Package 'pointblank'

January 23, 2022

Type Package

Version 0.10.0

Title Data Validation and Organization of Metadata for Local and Remote Tables

Description Validate data in data frames, 'tibble' objects, 'Spark' 'DataFrames', and database tables. 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://rich-iannone.github.io/pointblank/, https://github.com/rich-iannone/pointblank

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

Encoding UTF-8

LazyData true

ByteCompile true

RoxygenNote 7.1.2

Depends R (>= 3.5.0)

Imports base64enc (>= 0.1-3), blastula (>= 0.3.2), cli (>= 2.5.0), DBI (>= 1.1.0), digest (>= 0.6.27), dplyr (>= 1.0.6), dbplyr (>= 2.1.1), fs (>= 1.5.0), glue (>= 1.4.2), gt (>= 0.3.0), htmltools (>= 0.5.1.1), knitr (>= 1.30), rlang (>= 0.4.11), magrittr, scales (>= 1.1.1), testthat (>= 2.3.2), tibble (>= 3.1.2), tidyr (>= 1.1.3), tidyselect (>= 1.1.1), yaml (>= 2.2.1)

Suggests arrow, covr, crayon, data.table, duckdb, ggforce, ggplot2, jsonlite, log4r, lubridate, RSQLite, 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 2022-01-23 20:12:41 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 (which also has an `actions` argument). 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 operating directly on data (e.g., `mtcars %>% col_vals_lt("mpg",35)`). There are two helper functions that are convenient when using validation functions directly on data (the `agent`-less workflow): `warn_on_fail()` and `stop_on_fail()`. These helpers either warn or stop (default failure 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 before entering the `warn`, `stop`, or `notify` failure states. If this a decimal value between `0` and `1` then it's a proportional failure threshold (e.g., `0.15` indicates that if 15% percent of the test units are found to *fail*, then the designated failure state is entered). Absolute values starting from `1` can be used instead, and this constitutes an absolute failure threshold (e.g., `10` means that if 10 of the 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 provided as formulas (e.g., `list(warn = ~ warning("Too many failures."))`. A series of expressions for each named state can be used by enclosing the set of statements with `{ }`.

Details

The output of the `action_levels()` call in `actions` will be interpreted slightly differently if using an *agent* or using validation 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. Furthermore, 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 sufficient). 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: create_agent(), create_informant(), db_tbl(), draft_validation(), file_tbl(), scan_data(), tbl_get(), tbl_source(), tbl_store(), validate_rmd()

Examples

```
# For these examples, we will use the
# included `small_table` dataset
small_table

# Create an `action_levels` object
# 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
  )

# A summary of settings for the `al`
# object is shown by printing it
al

# Create a pointblank agent and
# apply the `al` object to `actions`;
# add two validation steps and
# interrogate the `small_table`
agent_1 <-
  create_agent(
    tbl = small_table,
    actions = al
  ) %>%
  col_vals_gt(
    vars(a), value = 2
  ) %>%
  col_vals_lt(
    vars(d), value = 20000
  ) %>%
  interrogate()

# The report from the agent will show
# that the `warn` state has been entered
# for the first validation step but not
# the second one; we can confirm this
# in the console by inspecting the
# `warn` component in the agent's x-list
x_list <- get_agent_x_list(agent_1)
x_list$warn

# Applying the `action_levels` object
```

```
# to the agent means that all validation
# steps will inherit these settings but
# we can override this by applying
# another such object to the validation
# step instead (this time using the
# `warn_on_fail()` shorthand)
agent_2 <-
  create_agent(
    tbl = small_table,
    actions = al
  ) %>%
  col_vals_gt(
    vars(a), value = 2,
    actions = warn_on_fail(warn_at = 0.5)
  ) %>%
  col_vals_lt(
    vars(d), value = 20000
  ) %>%
  interrogate()

# In this case, the first validation
# step has a less stringent failure
# threshold for the `warn` state and it's
# high enough that the condition is not
# entered; this can be confirmed in the
# console through inspection of the
# x-list `warn` component
x_list <- get_agent_x_list(agent_2)
x_list$warn

if (interactive()) {

# In the context of using validation
# functions directly on data (i.e., no
# involvement of an agent) we want to
# trigger warnings and raise errors; the
# following will yield a warning if
# it is executed (returning the
# `small_table` data)
small_table %>%
  col_vals_gt(
    vars(a), value = 2,
    actions = warn_on_fail(warn_at = 2)
  )

# With the same pipeline, not supplying
# anything for `actions` (it's `NULL` by
# default) will have the same effect as
# using `stop_on_fail(stop_at = 1)`
small_table %>%
  col_vals_gt(vars(a), value = 2)

small_table %>%
```

```
  col_vals_gt(
    vars(a), value = 2,
    actions = stop_on_fail(stop_at = 1)
  )

# This is because the `stop_on_fail()`
# call is auto-injected in the default
# case (when operating on data) for your
# convenience; behind the scenes a
# 'secret agent' uses 'covert actions':
# all so you can type less

}
```

`activate_steps` *Activate one or more of an* agent*'s validation steps*

Description

If certain validation steps need to be activated after the creation of the validation plan for an *agent*, use the `activate_steps()` function. This is equivalent to using the `active = TRUE` for the selected validation steps (`active` is an argument in all validation functions). This will replace any function that may have been defined for the `active` argument during creation of the targeted validation steps.

Usage

```
activate_steps(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 definition.

Value

A `ptblank_agent` object.

Function ID

9-5

See Also

For the opposite behavior, use the `deactivate_steps()` function.

Other Object Ops: `deactivate_steps()`, `export_report()`, `remove_steps()`, `set_tbl()`, `x_read_disk()`, `x_write_disk()`

Examples

```
# Create an agent that has the
# `small_table` object as the
# target table, add a few inactive
# validation steps, and then use
# `interrogate()`
agent_1 <-
  create_agent(
    tbl = small_table,
    tbl_name = "small_table",
    label = "An example."
  ) %>%
  col_exists(
    vars(date),
    active = FALSE
  ) %>%
  col_vals_regex(
    vars(b), regex = "[0-9]-[a-z]{3}-[0-9]{3}",
    active = FALSE
  ) %>%
  interrogate()

# In the above, the data is
# not actually interrogated
# because the `active` setting
# was `FALSE` in all steps; we
# can selectively change this
# with `activate_steps()`
agent_2 <-
  agent_1 %>%
  activate_steps(i = 1) %>%
  interrogate()
```

`affix_date` *Put the current date into a file name*

Description

This function helps to affix the current date to a filename. This is useful when writing *agent* and/or *informant* objects to disk as part of a continuous process. The date can be in terms of UTC time or the local system time. The date can be affixed either to the end of the filename (before the file extension) or at the beginning with a customizable delimiter.

The `x_write_disk()`, `yaml_write()` functions allow for the writing of **pointblank** objects to disk. Furthermore the `log4r_step()` function has the `append_to` argument that accepts filenames, and, it's reasonable that a series of log files could be differentiated by a date component in the naming scheme. The modification of the filename string takes effect immediately but not at the time of writing a file to disk. In most cases, especially when using `affix_date()` with the aforementioned file-writing functions, the file timestamps should approximate the time components affixed to the filenames.

Usage

```
affix_date(
  filename,
  position = c("end", "start"),
  format = "%Y-%m-%d",
  delimiter = "_",
  utc_time = TRUE
)
```

Arguments

filename	The filename to modify.
position	Where to place the formatted date. This could either be at the "end" of the filename (the default) or at the "start".
format	A base::strptime() format string for formatting the date. By default, this is "%Y-%m-%d" which expresses the date according to the ISO 8601 standard (as YYYY-MM-DD). Refer to the documentation on base::strptime() for conversion specifications if planning to use a different format string.
delimiter	The delimiter characters to use for separating the date string from the original file name.
utc_time	An option for whether to use the current UTC time to establish the date (the default, with TRUE), or, use the system's local time (FALSE).

Value

A character vector.

Function ID

13-3

See Also

The affix_datetime() function provides the same features except it produces a date-time string by default.

Other Utility and Helper Functions: affix_datetime(), col_schema(), from_github(), has_columns(), stop_if_not()

Examples

```
# Taking the generic `pb_file` name for
# a file, we add the current date to it
# as a suffix
affix_date(filename = "pb_file")

# File extensions won't get in the way:
affix_date(filename = "pb_file.rds")
```

```
# The date can be used as a prefix
affix_date(
  filename = "pb_file",
  position = "start"
)

# The date pattern can be changed and so
# can the delimiter
affix_date(
  filename = "pb_file.yml",
  format = "%Y%m%d",
  delimiter = "-"
)

if (interactive()) {

# We can use a file-naming convention
# involving dates when writing output
# files immediately after interrogating;
# useful when interrogating directly
# from YAML in a scheduled process
yaml_agent_interrogate(
  filename = system.file(
    "yaml", "agent-small_table.yml",
    package = "pointblank"
  )
) %>%
  x_write_disk(
    filename = affix_date(
      filename = "small_table_agent.rds",
      delimiter = "-"
    ),
    keep_tbl = TRUE,
    keep_extracts = TRUE
  )

}
```

`affix_datetime` *Put the current date-time into a file name*

Description

This function helps to affix the current date-time to a filename. This is useful when writing *agent* and/or *informant* objects to disk as part of a continuous process. The date-time string can be based on the current UTC time or the local system time. The date-time can be affixed either to the end of the filename (before the file extension) or at the beginning with a customizable delimiter. Optionally, the time zone information can be included. If the date-time is based on the local system time, the

user system time zone is shown with the format <time>(+/-)hhmm. If using UTC time, then the <time>Z format is adopted.

The `x_write_disk()`, `yaml_write()` functions allow for the writing of **pointblank** objects to disk. The modification of the filename string takes effect immediately but not at the time of writing a file to disk. In most cases, especially when using `affix_datetime()` with the aforementioned file-writing functions, the file timestamps should approximate the time components affixed to the filenames.

Usage

```
affix_datetime(
  filename,
  position = c("end", "start"),
  format = "%Y-%m-%d_%H-%M-%S",
  delimiter = "_",
  utc_time = TRUE,
  add_tz = FALSE
)
```

Arguments

`filename`	The filename to modify.
`position`	Where to place the formatted date-time. This could either be at the `"end"` of the filename (the default) or at the `"start"`.
`format`	A `base::strptime()` format string for formatting the date-time. By default, this is `"%Y-%m-%dT%H:%M:%S"` which expresses the date according to the ISO 8601 standard. For example, if the current date-time is 2020-12-04 13:11:23, the formatted string would become `"2020-12-04T13:11:23"`. Refer to the documentation on `base::strptime()` for conversion specifications if planning to use a different format string.
`delimiter`	The delimiter characters to use for separating the date-time string from the original file name.
`utc_time`	An option for whether to use the current UTC time to establish the date-time (the default, with `TRUE`), or, use the system's local time (`FALSE`).
`add_tz`	Should the time zone (as an offset from UTC) be provided? If `TRUE` then the UTC offset will be either provided as <time>Z (if `utc_time = TRUE`) or <time>(+/-)hhmm. By default, this is `FALSE`.

Value

A character vector.

Function ID

13-4

See Also

The affix_date() function provides the same features except it produces a date string by default.

Other Utility and Helper Functions: affix_date(), col_schema(), from_github(), has_columns(), stop_if_not()

Examples

```
# Taking the generic `pb_file` name for
# a file, we add the current date-time to it
# as a suffix
affix_datetime(filename = "pb_file")

# File extensions won't get in the way:
affix_datetime(filename = "pb_file.rds")

# The date-time can be used as a prefix
affix_datetime(
  filename = "pb_file",
  position = "start"
)

# The date-time pattern can be changed and so
# can the delimiter
affix_datetime(
  filename = "pb_file.yml",
  format = "%Y%m%d_%H%M%S",
  delimiter = "-"
)

# Time zone information can be included
affix_datetime(
  filename = "pb_file.yml",
  add_tz = TRUE
)

if (interactive()) {

# We can use a file-naming convention
# involving date-times when writing output
# files immediately after interrogating;
# useful when interrogating directly
# from YAML in a scheduled process
yaml_agent_interrogate(
  filename = system.file(
    "yaml", "agent-small_table.yml",
    package = "pointblank"
  )
) %>%
  x_write_disk(
    filename = affix_datetime(
      filename = "small_table_agent.rds",
      delimiter = "-"
```

```
    ),
    keep_tbl = TRUE,
    keep_extracts = TRUE
  )

}
```

`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 *failing* test units? Using the `all_passed()` function will let us know whether that's `TRUE` or not.

Usage

```
all_passed(agent, i = NULL)
```

Arguments

`agent`	An agent object of class `ptblank_agent`.
`i`	A vector of validation step numbers. These values are assigned to each validation step by **pointblank** in the order of definition. If `NULL` (the default), all validation steps will be used for the evaluation of complete *passing*.

Details

The `all_passed()` function provides a single logical value based on an interrogation performed in the *agent*-based workflow. For very large-scale validation (where data quality is a known issue, and is perhaps something to be tamed over time) this function is likely to be less useful since it is quite stringent (all test units must pass across all validation steps).

Should there be a requirement for logical values produced from validation, a more flexible alternative is in using the test (test_*()) variants of the validation functions. Each of those produce a single logical value and each and have a `threshold` option for failure levels. Another option is to utilize post-interrogation objects within the *agent*'s x-list (obtained by using the `get_agent_x_list()` function). This allows for many possibilities in producing a single logical value from an interrogation.

Value

A logical value.

Function ID

8-4

See Also

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

Examples

```
# Create a simple table with
# a column of numerical values
tbl <-
  dplyr::tibble(a = c(4, 5, 7, 8))

# Validate that values in column
# `a` are always greater than 4
agent <-
  create_agent(tbl = tbl) %>%
  col_vals_gt(vars(a), value = 3) %>%
  col_vals_lte(vars(a), value = 10) %>%
  col_vals_increasing(vars(a)) %>%
  interrogate()

# Determine if these column
# validations have all passed by
# using `all_passed()` (they do)
all_passed(agent = 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().
columns	One or more columns from the table in focus. This can be provided as a vector of column names using c() or bare column names enclosed in vars().
actions	A list containing threshold levels so that the validation step can react accordingly when exceeding the set levels. This is to be created with the action_levels() helper function.
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 validation 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 validation function will be operating directly on data (no *agent* involvement), then any step with active = FALSE will simply pass the data through with no validation whatsoever. Aside from a logical vector, a one-sided R formula using a leading ~ can be used with . (serving as the input data table) to evaluate to a single logical value. With this approach, the **pointblank** function has_columns() can be used to determine whether to make a validation step active on the basis of one or more columns existing in the table (e.g., ~ . %>% has_columns(vars(d,e))). The default for active 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.

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.

Column Names

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().

Actions

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).

Briefs

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.

