent(), create_informant(), scan_data(), validate_rmd()

---

email_blast                    *Send email at a validation step or at the end of an interrogation*

---

**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).

**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 the x-list object prepared by the agent. This version of the x-list is
                the same as that generated via get_agent_x_list(<agent>) except this version is
                internally generated and hence 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 mes-
                sage.

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_create() can be used with a **pointblank** agent object or the output obtained from using the get_agent_x_list() function.

## Function ID

4-1

## See Also

Other Emailing: email_create(), 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_create()` function
email_object <-
  create_agent(
    tbl = tbl,
    actions = al
  ) %>%
  col_vals_gt(vars(a), 5) %>%
  col_vals_lt(vars(b), 5) %>%
  interrogate() %>%
  email_create()
```

---

email_create                    *Create an email object from a* **pointblank** *agent*

---

### Description

The email_create() function produces an email message object that could be sent using the **blas-tula** package. The x that we need for this could either be a **pointblank** agent, the *agent* x-list (produced from the *agent* with the [get_agent_x_list()](get_agent_x_list()) function), or a **pointblank** *informant*. In all casesm, the email message will appear in the Viewer and a **blastula** email_message object will be returned.

## Usage

```
email_create(
  x,
  msg_header = NULL,
  msg_body = stock_msg_body(),
  msg_footer = stock_msg_footer()
)
```

## Arguments

x               A **pointblank** *agent*, an *agent* x-list, or a **pointblank** *informant*. The x-list object can be created with the `get_agent_x_list()` function. It is recommended that the option i = NULL be used with `get_agent_x_list()` if supplying an x-list as x. Furthermore, The option generate_report = TRUE could be used with `create_agent()` so that the agent report is available within the email.

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

4-2

## See Also

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

```
# In a workflow that involves an
# `agent` object, we can make use of
# the `end_fns` argument and
# programmatically email the report
# with the `email_blast()` function,
# however, an alternate workflow is to
# produce the email object and choose
# to send outside of the pointblank API;
# the `email_create()` function lets
# us do this with an `agent` object
# email_object_1 <-
#   create_agent(
#     tbl = tbl,
#     actions = al
#   ) %>%
#   col_vals_gt(vars(a), 5) %>%
#   col_vals_lt(vars(b), 5) %>%
#   interrogate() %>%
#   email_create()

# We can view the HTML email just
# by printing `email_object`; it
# should appear in the Viewer

# The `email_create()` function can
# also be used on an agent x-list to
# get the same email message object
# email_object_2 <-
#   create_agent(
#     tbl = tbl,
#     actions = al
#   ) %>%
#   col_vals_gt(vars(a), 5) %>%
#   col_vals_lt(vars(b), 5) %>%
#   interrogate() %>%
#   get_agent_x_list() %>%
#   email_create()

# An information report that's
# produced by the informant can
# made into an email message object;
# let's create an informant and use
# `email_create()`
# email_object_3 <-
#   create_informant(
#     tbl = tbl
#   ) %>%
#   info_tabular(
#     info = "A simple table in the
#     *Examples* section of the function
#     called `email_create()`."
#   ) %>%
```

```
#   info_columns(
#     columns = vars(a),
#     info = "Numbers. On the high side."
#   ) %>%
#   info_columns(
#     columns = vars(b),
#     info = "Lower numbers. Zeroes, even."
#   ) %>%
#   incorporate() %>%
#   email_create()
```

get_agent_report          *Get a summary report from an agent*

### Description

We can get an informative summary table from an agent by using the get_agent_report() function. The table can be provided in two substantially different forms: as a **gt** based display table (the default), or, as a tibble. The amount of fields with intel is different depending on whether or not the agent performed an interrogation (with the [interrogate()](#) function). Basically, before [interrogate()](#) is called, the agent will contain just the validation plan (however many rows it has depends on how many validation functions were supplied a part of that plan). Post-interrogation, information on the passing and failing test units is provided, along with indicators on whether certain failure states were entered (provided they were set through actions). The display table variant of the agent report, the default form, will have the following columns:

- i (unlabeled): the validation step number
- STEP: the name of the validation function used for the validation step
- COLUMNS: the names of the target columns used in the validation step (if applicable)
- VALUES: the values used in the validation step, where applicable; this could be as literal values, as column names, an expression, a set of sub-validations (for a [conjointly()](#) validation step), etc.
- TBL: indicates whether any there were any preconditions to apply before interrogation; if not, a script 'I' stands for 'identity' but, if so, a right-facing arrow appears
- EVAL: a character value that denotes the result of each validation step functions' evaluation during interrogation
- *N*: the total number of test units for the validation step
- PASS: the number of test units that received a *pass*
- FAIL: the fraction of test units that received a *pass*
- W, S, N: indicators that show whether the warn, stop, or notify states were entered; unset states appear as dashes, states that are set with thresholds appear as unfilled circles when not entered and filled when thresholds are exceeded (colors for W, S, and N are amber, red, and blue)

- EXT: a column that provides buttons with data extracts for each validation step where failed rows are available (as CSV files)

The small version of the display table (obtained using `size = "small"`) omits the COLUMNS, TBL, and EXT columns. The width of the small table is 575px; the standard table is 875px wide.

If choosing to get a tibble (with `display_table = FALSE`), it will have the following columns:

- i: the validation step number
- type: the name of the validation function used for the validation step
- columns: the names of the target columns used in the validation step (if applicable)
- 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 test units for the validation step
- n_pass: the number of test units that received a *pass*
- f_pass: the fraction of test units that received a *pass*
- W, S, N: logical value stating whether the `warn`, `stop`, or `notify` states were entered
- extract: a logical value that indicates whether a data extract is available for the validation step

## Usage

```
get_agent_report(
  agent,
  arrange_by = c("i", "severity"),
  keep = c("all", "fail_states"),
  display_table = TRUE,
  size = "standard",
  lang = NULL,
  locale = NULL
)
```

## 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. |
| size | The size of the display table, which can be either "standard" (the default) or "small". This only applies to a display table (where display_table = TRUE). |
| lang | The language to use for automatic creation of briefs (short descriptions for each validation step) and for the *agent report* (a summary table that provides the validation plan and the results from the interrogation. By default, NULL will create English ("en") text. Other options include French ("fr"), German ("de"), Italian ("it"), Spanish ("es"), Portuguese, ("pt"), Chinese ("zh"), and Russian ("ru"). This lang option will override any previously set lang value (e.g., by the [create_agent()](#) call). |
| locale | An optional locale ID to use for formatting values in the *agent report* summary table according the locale's rules. Examples include "en_US" for English (United States) and "fr_FR" for French (France); more simply, this can be a language identifier without a country designation, like "es" for Spanish (Spain, same as "es_ES"). This locale option will override any previously set locale value (e.g., by the [create_agent()](#) call). |

## Value

A **gt** table object if display_table = TRUE or a tibble if display_table = FALSE.

## Function ID

5-2

## See Also

Other Interrogate and Report: [interrogate](#)()

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

```
# View a the report by printing the
# `agent` object anytime, but, return a
# gt table object by using this with
# `display_table = TRUE` (the default)
report <- get_agent_report(agent)
class(report)

# What can you do with the report?
# Print it from an R Markdown code,
# use it in an email, put it in a
# webpage, or further modify it with
# the **gt** package

# The agent report as a **gt** display
# table comes in two sizes: "standard"
# (the default) and "small"
small_report <-
  get_agent_report(agent, size = "small")
class(small_report)

# The standard report is 875px wide
# the small one is 575px wide
```

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

```
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_start`: the time at which the interrogation began (POSIXct [0 or 1])

- `time_end`: the time at which the interrogation ended (POSIXct [0 or 1])

- `label`: the optional label given to the agent (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 validation step index (int [1])

- `type`: the type of validation, value is validation function name (chr [1])

- `columns`: the columns specified for the validation function (chr [variable length])

- `values`: the values specified for the validation function (mixed types [variable length])

- `briefs`: the brief for the validation step in the specified 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])

- `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

7-1

## See Also

Other Post-interrogation: all_passed(), 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

In an agent-based workflow, after interrogation with `interrogate()` we can get the row data that didn't pass row-based validation steps with the get_data_extracts() function. The amount of data available in a particular extract depends on both the fraction of test units that didn't pass a validation step and the level of sampling or explicit collection from that set of units.