YAML

A **pointblank** agent can be written to YAML with yaml_write() and the resulting YAML can be used to regenerate an agent (with yaml_read_agent()) or interrogate the target table (via yaml_agent_interrogate()). When col_exists() is represented in YAML (under the top-level steps key as a list member), the syntax closely follows the signature of the validation function. Here is an example of how a complex call of col_exists() as a validation step is expressed in R code and in the corresponding YAML representation.

```
# R statement
agent %>%
  col_exists(
    vars(a),
    actions = action_levels(warn_at = 0.1, stop_at = 0.2),
    label = "The `col_exists()` step.",
    active = FALSE
  )

# YAML representation
steps:
- col_exists:
    columns: vars(a)
    actions:
      warn_fraction: 0.1
      stop_fraction: 0.2
    label: The `col_exists()` step.
    active: false
```

In practice, both of these will often be shorter as only the `columns` argument requires a value. Arguments with default values won't be written to YAML when using `yaml_write()` (though it is acceptable to include them with their default when generating the YAML by other means). It is also possible to preview the transformation of an agent to YAML without any writing to disk by using the `yaml_agent_string()` function.

Function ID

2-29

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_decreasing()`, `col_vals_equal()`, `col_vals_expr()`, `col_vals_gte()`, `col_vals_gt()`, `col_vals_in_set()`, `col_vals_increasing()`, `col_vals_lte()`, `col_vals_lt()`, `col_vals_make_set()`, `col_vals_make_subset()`, `col_vals_not_between()`, `col_vals_not_equal()`, `col_vals_not_in_set()`, `col_vals_not_null()`, `col_vals_null()`, `col_vals_regex()`, `col_vals_within_spec()`, `conjointly()`, `row_count_match()`, `rows_complete()`, `rows_distinct()`, `serially()`, `specially()`, `tbl_match()`

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 function, 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 validation 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 validation function will be operating directly on data (no *agent* involvement), then any step with `active = FALSE` will simply pass the data through with no validation whatsoever. Aside from a logical vector, a one-sided R formula using a leading `~` can be used with `.` (serving as the input data table) to evaluate to a single logical value. With this approach, the **pointblank** function `has_columns()` can be used to determine whether to make a validation step active on the basis of one or more columns existing in the table (e.g., `~ . %>% has_columns(vars(d,e))`). The default for `active` 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.

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.

Column Names

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()`.

Actions

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 will `stop()`).

Briefs

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.

YAML

A **pointblank** agent can be written to YAML with `yaml_write()` and the resulting YAML can be used to regenerate an agent (with `yaml_read_agent()`) or interrogate the target table (via `yaml_agent_interrogate()`). When `col_is_character()` is represented in YAML (under the top-level `steps` key as a list member), the syntax closely follows the signature of the validation function. Here is an example of how a complex call of `col_is_character()` as a validation step is expressed in R code and in the corresponding YAML representation.

```
# R statement
agent %>%
  col_is_character(
    vars(a),
    actions = action_levels(warn_at = 0.1, stop_at = 0.2),
    label = "The `col_is_character()` step.",
    active = FALSE
  )

# YAML representation
steps:
- col_is_character:
    columns: vars(a)
    actions:
      warn_fraction: 0.1
      stop_fraction: 0.2
    label: The `col_is_character()` step.
    active: false
```

In practice, both of these will often be shorter as only the `columns` argument requires a value. Arguments with default values won't be written to YAML when using `yaml_write()` (though it is acceptable to include them with their default when generating the YAML by other means). It is also

possible to preview the transformation of an agent to YAML without any writing to disk by using the yaml_agent_string() function.

Function ID

2-22

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_decreasing(), col_vals_equal(), col_vals_expr(), col_vals_gte(), col_vals_gt(), col_vals_in_set(), col_vals_increasing(), col_vals_lte(), col_vals_lt(), col_vals_make_set(), col_vals_make_subset(), col_vals_not_between(), col_vals_not_equal(), col_vals_not_in_set(), col_vals_not_null(), col_vals_null(), col_vals_regex(), col_vals_within_spec(), conjointly(), row_count_match(), rows_complete(), rows_distinct(), serially(), specially(), tbl_match()

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 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 validation 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 validation function will be operating directly on data (no *agent* involvement), then any step with active = FALSE will simply pass the data through with no validation whatsoever. Aside from a logical vector, a one-sided R formula using a leading ~ can be used with . (serving as the input data table) to evaluate to a single logical value. With this approach, the **pointblank** function has_columns() can be used to determine whether to make a validation step active on the basis of one or more columns existing in the table (e.g., ~ . %>% has_columns(vars(d,e))). The default for active 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.

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.

Column Names

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().

Actions

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 will stop()).

Briefs

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.

YAML

A **pointblank** agent can be written to YAML with yaml_write() and the resulting YAML can be used to regenerate an agent (with yaml_read_agent()) or interrogate the target table (via yaml_agent_interrogate()). When col_is_date() is represented in YAML (under the top-level steps key as a list member), the syntax closely follows the signature of the validation function. Here is an example of how a complex call of col_is_date() as a validation step is expressed in R code and in the corresponding YAML representation.

```
# R statement
agent %>%
```

```
  col_is_date(
    vars(a),
    actions = action_levels(warn_at = 0.1, stop_at = 0.2),
    label = "The `col_is_date()` step.",
    active = FALSE
  )

# YAML representation
steps:
- col_is_date:
    columns: vars(a)
    actions:
      warn_fraction: 0.1
      stop_fraction: 0.2
    label: The `col_is_date()` step.
    active: false
```

In practice, both of these will often be shorter as only the `columns` argument requires a value. Arguments with default values won't be written to YAML when using `yaml_write()` (though it is acceptable to include them with their default when generating the YAML by other means). It is also possible to preview the transformation of an agent to YAML without any writing to disk by using the `yaml_agent_string()` function.

Function ID

2-26

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_decreasing()`, `col_vals_equal()`, `col_vals_expr()`, `col_vals_gte()`, `col_vals_gt()`, `col_vals_in_set()`, `col_vals_increasing()`, `col_vals_lte()`, `col_vals_lt()`, `col_vals_make_set()`, `col_vals_make_subset()`, `col_vals_not_between()`, `col_vals_not_equal()`, `col_vals_not_in_set()`, `col_vals_not_null()`, `col_vals_null()`, `col_vals_regex()`, `col_vals_within_spec()`, `conjointly()`, `row_count_match()`, `rows_complete()`, `rows_distinct()`, `serially()`, `specially()`, `tbl_match()`

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 validation 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 validation function will be operating directly on data (no *agent* involvement), then any step with `active = FALSE` will simply pass the data through with no validation whatsoever. Aside from a logical vector, a one-sided R formula using a leading `~` can be used with `.` (serving as the input data table) to evaluate to a single logical value. With this approach, the **pointblank** function `has_columns()` can be used to determine whether to make a validation step active on the basis of one or more columns existing in the table (e.g., `~ . %>% has_columns(vars(d,e))`). The default for `active` 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.

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.

Column Names

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()`.

Actions

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 will `stop()`).

Briefs

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.

YAML

A **pointblank** agent can be written to YAML with `yaml_write()` and the resulting YAML can be used to regenerate an agent (with `yaml_read_agent()`) or interrogate the target table (via `yaml_agent_interrogate()`). When `col_is_factor()` is represented in YAML (under the top-level `steps` key as a list member), the syntax closely follows the signature of the validation function. Here is an example of how a complex call of `col_is_factor()` as a validation step is expressed in R code and in the corresponding YAML representation.

```
# R statement
agent %>%
  col_is_factor(
    vars(a),
    actions = action_levels(warn_at = 0.1, stop_at = 0.2),
    label = "The `col_is_factor()` step.",
    active = FALSE
  )

# YAML representation
steps:
- col_is_factor:
    columns: vars(a)
    actions:
      warn_fraction: 0.1
      stop_fraction: 0.2
    label: The `col_is_factor()` step.
    active: false
```

In practice, both of these will often be shorter as only the `columns` argument requires a value. Arguments with default values won't be written to YAML when using `yaml_write()` (though it is acceptable to include them with their default when generating the YAML by other means). It is also possible to preview the transformation of an agent to YAML without any writing to disk by using the `yaml_agent_string()` function.

Function ID

2-28

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_decreasing(), col_vals_equal(), col_vals_expr(), col_vals_gte(), col_vals_gt(), col_vals_in_set(), col_vals_increasing(), col_vals_lte(), col_vals_lt(), col_vals_make_set(), col_vals_make_subset(), col_vals_not_between(), col_vals_not_equal(), col_vals_not_in_set(), col_vals_not_null(), col_vals_null(), col_vals_regex(), col_vals_within_spec(), conjointly(), row_count_match(), rows_complete(), rows_distinct(), serially(), specially(), tbl_match()

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().
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 validation 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 validation function will be operating directly on data (no *agent* involvement), then any step with active = FALSE will simply pass the data through with no validation whatsoever. Aside from a logical vector, a one-sided R formula using a leading ~ can be used with . (serving as the input data table) to evaluate to a single logical value. With this approach, the **pointblank** function has_columns() can be used to determine whether to make a validation step active on the basis of one or more columns existing in the table (e.g., ~ . %>% has_columns(vars(d,e))). The default for active 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.

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.

Column Names

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()`.

Actions

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 will `stop()`).

Briefs

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.

YAML

A **pointblank** agent can be written to YAML with `yaml_write()` and the resulting YAML can be used to regenerate an agent (with `yaml_read_agent()`) or interrogate the target table (via `yaml_agent_interrogate()`). When `col_is_integer()` is represented in YAML (under the top-level `steps` key as a list member), the syntax closely follows the signature of the validation function. Here is an example of how a complex call of `col_is_integer()` as a validation step is expressed in R code and in the corresponding YAML representation.

```
# R statement
agent %>%
  col_is_integer(
    vars(a),
    actions = action_levels(warn_at = 0.1, stop_at = 0.2),
    label = "The `col_is_integer()` step.",
    active = FALSE
  )
```

```
# YAML representation
steps:
- col_is_integer:
    columns: vars(a)
    actions:
      warn_fraction: 0.1
      stop_fraction: 0.2
    label: The `col_is_integer()` step.
    active: false
```

In practice, both of these will often be shorter as only the `columns` argument requires a value. Arguments with default values won't be written to YAML when using `yaml_write()` (though it is acceptable to include them with their default when generating the YAML by other means). It is also possible to preview the transformation of an agent to YAML without any writing to disk by using the `yaml_agent_string()` function.

Function ID

2-24

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_decreasing()`, `col_vals_equal()`, `col_vals_expr()`, `col_vals_gte()`, `col_vals_gt()`, `col_vals_in_set()`, `col_vals_increasing()`, `col_vals_lte()`, `col_vals_lt()`, `col_vals_make_set()`, `col_vals_make_subset()`, `col_vals_not_between()`, `col_vals_not_equal()`, `col_vals_not_in_set()`, `col_vals_not_null()`, `col_vals_null()`, `col_vals_regex()`, `col_vals_within_spec()`, `conjointly()`, `row_count_match()`, `rows_complete()`, `rows_distinct()`, `serially()`, `specially()`, `tbl_match()`

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 validation 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 validation function will be operating directly on data (no *agent* involvement), then any step with `active = FALSE` will simply pass the data through with no validation whatsoever. Aside from a logical vector, a one-sided R formula using a leading `~` can be used with `.` (serving as the input data table) to evaluate to a single logical value. With this approach, the **pointblank** function `has_columns()` can be used to determine whether to make a validation step active on the basis of one or more columns existing in the table (e.g., `~ . %>% has_columns(vars(d,e))`). The default for `active` 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.

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.

Column Names

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()`.

Actions

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 will `stop()`).

Briefs

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.

YAML

A **pointblank** agent can be written to YAML with `yaml_write()` and the resulting YAML can be used to regenerate an agent (with `yaml_read_agent()`) or interrogate the target table (via `yaml_agent_interrogate()`). When `col_is_logical()` is represented in YAML (under the top-level `steps` key as a list member), the syntax closely follows the signature of the validation function. Here is an example of how a complex call of `col_is_logical()` as a validation step is expressed in R code and in the corresponding YAML representation.

```
# R statement
agent %>%
  col_is_logical(
    vars(a),
    actions = action_levels(warn_at = 0.1, stop_at = 0.2),
    label = "The `col_is_logical()` step.",
    active = FALSE
  )

# YAML representation
steps:
- col_is_logical:
    columns: vars(a)
    actions:
      warn_fraction: 0.1
      stop_fraction: 0.2
    label: The `col_is_logical()` step.
    active: false
```

In practice, both of these will often be shorter as only the `columns` argument requires a value. Arguments with default values won't be written to YAML when using `yaml_write()` (though it is acceptable to include them with their default when generating the YAML by other means). It is also possible to preview the transformation of an agent to YAML without any writing to disk by using the `yaml_agent_string()` function.

Function ID

2-25

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_decreasing()`, `col_vals_equal()`, `col_vals_expr()`, `col_vals_gte()`, `col_vals_gt()`,

col_vals_in_set(), col_vals_increasing(), col_vals_lte(), col_vals_lt(), col_vals_make_set(), col_vals_make_subset(), col_vals_not_between(), col_vals_not_equal(), col_vals_not_in_set(), col_vals_not_null(), col_vals_null(), col_vals_regex(), col_vals_within_spec(), conjointly(), row_count_match(), rows_complete(), rows_distinct(), serially(), specially(), tbl_match()

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 validation 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 validation function will be operating directly on data (no *agent* involvement), then any step with `active = FALSE` will simply pass the data through with no validation whatsoever. Aside from a logical vector, a one-sided R formula using a leading `~` can be used with `.` (serving as the input data table) to evaluate to a single logical value. With this approach, the **pointblank** function `has_columns()` can be used to determine whether to make a validation step active on the basis of one or more columns existing in the table (e.g., `~ . %>% has_columns(vars(d,e))`). The default for `active` 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.

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.

Column Names

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()`.

Actions

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 will `stop()`).

Briefs

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.

YAML

A **pointblank** agent can be written to YAML with `yaml_write()` and the resulting YAML can be used to regenerate an agent (with `yaml_read_agent()`) or interrogate the target table (via `yaml_agent_interrogate()`). When `col_is_numeric()` is represented in YAML (under the top-level `steps` key as a list member), the syntax closely follows the signature of the validation function. Here is an example of how a complex call of `col_is_numeric()` as a validation step is expressed in R code and in the corresponding YAML representation.

```
# R statement
agent %>%
  col_is_numeric(
    vars(a),
    actions = action_levels(warn_at = 0.1, stop_at = 0.2),
    label = "The `col_is_numeric()` step.",
    active = FALSE
  )
```

```
# YAML representation
steps:
- col_is_numeric:
    columns: vars(a)
    actions:
      warn_fraction: 0.1
      stop_fraction: 0.2
    label: The `col_is_numeric()` step.
    active: false
```

In practice, both of these will often be shorter as only the `columns` argument requires a value. Arguments with default values won't be written to YAML when using `yaml_write()` (though it is acceptable to include them with their default when generating the YAML by other means). It is also possible to preview the transformation of an agent to YAML without any writing to disk by using the `yaml_agent_string()` function.

Function ID

2-23

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_decreasing()`, `col_vals_equal()`, `col_vals_expr()`, `col_vals_gte()`, `col_vals_gt()`, `col_vals_in_set()`, `col_vals_increasing()`, `col_vals_lte()`, `col_vals_lt()`, `col_vals_make_set()`, `col_vals_make_subset()`, `col_vals_not_between()`, `col_vals_not_equal()`, `col_vals_not_in_set()`, `col_vals_not_null()`, `col_vals_null()`, `col_vals_regex()`, `col_vals_within_spec()`, `conjointly()`, `row_count_match()`, `rows_complete()`, `rows_distinct()`, `serially()`, `specially()`, `tbl_match()`

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 validation 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 validation function will be operating directly on data (no *agent* involvement), then any step with `active = FALSE` will simply pass the data through with no validation whatsoever. Aside from a logical vector, a one-sided R formula using a leading `~` can be used with `.` (serving as the input data table) to evaluate to a single logical value. With this approach, the **pointblank** function `has_columns()` can be used to determine whether to make a validation step active on the basis of one or more columns existing in the table (e.g., `~ . %>% has_columns(vars(d,e))`). The default for `active` 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.

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.

Column Names

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()`.

Actions

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 will `stop()`).

Briefs

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.

YAML

A **pointblank** agent can be written to YAML with `yaml_write()` and the resulting YAML can be used to regenerate an agent (with `yaml_read_agent()`) or interrogate the target table (via `yaml_agent_interrogate()`). When `col_is_posix()` is represented in YAML (under the top-level `steps` key as a list member), the syntax closely follows the signature of the validation function. Here is an example of how a complex call of `col_is_posix()` as a validation step is expressed in R code and in the corresponding YAML representation.

```
# R statement
agent %>%
  col_is_posix(
    vars(a),
    actions = action_levels(warn_at = 0.1, stop_at = 0.2),
    label = "The `col_is_posix()` step.",
    active = FALSE
  )

# YAML representation
steps:
- col_is_posix:
    columns: vars(a)
    actions:
      warn_fraction: 0.1
      stop_fraction: 0.2
    label: The `col_is_posix()` step.
    active: false
```

In practice, both of these will often be shorter as only the `columns` argument requires a value. Arguments with default values won't be written to YAML when using `yaml_write()` (though it is acceptable to include them with their default when generating the YAML by other means). It is also possible to preview the transformation of an agent to YAML without any writing to disk by using the `yaml_agent_string()` function.

Function ID

2-27

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_decreasing()`, `col_vals_equal()`, `col_vals_expr()`, `col_vals_gte()`, `col_vals_gt()`,

col_vals_in_set(), col_vals_increasing(), col_vals_lte(), col_vals_lt(), col_vals_make_set(), col_vals_make_subset(), col_vals_not_between(), col_vals_not_equal(), col_vals_not_in_set(), col_vals_not_null(), col_vals_null(), col_vals_regex(), col_vals_within_spec(), conjointly(), row_count_match(), rows_complete(), rows_distinct(), serially(), specially(), tbl_match()

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. This can either be a table object, a table-prep formula.This can be a table object such as a data frame, a tibble, a `tbl_dbi` object, or a `tbl_spark` object. Alternatively, a table-prep formula (~ <table reading code>) or a function (function() <table reading code>) can be used to lazily read in the table at interrogation time.
`.db_col_types`	Determines whether the column types refer to R column types (`"r"`) or SQL column types (`"sql"`).

Function ID

13-1

See Also

Other Utility and Helper Functions: affix_datetime(), affix_date(), from_github(), has_columns(), stop_if_not()

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 this validation step
# 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`).

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.

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 validation 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 validation function will be operating directly on data (no *agent* involvement), then any step with `active = FALSE` will simply pass the data through with no validation whatsoever. Aside from a logical vector, a one-sided R formula using a leading `~` can be used with `.` (serving as the input data table) to evaluate to a single logical value. With this approach, the **pointblank** function `has_columns()` can be used to determine whether to make a validation step active on the basis of one or more columns existing in the table (e.g., `~ . %>% has_columns(vars(d,e))`). The default for `active` 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.

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.

Actions

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).

Briefs

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.

YAML

A **pointblank** agent can be written to YAML with `yaml_write()` and the resulting YAML can be used to regenerate an agent (with `yaml_read_agent()`) or interrogate the target table (via `yaml_agent_interrogate()`). When `col_schema_match()` is represented in YAML (under the top-level `steps` key as a list member), the syntax closely follows the signature of the validation function. Here is an example of how a complex call of `col_schema_match()` as a validation step is expressed in R code and in the corresponding YAML representation.

```
# R statement
agent %>%
  col_schema_match(
    schema = col_schema(
      a = "integer",
      b = "character"
    ),
    complete = FALSE,
    in_order = FALSE,
    is_exact = FALSE,
    actions = action_levels(stop_at = 1),
    label = "The `col_schema_match()` step.",
    active = FALSE
  )

# YAML representation
steps:
- col_schema_match:
    schema:
      a: integer
      b: character
    complete: false
    in_order: false
    is_exact: false
    actions:
      stop_count: 1.0
    label: The `col_schema_match()` step.
    active: false
```

In practice, both of these will often be shorter as only the schema argument requires a value. Arguments with default values won't be written to YAML when using yaml_write() (though it is acceptable to include them with their default when generating the YAML by other means). It is also possible to preview the transformation of an agent to YAML without any writing to disk by using the yaml_agent_string() function.

Function ID

2-30

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_decreasing(), col_vals_equal(), col_vals_expr(), col_vals_gte(), col_vals_gt(), col_vals_in_set(), col_vals_increasing(), col_vals_lte(), col_vals_lt(), col_vals_make_set(), col_vals_make_subset(), col_vals_not_between(), col_vals_not_equal(), col_vals_not_in_set(), col_vals_not_null(), col_vals_null(), col_vals_regex(), col_vals_within_spec(), conjointly(), row_count_match(), rows_complete(), rows_distinct(), serially(), specially(), tbl_match()

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` *Do column data lie between two specified values or data in other columns?*

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 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,
  segments = 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	An optional expression for mutating the input table before proceeding with the validation. This can either be provided as a one-sided R formula using a leading ~ (e.g., ~ . %>% dplyr::mutate(col = col + 10) or as a function (e.g., function(x) dplyr::mutate(x,col = col + 10). See the *Preconditions* section for more information.
segments	An optional expression or set of expressions (held in a list) that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on. See the *Segments* section for more details on this.
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 validation 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 validation function will be operating directly on data (no *agent* involvement), then any step with `active = FALSE` will simply pass the data through with no validation whatsoever. Aside from a logical vector, a one-sided R formula using a leading `~` can be used with `.` (serving as the input data table) to evaluate to a single logical value. With this approach, the **pointblank** function `has_columns()` can be used to determine whether to make a validation step active on the basis of one or more columns existing in the table (e.g., `~ . %>% has_columns(vars(d,e))`). The default for `active` 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.

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.

Column Names

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()`.

Missing Values

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 `NA`s encountered will accumulate failing test units.

Preconditions

Providing expressions as `preconditions` means **pointblank** will preprocess the target table during interrogation as a preparatory step. It might happen that a particular validation requires a calculated column, some filtering of rows, or the addition of columns via a join, etc. Especially for an *agent*-based report this can be advantageous since we can develop a large validation plan with a single target table and make minor adjustments to it, as needed, along the way.

The table mutation is totally isolated in scope to the validation step(s) where `preconditions` is used. Using **dplyr** code is suggested here since the statements can be translated to SQL if necessary (i.e., if the target table resides in a database). 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_b = col_a + 10)`). Alternatively, a function could instead be supplied (e.g., `function(x) dplyr::mutate(x,col_b = col_a + 10)`).

Segments

By using the `segments` argument, it's possible to define a particular validation with segments (or row slices) of the target table. An optional expression or set of expressions that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on.

As an example of the first type of expression that can be used, `vars(a_column)` will segment the target table in however many unique values are present in the column called `a_column`. This is great if every unique value in a particular column (like different locations, or different dates) requires it's own repeating validation.

With a formula, we can be more selective with which column values should be used for segmentation. Using `a_column ~ c("group_1","group_2")` will attempt to obtain two segments where one is a slice of data where the value `"group_1"` exists in the column named `"a_column"`, and, the other is a slice where `"group_2"` exists in the same column. Each group of rows resolved from the formula will result in a separate validation step.

If there are multiple `columns` specified then the potential number of validation steps will be `m` columns multiplied by `n` segments resolved.

Segmentation will always occur after `preconditions` (i.e., statements that mutate the target table), if any, are applied. With this type of one-two combo, it's possible to generate labels for segmentation using an expression for `preconditions` and refer to those labels in `segments` without having to generate a separate version of the target table.

Actions

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).

Briefs

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.

YAML

A **pointblank** agent can be written to YAML with `yaml_write()` and the resulting YAML can be used to regenerate an agent (with `yaml_read_agent()`) or interrogate the target table (via `yaml_agent_interrogate()`). When `col_vals_between()` is represented in YAML (under the top-level `steps` key as a list member), the syntax closely follows the signature of the validation function. Here is an example of how a complex call of `col_vals_between()` as a validation step is expressed in R code and in the corresponding YAML representation.

```
# R statement
agent %>%
  col_vals_between(
    columns = vars(a),
    left = 1,
    right = 2,
    inclusive = c(TRUE, FALSE),
    na_pass = TRUE,
    preconditions = ~ . %>% dplyr::filter(a < 10),
    segments = b ~ c("group_1", "group_2"),
    actions = action_levels(warn_at = 0.1, stop_at = 0.2),
    label = "The `col_vals_between()` step.",
    active = FALSE
  )

# YAML representation
steps:
- col_vals_between:
    columns: vars(a)
    left: 1.0
    right: 2.0
    inclusive:
    - true
    - false
    na_pass: true
    preconditions: ~. %>% dplyr::filter(a < 10)
    segments: b ~ c("group_1", "group_2")
    actions:
      warn_fraction: 0.1
      stop_fraction: 0.2
    label: The `col_vals_between()` step.
    active: false
```

In practice, both of these will often be shorter as only the `columns`, `left`, and `right` arguments require values. Arguments with default values won't be written to YAML when using `yaml_write()` (though it is acceptable to include them with their default when generating the YAML by other means). It is also possible to preview the transformation of an agent to YAML without any writing to disk by using the `yaml_agent_string()` function.

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_decreasing()`, `col_vals_equal()`, `col_vals_expr()`, `col_vals_gte()`, `col_vals_gt()`, `col_vals_in_set()`, `col_vals_increasing()`, `col_vals_lte()`, `col_vals_lt()`, `col_vals_make_set()`, `col_vals_make_subset()`, `col_vals_not_between()`, `col_vals_not_equal()`, `col_vals_not_in_set()`, `col_vals_not_null()`, `col_vals_null()`, `col_vals_regex()`, `col_vals_within_spec()`, `conjointly()`, `row_count_match()`, `rows_complete()`, `rows_distinct()`, `serially()`, `specially()`, `tbl_match()`

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_decreasing` *Are column data decreasing by row?*

Description

The `col_vals_decreasing()` validation function, the `expect_col_vals_decreasing()` expectation function, and the `test_col_vals_decreasing()` test function all check whether column values in a table are decreasing when moving down a table. There are options for allowing `NA` values in the target column, allowing stationary phases (where consecutive values don't change), and even on for allowing increasing movements up to a certain threshold. 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_decreasing(
  x,
  columns,
  allow_stationary = FALSE,
  increasing_tol = NULL,
  na_pass = FALSE,
  preconditions = NULL,
  segments = NULL,
  actions = NULL,
  step_id = NULL,
  label = NULL,
  brief = NULL,
  active = TRUE
)

expect_col_vals_decreasing(
  object,
  columns,
  allow_stationary = FALSE,
  increasing_tol = NULL,
  na_pass = FALSE,
```

```
  preconditions = NULL,
  threshold = 1
)

test_col_vals_decreasing(
  object,
  columns,
  allow_stationary = FALSE,
  increasing_tol = NULL,
  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.
allow_stationary	An option to allow pauses in decreasing values. For example if the values for the test units are [85, 82, 82, 80, 77] then the third unit (82, appearing a second time) would be marked with *fail* when allow_stationary is FALSE (the default). Using allow_stationary = TRUE will result in all the test units in [85, 82, 82, 80, 77] to be marked with *pass*.
increasing_tol	An optional threshold value that allows for movement of numerical values in the positive direction. By default this is NULL but using a numerical value with set the absolute threshold of positive travel allowed across numerical test units. Note that setting a value here also has the effect of setting allow_stationary to TRUE.
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	An optional expression for mutating the input table before proceeding with the validation. This can either be provided as a one-sided R formula using a leading ~ (e.g., ~ . %>% dplyr::mutate(col = col + 10) or as a function (e.g., function(x) dplyr::mutate(x,col = col + 10). See the *Preconditions* section for more information.
segments	An optional expression or set of expressions (held in a list) that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on. See the *Segments* section for more details on this.
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 validation 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 validation function will be operating directly on data (no *agent* involvement), then any step with `active = FALSE` will simply pass the data through with no validation whatsoever. Aside from a logical vector, a one-sided R formula using a leading `~` can be used with `.` (serving as the input data table) to evaluate to a single logical value. With this approach, the **pointblank** function `has_columns()` can be used to determine whether to make a validation step active on the basis of one or more columns existing in the table (e.g., `~ . %>% has_columns(vars(d,e))`). The default for `active` 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.

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.

Column Names

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()`.

Missing Values

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 `NA`s encountered will accumulate failing test units.

Preconditions

Providing expressions as `preconditions` means **pointblank** will preprocess the target table during interrogation as a preparatory step. It might happen that a particular validation requires a calculated column, some filtering of rows, or the addition of columns via a join, etc. Especially for an *agent*-based report this can be advantageous since we can develop a large validation plan with a single target table and make minor adjustments to it, as needed, along the way.

The table mutation is totally isolated in scope to the validation step(s) where `preconditions` is used. Using **dplyr** code is suggested here since the statements can be translated to SQL if necessary (i.e., if the target table resides in a database). 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_b = col_a + 10)`). Alternatively, a function could instead be supplied (e.g., `function(x) dplyr::mutate(x,col_b = col_a + 10)`).

Segments

By using the `segments` argument, it's possible to define a particular validation with segments (or row slices) of the target table. An optional expression or set of expressions that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on.

As an example of the first type of expression that can be used, `vars(a_column)` will segment the target table in however many unique values are present in the column called `a_column`. This is great if every unique value in a particular column (like different locations, or different dates) requires it's own repeating validation.