The availability of data extracts for each row-based validation step is depends on whether extract_failed is set to TRUE within the `interrogate()` call (it is by default). The amount of *fail* rows extracted depends on the collection parameters in `interrogate()`, and the default behavior is to collect up to the first 5000 *fail* rows.

Row-based validation steps are based on the validation functions of the form col_vals_*() and also include `conjointly()` and `rows_distinct()`. Only those types of validation steps can provide data extracts.

### 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

7-2

### See Also

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

**Examples**

```
# 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_informant_report        *Get a table information report from an* informant *object*

---

**Description**

We can get a table information report from an informant object that's generated by the create_informant()
function. The report is provided as a **gt** based display table. The amount of information shown de-
pends on the extent of that added via the use of the info_*() functions or through direct editing of a
**pointblank** YAML file (an informant can be written to **pointblank** YAML with yaml_write(informant = <informant>, ...)).

**Usage**

```
get_informant_report(informant, size = "standard", lang = NULL, locale = NULL)
```

**Arguments**

| | |
|---|---|
| informant | An informant object of class ptblank_informant. |
| size | The size of the display table, which can be either "standard" (the default, with a width of 875px) or "small" (width of 575px). |
| lang | The language to use for the *information report* (a summary table that provides the validation plan and the results from the interrogation. By default, NULL will create English ("en") text. Other options include French ("fr"), German ("de"), Italian ("it"), Spanish ("es"), Portuguese, ("pt"), Chinese ("zh"), and Russian ("ru"). This lang option will override any previously set lang value (e.g., by the create_agent() call). |

locale            An optional locale ID to use for formatting values in the *information report* sum-
                  mary table according the locale's rules. Examples include ″en_US″ for English
                  (United States) and ″fr_FR″ for French (France); more simply, this can be a
                  language identifier without a country designation, like ″es″ for Spanish (Spain,
                  same as ″es_ES″). This locale option will override any previously set locale
                  value (e.g., by the [create_agent()](#) call).

## Value

A **gt** table object.

## Function ID

6-2

## See Also

Other Incorporate and Report: [incorporate](#)()

## Examples

```
# Generate an informant object using
# the `small_table` dataset
informant <- create_informant(small_table)

# This function creates some information
# without any extra help by profiling
# the supplied table object; it adds
# the sections 'table' and columns' and
# we can print the object to see the
# table information report

# Alternatively, we can get the same report
# by using `get_informant_report()`
report <- get_informant_report(informant)
class(report)
```

---

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 test units
across all row-based validation functions), or, the 'fail' data piece (rows with at least one failing test
unit across the same series of validations).

## Usage

```
get_sundered_data(
  agent,
  type = c("pass", "fail", "combined"),
  pass_fail = c("pass", "fail"),
  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 for returning a
             single table are "pass" (the default) and "fail". Each of these options return
             a single table with, in the "pass" case, only the rows that passed across all
             validation steps (i.e., had no failing test units in any part of a row for any vali-
             dation step), or, the complementary set of rows in the "fail" case. Providing
             NULL returns both of the split data tables in a list (with the names of "pass" and
             "fail"). The option "combined" applies a categorical (pass/fail) label (settable
             in the pass_fail argument) in a new .pb_combined flag column. For this case
             the ordering of rows is fully retained from the input table.

pass_fail    A vector for encoding the flag column with 'pass' and 'fail' values when type
             = "combined". The default is c("pass","fail") but other options could be
             c(TRUE,FALSE), c(1,0), or c(1L,0L).

id_cols      An optional specification of one or more identifying columns. When taken to-
             gether, we can count on this single column or grouping of columns to distin-
             guish rows. If the table undergoing validation is not a data frame or tibble, then
             columns need to be specified for id_cols.

## Details

There are some caveats to sundering. The validation steps considered for this splitting has to be of
the row-based variety (e.g., the col_vals_*() functions or [conjointly()](#), but not rows_distinct()).
Furthermore, validation steps that experienced evaluation issues during interrogation are not consid-
ered, and, validation steps where active = FALSE will be disregarded. The collection of validation
steps that fulfill the above requirements for sundering are termed in-consideration validation steps.

If using any preconditions for validation steps, we must ensure that all in-consideration validation
steps use the same specified preconditions function. Put another way, we cannot split the target
table using a collection of in-consideration validation steps that use different forms of the input
table.

## Value

A list of table objects if type is NULL, or, a single table if a type is given.

## Function ID

7-3

### See Also

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

### Examples

```
# 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) %>%
  col_vals_equal(
    vars(d), vars(d),
    na_pass = TRUE
  ) %>%
  col_vals_between(
    vars(c), left = vars(a), right = vars(d),
    na_pass = TRUE
  ) %>%
  interrogate()

# Get the sundered data piece that
# contains only rows that passed all
# validation steps (the default piece)
agent %>% get_sundered_data()
```

---

incorporate *Given an* informant *object, update and incorporate table snippets*

---

### Description

When the *informant* object has a number of snippets available (by using `info_snippet()`) and the strings to use them (by using the info_*() functions and {<snippet_name>} in the text elements), the process of incorporating aspects of the table into the info text can occur by using the `incorporate()` function. After that, the information will be fully updated (getting the current state of table dimensions, re-rendering the info text, etc.) and we can print the *informant* object or use the `get_informant_report()` function to see the information report.

### Usage

```
incorporate(informant)
```

### Arguments

informant       An informant object of class `ptblank_informant`.

## Value

A `ptblank_informant` object.

## Function ID

6-1

## See Also

Other Incorporate and Report: `get_informant_report()`

## Examples

```
# Take the `small_table` and
# assign it to `test_table`; we'll
# modify it later
test_table <- small_table

# Generate an informant object, add
# two snippets with `info_snippet()`,
# add information with some other
# `info_*()` functions and then
# `incorporate()` the snippets into
# the info text
informant <-
  create_informant(
    read_fn = ~ test_table,
    tbl_name = "test_table"
  ) %>%
  info_snippet(
    snippet_name = "row_count",
    fn = ~ . %>% nrow()
  ) %>%
  info_snippet(
    snippet_name = "col_count",
    fn = ~ . %>% ncol()
  ) %>%
  info_columns(
    columns = vars(a),
    info = "In the range of 1 to 10. (SIMPLE)"
  ) %>%
  info_columns(
    columns = starts_with("date"),
    info = "Time-based values (e.g., `Sys.time()`)."
  ) %>%
  info_columns(
    columns = "date",
    info = "The date part of `date_time`. (CALC)"
  ) %>%
  info_section(
    section_name = "rows",
    row_count = "There are {row_count} rows available."
```

```
  ) %>%
  incorporate()

# We can print the `informant` object
# to see the information report

# Let's modify `test_table` to give
# it more rows and an extra column
test_table <-
  dplyr::bind_rows(test_table, test_table) %>%
  dplyr::mutate(h = a + c)

# Using `incorporate()` will cause
# the snippets to be reprocessed, and,
# the strings to be updated
informant <-
  informant %>% incorporate()

# When printed again, we'll see that the
# row and column counts in the header
# have been updated to reflect the
# changed `test_table`
```

---

info_columns          *Add information that focuses on aspects of a data table's columns*

---

### Description

Upon creation of an *informant* object (with the `create_informant()` function), there are two sections containing properties: (1) 'table' and (2) 'columns'. The 'columns' section is initialized with the table's column names and their types (as _type). Beyond that, it is useful to provide details about the nature of each column and we can do that with the info_columns() function. A single column (or multiple columns) is targeted, and then a series of named arguments (in the form entry_name = "The *info text*.") serves as additional information for the column or columns.

### Usage

```
info_columns(x, columns, ..., .add = TRUE)
```

### Arguments

x            An informant object of class ptblank_informant.

columns      The column or set of columns to focus on. Can be defined as a column name
             in quotes (e.g., "<column_name>"), one or more column names in vars() (e.g.,
             vars(<column_name>)), or with a select helper (e.g., starts_with("date")).

| ... | Information entries as a series of named arguments. The names refer to sub-section titles within `COLUMN -> <COLUMN_NAME>` and the RHS contains the *info text* (informational text that can be written as Markdown and further styled with *Text Tricks*). |
| `.add` | Should new text be added to existing text? This is `TRUE` by default; setting to `FALSE` replaces any existing text for a property. |

## Details

The *info text* readily accepts Markdown formatting. Also, there are a few *Text Tricks* that are good to know. Markdown links written as < link url > or [ link text ]( link url ) will get nicely-styled links. Any dates expressed in the ISO-8601 standard with parentheses, `"(2004-12-01)"`, will be styled with a font variation (monospaced) and underlined in purple. Spans of text can be converted to label text by using: (1) double parentheses around text for a rectangular label as in ((label text)), or (2) triple parentheses around text for a rounded-rectangular label like (((label text))). Finally, CSS styles can be applied to spans of *info text* with the following form:
[[ info text ]]<< CSS style rules >>

As an example of this in practice suppose you'd like to change the color of some text to red and make the font appear somewhat thinner. A variation on the following might be used:

`"This is a [[factor]]<<color: red; font-weight: 300;>> value."`

## Value

A `ptblank_informant` object.

## Figures

## Function ID

3-2

## See Also

Other Information Functions: `info_section()`, `info_snippet()`, `info_tabular()`

## Examples

```
# Create a pointblank `informant`
# object with `create_informant()`;
# we specify a `read_fn` with the
# `~` followed by a statement that
# gets the `small_table` dataset
informant <-
  create_informant(
    read_fn = ~ small_table,
    tbl_name = "small_table",
    label = "An example."
  )
```

```
# The `informant` object has the 'table'
# and 'columns' sections; we can add more
# properties to individual columns in
# the 'columns' section
informant <-
  informant %>%
  info_columns(
    columns = vars(a),
    info = "In the range of 1 to 10. (SIMPLE)"
  ) %>%
  info_columns(
    columns = starts_with("date"),
    info = "Time-based values (e.g., `Sys.time()`)."
  ) %>%
  info_columns(
    columns = "date",
    info = "The date part of `date_time`. (CALC)"
  )

# Upon printing the `informant` object, we see
# the additions made to the 'Columns' section

# The `informant` object can be written to
# a YAML file with the `yaml_write()`
# function; then, information can
# be directly edited or modified
# yaml_write(
#   informant = informant,
#   filename = "informant.yml"
# )

# The YAML file can then be read back
# into an informant object with the
# `yaml_read_informant()` function
# informant <-
#   yaml_read_informant(path = "informant.yml")
```

---

info_section                *Add information that focuses on some key aspect of the data table*

---

## Description

While the [info_tabular()](#) and [info_columns()](#) functions allow us to add/modify info text for specific sections, the info_section() makes it possible to add sections of our own choosing and the information that make sense for those sections. Define a section_name and provide a series of named arguments (in the form entry_name = "The *info text*.") to build the informational content for that section.

## Usage

```
info_section(x, section_name, ...)
```

## Arguments

| | |
|---|---|
| x | An informant object of class ptblank_informant. |
| section_name | The name of the section for which this information pertains. |
| ... | Information entries as a series of named arguments. The names refer to subsection titles within the section defined as section_name and the RHS is the *info text* (informational text that can be written as Markdown and further styled with *Text Tricks*). |

## Details

The *info text* readily accepts Markdown formatting. Also, there are a few *Text Tricks* that are good to know. Markdown links written as < link url > or [ link text ]( link url ) will get nicely-styled links. Any dates expressed in the ISO-8601 standard with parentheses, "(2004-12-01)", will be styled with a font variation (monospaced) and underlined in purple. Spans of text can be converted to label text by using: (1) double parentheses around text for a rectangular label as in ((label text)), or (2) triple parentheses around text for a rounded-rectangular label like (((label text))). Finally, CSS styles can be applied to spans of *info text* with the following form:
[[ info text ]]<< CSS style rules >>

As an example of this in practice suppose you'd like to change the color of some text to red and make the font appear somewhat thinner. A variation on the following might be used:

```
"This is a [[factor]]<<color: red; font-weight: 300;>> value."
```

## Value

A ptblank_informant object.

## Figures

## Function ID

3-3

## See Also

Other Information Functions: `info_columns()`, `info_snippet()`, `info_tabular()`

## Examples

```
# Create a pointblank `informant`
# object with `create_informant()`;
# we specify a `read_fn` with the
# `~` followed by a statement that
# gets the `small_table` dataset
```

```
informant <-
  create_informant(
    read_fn = ~ small_table,
    tbl_name = "small_table",
    label = "An example."
  )

# The `informant` object has the 'table'
# and 'columns' sections; we can create
# entirely different sections with their
# own properties using `info_section()`
informant <-
  informant %>%
  info_section(
    section_name = "notes",
    creation = "Dataset generated on (2020-01-15).",
    usage = "`small_table %>% dplyr::glimpse()`"
  )

# Upon printing the `informant` object, we see
# the addition of the 'Notes' section and its
# own information

# The `informant` object can be written to
# a YAML file with the `yaml_write()`
# function; then, information can
# be directly edited or modified
# yaml_write(
#   informant = informant,
#   filename = "informant.yml"
# )

# The YAML file can then be read back
# into an informant object with the
# `yaml_read_informant()` function
# informant <-
#   yaml_read_informant(path = "informant.yml")
```

---

info_snippet                    *Generate a useful text 'snippet' from the target table*

---

### Description

Getting little snippets of information from a table goes hand-in-hand with mixing those bits of info with your table info. Call info_snippet() to define a snippet and how you'll get that from the target table (it's with a function). So long as you know how to interact with a table and extract information, you can easily define snippets for a *informant* object. And once those snippets are defined, you can insert them into the info text as defined through the info_*() functions. Just use curly braces with the snippet_name inside (e.g., "This column has {n_cat} categories.").

## Usage

```
info_snippet(x, snippet_name, fn)
```

## Arguments

| | |
|---|---|
| x | An informant object of class ptblank_informant. |
| snippet_name | The name for snippet, which is used for interpolating the snippet itself into info text. |
| fn | A function that obtains a snippet of data from the target table. |

## Value

A ptblank_informant object.

## Figures

## Function ID

3-4

## See Also

Other Information Functions: info_columns(), info_section(), info_tabular()

## Examples

```
# Take the `small_table` and
# assign it to `test_table`; we'll
# modify it later
test_table <- small_table

# Generate an informant object, add
# two snippets with `info_snippet()`,
# add information with some other
# `info_*()` functions and then
# `incorporate()` the snippets into
# the info text
informant <-
  create_informant(
    read_fn = ~test_table,
    tbl_name = "test_table",
    label = "An example."
  ) %>%
  info_snippet(
    snippet_name = "row_count",
    fn = ~ . %>% nrow()
  ) %>%
  info_snippet(
    snippet_name = "col_count",
```

```
    fn = ~ . %>% ncol()
  ) %>%
  info_columns(
    columns = vars(a),
    info = "In the range of 1 to 10. (SIMPLE)"
  ) %>%
  info_columns(
    columns = starts_with("date"),
    info = "Time-based values (e.g., `Sys.time()`)."
  ) %>%
  info_columns(
    columns = "date",
    info = "The date part of `date_time`. (CALC)"
  ) %>%
  info_section(
    section_name = "rows",
    row_count = "There are {row_count} rows available."
  ) %>%
  incorporate()

# We can print the `informant` object
# to see the information report

# Let's modify `test_table` to give
# it more rows and an extra column
test_table <-
  dplyr::bind_rows(test_table, test_table) %>%
  dplyr::mutate(h = a + c)

# Using `incorporate()` will cause
# the snippets to be reprocessed, and,
# the info text to be updated
informant <-
  informant %>% incorporate()
```

---

info_tabular                *Add information that focuses on aspects of the data table as a whole*

---

### Description

When an *informant* object is created with the [create_informant()](#) function, it has two starter
sections: (1) 'table' and (2) 'columns'. The 'table' section should contain a few properties upon
creation, such as the supplied table name (name) and table dimensions (as _columns and _rows). We
can add more table-based properties with the info_tabular() function. By providing a series of
named arguments (in the form entry_name = "The *info text*."), we can add more information
that makes sense for describing the table as a whole.

### Usage

```
info_tabular(x, ...)
```

**Arguments**

x                        An informant object of class `ptblank_informant`.

...                      Information entries as a series of named arguments. The names refer to subsec-
                         tion titles within the `TABLE` section and the RHS is the *info text* (informational
                         text that can be written as Markdown and further styled with *Text Tricks*).