With a formula, we can be more selective with which column values should be used for segmentation. Using `a_column ~ c("group_1","group_2")` will attempt to obtain two segments where one is a slice of data where the value `"group_1"` exists in the column named `"a_column"`, and, the other is a slice where `"group_2"` exists in the same column. Each group of rows resolved from the formula will result in a separate validation step.

If there are multiple `columns` specified then the potential number of validation steps will be `m` columns multiplied by `n` segments resolved.

Segmentation will always occur after `preconditions` (i.e., statements that mutate the target table), if any, are applied. With this type of one-two combo, it's possible to generate labels for segmentation

using an expression for `preconditions` and refer to those labels in `segments` without having to generate a separate version of the target table.

Actions

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).

Briefs

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.

YAML

A **pointblank** agent can be written to YAML with `yaml_write()` and the resulting YAML can be used to regenerate an agent (with `yaml_read_agent()`) or interrogate the target table (via `yaml_agent_interrogate()`). When `col_vals_decreasing()` is represented in YAML (under the top-level `steps` key as a list member), the syntax closely follows the signature of the validation function. Here is an example of how a complex call of `col_vals_decreasing()` as a validation step is expressed in R code and in the corresponding YAML representation.

```
# R statement
agent %>%
  col_vals_decreasing(
    columns = vars(a),
    allow_stationary = TRUE,
    increasing_tol = 0.5,
    na_pass = TRUE,
    preconditions = ~ . %>% dplyr::filter(a < 10),
    segments = b ~ c("group_1", "group_2"),
    actions = action_levels(warn_at = 0.1, stop_at = 0.2),
    label = "The `col_vals_decreasing()` step.",
    active = FALSE
  ) %>% yaml_agent_string()

# YAML representation
steps:
- col_vals_decreasing:
    columns: vars(a)
```

```
  allow_stationary: true
  increasing_tol: 0.5
  na_pass: true
  preconditions: ~. %>% dplyr::filter(a < 10)
  segments: b ~ c("group_1", "group_2")
  actions:
    warn_fraction: 0.1
    stop_fraction: 0.2
  label: The `col_vals_decreasing()` step.
  active: false
```

In practice, both of these will often be shorter as only the `columns` argument requires a value. Arguments with default values won't be written to YAML when using `yaml_write()` (though it is acceptable to include them with their default when generating the YAML by other means). It is also possible to preview the transformation of an agent to YAML without any writing to disk by using the `yaml_agent_string()` function.

Function ID

2-14

See Also

The analogous function that moves in the opposite direction: `col_vals_increasing()`.

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_increasing()`, `col_vals_lte()`, `col_vals_lt()`, `col_vals_make_set()`, `col_vals_make_subset()`, `col_vals_not_between()`, `col_vals_not_equal()`, `col_vals_not_in_set()`, `col_vals_not_null()`, `col_vals_null()`, `col_vals_regex()`, `col_vals_within_spec()`, `conjointly()`, `row_count_match()`, `rows_complete()`, `rows_distinct()`, `serially()`, `specially()`, `tbl_match()`

Examples

```
# The `game_revenue` dataset in
# the package has the column
# `session_start`, which contains
# date-time values; let's create
# a column of difftime values (in
# `time_left`) that describes the
# time remaining in the month
# relative to the session start
game_revenue_2 <-
  game_revenue %>%
  dplyr::mutate(
    time_left =
      lubridate::ymd_hms(
        "2015-02-01 00:00:00"
      ) - session_start
  )
```

```
# Let's ensure that the difftime
# values in the new `time_left`
# column has values that are
# decreasing from top to bottom

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

# Validate that all difftime values
# in the column `time_left` are
# decreasing, and, allow for repeating
# values (`allow_stationary` will be
# set to `TRUE`)
agent <-
  create_agent(game_revenue_2) %>%
  col_vals_decreasing(
    vars(time_left),
    allow_stationary = TRUE
  ) %>%
  interrogate()

# Determine if this validation
# had no failing test units (there
# are 2000 test units)
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
game_revenue_2 %>%
  col_vals_decreasing(
    vars(time_left),
    allow_stationary = TRUE
  ) %>%
  dplyr::select(time_left) %>%
  dplyr::distinct() %>%
  dplyr::count()

# 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_decreasing(
  game_revenue_2,
  vars(time_left),
  allow_stationary = TRUE
)

# D: Using the test function

# With the `test_*()` form, we should
# get a single logical value returned
# to us
game_revenue_2 %>%
  test_col_vals_decreasing(
    vars(time_left),
    allow_stationary = TRUE
  )
```

`col_vals_equal` *Are column data equal to a fixed value or data in another column?*

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,
  segments = 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 value used for this test of equality. This can be a single value or a compatible column given in vars(). Any column values equal to what is specified here will pass validation.
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	An optional expression for mutating the input table before proceeding with the validation. This can either be provided as a one-sided R formula using a leading ~ (e.g., ~ . %>% dplyr::mutate(col = col + 10) or as a function (e.g., function(x) dplyr::mutate(x,col = col + 10). See the *Preconditions* section for more information.
segments	An optional expression or set of expressions (held in a list) that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on. See the *Segments* section for more details on this.
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 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 validation 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 validation function will be operating directly on data (no *agent* involvement), then any step with `active = FALSE` will simply pass the data through with no validation whatsoever. Aside from a logical vector, a one-sided R formula using a leading `~` can be used with `.` (serving as the input data table) to evaluate to a single logical value. With this approach, the **pointblank** function `has_columns()` can be used to determine whether to make a validation step active on the basis of one or more columns existing in the table (e.g., `~ . %>% has_columns(vars(d,e))`). The default for `active` 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.

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.

Column Names

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()`.

Missing Values

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 `NA`s encountered will accumulate failing test units.

Preconditions

Providing expressions as `preconditions` means **pointblank** will preprocess the target table during interrogation as a preparatory step. It might happen that a particular validation requires a calculated column, some filtering of rows, or the addition of columns via a join, etc. Especially for an *agent*-based report this can be advantageous since we can develop a large validation plan with a single target table and make minor adjustments to it, as needed, along the way.

The table mutation is totally isolated in scope to the validation step(s) where `preconditions` is used. Using **dplyr** code is suggested here since the statements can be translated to SQL if necessary (i.e., if the target table resides in a database). 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_b = col_a + 10)`). Alternatively, a function could instead be supplied (e.g., `function(x) dplyr::mutate(x,col_b = col_a + 10)`).

Segments

By using the `segments` argument, it's possible to define a particular validation with segments (or row slices) of the target table. An optional expression or set of expressions that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on.

As an example of the first type of expression that can be used, `vars(a_column)` will segment the target table in however many unique values are present in the column called `a_column`. This is great if every unique value in a particular column (like different locations, or different dates) requires it's own repeating validation.

With a formula, we can be more selective with which column values should be used for segmentation. Using `a_column ~ c("group_1","group_2")` will attempt to obtain two segments where one is a slice of data where the value `"group_1"` exists in the column named `"a_column"`, and, the other is a slice where `"group_2"` exists in the same column. Each group of rows resolved from the formula will result in a separate validation step.

If there are multiple `columns` specified then the potential number of validation steps will be `m` columns multiplied by `n` segments resolved.

Segmentation will always occur after `preconditions` (i.e., statements that mutate the target table), if any, are applied. With this type of one-two combo, it's possible to generate labels for segmentation using an expression for `preconditions` and refer to those labels in `segments` without having to generate a separate version of the target table.

Actions

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).

Briefs

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.

YAML

A **pointblank** agent can be written to YAML with `yaml_write()` and the resulting YAML can be used to regenerate an agent (with `yaml_read_agent()`) or interrogate the target table (via `yaml_agent_interrogate()`). When `col_vals_equal()` is represented in YAML (under the top-level `steps` key as a list member), the syntax closely follows the signature of the validation function. Here is an example of how a complex call of `col_vals_equal()` as a validation step is expressed in R code and in the corresponding YAML representation.

```
# R statement
agent %>%
  col_vals_equal(
    columns = vars(a),
    value = 1,
    na_pass = TRUE,
    preconditions = ~ . %>% dplyr::filter(a < 10),
    segments = b ~ c("group_1", "group_2"),
    actions = action_levels(warn_at = 0.1, stop_at = 0.2),
    label = "The `col_vals_equal()` step.",
    active = FALSE
  )

# YAML representation
steps:
- col_vals_equal:
    columns: vars(a)
    value: 1.0
    na_pass: true
    preconditions: ~. %>% dplyr::filter(a < 10)
    segments: b ~ c("group_1", "group_2")
```

```
actions:
  warn_fraction: 0.1
  stop_fraction: 0.2
label: The `col_vals_equal()` step.
active: false
```

In practice, both of these will often be shorter as only the `columns` and `value` arguments require values. Arguments with default values won't be written to YAML when using `yaml_write()` (though it is acceptable to include them with their default when generating the YAML by other means). It is also possible to preview the transformation of an agent to YAML without any writing to disk by using the `yaml_agent_string()` function.

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_decreasing()`, `col_vals_expr()`, `col_vals_gte()`, `col_vals_gt()`, `col_vals_in_set()`, `col_vals_increasing()`, `col_vals_lte()`, `col_vals_lt()`, `col_vals_make_set()`, `col_vals_make_subset()`, `col_vals_not_between()`, `col_vals_not_equal()`, `col_vals_not_in_set()`, `col_vals_not_null()`, `col_vals_null()`, `col_vals_regex()`, `col_vals_within_spec()`, `conjointly()`, `row_count_match()`, `rows_complete()`, `rows_distinct()`, `serially()`, `specially()`, `tbl_match()`

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, the `expect_col_vals_expr()` expectation function, and the `test_col_vals_expr()` test function all check whether column values in a table agree

with 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,
  segments = 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 validation. 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`	An optional expression for mutating the input table before proceeding with the validation. This can either be provided as a one-sided R formula using a leading `~` (e.g., `~ . %>% dplyr::mutate(col = col + 10)` or as a function (e.g., `function(x) dplyr::mutate(x,col = col + 10)`. See the *Preconditions* section for more information.
`segments`	An optional expression or set of expressions (held in a list) that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on. See the *Segments* section for more details on this.
`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 validation 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 validation function will be operating directly on data (no *agent* involvement), then any step with `active = FALSE` will simply pass the data through with no validation whatsoever. Aside from a logical vector, a one-sided R formula using a leading `~` can be used with `.` (serving as the input data table) to evaluate to a single logical value. With this approach, the **pointblank** function `has_columns()` can be used to determine whether to make a validation step active on the basis of one or more columns existing in the table (e.g., `~ . %>% has_columns(vars(d,e))`). The default for `active` 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.

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.

Preconditions

Providing expressions as `preconditions` means **pointblank** will preprocess the target table during interrogation as a preparatory step. It might happen that a particular validation requires a calculated

column, some filtering of rows, or the addition of columns via a join, etc. Especially for an *agent*-based report this can be advantageous since we can develop a large validation plan with a single target table and make minor adjustments to it, as needed, along the way.

The table mutation is totally isolated in scope to the validation step(s) where `preconditions` is used. Using **dplyr** code is suggested here since the statements can be translated to SQL if necessary (i.e., if the target table resides in a database). 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_b = col_a + 10)`). Alternatively, a function could instead be supplied (e.g., `function(x) dplyr::mutate(x,col_b = col_a + 10)`).

Segments

By using the `segments` argument, it's possible to define a particular validation with segments (or row slices) of the target table. An optional expression or set of expressions that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on.

As an example of the first type of expression that can be used, `vars(a_column)` will segment the target table in however many unique values are present in the column called `a_column`. This is great if every unique value in a particular column (like different locations, or different dates) requires it's own repeating validation.

With a formula, we can be more selective with which column values should be used for segmentation. Using `a_column ~ c("group_1","group_2")` will attempt to obtain two segments where one is a slice of data where the value `"group_1"` exists in the column named `"a_column"`, and, the other is a slice where `"group_2"` exists in the same column. Each group of rows resolved from the formula will result in a separate validation step.

If there are multiple `columns` specified then the potential number of validation steps will be `m` columns multiplied by `n` segments resolved.

Segmentation will always occur after `preconditions` (i.e., statements that mutate the target table), if any, are applied. With this type of one-two combo, it's possible to generate labels for segmentation using an expression for `preconditions` and refer to those labels in `segments` without having to generate a separate version of the target table.

Actions

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).

Briefs

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.

YAML

A **pointblank** agent can be written to YAML with `yaml_write()` and the resulting YAML can be used to regenerate an agent (with `yaml_read_agent()`) or interrogate the target table (via `yaml_agent_interrogate()`). When `col_vals_expr()` is represented in YAML (under the top-level `steps` key as a list member), the syntax closely follows the signature of the validation function. Here is an example of how a complex call of `col_vals_expr()` as a validation step is expressed in R code and in the corresponding YAML representation.

```
# R statement
agent %>%
  col_vals_expr(
    expr = ~ a %% 1 == 0,
    preconditions = ~ . %>% dplyr::filter(a < 10),
    segments = b ~ c("group_1", "group_2"),
    actions = action_levels(warn_at = 0.1, stop_at = 0.2),
    label = "The `col_vals_expr()` step.",
    active = FALSE
  )

# YAML representation
steps:
- col_vals_expr:
    expr: ~a%%1 == 0
    preconditions: ~. %>% dplyr::filter(a < 10)
    segments: b ~ c("group_1", "group_2")
    actions:
      warn_fraction: 0.1
      stop_fraction: 0.2
    label: The `col_vals_expr()` step.
    active: false
```

In practice, both of these will often be shorter as only the `expr` argument requires a value. Arguments with default values won't be written to YAML when using `yaml_write()` (though it is acceptable to include them with their default when generating the YAML by other means). It is also possible to preview the transformation of an agent to YAML without any writing to disk by using the `yaml_agent_string()` function.

Function ID

2-19

See Also

These reexported functions (from **rlang** and **dplyr**) work nicely within `col_vals_expr()` and its variants: `rlang::expr()`, `dplyr::between()`, and `dplyr::case_when()`.

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_decreasing()`, `col_vals_equal()`, `col_vals_gte()`, `col_vals_gt()`, `col_vals_in_set()`, `col_vals_increasing()`, `col_vals_lte()`, `col_vals_lt()`, `col_vals_make_set()`, `col_vals_make_subset()`, `col_vals_not_between()`, `col_vals_not_equal()`, `col_vals_not_in_set()`, `col_vals_not_null()`, `col_vals_null()`, `col_vals_regex()`, `col_vals_within_spec()`, `conjointly()`, `row_count_match()`, `rows_complete()`, `rows_distinct()`, `serially()`, `specially()`, `tbl_match()`

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 fixed value or data in another column?*

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,
  segments = 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 value used for this comparison. This can be a single value or a compatible column given in vars(). Any column values greater than what is specified here will pass validation.
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	An optional expression for mutating the input table before proceeding with the validation. This can either be provided as a one-sided R formula using a leading ~ (e.g., ~ . %>% dplyr::mutate(col = col + 10) or as a function (e.g., function(x) dplyr::mutate(x,col = col + 10). See the *Preconditions* section for more information.
segments	An optional expression or set of expressions (held in a list) that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on. See the *Segments* section for more details on this.
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 validation 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 validation function will be operating directly on data (no *agent*

involvement), then any step with `active = FALSE` will simply pass the data through with no validation whatsoever. Aside from a logical vector, a one-sided R formula using a leading `~` can be used with `.` (serving as the input data table) to evaluate to a single logical value. With this approach, the **pointblank** function `has_columns()` can be used to determine whether to make a validation step active on the basis of one or more columns existing in the table (e.g., `~ . %>% has_columns(vars(d,e))`). The default for `active` 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.