**Details**

The *info text* readily accepts Markdown formatting. Also, there are a few *Text Tricks* that are good
to know. Markdown links written as < link url > or [ link text ]( link url ) will get nicely-styled
links. Any dates expressed in the ISO-8601 standard with parentheses, `"(2004-12-01)"`, will be
styled with a font variation (monospaced) and underlined in purple. Spans of text can be converted
to label text by using: (1) double parentheses around text for a rectangular label as in ((label text)),
or (2) triple parentheses around text for a rounded-rectangular label like (((label text))). Finally,
CSS styles can be applied to spans of *info text* with the following form:
[[ info text ]]<< CSS style rules >>

As an example of this in practice suppose you'd like to change the color of some text to red and
make the font appear somewhat thinner. A variation on the following might be used:

`"This is a [[factor]]<<color: red; font-weight: 300;>> value."`

**Value**

A `ptblank_informant` object.

**Figures**

**Function ID**

3-1

**See Also**

Other Information Functions: `info_columns()`, `info_section()`, `info_snippet()`

**Examples**

```
# Create a pointblank `informant`
# object with `create_informant()`;
# we specify a `read_fn` with the
# `~` followed by a statement that
# gets the `small_table` dataset
informant <-
  create_informant(
    read_fn = ~ small_table,
    tbl_name = "small_table",
    label = "An example."
  )
```

```
# The `informant` object has the 'table'
# and 'columns' sections; we can add more
# properties to the 'table' section
informant <-
  informant %>%
  info_tabular(
    row_definition = "A row has randomized values.",
    source = c(
      "- From the **pointblank** package.",
      "- [https://rich-iannone.github.io/pointblank/]()"
    )
  )

# Upon printing the `informant` object, we see
# the additions made to the 'Table' section

# The `informant` object can be written to
# a YAML file with the `yaml_write()`
# function; then information can
# be directly edited or modified
# yaml_write(
#   informant = informant,
#   filename = "informant.yml"
# )

# The YAML file can then be read back
# into an informant object with the
# `yaml_read_informant()` function
# informant <-
#   yaml_read_informant(path = "informant.yml")
```

---

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

---

### Description

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

### Usage

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

## Value

A ptblank_agent object.

## Function ID

5-1

## See Also

Other Interrogate and Report: [get_agent_report]()

## 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)
  )

# 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()
```

---

log4r_step                    *Enable logging of failure conditions at the validation step level*

---

#### Description

The log4r_step() function can be used as an action in the [action_levels()](#) function (as a list
component for the fns list). Place a call to this function in every failure condition that should
produce a log (i.e., warn, stop, notify). Only the failure condition with the highest severity for a
given validation step will produce a log entry (skipping failure conditions with lower severity) so
long as the call to log4r_step() is present.

#### Usage

```
log4r_step(x, message = NULL, append_to = "pb_log_file")
```

#### Arguments

| | |
|---|---|
| x | A reference to the x-list object prepared by the agent. This version of the x-list is the same as that generated via get_agent_x_list(<agent>, i = <step>) except this version is internally generated and hence only available in an internal evaluation context. |
| message | The message to use for the log entry. When not provided, a default glue string is used for the messaging. This is dynamic since the internal glue::glue() call occurs in the same environment as x, the x-list that's constrained to the validation step. The default message, used when message = NULL is the glue string "Step {x$i} exceeded the {level} failure threshold (f_failed = {x$f_failed}) ['{x$type}']". As can be seen, a custom message can be crafted that uses other elements of the x-list with the {x$<component>} construction. |
| append_to | The file to which log entries at the warn level are appended. This can alternatively be one or more **log4r** appenders. |

---

remove_read_fn | *Remove a table-reading function associated with an* agent *or* infor-
mant

---

### Description

Removing an *agent* or an *informant*'s association to a table-reading function can be done with
remove_read_fn(). This may be good idea in an interactive session when instead relying on the
direct association of a data table (settable in [create_agent()](#) and [create_informant()](#)'s tbl
argument or with [set_tbl()](#)). The table-reading function can be set again with [set_read_fn()](#).

### Usage

```
remove_read_fn(x)
```

### Arguments

x      An *agent* object of class ptblank_agent, or, an *informant* of class ptblank_informant.

### Function ID

8-6

### See Also

Other Object Ops: [remove_tbl()](#), [set_read_fn()](#), [set_tbl()](#), [x_read_disk()](#), [x_write_disk()](#)

---

remove_tbl | *Remove a data table associated with an* agent *or* informant

---

### Description

Removing an *agent* or *informant*'s association to a data table can be done with the remove_tbl()
function. This can be useful to ensure that the table data isn't unintentionally written to disk. It is
usually best to avoid directly associating a table to an *agent* or *informant*, instead opting for setting
a table-reading function (via [create_agent()](#) and [create_informant()](#)'s read_fn argument, or,
with [set_read_fn()](#)). The association to a table can be set again with [set_tbl()](#).

### Usage

```
remove_tbl(x)
```

### Arguments

x      An *agent* object of class ptblank_agent, or, an *informant* of class ptblank_informant.

**Function ID**

8-4

**See Also**

Other Object Ops: remove_read_fn(), set_read_fn(), set_tbl(), x_read_disk(), x_write_disk()

---

rows_distinct          *Are row data distinct?*

---

**Description**

The rows_distinct() validation function, the expect_rows_distinct() expectation function, and the test_rows_distinct() test function all check 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. The validation function can be used directly on a data table or with an *agent* object (technically, a ptblank_agent object) whereas the expectation and test functions can only be used with a data table. The types of data tables that can be used include data frames, tibbles, database tables (tbl_dbi), and Spark DataFrames (tbl_spark). As a validation step or as an expectation, this will operate over the number of test units that is equal to the number of rows in the table (after any preconditions have been applied).

**Usage**

```
rows_distinct(
  x,
  columns = NULL,
  preconditions = NULL,
  actions = NULL,
  step_id = NULL,
  label = NULL,
  brief = NULL,
  active = TRUE
)

expect_rows_distinct(
  object,
  columns = NULL,
  preconditions = NULL,
  threshold = 1
)

test_rows_distinct(object, columns = NULL, preconditions = NULL, threshold = 1)
```

**Arguments**

x                        A data frame, tibble (`tbl_df` or `tbl_dbi`), Spark DataFrame (`tbl_spark`), or,
                         an agent object of class ptblank_agent that is created with [`create_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 valida-
                         tion. This is ideally as a one-sided R formula using a leading ~. In the formula
                         representation, the . serves as the input data table to be transformed (e.g., ~ .
                         `%>% dplyr::mutate(col = col + 10)`.

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.

step_id                  One or more optional identifiers for the single or multiple validation steps gen-
                         erated from calling a validation function. The use of step IDs serves to distin-
                         guish validation steps from each other and provide an opportunity for supplying
                         a more meaningful label compared to the step index. By default this is NULL,
                         and **pointblank** will automatically generate the step ID value (based on the step
                         index) in this case. One or more values can be provided, and the exact number
                         of ID values should (1) match the number of validation steps that the validation
                         function call will produce (influenced by the number of `columns` provided), (2)
                         be an ID string not used in any previous validation step, and (3) be a vector with
                         unique values.

label                    An optional label for the validation step. This label appears in the agent report
                         and for the best appearance it should be kept short.

brief                    An optional, text-based description for the validation step. If nothing is provided
                         here then an *autobrief* is generated by the agent, using the language provided
                         in [`create_agent()`](#)'s lang argument (which defaults to "en" or English). The
                         *autobrief* incorporates details of the validation step so it's often the preferred
                         option in most cases (where a label might be better suited to succinctly describe
                         the validation).

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.

object                   A data frame, tibble (`tbl_df` or `tbl_dbi`), or Spark DataFrame (`tbl_spark`)
                         that serves as the target table for the expectation function or the test function.

threshold                A simple failure threshold value for use with the expectation (`expect_`) and
                         the test (`test_`) function variants. By default, this is set to 1 meaning that any
                         single unit of failure in data validation results in an overall test failure. Whole
                         numbers beyond 1 indicate that any failing units up to that absolute threshold
                         value will result in a succeeding **testthat** test or evaluate to TRUE. Likewise,
                         fractional values (between 0 and 1) act as a proportional failure threshold, where
                         0.15 means that 15 percent of failing test units results in an overall test failure.

### Details

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

Having table preconditions means **pointblank** will mutate the table just before interrogation. Such a table mutation is isolated in scope to the validation step(s) produced by the validation function call. Using **dplyr** code is suggested here since the statements can be translated to SQL if necessary. The code is most easily supplied as a one-sided **R** formula (using a leading ~). In the formula representation, the . serves as the input data table to be transformed (e.g., ~ . %>% dplyr::mutate(col_a = col_b + 10)). Alternatively, a function could instead be supplied (e.g., function(x) dplyr::mutate(x, col_a = col_b + 10)).

Often, we will want to specify actions for the validation. This argument, present in every validation 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 of 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

For the validation function, the return value is either a ptblank_agent object or a table object (depending on whether an agent object or a table was passed to x). The expectation function invisibly returns its input but, in the context of testing data, the function is called primarily for its potential side-effects (e.g., signaling failure). The test function returns a logical value.

### Function ID

2-15

### See Also

Other validation 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_expr()](#), [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