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.

Column Names

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()`.

Missing Values

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 `NA`s encountered will accumulate failing test units.

Preconditions

Providing expressions as `preconditions` means **pointblank** will preprocess the target table during interrogation as a preparatory step. It might happen that a particular validation requires a calculated column, some filtering of rows, or the addition of columns via a join, etc. Especially for an *agent*-based report this can be advantageous since we can develop a large validation plan with a single target table and make minor adjustments to it, as needed, along the way.

The table mutation is totally isolated in scope to the validation step(s) where `preconditions` is used. Using **dplyr** code is suggested here since the statements can be translated to SQL if necessary (i.e., if the target table resides in a database). 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_b = col_a + 10)). Alternatively, a function could instead be supplied (e.g., function(x) dplyr::mutate(x,col_b = col_a + 10)).

Segments

By using the segments argument, it's possible to define a particular validation with segments (or row slices) of the target table. An optional expression or set of expressions that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on.

As an example of the first type of expression that can be used, vars(a_column) will segment the target table in however many unique values are present in the column called a_column. This is great if every unique value in a particular column (like different locations, or different dates) requires it's own repeating validation.

With a formula, we can be more selective with which column values should be used for segmentation. Using a_column ~ c("group_1","group_2") will attempt to obtain two segments where one is a slice of data where the value "group_1" exists in the column named "a_column", and, the other is a slice where "group_2" exists in the same column. Each group of rows resolved from the formula will result in a separate validation step.

If there are multiple columns specified then the potential number of validation steps will be m columns multiplied by n segments resolved.

Segmentation will always occur after preconditions (i.e., statements that mutate the target table), if any, are applied. With this type of one-two combo, it's possible to generate labels for segmentation using an expression for preconditions and refer to those labels in segments without having to generate a separate version of the target table.

Actions

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).

Briefs

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.

YAML

A **pointblank** agent can be written to YAML with `yaml_write()` and the resulting YAML can be used to regenerate an agent (with `yaml_read_agent()`) or interrogate the target table (via `yaml_agent_interrogate()`). When `col_vals_gt()` is represented in YAML (under the top-level `steps` key as a list member), the syntax closely follows the signature of the validation function. Here is an example of how a complex call of `col_vals_gt()` as a validation step is expressed in R code and in the corresponding YAML representation.

```
# R statement
agent %>%
  col_vals_gt(
    columns = vars(a),
    value = 1,
    na_pass = TRUE,
    preconditions = ~ . %>% dplyr::filter(a < 10),
    segments = b ~ c("group_1", "group_2"),
    actions = action_levels(warn_at = 0.1, stop_at = 0.2),
    label = "The `col_vals_gt()` step.",
    active = FALSE
  )

# YAML representation
steps:
- col_vals_gt:
    columns: vars(a)
    value: 1.0
    na_pass: true
    preconditions: ~. %>% dplyr::filter(a < 10)
    segments: b ~ c("group_1", "group_2")
    actions:
      warn_fraction: 0.1
      stop_fraction: 0.2
    label: The `col_vals_gt()` step.
    active: false
```

In practice, both of these will often be shorter as only the `columns` and `value` arguments require values. Arguments with default values won't be written to YAML when using `yaml_write()` (though it is acceptable to include them with their default when generating the YAML by other means). It is also possible to preview the transformation of an agent to YAML without any writing to disk by using the `yaml_agent_string()` function.

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_decreasing(), col_vals_equal(), col_vals_expr(), col_vals_gte(), col_vals_in_set(), col_vals_increasing(), col_vals_lte(), col_vals_lt(), col_vals_make_set(), col_vals_make_subset(), col_vals_not_between(), col_vals_not_equal(), col_vals_not_in_set(), col_vals_not_null(), col_vals_null(), col_vals_regex(), col_vals_within_spec(), conjointly(), row_count_match(), rows_complete(), rows_distinct(), serially(), specially(), tbl_match()

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), value = 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), value = 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),
  value = 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),
  value = 4
)
```

`col_vals_gte` *Are column data greater than or equal to a fixed value or data in another column?*

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 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,
  segments = 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 value used for this comparison. This can be a single value or a compatible column given in vars(). Any column values greater than or equal to what is specified here will pass validation.
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	An optional expression for mutating the input table before proceeding with the validation. This can either be provided as a one-sided R formula using a leading ~ (e.g., ~ . %>% dplyr::mutate(col = col + 10) or as a function (e.g., function(x) dplyr::mutate(x,col = col + 10). See the *Preconditions* section for more information.
segments	An optional expression or set of expressions (held in a list) that serve to segment the target table by column values. Each expression can be given in one of two

ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on. See the *Segments* section for more details on this.

`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 validation 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 validation function will be operating directly on data (no *agent* involvement), then any step with `active = FALSE` will simply pass the data through with no validation whatsoever. Aside from a logical vector, a one-sided R formula using a leading `~` can be used with `.` (serving as the input data table) to evaluate to a single logical value. With this approach, the **pointblank** function `has_columns()` can be used to determine whether to make a validation step active on the basis of one or more columns existing in the table (e.g., `~ . %>% has_columns(vars(d,e))`). The default for `active` 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.

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.

Column Names

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()`.

Missing Values

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 `NA`s encountered will accumulate failing test units.

Preconditions

Providing expressions as `preconditions` means **pointblank** will preprocess the target table during interrogation as a preparatory step. It might happen that a particular validation requires a calculated column, some filtering of rows, or the addition of columns via a join, etc. Especially for an *agent*-based report this can be advantageous since we can develop a large validation plan with a single target table and make minor adjustments to it, as needed, along the way.

The table mutation is totally isolated in scope to the validation step(s) where `preconditions` is used. Using **dplyr** code is suggested here since the statements can be translated to SQL if necessary (i.e., if the target table resides in a database). 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_b = col_a + 10)`). Alternatively, a function could instead be supplied (e.g., `function(x) dplyr::mutate(x,col_b = col_a + 10)`).

Segments

By using the `segments` argument, it's possible to define a particular validation with segments (or row slices) of the target table. An optional expression or set of expressions that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on.

As an example of the first type of expression that can be used, `vars(a_column)` will segment the target table in however many unique values are present in the column called `a_column`. This is great if every unique value in a particular column (like different locations, or different dates) requires it's own repeating validation.

With a formula, we can be more selective with which column values should be used for segmentation. Using `a_column ~ c("group_1","group_2")` will attempt to obtain two segments where one is a slice of data where the value `"group_1"` exists in the column named `"a_column"`, and, the

other is a slice where `"group_2"` exists in the same column. Each group of rows resolved from the formula will result in a separate validation step.

If there are multiple `columns` specified then the potential number of validation steps will be `m` columns multiplied by `n` segments resolved.

Segmentation will always occur after `preconditions` (i.e., statements that mutate the target table), if any, are applied. With this type of one-two combo, it's possible to generate labels for segmentation using an expression for `preconditions` and refer to those labels in `segments` without having to generate a separate version of the target table.

Actions

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).

Briefs

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.

YAML

A **pointblank** agent can be written to YAML with `yaml_write()` and the resulting YAML can be used to regenerate an agent (with `yaml_read_agent()`) or interrogate the target table (via `yaml_agent_interrogate()`). When `col_vals_gte()` is represented in YAML (under the top-level `steps` key as a list member), the syntax closely follows the signature of the validation function. Here is an example of how a complex call of `col_vals_gte()` as a validation step is expressed in R code and in the corresponding YAML representation.

```
# R statement
agent %>%
  col_vals_gte(
    columns = vars(a),
    value = 1,
    na_pass = TRUE,
    preconditions = ~ . %>% dplyr::filter(a < 10),
    segments = b ~ c("group_1", "group_2"),
    actions = action_levels(warn_at = 0.1, stop_at = 0.2),
    label = "The `col_vals_gte()` step.",
    active = FALSE
  )
```

```
# YAML representation
steps:
- col_vals_gte:
    columns: vars(a)
    value: 1.0
    na_pass: true
    preconditions: ~. %>% dplyr::filter(a < 10)
    segments: b ~ c("group_1", "group_2")
    actions:
      warn_fraction: 0.1
      stop_fraction: 0.2
    label: The `col_vals_gte()` step.
    active: false
```

In practice, both of these will often be shorter as only the `columns` and `value` arguments require values. Arguments with default values won't be written to YAML when using `yaml_write()` (though it is acceptable to include them with their default when generating the YAML by other means). It is also possible to preview the transformation of an agent to YAML without any writing to disk by using the `yaml_agent_string()` function.

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_decreasing()`, `col_vals_equal()`, `col_vals_expr()`, `col_vals_gt()`, `col_vals_in_set()`, `col_vals_increasing()`, `col_vals_lte()`, `col_vals_lt()`, `col_vals_make_set()`, `col_vals_make_subset()`, `col_vals_not_between()`, `col_vals_not_equal()`, `col_vals_not_in_set()`, `col_vals_not_null()`, `col_vals_null()`, `col_vals_regex()`, `col_vals_within_spec()`, `conjointly()`, `row_count_match()`, `rows_complete()`, `rows_distinct()`, `serially()`, `specially()`, `tbl_match()`

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_increasing *Are column data increasing by row?*

Description

The col_vals_increasing() validation function, the expect_col_vals_increasing() expectation function, and the test_col_vals_increasing() test function all check whether column values in a table are increasing when moving down a table. There are options for allowing NA values in the target column, allowing stationary phases (where consecutive values don't change), and even on for allowing decreasing movements up to a certain threshold. 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_increasing(
  x,
  columns,
  allow_stationary = FALSE,
  decreasing_tol = NULL,
  na_pass = FALSE,
  preconditions = NULL,
  segments = NULL,
  actions = NULL,
  step_id = NULL,
  label = NULL,
  brief = NULL,
  active = TRUE
)

expect_col_vals_increasing(
  object,
  columns,
  allow_stationary = FALSE,
  decreasing_tol = NULL,
  na_pass = FALSE,
  preconditions = NULL,
  threshold = 1
)

test_col_vals_increasing(
  object,
  columns,
  allow_stationary = FALSE,
```

```
  decreasing_tol = NULL,
  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.
allow_stationary	An option to allow pauses in decreasing values. For example if the values for the test units are [80, 82, 82, 85, 88] then the third unit (82, appearing a second time) would be marked with *fail* when allow_stationary is FALSE (the default). Using allow_stationary = TRUE will result in all the test units in [80, 82, 82, 85, 88] to be marked with *pass*.
decreasing_tol	An optional threshold value that allows for movement of numerical values in the negative direction. By default this is NULL but using a numerical value with set the absolute threshold of negative travel allowed across numerical test units. Note that setting a value here also has the effect of setting allow_stationary to TRUE.
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	An optional expression for mutating the input table before proceeding with the validation. This can either be provided as a one-sided R formula using a leading ~ (e.g., ~ . %>% dplyr::mutate(col = col + 10) or as a function (e.g., function(x) dplyr::mutate(x,col = col + 10). See the *Preconditions* section for more information.
segments	An optional expression or set of expressions (held in a list) that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on. See the *Segments* section for more details on this.
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 validation 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 validation function will be operating directly on data (no *agent* involvement), then any step with `active = FALSE` will simply pass the data through with no validation whatsoever. Aside from a logical vector, a one-sided R formula using a leading `~` can be used with `.` (serving as the input data table) to evaluate to a single logical value. With this approach, the **pointblank** function `has_columns()` can be used to determine whether to make a validation step active on the basis of one or more columns existing in the table (e.g., `~ . %>% has_columns(vars(d,e))`). The default for `active` 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.

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.

Column Names

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()`.

Missing Values

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.

Preconditions

Providing expressions as preconditions means **pointblank** will preprocess the target table during interrogation as a preparatory step. It might happen that a particular validation requires a calculated column, some filtering of rows, or the addition of columns via a join, etc. Especially for an *agent*-based report this can be advantageous since we can develop a large validation plan with a single target table and make minor adjustments to it, as needed, along the way.

The table mutation is totally isolated in scope to the validation step(s) where preconditions is used. Using **dplyr** code is suggested here since the statements can be translated to SQL if necessary (i.e., if the target table resides in a database). 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_b = col_a + 10)). Alternatively, a function could instead be supplied (e.g., function(x) dplyr::mutate(x,col_b = col_a + 10)).

Segments

By using the segments argument, it's possible to define a particular validation with segments (or row slices) of the target table. An optional expression or set of expressions that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on.

As an example of the first type of expression that can be used, vars(a_column) will segment the target table in however many unique values are present in the column called a_column. This is great if every unique value in a particular column (like different locations, or different dates) requires it's own repeating validation.

With a formula, we can be more selective with which column values should be used for segmentation. Using a_column ~ c("group_1","group_2") will attempt to obtain two segments where one is a slice of data where the value "group_1" exists in the column named "a_column", and, the other is a slice where "group_2" exists in the same column. Each group of rows resolved from the formula will result in a separate validation step.

If there are multiple columns specified then the potential number of validation steps will be m columns multiplied by n segments resolved.

Segmentation will always occur after preconditions (i.e., statements that mutate the target table), if any, are applied. With this type of one-two combo, it's possible to generate labels for segmentation using an expression for preconditions and refer to those labels in segments without having to generate a separate version of the target table.

Actions

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).

Briefs

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.

YAML

A **pointblank** agent can be written to YAML with `yaml_write()` and the resulting YAML can be used to regenerate an agent (with `yaml_read_agent()`) or interrogate the target table (via `yaml_agent_interrogate()`). When `col_vals_increasing()` is represented in YAML (under the top-level `steps` key as a list member), the syntax closely follows the signature of the validation function. Here is an example of how a complex call of `col_vals_increasing()` as a validation step is expressed in R code and in the corresponding YAML representation.