```
# 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 <-
  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)
```

---

scan_data                    *Thoroughly scan a table to better understand it*

---

### Description

Generate 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 counts, 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 will appear in the RStudio Viewer and can also be integrated in R Markdown HTML output. If you need the output HTML as a string, it's possible to get that by using as.character() (e.g., scan_data(tbl = mtcars) %>% as.character()). The resulting HTML string is a complete HTML document where **Bootstrap** and **jQuery** are embedded within.

### Usage

```
scan_data(tbl, sections = "OVICMS", navbar = TRUE, lang = NULL, locale = NULL)
```

## Arguments

| | |
|---|---|
| tbl | The input table. This can be a data frame, tibble, a tbl_dbi object, or a tbl_spark object. |
| sections | The sections to include in the finalized Table Scan report. A string with key characters representing section names is required here. The default string is "OVICMS" wherein each letter stands for the following sections in their default order: "O": "overview"; "V": "variables"; "I": "interactions"; "C": "correlations"; "M": "missing"; and "S": "sample". This string can be comprised of less characters and the order can be changed to suit the desired layout of the report. For tbl_dbi and tbl_spark objects supplied to tbl, the "interactions" and "correlations" sections are currently excluded. |
| navbar | Should there be a navigation bar anchored to the top of the report page? By default this is TRUE. |
| 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"), Spanish ("es"), Portuguese, ("pt"), Chinese ("zh"), and Russian ("ru"). |
| locale | An optional locale ID to use for formatting values in the report according the locale's rules. Examples include "en_US" for English (United States) and "fr_FR" for French (France); more simply, this can be a language identifier without a country designation, like "es" for Spanish (Spain, same as "es_ES"). |

## Function ID

1-1

## See Also

Other Planning and Prep: action_levels(), col_schema(), create_agent(), create_informant(), db_tbl(), validate_rmd()

## Examples

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

---

| | |
|---|---|
| set_read_fn | *Set a table-reading function to an* agent *or* informant |

---

## Description

A table-reading function can be associated with an *agent* or *informant* with set_read_fn(). Should both a tbl *and* a read_fn be associated with the *agent* or *informant*, the read_fn will take priority. There are two ways to specify a read_fn: (1) using a function (e.g., function() { <table reading code> }) or, (2) with an R formula expression (e.g., ~ { <table reading code> }). The table-reading function can removed with remove_read_fn().

## Usage

```
set_read_fn(x, read_fn)
```

## Arguments

| | |
|---|---|
| x | An *agent* object of class ptblank_agent, or, an *informant* of class ptblank_informant. |
| read_fn | A function that's used for reading in the data. This can be specified by using a function (e.g., function() { <table reading code> }) or an R formula expression (e.g., ~ { <table reading code> }). |

## Function ID

8-5

## See Also

Other Object Ops: remove_read_fn(), remove_tbl(), set_tbl(), x_read_disk(), x_write_disk()

---

set_tbl                          *Set a data table to an* agent *or* informant

---

## Description

Setting a data table to an *agent* or *informant* with set_tbl() replaces any associated table (a data frame, a tibble, objects of class tbl_dbi or tbl_spark). If a data table is associated with an *agent* or *informant* along with a table-reading function (settable in create_agent() and create_informant()'s read_fn argument or with set_read_fn()), the table-reading function will take precedence. If this is undesirable, it be removed with the remove_read_fn() function. The association to a table can be removed with with remove_tbl().

## Usage

```
set_tbl(x, tbl)
```

## Arguments

| | |
|---|---|
| x | An *agent* object of class ptblank_agent, or, an *informant* of class ptblank_informant. |
| tbl | The input table for the agent. This can be a data frame, a tibble, a tbl_dbi object, or a tbl_spark object. Any table already associated with the *agent* or *informant* will be overwritten. |

## Function ID

8-3

## See Also

Other Object Ops: remove_read_fn(), remove_tbl(), set_read_fn(), x_read_disk(), x_write_disk()

## 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

10-1

### See Also

Other Datasets: [small_table_sqlite](#)()

### Examples

```
# 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

10-2

### 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()
```

---

snip_highest                  *A* fn *for* info_snippet()*: get the highest value from a column*

---

### Description

The snip_lowest() function can be used as an [info_snippet()](#) function (i.e., provided to fn) to
get the highest numerical, time value, or alphabetical value from a column in the target table.

### Usage

```
snip_highest(column)
```

### Arguments

column            The name of the column that contains the target values.

## Value

A formula needed for [`info_snippet()`](info_snippet())'s fn argument.

---

| snip_list | *A* fn *for* info_snippet()*: get a list of column categories* |
|---|---|

---

## Description

The snip_list() function can be used as an [`info_snippet()`](info_snippet()) function (i.e., provided to fn) to get a catalog list from a table column. You can limit the of items in that list with the limit value.

## Usage

```
snip_list(column, limit = 5)
```

## Arguments

| column | The name of the column that contains the target values. |
|---|---|
| limit | A limit of items put into the generated list. The returned text will state the remaining number of items beyond the limit. |

## Value

A formula needed for [`info_snippet()`](info_snippet())'s fn argument.

---

| snip_lowest | *A* fn *for* info_snippet()*: get the lowest value from a column* |
|---|---|

---

## Description

The snip_lowest() function can be used as an [`info_snippet()`](info_snippet()) function (i.e., provided to fn) to get the lowest numerical, time value, or alphabetical value from a column in the target table.

## Usage

```
snip_lowest(column)
```

## Arguments

| column | The name of the column that contains the target values. |
|---|---|

## Value

A formula needed for [`info_snippet()`](info_snippet())'s fn argument.

---

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 obtained as a standalone object through email_create().

### Usage

```
stock_msg_body()
```

### Value

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

### Function ID

4-3

### See Also

Other Emailing: email_blast(), email_create(), 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 obtained as a standalone object through email_create().

### Usage

```
stock_msg_footer()
```

### Value

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

### Function ID

4-4

### See Also

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

---

stop_if_not                 *The next generation of* stopifnot()*-type functions:* stop_if_not()

---

### Description

This is stopifnot() but with a twist: it works well as a standalone, replacement for stopifnot() but is also customized for use in validation checks in R Markdown documents where **pointblank** is loaded. Using stop_if_not() in a code chunk where the validate = TRUE option is set will yield the correct reporting of successes and failures whereas stopifnot() *does not*.

### Usage

```
stop_if_not(...)
```

### Arguments

...                 R expressions that should each evaluate to (a logical vector of all) TRUE.

### Value

NULL if all statements in ... are TRUE.

### Examples

```
# This checks whether the number of
# rows in `small_table` is greater
# than `10`
stop_if_not(nrow(small_table) > 10)

# This will stop for sure: there
# isn't a `time` column in `small_table`
# (but there are the `date_time` and
# `date` columns)
# stop_if_not("time" %in% colnames(small_table))

# You're not bound to using tabular
# data here, any statements that
# evaluate to logical vectors will work
stop_if_not(1:5 < 20:25)
```

---

validate_rmd                    *Modify* **pointblank** *validation testing options within R Markdown documents*

---

**Description**

Using **pointblank** in an R Markdown workflow is enabled by default once the **pointblank** library is loaded. The framework allows for validation testing within specialized validation code chunks where the validate = TRUE option is set. Using **pointblank** validation functions on data in these marked code chunks will flag overall failure if the stop threshold is exceeded anywhere. All errors are reported in the validation code chunk after rendering the document to HTML, where green or red status buttons indicate whether all validations succeeded or failures occurred. Clicking any such button reveals the otherwise hidden validation statements and their error messages (if any). While the framework for such testing is set up by default, the validate_rmd() function offers an opportunity to set UI and logging options.

**Usage**

```
validate_rmd(summary = TRUE, log_to_file = NULL)
```

**Arguments**

summary        If TRUE (the default), then there will be a leading summary of all validations in
               the rendered R Markdown document. With FALSE, this element is not shown.

log_to_file    An option to log errors to a text file. By default, no logging is done but TRUE
               will write log entries to "validation_errors.log" in the working directory.
               To both enable logging and to specify a file name, include a path to a log file of
               the desired name.

**Function ID**

1-4

**See Also**

Other Planning and Prep: [action_levels](), [col_schema](), [create_agent](), [create_informant](),
[db_tbl](), [scan_data]()

---

x_read_disk                     *Read a* **pointblank** *agent or* informant *from disk*

---

### Description

An *agent* or *informant* that has been written to disk (with x_write_disk()) can be read back into memory with the x_read_disk() function. Once the *agent* or *informant* has been generated in this way, it may not have a data table associated with it (depending on whether the keep_tbl option was TRUE or FALSE when writing to disk) but it should still be able to produce reporting (by printing the *agent* or *informant* to the console or using get_agent_report()/get_informant_report()). An *agent* will return an x-list with get_agent_x_list() and yield any available data extracts with get_data_extracts(). Furthermore, all of an *agent*'s validation steps will still be present (along with results from the last interrogation).

### Usage

```
x_read_disk(path)
```

### Arguments

path            The path to a file that was previously written by x_write_disk().

### Details

Should the *agent* or *informant* possess a table-reading function (can be set any time with set_read_fn()) or a specific table (settable with set_tbl()) we could use the interrogate() or incorporate() function again. For a *data quality reporting* workflow, it is useful to interrogate() target tables that evolve over time. While the same validation steps will be used, more can be added before calling interrogate(). For an *information management* workflow with an *informant* object, using incorporate() will update aspects of the reporting such as table dimensions, and info snippets/text will be regenerated.

### Function ID

8-2

### See Also

Other Object Ops: remove_read_fn(), remove_tbl(), set_read_fn(), set_tbl(), x_write_disk()

---

x_write_disk                    *Write a* **pointblank** *agent or* informant *to disk*

---

### Description

Writing an *agent* or *informant* to disk with x_write_disk() can be useful for keeping data valida-
tion intel or table information close at hand for later retrieval (with x_read_disk()). By default,
any data table that the *agent* or *informant* may have held before being committed to disk will be
expunged. This behavior can be changed by setting keep_tbl to TRUE but this only works in the
case where the table is not of the tbl_dbi or the tbl_spark class.

### Usage

```
x_write_disk(x, filename, path = NULL, keep_tbl = FALSE, keep_extracts = FALSE)
```

### Arguments

| | |
|---|---|
| x | An *agent* object of class ptblank_agent, or, an *informant* of class ptblank_informant. |
| filename | The filename to create on disk for the agent or informant. |
| path | An optional path to which the file should be saved (this is automatically com-bined with filename). |
| keep_tbl | An option to keep a data table that is associated with the *agent* or *informant* (which is the case when the *agent*, for example, is created using create_agent(tbl = <data ta-ble, ...)). The default is FALSE where the data table is removed before writing to disk. For database tables of the class tbl_dbi and for Spark DataFrames (tbl_spark) the table is always removed (even if keep_tbl is set to TRUE). |
| keep_extracts | An option to keep any collected extract data for failing rows. By default, this is FALSE. |

### Details

It is recommended to set a table-reading function for later reuse of the *agent* and *informant* after
being read from disk through x_read_disk(). This can be done initially with the read_fn argu-
ment of create_agent()/create_informant() or, later, with set_read_fn(). Alternatively, we
can reintroduce the *agent* or *informant* to a data table with the set_tbl() function.

### Function ID

8-1

### See Also

Other Object Ops: remove_read_fn(), remove_tbl(), set_read_fn(), set_tbl(), x_read_disk()

yaml_agent_interrogate

*Get an* agent *from* **pointblank** *YAML and* `interrogate()`

## Description

The `yaml_agent_interrogate()` function operates much like the [yaml_read_agent()](#) function (reading a **pointblank** YAML file and generating an *agent* with a validation plan in place). The key difference is that this function takes things a step further and interrogates the target table (defined by table-reading, `read_fn`, function that is required in the YAML file). The additional auto-invocation of [interrogate()](#) uses the default options of that function. As with [yaml_read_agent()](#) the agent is returned except, this time, it has intel from the interrogation.

## Usage

```
yaml_agent_interrogate(path)
```

## Arguments

path            A path to a **pointblank** YAML file that contains fields related to an *agent*.

## Function ID

9-4

## See Also

Other pointblank YAML: [yaml_agent_show_exprs()](#), [yaml_agent_string()](#), [yaml_informant_incorporate()](#), [yaml_read_agent()](#), [yaml_read_informant()](#), [yaml_write()](#)

## Examples

```
# Let's go through the process of
# developing an agent with a validation
# plan (to be used for the data quality
# analysis of the `small_table` dataset),
# and then offloading that validation
# plan to a pointblank YAML file; this
# will later be read in as a new agent and
# the target data will be interrogated
# (one step) with `yaml_agent_interrogate()`

# We ought to think about what's
# tolerable in terms of data quality so
# let's designate proportional failure
# thresholds to the `warn`, `stop`, and
# `notify` states using `action_levels()`
al <-
  action_levels(
```

```
      warn_at = 0.10,
      stop_at = 0.25,
      notify_at = 0.35
    )

# Now create a pointblank `agent` object
# and give it the `al` object (which
# serves as a default for all validation
# steps which can be overridden); the
# data will be referenced in a `read_fn`
# (a requirement for writing to YAML)
agent <-
  create_agent(
    read_fn = ~small_table,
    label = "A simple example with the `small_table`.",
    actions = al
  )

# Then, as with any `agent` object, we
# can add steps to the validation plan by
# using as many validation functions as we
# want
agent <-
  agent %>%
  col_exists(vars(date, date_time)) %>%
  col_vals_regex(
    vars(b), "[0-9]-[a-z]{3}-[0-9]{3}"
  ) %>%
  rows_distinct() %>%
  col_vals_gt(vars(d), 100) %>%
  col_vals_lte(vars(c), 5)

# The agent can be written to a pointblank
# YAML file with `yaml_write()`
# yaml_write(
#   agent = agent,
#   filename = "agent-small_table.yml"
# )

# The 'agent-small_table.yml' file is
# available in the package through `system.file()`
yml_file <-
  system.file(
    "agent-small_table.yml",
    package = "pointblank"
  )

# We can view the YAML file in the console
# with the `yaml_agent_string()` function
yaml_agent_string(path = yml_file)

# The YAML can also be printed in the console
# by supplying the agent as the input
```

```
yaml_agent_string(agent = agent)

# We can interrogate the data (which
# is accessible through the `read_fn`)
# through direct use of the YAML file
# with `yaml_agent_interrogate()`
agent <-
  yaml_agent_interrogate(path = yml_file)

class(agent)

# If it's desired to only create a new
# agent with the validation plan in place
# (stopping short of interrogating the data),
# then the `yaml_read_agent()` function
# will be useful
agent <- yaml_read_agent(path = yml_file)
class(agent)
```

---

yaml_agent_show_exprs  *Display validation expressions using* **pointblank** *YAML*

---

## Description

The yaml_agent_show_exprs() function follows the specifications of a **pointblank** YAML file to generate and show the **pointblank** expressions for generating the described validation plan. The expressions are shown in the console, providing an opportunity to copy the statements and extend as needed. A **pointblank** YAML file can itself be generated by using the [yaml_write()](#) function with a pre-existing *agent*, or, it can be carefully written by hand.

## Usage

```
yaml_agent_show_exprs(path)
```

## Arguments

path          A path to a **pointblank** YAML file that contains fields related to an *agent*.

## Function ID

9-6

## See Also

Other pointblank YAML: [yaml_agent_interrogate()](#), [yaml_agent_string()](#), [yaml_informant_incorporate()](#), [yaml_read_agent()](#), [yaml_read_informant()](#), [yaml_write()](#)

**Examples**

```
# Let's create a validation plan for the
# data quality analysis of the `small_table`
# dataset; we need an agent and its
# table-reading function enables retrieval
# of the target table
agent <-
  create_agent(
    read_fn = ~small_table,
    label = "A simple example with the `small_table`.",
    actions = action_levels(
      warn_at = 0.10,
      stop_at = 0.25,
      notify_at = 0.35
    )
  ) %>%
  col_exists(vars(date, date_time)) %>%
  col_vals_regex(
    vars(b), "[0-9]-[a-z]{3}-[0-9]{3}"
  ) %>%
  rows_distinct() %>%
  col_vals_gt(vars(d), 100) %>%
  col_vals_lte(vars(c), 5)