```
# R statement
agent %>%
  col_vals_increasing(
    columns = vars(a),
    allow_stationary = TRUE,
    decreasing_tol = 0.5,
    na_pass = TRUE,
    preconditions = ~ . %>% dplyr::filter(a < 10),
    segments = b ~ c("group_1", "group_2"),
    actions = action_levels(warn_at = 0.1, stop_at = 0.2),
    label = "The `col_vals_increasing()` step.",
    active = FALSE
  )

# YAML representation
steps:
- col_vals_increasing:
    columns: vars(a)
    allow_stationary: true
    decreasing_tol: 0.5
    na_pass: true
    preconditions: ~. %>% dplyr::filter(a < 10)
    segments: b ~ c("group_1", "group_2")
    actions:
      warn_fraction: 0.1
```

```
  stop_fraction: 0.2
label: The `col_vals_increasing()` step.
active: false
```

In practice, both of these will often be shorter as only the `columns` argument requires a value. Arguments with default values won't be written to YAML when using `yaml_write()` (though it is acceptable to include them with their default when generating the YAML by other means). It is also possible to preview the transformation of an agent to YAML without any writing to disk by using the `yaml_agent_string()` function.

Function ID

2-13

See Also

The analogous function that moves in the opposite direction: `col_vals_decreasing()`.

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_decreasing()`, `col_vals_equal()`, `col_vals_expr()`, `col_vals_gte()`, `col_vals_gt()`, `col_vals_in_set()`, `col_vals_lte()`, `col_vals_lt()`, `col_vals_make_set()`, `col_vals_make_subset()`, `col_vals_not_between()`, `col_vals_not_equal()`, `col_vals_not_in_set()`, `col_vals_not_null()`, `col_vals_null()`, `col_vals_regex()`, `col_vals_within_spec()`, `conjointly()`, `row_count_match()`, `rows_complete()`, `rows_distinct()`, `serially()`, `specially()`, `tbl_match()`

Examples

```
# The `game_revenue` dataset in
# the package has the column
# `session_start`, which contains
# date-time values; let's ensure
# that this column has values that
# are increasing from top to bottom

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

# Validate that all date-time values
# in the column `session_start` are
# increasing, and, allow for repeating
# values (`allow_stationary` will be
# set to `TRUE`)
agent <-
  create_agent(game_revenue) %>%
  col_vals_increasing(
    vars(session_start),
    allow_stationary = TRUE
  ) %>%
  interrogate()

# Determine if this validation
```

```
# had no failing test units (there
# are 2000 test units)
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
game_revenue %>%
  col_vals_increasing(
    vars(session_start),
    allow_stationary = TRUE
  ) %>%
  dplyr::select(session_start) %>%
  dplyr::distinct() %>%
  dplyr::count()

# 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_increasing(
  game_revenue,
  vars(session_start),
  allow_stationary = TRUE
)

# D: Using the test function

# With the `test_*()` form, we should
# get a single logical value returned
# to us
game_revenue %>%
  test_col_vals_increasing(
    vars(session_start),
    allow_stationary = TRUE
  )
```

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 function, 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 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_in_set(
  x,
  columns,
  set,
  preconditions = NULL,
  segments = 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().
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` An optional expression for mutating the input table before proceeding with the validation. This can either be provided as a one-sided R formula using a leading `~` (e.g., `~ . %>% dplyr::mutate(col = col + 10)` or as a function (e.g., `function(x) dplyr::mutate(x,col = col + 10)`. See the *Preconditions* section for more information.

`segments` An optional expression or set of expressions (held in a list) that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on. See the *Segments* section for more details on this.

`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 validation 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 validation function will be operating directly on data (no *agent* involvement), then any step with `active = FALSE` will simply pass the data through with no validation whatsoever. Aside from a logical vector, a one-sided R formula using a leading `~` can be used with `.` (serving as the input data table) to evaluate to a single logical value. With this approach, the **pointblank** function `has_columns()` can be used to determine whether to make a validation step active on the basis of one or more columns existing in the table (e.g., `~ . %>% has_columns(vars(d,e))`). The default for `active` 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.

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.

Column Names

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()`.

Preconditions

Providing expressions as `preconditions` means **pointblank** will preprocess the target table during interrogation as a preparatory step. It might happen that a particular validation requires a calculated column, some filtering of rows, or the addition of columns via a join, etc. Especially for an *agent*-based report this can be advantageous since we can develop a large validation plan with a single target table and make minor adjustments to it, as needed, along the way.

The table mutation is totally isolated in scope to the validation step(s) where `preconditions` is used. Using **dplyr** code is suggested here since the statements can be translated to SQL if necessary (i.e., if the target table resides in a database). 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_b = col_a + 10)`). Alternatively, a function could instead be supplied (e.g., `function(x) dplyr::mutate(x,col_b = col_a + 10)`).

Segments

By using the `segments` argument, it's possible to define a particular validation with segments (or row slices) of the target table. An optional expression or set of expressions that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on.

As an example of the first type of expression that can be used, `vars(a_column)` will segment the target table in however many unique values are present in the column called `a_column`. This is great if every unique value in a particular column (like different locations, or different dates) requires it's own repeating validation.

With a formula, we can be more selective with which column values should be used for segmentation. Using `a_column ~ c("group_1","group_2")` will attempt to obtain two segments where

one is a slice of data where the value `"group_1"` exists in the column named `"a_column"`, and, the other is a slice where `"group_2"` exists in the same column. Each group of rows resolved from the formula will result in a separate validation step.

If there are multiple `columns` specified then the potential number of validation steps will be `m` columns multiplied by `n` segments resolved.

Segmentation will always occur after `preconditions` (i.e., statements that mutate the target table), if any, are applied. With this type of one-two combo, it's possible to generate labels for segmentation using an expression for `preconditions` and refer to those labels in `segments` without having to generate a separate version of the target table.

Actions

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).

Briefs

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.

YAML

A **pointblank** agent can be written to YAML with `yaml_write()` and the resulting YAML can be used to regenerate an agent (with `yaml_read_agent()`) or interrogate the target table (via `yaml_agent_interrogate()`). When `col_vals_in_set()` is represented in YAML (under the top-level `steps` key as a list member), the syntax closely follows the signature of the validation function. Here is an example of how a complex call of `col_vals_in_set()` as a validation step is expressed in R code and in the corresponding YAML representation.

```
# R statement
agent %>%
  col_vals_in_set(
    columns = vars(a),
    set = c(1, 2, 3, 4),
    preconditions = ~ . %>% dplyr::filter(a < 10),
    segments = b ~ c("group_1", "group_2"),
    actions = action_levels(warn_at = 0.1, stop_at = 0.2),
    label = "The `col_vals_in_set()` step.",
    active = FALSE
  )
```

```
# YAML representation
steps:
- col_vals_in_set:
    columns: vars(a)
   set:
   - 1.0
   - 2.0
   - 3.0
   - 4.0
   preconditions: ~. %>% dplyr::filter(a < 10)
   segments: b ~ c("group_1", "group_2")
   actions:
     warn_fraction: 0.1
     stop_fraction: 0.2
   label: The `col_vals_in_set()` step.
   active: false
```

In practice, both of these will often be shorter as only the `columns` and `set` arguments require values. Arguments with default values won't be written to YAML when using `yaml_write()` (though it is acceptable to include them with their default when generating the YAML by other means). It is also possible to preview the transformation of an agent to YAML without any writing to disk by using the `yaml_agent_string()` function.

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_decreasing()`, `col_vals_equal()`, `col_vals_expr()`, `col_vals_gte()`, `col_vals_gt()`, `col_vals_increasing()`, `col_vals_lte()`, `col_vals_lt()`, `col_vals_make_set()`, `col_vals_make_subset()`, `col_vals_not_between()`, `col_vals_not_equal()`, `col_vals_not_in_set()`, `col_vals_not_null()`, `col_vals_null()`, `col_vals_regex()`, `col_vals_within_spec()`, `conjointly()`, `row_count_match()`, `rows_complete()`, `rows_distinct()`, `serially()`, `specially()`, `tbl_match()`

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 fixed value or data in another column?*

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,
  segments = 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 value used for this comparison. This can be a single value or a compatible column given in vars(). Any column values less than what is specified here will pass validation.
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	An optional expression for mutating the input table before proceeding with the validation. This can either be provided as a one-sided R formula using a leading ~ (e.g., ~ . %>% dplyr::mutate(col = col + 10) or as a function (e.g., function(x) dplyr::mutate(x,col = col + 10). See the *Preconditions* section for more information.
segments	An optional expression or set of expressions (held in a list) that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on. See the *Segments* section for more details on this.
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 validation 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 validation function will be operating directly on data (no *agent* involvement), then any step with `active = FALSE` will simply pass the data through with no validation whatsoever. Aside from a logical vector, a one-sided R formula using a leading `~` can be used with `.` (serving as the input data table) to evaluate to a single logical value. With this approach, the **pointblank** function `has_columns()` can be used to determine whether to make a validation step active on the basis of one or more columns existing in the table (e.g., `~ . %>% has_columns(vars(d,e))`). The default for `active` 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.

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.

Column Names

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()`.

Missing Values

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 `NA`s encountered will accumulate failing test units.

Preconditions

Providing expressions as `preconditions` means **pointblank** will preprocess the target table during interrogation as a preparatory step. It might happen that a particular validation requires a calculated column, some filtering of rows, or the addition of columns via a join, etc. Especially for an *agent*-based report this can be advantageous since we can develop a large validation plan with a single target table and make minor adjustments to it, as needed, along the way.

The table mutation is totally isolated in scope to the validation step(s) where `preconditions` is used. Using **dplyr** code is suggested here since the statements can be translated to SQL if necessary (i.e., if the target table resides in a database). 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_b = col_a + 10)`). Alternatively, a function could instead be supplied (e.g., `function(x) dplyr::mutate(x,col_b = col_a + 10)`).

Segments

By using the `segments` argument, it's possible to define a particular validation with segments (or row slices) of the target table. An optional expression or set of expressions that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on.

As an example of the first type of expression that can be used, `vars(a_column)` will segment the target table in however many unique values are present in the column called `a_column`. This is great if every unique value in a particular column (like different locations, or different dates) requires it's own repeating validation.

With a formula, we can be more selective with which column values should be used for segmentation. Using `a_column ~ c("group_1","group_2")` will attempt to obtain two segments where one is a slice of data where the value `"group_1"` exists in the column named `"a_column"`, and, the other is a slice where `"group_2"` exists in the same column. Each group of rows resolved from the formula will result in a separate validation step.

If there are multiple `columns` specified then the potential number of validation steps will be `m` columns multiplied by `n` segments resolved.

Segmentation will always occur after `preconditions` (i.e., statements that mutate the target table), if any, are applied. With this type of one-two combo, it's possible to generate labels for segmentation using an expression for `preconditions` and refer to those labels in `segments` without having to generate a separate version of the target table.

Actions

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).

Briefs

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.

YAML

A **pointblank** agent can be written to YAML with `yaml_write()` and the resulting YAML can be used to regenerate an agent (with `yaml_read_agent()`) or interrogate the target table (via `yaml_agent_interrogate()`). When `col_vals_lt()` is represented in YAML (under the top-level `steps` key as a list member), the syntax closely follows the signature of the validation function. Here is an example of how a complex call of `col_vals_lt()` as a validation step is expressed in R code and in the corresponding YAML representation.

```
# R statement
agent %>%
  col_vals_lt(
    columns = vars(a),
    value = 1,
    na_pass = TRUE,
    preconditions = ~ . %>% dplyr::filter(a < 10),
    segments = b ~ c("group_1", "group_2"),
    actions = action_levels(warn_at = 0.1, stop_at = 0.2),
    label = "The `col_vals_lt()` step.",
    active = FALSE
  )

# YAML representation
steps:
- col_vals_lt:
    columns: vars(a)
    value: 1.0
    na_pass: true
    preconditions: ~. %>% dplyr::filter(a < 10)
    segments: b ~ c("group_1", "group_2")
    actions:
      warn_fraction: 0.1
      stop_fraction: 0.2
    label: The `col_vals_lt()` step.
    active: false
```

In practice, both of these will often be shorter as only the `columns` and `value` arguments require values. Arguments with default values won't be written to YAML when using `yaml_write()` (though it is acceptable to include them with their default when generating the YAML by other means). It is also possible to preview the transformation of an agent to YAML without any writing to disk by using the `yaml_agent_string()` function.

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_decreasing(), col_vals_equal(), col_vals_expr(), col_vals_gte(), col_vals_gt(), col_vals_in_set(), col_vals_increasing(), col_vals_lte(), col_vals_make_set(), col_vals_make_subset(), col_vals_not_between(), col_vals_not_equal(), col_vals_not_in_set(), col_vals_not_null(), col_vals_null(), col_vals_regex(), col_vals_within_spec(), conjointly(), row_count_match(), rows_complete(), rows_distinct(), serially(), specially(), tbl_match()

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 fixed value or data in another column?*

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 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_lte(
  x,
  columns,
  value,
  na_pass = FALSE,
  preconditions = NULL,
  segments = 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 value used for this comparison. This can be a single value or a compatible column given in vars(). Any column values less than or equal to what is specified here will pass validation.
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	An optional expression for mutating the input table before proceeding with the validation. This can either be provided as a one-sided R formula using a leading ~ (e.g., ~ . %>% dplyr::mutate(col = col + 10) or as a function (e.g., function(x) dplyr::mutate(x,col = col + 10). See the *Preconditions* section for more information.
segments	An optional expression or set of expressions (held in a list) that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on. See the *Segments* section for more details on this.
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 validation 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 validation function will be operating directly on data (no *agent* involvement), then any step with `active = FALSE` will simply pass the data through with no validation whatsoever. Aside from a logical vector, a one-sided R formula using a leading `~` can be used with `.` (serving as the input data table) to evaluate to a single logical value. With this approach, the **pointblank** function `has_columns()` can be used to determine whether to make a validation step active on the basis of one or more columns existing in the table (e.g., `~ . %>% has_columns(vars(d,e))`). The default for `active` 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.

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.

Column Names

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()`.

Missing Values

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 `NA`s encountered will accumulate failing test units.

Preconditions

Providing expressions as `preconditions` means **pointblank** will preprocess the target table during interrogation as a preparatory step. It might happen that a particular validation requires a calculated column, some filtering of rows, or the addition of columns via a join, etc. Especially for an *agent*-based report this can be advantageous since we can develop a large validation plan with a single target table and make minor adjustments to it, as needed, along the way.

The table mutation is totally isolated in scope to the validation step(s) where `preconditions` is used. Using **dplyr** code is suggested here since the statements can be translated to SQL if necessary (i.e., if the target table resides in a database). 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_b = col_a + 10)`). Alternatively, a function could instead be supplied (e.g., `function(x) dplyr::mutate(x,col_b = col_a + 10)`).

Segments

By using the `segments` argument, it's possible to define a particular validation with segments (or row slices) of the target table. An optional expression or set of expressions that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on.

As an example of the first type of expression that can be used, `vars(a_column)` will segment the target table in however many unique values are present in the column called `a_column`. This is great if every unique value in a particular column (like different locations, or different dates) requires it's own repeating validation.

With a formula, we can be more selective with which column values should be used for segmentation. Using `a_column ~ c("group_1","group_2")` will attempt to obtain two segments where one is a slice of data where the value `"group_1"` exists in the column named `"a_column"`, and, the other is a slice where `"group_2"` exists in the same column. Each group of rows resolved from the formula will result in a separate validation step.

If there are multiple `columns` specified then the potential number of validation steps will be `m` columns multiplied by `n` segments resolved.

Segmentation will always occur after `preconditions` (i.e., statements that mutate the target table), if any, are applied. With this type of one-two combo, it's possible to generate labels for segmentation

using an expression for `preconditions` and refer to those labels in `segments` without having to generate a separate version of the target table.

Actions

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).

Briefs

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.

YAML

A **pointblank** agent can be written to YAML with `yaml_write()` and the resulting YAML can be used to regenerate an agent (with `yaml_read_agent()`) or interrogate the target table (via `yaml_agent_interrogate()`). When `col_vals_lte()` is represented in YAML (under the top-level `steps` key as a list member), the syntax closely follows the signature of the validation function. Here is an example of how a complex call of `col_vals_lte()` as a validation step is expressed in R code and in the corresponding YAML representation.

```
# R statement
agent %>%
  col_vals_lte(
    columns = vars(a),
    value = 1,
    na_pass = TRUE,
    preconditions = ~ . %>% dplyr::filter(a < 10),
    segments = b ~ c("group_1", "group_2"),
    actions = action_levels(warn_at = 0.1, stop_at = 0.2),
    label = "The `col_vals_lte()` step.",
    active = FALSE
  )

# YAML representation
steps:
- col_vals_lte:
    columns: vars(a)
    value: 1.0
```

```
  na_pass: true
  preconditions: ~. %>% dplyr::filter(a < 10)
  segments: b ~ c("group_1", "group_2")
  actions:
    warn_fraction: 0.1
    stop_fraction: 0.2
  label: The `col_vals_lte()` step.
  active: false
```

In practice, both of these will often be shorter as only the `columns` and `value` arguments require values. Arguments with default values won't be written to YAML when using `yaml_write()` (though it is acceptable to include them with their default when generating the YAML by other means). It is also possible to preview the transformation of an agent to YAML without any writing to disk by using the `yaml_agent_string()` function.

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_decreasing()`, `col_vals_equal()`, `col_vals_expr()`, `col_vals_gte()`, `col_vals_gt()`, `col_vals_in_set()`, `col_vals_increasing()`, `col_vals_lt()`, `col_vals_make_set()`, `col_vals_make_subset()`, `col_vals_not_between()`, `col_vals_not_equal()`, `col_vals_not_in_set()`, `col_vals_not_null()`, `col_vals_null()`, `col_vals_regex()`, `col_vals_within_spec()`, `conjointly()`, `row_count_match()`, `rows_complete()`, `rows_distinct()`, `serially()`, `specially()`, `tbl_match()`

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_make_set` *Is a set of values entirely accounted for in a column of values?*

Description

The col_vals_make_set() validation function, the expect_col_vals_make_set() expectation function, and the test_col_vals_make_set() test function all check whether set values are all seen at least once in a table column. A necessary criterion here is that no *additional* values (outside those definied in the set) should be seen (this requirement is relaxed in the col_vals_make_subset() validation function and in its expectation and test variants). 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 elements in the set plus a test unit reserved for detecting column values outside of the set (any outside value seen will make this additional test unit fail).

Usage

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

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

test_col_vals_make_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 elements that is expected to be equal to the set of unique values in the target column.

`preconditions` An optional expression for mutating the input table before proceeding with the validation. This can either be provided as a one-sided R formula using a leading `~` (e.g., `~ . %>% dplyr::mutate(col = col + 10)` or as a function (e.g., `function(x) dplyr::mutate(x,col = col + 10)`. See the *Preconditions* section for more information.

`segments` An optional expression or set of expressions (held in a list) that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on. See the *Segments* section for more details on this.

`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 validation 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 validation function will be operating directly on data (no *agent* involvement), then any step with `active = FALSE` will simply pass the data through with no validation whatsoever. Aside from a logical vector, a one-sided R formula using a leading `~` can be used with `.` (serving as the input data table) to evaluate to a single logical value. With this approach, the **pointblank** function `has_columns()` can be used to determine whether to make a validation step active on the basis of one or more columns existing in the table (e.g., `~ . %>% has_columns(vars(d,e))`). The default for `active` 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.

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.

Column Names

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()`.

Preconditions

Providing expressions as `preconditions` means **pointblank** will preprocess the target table during interrogation as a preparatory step. It might happen that a particular validation requires a calculated column, some filtering of rows, or the addition of columns via a join, etc. Especially for an *agent*-based report this can be advantageous since we can develop a large validation plan with a single target table and make minor adjustments to it, as needed, along the way.

The table mutation is totally isolated in scope to the validation step(s) where `preconditions` is used. Using **dplyr** code is suggested here since the statements can be translated to SQL if necessary (i.e., if the target table resides in a database). 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_b = col_a + 10)`). Alternatively, a function could instead be supplied (e.g., `function(x) dplyr::mutate(x,col_b = col_a + 10)`).

Segments

By using the `segments` argument, it's possible to define a particular validation with segments (or row slices) of the target table. An optional expression or set of expressions that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on.

As an example of the first type of expression that can be used, `vars(a_column)` will segment the target table in however many unique values are present in the column called `a_column`. This is great

if every unique value in a particular column (like different locations, or different dates) requires it's own repeating validation.

With a formula, we can be more selective with which column values should be used for segmentation. Using `a_column ~ c("group_1","group_2")` will attempt to obtain two segments where one is a slice of data where the value `"group_1"` exists in the column named `"a_column"`, and, the other is a slice where `"group_2"` exists in the same column. Each group of rows resolved from the formula will result in a separate validation step.

If there are multiple `columns` specified then the potential number of validation steps will be `m` columns multiplied by `n` segments resolved.

Segmentation will always occur after `preconditions` (i.e., statements that mutate the target table), if any, are applied. With this type of one-two combo, it's possible to generate labels for segmentation using an expression for `preconditions` and refer to those labels in `segments` without having to generate a separate version of the target table.

Actions

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).

Briefs

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.

YAML

A **pointblank** agent can be written to YAML with `yaml_write()` and the resulting YAML can be used to regenerate an agent (with `yaml_read_agent()`) or interrogate the target table (via `yaml_agent_interrogate()`). When `col_vals_make_set()` is represented in YAML (under the top-level `steps` key as a list member), the syntax closely follows the signature of the validation function. Here is an example of how a complex call of `col_vals_make_set()` as a validation step is expressed in R code and in the corresponding YAML representation.

```
# R statement
agent %>%
  col_vals_make_set(
    columns = vars(a),
    set = c(1, 2, 3, 4),
    preconditions = ~ . %>% dplyr::filter(a < 10),
```

```
    segments = b ~ c("group_1", "group_2"),
    actions = action_levels(warn_at = 0.1, stop_at = 0.2),
    label = "The `col_vals_make_set()` step.",
    active = FALSE
  )

# YAML representation
steps:
- col_vals_make_set:
    columns: vars(a)
   set:
   - 1.0
   - 2.0
   - 3.0
   - 4.0
   preconditions: ~. %>% dplyr::filter(a < 10)
   segments: b ~ c("group_1", "group_2")
   actions:
     warn_fraction: 0.1
     stop_fraction: 0.2
   label: The `col_vals_make_set()` step.
   active: false
```

In practice, both of these will often be shorter as only the `columns` and `set` arguments require values. Arguments with default values won't be written to YAML when using `yaml_write()` (though it is acceptable to include them with their default when generating the YAML by other means). It is also possible to preview the transformation of an agent to YAML without any writing to disk by using the `yaml_agent_string()` function.

Function ID

2-11

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_decreasing()`, `col_vals_equal()`, `col_vals_expr()`, `col_vals_gte()`, `col_vals_gt()`, `col_vals_in_set()`, `col_vals_increasing()`, `col_vals_lte()`, `col_vals_lt()`, `col_vals_make_subset()`, `col_vals_not_between()`, `col_vals_not_equal()`, `col_vals_not_in_set()`, `col_vals_not_null()`, `col_vals_null()`, `col_vals_regex()`, `col_vals_within_spec()`, `conjointly()`, `row_count_match()`, `rows_complete()`, `rows_distinct()`, `serially()`, `specially()`, `tbl_match()`

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`
# comprise the values of `low`, `mid`,
# and `high`, and, no other values
agent <-
  create_agent(small_table) %>%
  col_vals_make_set(
    vars(f), c("low", "mid", "high")
  ) %>%
  interrogate()

# Determine if this validation
# had no failing test units (there
# are 4 test units)
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_make_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_make_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_make_set(
    vars(f), c("low", "mid", "high")
  )
```

`col_vals_make_subset` *Is a set of values a subset of a column of values?*

Description

The `col_vals_make_subset()` validation function, the `expect_col_vals_make_subset()` expectation function, and the `test_col_vals_make_subset()` test function all check whether all `set` values are seen at least once in a table column. 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 elements in the `set`.

Usage

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

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

test_col_vals_make_subset(
  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 elements that is expected to be a subset of the unique values in the target column.
preconditions	An optional expression for mutating the input table before proceeding with the validation. This can either be provided as a one-sided R formula using a leading ~ (e.g., ~ . %>% dplyr::mutate(col = col + 10) or as a function (e.g., function(x) dplyr::mutate(x,col = col + 10). See the *Preconditions* section for more information.
segments	An optional expression or set of expressions (held in a list) that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on. See the *Segments* section for more details on this.
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 validation 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 validation function will be operating directly on data (no *agent* involvement), then any step with `active = FALSE` will simply pass the data through with no validation whatsoever. Aside from a logical vector, a one-sided R formula using a leading `~` can be used with `.` (serving as the input data table) to evaluate to a single logical value. With this approach, the **pointblank** function `has_columns()` can be used to determine whether to make a validation step active on the basis of one or more columns existing in the table (e.g., `~ . %>% has_columns(vars(d,e))`). The default for `active` 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.

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.

Column Names

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()`.

Preconditions

Providing expressions as `preconditions` means **pointblank** will preprocess the target table during interrogation as a preparatory step. It might happen that a particular validation requires a calculated column, some filtering of rows, or the addition of columns via a join, etc. Especially for an *agent*-based report this can be advantageous since we can develop a large validation plan with a single target table and make minor adjustments to it, as needed, along the way.

The table mutation is totally isolated in scope to the validation step(s) where `preconditions` is used. Using **dplyr** code is suggested here since the statements can be translated to SQL if necessary (i.e., if the target table resides in a database). 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_b = col_a + 10)`). Alternatively, a function could instead be supplied (e.g., `function(x) dplyr::mutate(x,col_b = col_a + 10)`).

Segments

By using the `segments` argument, it's possible to define a particular validation with segments (or row slices) of the target table. An optional expression or set of expressions that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on.

As an example of the first type of expression that can be used, `vars(a_column)` will segment the target table in however many unique values are present in the column called `a_column`. This is great if every unique value in a particular column (like different locations, or different dates) requires it's own repeating validation.

With a formula, we can be more selective with which column values should be used for segmentation. Using `a_column ~ c("group_1","group_2")` will attempt to obtain two segments where one is a slice of data where the value `"group_1"` exists in the column named `"a_column"`, and, the other is a slice where `"group_2"` exists in the same column. Each group of rows resolved from the formula will result in a separate validation step.

If there are multiple `columns` specified then the potential number of validation steps will be `m` columns multiplied by `n` segments resolved.

Segmentation will always occur after `preconditions` (i.e., statements that mutate the target table), if any, are applied. With this type of one-two combo, it's possible to generate labels for segmentation using an expression for `preconditions` and refer to those labels in `segments` without having to generate a separate version of the target table.

Actions

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).

Briefs

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.

YAML

A **pointblank** agent can be written to YAML with `yaml_write()` and the resulting YAML can be used to regenerate an agent (with `yaml_read_agent()`) or interrogate the target table (via `yaml_agent_interrogate()`). When `col_vals_make_subset()` is represented in YAML (under the top-level `steps` key as a list member), the syntax closely follows the signature of the validation

function. Here is an example of how a complex call of `col_vals_make_subset()` as a validation step is expressed in R code and in the corresponding YAML representation.

```
# R statement
agent %>%
  col_vals_make_subset(
    columns = vars(a),
    set = c(1, 2, 3, 4),
    preconditions = ~ . %>% dplyr::filter(a < 10),
    segments = b ~ c("group_1", "group_2"),
    actions = action_levels(warn_at = 0.1, stop_at = 0.2),
    label = "The `col_vals_make_subset()` step.",
    active = FALSE
  )

# YAML representation
steps:
- col_vals_make_subset:
    columns: vars(a)
   set:
   - 1.0
   - 2.0
   - 3.0
   - 4.0
   preconditions: ~. %>% dplyr::filter(a < 10)
   segments: b ~ c("group_1", "group_2")
   actions:
     warn_fraction: 0.1
     stop_fraction: 0.2
   label: The `col_vals_make_subset()` step.
   active: false
```

In practice, both of these will often be shorter as only the `columns` and `set` arguments require values. Arguments with default values won't be written to YAML when using `yaml_write()` (though it is acceptable to include them with their default when generating the YAML by other means). It is also possible to preview the transformation of an agent to YAML without any writing to disk by using the `yaml_agent_string()` function.