# The agent can be written to a pointblank
# YAML file with `yaml_write()`
# yaml_write(
#   agent = agent,
#   filename = "agent-small_table.yml"
# )

# The 'agent-small_table.yml' file is
# available in the package through
# `system.file()`
yml_file <-
  system.file(
    "agent-small_table.yml",
    package = "pointblank"
  )

# At a later time, the YAML file can
# be read into a new agent with the
# `yaml_read_agent()` function
agent <- yaml_read_agent(path = yml_file)

class(agent)

# To get a sense of which expressions are
# being used to generate the new agent, we
# can use `yaml_agent_show_exprs()`
yaml_agent_show_exprs(path = yml_file)
```

---

yaml_agent_string          *Display* **pointblank** *YAML using an agent or a YAML file*

---

### Description

With **pointblank** YAML, we can serialize an agent's validation plan (with yaml_write()), read it
back later with a new agent (with yaml_read_agent()), or perform an interrogation on the target
data table directly with the YAML file (with yaml_agent_interrogate()). The yaml_agent_string()
function allows us to inspect the YAML generated by yaml_write() in the console, giving us a look
at the YAML without needing to open the file directly. Alternatively, we can provide an *agent* to the
yaml_agent_string() and view the YAML representation of the validation plan without needing
to write the YAML to disk beforehand.

### Usage

```
yaml_agent_string(agent = NULL, path = NULL)
```

### Arguments

agent          An *agent* object of class ptblank_agent.

path           A path to a YAML file that specifies a validation plan for an *agent*.

### Function ID

9-5

### See Also

Other pointblank YAML: yaml_agent_interrogate(), yaml_agent_show_exprs(), yaml_informant_incorporate(),
yaml_read_agent(), yaml_read_informant(), yaml_write()

### Examples

```
# Let's create a validation plan for the
# data quality analysis of the `small_table`
# dataset; we need an agent and its
# table-reading function enables retrieval
# of the target table
agent <-
  create_agent(
    read_fn = ~small_table,
    label = "A simple example with the `small_table`.",
    actions = action_levels(
      warn_at = 0.10,
      stop_at = 0.25,
      notify_at = 0.35
    )
  ) %>%
```

```
    col_exists(vars(date, date_time)) %>%
    col_vals_regex(
      vars(b), "[0-9]-[a-z]{3}-[0-9]{3}"
    ) %>%
    rows_distinct() %>%
    col_vals_gt(vars(d), 100) %>%
    col_vals_lte(vars(c), 5)

# We can view the YAML file in the console
# with the `yaml_agent_string()` function,
# providing the `agent` object as the input
yaml_agent_string(agent = agent)

# The agent can be written to a pointblank
# YAML file with `yaml_write()`
# yaml_write(
#   agent = agent,
#   filename = "agent-small_table.yml"
# )

# The 'agent-small_table.yml' file is
# available in the package through `system.file()`
yml_file <-
  system.file(
    "agent-small_table.yml",
    package = "pointblank"
  )

# The `yaml_agent_string()` function can
# be used with the YAML file as well
yaml_agent_string(path = yml_file)

# At a later time, the YAML file can
# be read into a new agent with the
# `yaml_read_agent()` function
agent <- yaml_read_agent(path = yml_file)
class(agent)
```

---

yaml_informant_incorporate

*Get an* informant *from* **pointblank** *YAML and* incorporate()

---

### Description

The yaml_informant_incorporate() function operates much like the yaml_read_informant() function (reading a **pointblank** YAML file and generating an *informant* with all information in place). The key difference is that this function takes things a step further and incorporates aspects from the the target table (defined by table-reading, read_fn, function that is required in the YAML file). The additional auto-invocation of incorporate() uses the default options of that function.

As with `yaml_read_informant()` the informant is returned except, this time, it has been updated with the latest information from the target table.

## Usage

```
yaml_informant_incorporate(path)
```

## Arguments

path              A path to a YAML file that specifies a information for an *informant*.

## Function ID

9-7

## See Also

Other pointblank YAML: `yaml_agent_interrogate()`, `yaml_agent_show_exprs()`, `yaml_agent_string()`, `yaml_read_agent()`, `yaml_read_informant()`, `yaml_write()`

## Examples

```
# Let's go through the process of
# developing an informant with information
# about the `small_table` dataset and then
# move all that to a pointblank YAML
# file; this will later be read in as a
# new informant and the target data will
# be incorporated into the info text
# (in one step) with
# `yaml_informant_incorporate()`

# Now create a pointblank `informant`
# object; the data will be referenced
# in a `read_fn` (a requirement for
# writing to YAML)
informant <-
  create_informant(
    read_fn = ~small_table,
    label = "A simple example with the `small_table`."
  )

# Then, as with any `informant` object, we
# can add information by using as many
# `info_*()` functions as we want
informant <-
  informant %>%
  info_columns(
   columns = vars(a),
   info = "In the range of 1 to 10. (SIMPLE)"
  ) %>%
  info_columns(
```

```
    columns = starts_with("date"),
    info = "Time-based values (e.g., `Sys.time()`)."
  ) %>%
  info_columns(
    columns = "date",
    info = "The date part of `date_time`. (CALC)"
  ) %>%
  info_section(
    section_name = "rows",
    row_count = "There are {row_count} rows available."
  ) %>%
  info_snippet(
    snippet_name = "row_count",
    fn = ~ . %>% nrow()
  ) %>%
  incorporate()

# The informant can be written to a pointblank
# YAML file with `yaml_write()`
# yaml_write(
#   informant = informant,
#   filename = "informant-small_table.yml"
# )

# The 'informant-small_table.yml' file
# is available in the package through
# `system.file()`
yml_file <-
  system.file(
    "informant-small_table.yml",
    package = "pointblank"
  )

# We can incorporate the data (which
# is accessible through the `read_fn`)
# into the info text through direct
# use of the YAML file with
# `yaml_informant_incorporate()`
informant <-
  yaml_informant_incorporate(path = yml_file)

class(informant)

# If it's desired to only create a new
# informant with the information in place
# (stopping short of processing), then the
# `yaml_read_informant()` function will
# be useful
informant <-
  yaml_read_informant(path = yml_file)

class(informant)
```

---

yaml_read_agent                *Read a* **pointblank** *YAML file to create an* agent *object*

---

### Description

With yaml_read_agent() we can read a **pointblank** YAML file that describes a validation plan to be carried out by an *agent* (typically generated by the yaml_write() function. What's returned is a new *agent* with that validation plan, ready to interrogate the target table at will (using the table-reading function stored as the read_fn). The agent can be given more validation steps if needed before using interrogate() or taking part in any other agent ops (e.g., writing to disk with outputs intact via x_write_disk() or again to **pointblank** YAML with yaml_write()).

To get a picture of how yaml_read_agent() is interpreting the validation plan specified in the **pointblank** YAML, we can use the yaml_agent_show_exprs() function. That function shows us (in the console) the **pointblank** expressions for generating the described validation plan.

### Usage

```
yaml_read_agent(path)
```

### Arguments

path                A path to a **pointblank** YAML file that contains fields related to an *agent*.

### Function ID

9-2

### See Also

Other pointblank YAML: yaml_agent_interrogate(), yaml_agent_show_exprs(), yaml_agent_string(), yaml_informant_incorporate(), yaml_read_informant(), yaml_write()

### Examples

```
# Let's go through the process of
# developing an agent with a validation
# plan (to be used for the data quality
# analysis of the `small_table` dataset),
# and then offloading that validation
# plan to a pointblank YAML file; this
# will be read in with `yaml_read_agent()`

# We ought to think about what's
# tolerable in terms of data quality so
# let's designate proportional failure
# thresholds to the `warn`, `stop`, and
# `notify` states using `action_levels()`
al <-
```

```
  action_levels(
    warn_at = 0.10,
    stop_at = 0.25,
    notify_at = 0.35
  )

# Now create a pointblank `agent` object
# and give it the `al` object (which
# serves as a default for all validation
# steps which can be overridden); the
# data will be referenced in a `read_fn`
# (a requirement for writing to YAML)
agent <-
  create_agent(
    read_fn = ~small_table,
    label = "A simple example with the `small_table`.",
    actions = al
  )

# Then, as with any `agent` object, we
# can add steps to the validation plan by
# using as many validation functions as we
# want
agent <-
  agent %>%
  col_exists(vars(date, date_time)) %>%
  col_vals_regex(
    vars(b), "[0-9]-[a-z]{3}-[0-9]{3}"
  ) %>%
  rows_distinct() %>%
  col_vals_gt(vars(d), 100) %>%
  col_vals_lte(vars(c), 5)

# The agent can be written to a pointblank
# YAML file with `yaml_write()`
# yaml_write(
#   agent = agent,
#   filename = "agent-small_table.yml"
# )

# The 'agent-small_table.yml' file is
# available in the package through `system.file()`
yml_file <-
  system.file(
    "agent-small_table.yml",
    package = "pointblank"
  )

# We can view the YAML file in the console
# with the `yaml_agent_string()` function
yaml_agent_string(path = yml_file)

# The YAML can also be printed in the console
```

```
# by supplying the agent as the input
yaml_agent_string(agent = agent)

# At a later time, the YAML file can
# be read into a new agent with the
# `yaml_read_agent()` function
agent <- yaml_read_agent(path = yml_file)

class(agent)

# We can interrogate the data (which
# is accessible through the `read_fn`)
# with `interrogate()` and get an
# agent with intel, or, we can
# interrogate directly from the YAML
# file with `yaml_agent_interrogate()`
agent <-
  yaml_agent_interrogate(path = yml_file)

class(agent)
```

yaml_read_informant    *Read a* **pointblank** *YAML file to create an* informant *object*

### Description

With yaml_read_informant() we can read a **pointblank** YAML file that describes table informa-
tion (typically generated by the [yaml_write()](#) function. What's returned is a new *informant* object
with the information intact. The *informant* object can be given more information through use of the
info_*() functions.

### Usage

```
yaml_read_informant(path)
```

### Arguments

path          A path to a **pointblank** YAML file that contains fields related to an *informant*.

### Function ID

9-3

### See Also

Other pointblank YAML: [yaml_agent_interrogate()](#), [yaml_agent_show_exprs()](#), [yaml_agent_string()](#),
[yaml_informant_incorporate()](#), [yaml_read_agent()](#), [yaml_write()](#)

**Examples**

```
# Create a pointblank `informant`
# object with `create_informant()`
# and the `small_table` dataset
informant <- create_informant(small_table)

# An `informant` object can be written
# to a YAML file with the `yaml_write()`
# function
# yaml_write(
#   informant = informant,
#   filename = "informant-small_table.yml"
# )

# The `informant-small_table.yml` file
# looks like this when written

#> info_label: '[2020-09-06|13:37:38]'
#> table:
#>   name: small_table
#> _columns: 8
#> _rows: 13
#> _type: tbl_df
#> columns:
#>   date_time:
#>     _type: POSIXct, POSIXt
#>   date:
#>     _type: Date
#>   a:
#>     _type: integer
#>   b:
#>     _type: character
#>   c:
#>     _type: numeric
#>   d:
#>     _type: numeric
#>   e:
#>     _type: logical
#>   f:
#>     _type: character

# We can add keys and values to
# add more pertinent information; with
# some direct editing of the file we get:

#> info_label: '[2020-09-06|13:37:38]'
#> table:
#>   name: small_table
#>   _columns: 8
#>   _rows: 13
#>   _type: tbl_df
#> columns:
```

```
#>   date_time:
#>     _type: POSIXct, POSIXt
#>     info: Date-time values.
#>   date:
#>     _type: Date
#>     info: Date values (the date part of `date_time`).
#>   a:
#>     _type: integer
#>     info: Small integer values (no missing values).
#>   b:
#>     _type: character
#>     info: Strings with a common pattern.
#>   c:
#>     _type: numeric
#>     info: Small numeric values (contains missing values).
#>   d:
#>     _type: numeric
#>     info: Large numeric values (much greater than `c`).
#>   e:
#>     _type: logical
#>     info: TRUE and FALSE values.
#>   f:
#>     _type: character
#>     info: Strings of the set `"low"`, `"mid"`, and `"high"`.

# We could also have done the same
# with the `informant` object by use of
# the `info_columns()` function

# The 'informant-small_table.yml' file
# is available in the package through
# `system.file()`
yml_file <-
  system.file(
    "informant-small_table.yml",
    package = "pointblank"
  )

# We can read this YAML file back
# as an `informant` object by using
# `yaml_read_informant()`
informant <-
  yaml_read_informant(path = yml_file)

class(informant)
```

---

yaml_write                  *Write an* agent *and* informant *to a* **pointblank** *YAML file*

---

**Description**

With `yaml_write()` we can take an existing *agent* and write that *agent*'s validation plan to a YAML file. With **pointblank** YAML, we can modify the YAML markup if so desired, or, use as is to create a new agent with the `yaml_read_agent()` function. That *agent* will have a validation plan and is ready to `interrogate()` the data. We can go a step further and perform an interrogation directly from the YAML file with the `yaml_agent_interrogate()` function. That returns an agent with intel (having already interrogated the target data table). An *informant* object can also be written to YAML with `yaml_write()`.

One requirement for writing the *agent* to YAML is that we need to have a table-reading function (`read_fn`) specified (it's a function that is used to read the target table when `interrogate()` is called). This option can be set when using `create_agent()` or with `set_read_fn()` (for use with an existing *agent*).

**Usage**

```
yaml_write(agent = NULL, informant = NULL, filename, path = NULL)
```

**Arguments**

| | |
|---|---|
| agent | An *agent* object of class ptblank_agent. |
| informant | An *informant* object of class ptblank_informant. |
| filename | The name of the YAML file to create on disk. It is recommended that either the .yaml or .yml extension be used for this file. |
| path | An optional path to which the YAML file should be saved (combined with filename). |

**Function ID**

9-1

**See Also**

Other pointblank YAML: `yaml_agent_interrogate()`, `yaml_agent_show_exprs()`, `yaml_agent_string()`, `yaml_informant_incorporate()`, `yaml_read_agent()`, `yaml_read_informant()`

**Examples**

```
# Let's go through the process of
# developing an agent with a validation
# plan (to be used for the data quality
# analysis of the `small_table` dataset),
# and then offloading that validation
# plan to a pointblank YAML file

# We ought to think about what's
# tolerable in terms of data quality so
# let's designate proportional failure
# thresholds to the `warn`, `stop`, and
# `notify` states using `action_levels()`
```

```
al <-
  action_levels(
    warn_at = 0.10,
    stop_at = 0.25,
    notify_at = 0.35
  )

# Now create a pointblank `agent` object
# and give it the `al` object (which
# serves as a default for all validation
# steps which can be overridden); the
# data will be referenced in a `read_fn`
# (a requirement for writing to YAML)
agent <-
  create_agent(
    read_fn = ~small_table,
    label = "A simple example with the `small_table`.",
    actions = al
  )

# Then, as with any `agent` object, we
# can add steps to the validation plan by
# using as many validation functions as we
# want
agent <-
  agent %>%
  col_exists(vars(date, date_time)) %>%
  col_vals_regex(
    vars(b), "[0-9]-[a-z]{3}-[0-9]{3}"
  ) %>%
  rows_distinct() %>%
  col_vals_gt(vars(d), 100) %>%
  col_vals_lte(vars(c), 5)

# The agent can be written to a pointblank
# YAML file with `yaml_write()`
# yaml_write(
#   agent = agent,
#   filename = "agent-small_table.yml"
# )

# The 'agent-small_table.yml' file is
# available in the package through
# `system.file()`
yml_file <-
  system.file(
    "agent-small_table.yml",
    package = "pointblank"
  )

# We can view the YAML file in the console
# with the `yaml_agent_string()` function
yaml_agent_string(path = yml_file)
```

```
# The YAML can also be printed in the console
# by supplying the agent as the input
yaml_agent_string(agent = agent)

# At a later time, the YAML file can
# be read into a new agent with the
# `yaml_read_agent()` function
agent <-
  yaml_read_agent(path = yml_file)

class(agent)

# We can interrogate the data (which
# is accessible through the `read_fn`)
# with `interrogate()` and get an
# agent with intel, or, we can
# interrogate directly from the YAML
# file with `yaml_agent_interrogate()`
agent <-
  yaml_agent_interrogate(path = yml_file)

class(agent)
```

# Index