Function ID

2-12

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_decreasing()`, `col_vals_equal()`, `col_vals_expr()`, `col_vals_gte()`, `col_vals_gt()`, `col_vals_in_set()`, `col_vals_increasing()`, `col_vals_lte()`, `col_vals_lt()`, `col_vals_make_set()`, `col_vals_not_between()`, `col_vals_not_equal()`, `col_vals_not_in_set()`,

col_vals_not_null(), col_vals_null(), col_vals_regex(), col_vals_within_spec(), conjointly(), row_count_match(), rows_complete(), rows_distinct(), serially(), specially(), tbl_match()

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 the distinct set of values
# in column `f` contains at least the
# subset defined as `low` and `high` (the
# column actually has both of those and
# some `mid` values)
agent <-
  create_agent(small_table) %>%
  col_vals_make_subset(
    vars(f), c("low", "high")
  ) %>%
  interrogate()

# Determine if this validation
# had no failing test units (there
# are 2 test units, one per element
# in the `set`)
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_make_subset(
    vars(f), c("low", "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_make_subset(
  small_table,
  vars(f), c("low", "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_make_subset(
    vars(f), c("low", "high")
  )
```

`col_vals_not_between` *Do column data lie outside of two specified values or data in other columns?*

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 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_between(
  x,
  columns,
  left,
  right,
  inclusive = c(TRUE, TRUE),
```

```
  na_pass = FALSE,
  preconditions = NULL,
  segments = 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 (or left) and upper (or right) boundary values for the range. These can be expressed as single values, compatible columns given in vars(), or a combination of both. By default, any column values greater than or equal to left *and* less than or equal to right will fail validation. The inclusivity of the bounds can be modified by the inclusive option.
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` An optional expression for mutating the input table before proceeding with the validation. This can either be provided as a one-sided R formula using a leading `~` (e.g., `~ . %>% dplyr::mutate(col = col + 10)` or as a function (e.g., `function(x) dplyr::mutate(x,col = col + 10)`. See the *Preconditions* section for more information.

`segments` An optional expression or set of expressions (held in a list) that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on. See the *Segments* section for more details on this.

`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 validation 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 validation function will be operating directly on data (no *agent* involvement), then any step with `active = FALSE` will simply pass the data through with no validation whatsoever. Aside from a logical vector, a one-sided R formula using a leading `~` can be used with `.` (serving as the input data table) to evaluate to a single logical value. With this approach, the **pointblank** function `has_columns()` can be used to determine whether to make a validation step active on the basis of one or more columns existing in the table (e.g., `~ . %>% has_columns(vars(d,e))`). The default for `active` 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.

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.

Column Names

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()`.

Missing Values

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 `NA`s encountered will accumulate failing test units.

Preconditions

Providing expressions as `preconditions` means **pointblank** will preprocess the target table during interrogation as a preparatory step. It might happen that a particular validation requires a calculated column, some filtering of rows, or the addition of columns via a join, etc. Especially for an *agent*-based report this can be advantageous since we can develop a large validation plan with a single target table and make minor adjustments to it, as needed, along the way.

The table mutation is totally isolated in scope to the validation step(s) where `preconditions` is used. Using **dplyr** code is suggested here since the statements can be translated to SQL if necessary (i.e., if the target table resides in a database). 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_b = col_a + 10)`). Alternatively, a function could instead be supplied (e.g., `function(x) dplyr::mutate(x,col_b = col_a + 10)`).

Segments

By using the `segments` argument, it's possible to define a particular validation with segments (or row slices) of the target table. An optional expression or set of expressions that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on.

As an example of the first type of expression that can be used, `vars(a_column)` will segment the target table in however many unique values are present in the column called `a_column`. This is great if every unique value in a particular column (like different locations, or different dates) requires it's own repeating validation.

With a formula, we can be more selective with which column values should be used for segmentation. Using `a_column ~ c("group_1","group_2")` will attempt to obtain two segments where one is a slice of data where the value `"group_1"` exists in the column named `"a_column"`, and, the other is a slice where `"group_2"` exists in the same column. Each group of rows resolved from the formula will result in a separate validation step.

If there are multiple `columns` specified then the potential number of validation steps will be `m` columns multiplied by `n` segments resolved.

Segmentation will always occur after `preconditions` (i.e., statements that mutate the target table), if any, are applied. With this type of one-two combo, it's possible to generate labels for segmentation using an expression for `preconditions` and refer to those labels in `segments` without having to generate a separate version of the target table.

Actions

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).

Briefs

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.

YAML

A **pointblank** agent can be written to YAML with `yaml_write()` and the resulting YAML can be used to regenerate an agent (with `yaml_read_agent()`) or interrogate the target table (via `yaml_agent_interrogate()`). When `col_vals_not_between()` is represented in YAML (under the top-level `steps` key as a list member), the syntax closely follows the signature of the validation function. Here is an example of how a complex call of `col_vals_not_between()` as a validation step is expressed in R code and in the corresponding YAML representation.

```
# R statement
agent %>%
  col_vals_not_between(
    columns = vars(a),
```

```
    left = 1,
    right = 2,
    inclusive = c(TRUE, FALSE),
    na_pass = TRUE,
    preconditions = ~ . %>% dplyr::filter(a < 10),
    segments = b ~ c("group_1", "group_2"),
    actions = action_levels(warn_at = 0.1, stop_at = 0.2),
    label = "The `col_vals_not_between()` step.",
    active = FALSE
  )

# YAML representation
steps:
- col_vals_not_between:
    columns: vars(a)
    left: 1.0
    right: 2.0
    inclusive:
    - true
    - false
    na_pass: true
    preconditions: ~. %>% dplyr::filter(a < 10)
    segments: b ~ c("group_1", "group_2")
    actions:
      warn_fraction: 0.1
      stop_fraction: 0.2
    label: The `col_vals_not_between()` step.
    active: false
```

In practice, both of these will often be shorter as only the `columns`, `left`, and `right` arguments require values. Arguments with default values won't be written to YAML when using `yaml_write()` (though it is acceptable to include them with their default when generating the YAML by other means). It is also possible to preview the transformation of an agent to YAML without any writing to disk by using the `yaml_agent_string()` function.

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_decreasing()`, `col_vals_equal()`, `col_vals_expr()`, `col_vals_gte()`, `col_vals_gt()`, `col_vals_in_set()`, `col_vals_increasing()`, `col_vals_lte()`, `col_vals_lt()`, `col_vals_make_set()`, `col_vals_make_subset()`, `col_vals_not_equal()`, `col_vals_not_in_set()`, `col_vals_not_null()`, `col_vals_null()`, `col_vals_regex()`, `col_vals_within_spec()`, `conjointly()`, `row_count_match()`, `rows_complete()`, `rows_distinct()`, `serially()`, `specially()`, `tbl_match()`

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 fixed value or data in another column?*

Description

The `col_vals_not_equal()` validation function, the `expect_col_vals_not_equal()` expectation 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 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,
  segments = 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 value used for this test of inequality. This can be a single value or a compatible column given in `vars()`. Any column values not equal to what is specified here will pass validation.

`na_pass` Should any encountered `NA` values be considered as passing test units? This is by default `FALSE`. Set to `TRUE` to give `NA`s a pass.

`preconditions` An optional expression for mutating the input table before proceeding with the validation. This can either be provided as a one-sided R formula using a leading `~` (e.g., `~ . %>% dplyr::mutate(col = col + 10)` or as a function (e.g., `function(x) dplyr::mutate(x,col = col + 10)`. See the *Preconditions* section for more information.

`segments` An optional expression or set of expressions (held in a list) that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on. See the *Segments* section for more details on this.

`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 validation 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 validation function will be operating directly on data (no *agent* involvement), then any step with `active = FALSE` will simply pass the data through with no validation whatsoever. Aside from a logical vector, a one-sided R formula using a leading `~` can be used with `.` (serving as the input data table) to evaluate to a single logical value. With this approach, the **pointblank** function `has_columns()` can be used to determine whether to make a validation step active on the basis of one or more columns existing in the table (e.g., `~ . %>% has_columns(vars(d,e))`). The default for `active` 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.

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.

Column Names

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().

Missing Values

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.

Preconditions

Providing expressions as preconditions means **pointblank** will preprocess the target table during interrogation as a preparatory step. It might happen that a particular validation requires a calculated column, some filtering of rows, or the addition of columns via a join, etc. Especially for an *agent*-based report this can be advantageous since we can develop a large validation plan with a single target table and make minor adjustments to it, as needed, along the way.

The table mutation is totally isolated in scope to the validation step(s) where preconditions is used. Using **dplyr** code is suggested here since the statements can be translated to SQL if necessary (i.e., if the target table resides in a database). 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_b = col_a + 10)). Alternatively, a function could instead be supplied (e.g., function(x) dplyr::mutate(x,col_b = col_a + 10)).

Segments

By using the segments argument, it's possible to define a particular validation with segments (or row slices) of the target table. An optional expression or set of expressions that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column

names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on.

As an example of the first type of expression that can be used, `vars(a_column)` will segment the target table in however many unique values are present in the column called `a_column`. This is great if every unique value in a particular column (like different locations, or different dates) requires it's own repeating validation.

With a formula, we can be more selective with which column values should be used for segmentation. Using `a_column ~ c("group_1","group_2")` will attempt to obtain two segments where one is a slice of data where the value `"group_1"` exists in the column named `"a_column"`, and, the other is a slice where `"group_2"` exists in the same column. Each group of rows resolved from the formula will result in a separate validation step.

If there are multiple `columns` specified then the potential number of validation steps will be `m` columns multiplied by `n` segments resolved.

Segmentation will always occur after `preconditions` (i.e., statements that mutate the target table), if any, are applied. With this type of one-two combo, it's possible to generate labels for segmentation using an expression for `preconditions` and refer to those labels in `segments` without having to generate a separate version of the target table.

Actions

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).

Briefs

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.

YAML

A **pointblank** agent can be written to YAML with `yaml_write()` and the resulting YAML can be used to regenerate an agent (with `yaml_read_agent()`) or interrogate the target table (via `yaml_agent_interrogate()`). When `col_vals_not_equal()` is represented in YAML (under the top-level `steps` key as a list member), the syntax closely follows the signature of the validation function. Here is an example of how a complex call of `col_vals_not_equal()` as a validation step is expressed in R code and in the corresponding YAML representation.

```
# R statement
agent %>%
```

```
  col_vals_not_equal(
    columns = vars(a),
    value = 1,
    na_pass = TRUE,
    preconditions = ~ . %>% dplyr::filter(a < 10),
    segments = b ~ c("group_1", "group_2"),
    actions = action_levels(warn_at = 0.1, stop_at = 0.2),
    label = "The `col_vals_not_equal()` step.",
    active = FALSE
  )

# YAML representation
steps:
- col_vals_not_equal:
    columns: vars(a)
    value: 1.0
    na_pass: true
    preconditions: ~. %>% dplyr::filter(a < 10)
    segments: b ~ c("group_1", "group_2")
    actions:
      warn_fraction: 0.1
      stop_fraction: 0.2
    label: The `col_vals_not_equal()` step.
    active: false
```

In practice, both of these will often be shorter as only the `columns` and `value` arguments require values. Arguments with default values won't be written to YAML when using `yaml_write()` (though it is acceptable to include them with their default when generating the YAML by other means). It is also possible to preview the transformation of an agent to YAML without any writing to disk by using the `yaml_agent_string()` function.

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_decreasing()`, `col_vals_equal()`, `col_vals_expr()`, `col_vals_gte()`, `col_vals_gt()`, `col_vals_in_set()`, `col_vals_increasing()`, `col_vals_lte()`, `col_vals_lt()`, `col_vals_make_set()`, `col_vals_make_subset()`, `col_vals_not_between()`, `col_vals_not_in_set()`, `col_vals_not_null()`, `col_vals_null()`, `col_vals_regex()`, `col_vals_within_spec()`, `conjointly()`, `row_count_match()`, `rows_complete()`, `rows_distinct()`, `serially()`, `specially()`, `tbl_match()`

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,
  segments = 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	An optional expression for mutating the input table before proceeding with the validation. This can either be provided as a one-sided R formula using a leading ~ (e.g., ~ . %>% dplyr::mutate(col = col + 10) or as a function (e.g., function(x) dplyr::mutate(x,col = col + 10). See the *Preconditions* section for more information.
segments	An optional expression or set of expressions (held in a list) that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on. See the *Segments* section for more details on this.
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 validation 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 validation function will be operating directly on data (no *agent* involvement), then any step with `active = FALSE` will simply pass the data through with no validation whatsoever. Aside from a logical vector, a one-sided R formula using a leading `~` can be used with `.` (serving as the input data table) to evaluate to a single logical value. With this approach, the **pointblank** function `has_columns()` can be used to determine whether to make a validation step active on the basis of one or more columns existing in the table (e.g., `~ . %>% has_columns(vars(d,e))`). The default for `active` 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.

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.

Column Names

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()`.

Preconditions

Providing expressions as `preconditions` means **pointblank** will preprocess the target table during interrogation as a preparatory step. It might happen that a particular validation requires a calculated column, some filtering of rows, or the addition of columns via a join, etc. Especially for an *agent*-based report this can be advantageous since we can develop a large validation plan with a single target table and make minor adjustments to it, as needed, along the way.

The table mutation is totally isolated in scope to the validation step(s) where `preconditions` is used. Using **dplyr** code is suggested here since the statements can be translated to SQL if necessary (i.e., if the target table resides in a database). 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_b = col_a + 10)). Alternatively, a function could instead be supplied (e.g., function(x) dplyr::mutate(x,col_b = col_a + 10)).

Segments

By using the segments argument, it's possible to define a particular validation with segments (or row slices) of the target table. An optional expression or set of expressions that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on.

As an example of the first type of expression that can be used, vars(a_column) will segment the target table in however many unique values are present in the column called a_column. This is great if every unique value in a particular column (like different locations, or different dates) requires it's own repeating validation.

With a formula, we can be more selective with which column values should be used for segmentation. Using a_column ~ c("group_1","group_2") will attempt to obtain two segments where one is a slice of data where the value "group_1" exists in the column named "a_column", and, the other is a slice where "group_2" exists in the same column. Each group of rows resolved from the formula will result in a separate validation step.

If there are multiple columns specified then the potential number of validation steps will be m columns multiplied by n segments resolved.

Segmentation will always occur after preconditions (i.e., statements that mutate the target table), if any, are applied. With this type of one-two combo, it's possible to generate labels for segmentation using an expression for preconditions and refer to those labels in segments without having to generate a separate version of the target table.

Actions

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).

Briefs

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.

YAML

A **pointblank** agent can be written to YAML with `yaml_write()` and the resulting YAML can be used to regenerate an agent (with `yaml_read_agent()`) or interrogate the target table (via `yaml_agent_interrogate()`). When `col_vals_not_in_set()` is represented in YAML (under the top-level `steps` key as a list member), the syntax closely follows the signature of the validation function. Here is an example of how a complex call of `col_vals_not_in_set()` as a validation step is expressed in R code and in the corresponding YAML representation.

```
# R statement
agent %>%
  col_vals_not_in_set(
    columns = vars(a),
    set = c(1, 2, 3, 4),
    preconditions = ~ . %>% dplyr::filter(a < 10),
    segments = b ~ c("group_1", "group_2"),
    actions = action_levels(warn_at = 0.1, stop_at = 0.2),
    label = "The `col_vals_not_in_set()` step.",
    active = FALSE
  )

# YAML representation
steps:
- col_vals_not_in_set:
    columns: vars(a)
   set:
   - 1.0
   - 2.0
   - 3.0
   - 4.0
   preconditions: ~. %>% dplyr::filter(a < 10)
   segments: b ~ c("group_1", "group_2")
   actions:
     warn_fraction: 0.1
     stop_fraction: 0.2
   label: The `col_vals_not_in_set()` step.
   active: false
```

In practice, both of these will often be shorter as only the `columns` and `set` arguments require values. Arguments with default values won't be written to YAML when using `yaml_write()` (though it is acceptable to include them with their default when generating the YAML by other means). It is also possible to preview the transformation of an agent to YAML without any writing to disk by using the `yaml_agent_string()` function.

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_decreasing(), col_vals_equal(), col_vals_expr(), col_vals_gte(), col_vals_gt(), col_vals_in_set(), col_vals_increasing(), col_vals_lte(), col_vals_lt(), col_vals_make_set(), col_vals_make_subset(), col_vals_not_between(), col_vals_not_equal(), col_vals_not_null(), col_vals_null(), col_vals_regex(), col_vals_within_spec(), conjointly(), row_count_match(), rows_complete(), rows_distinct(), serially(), specially(), tbl_match()

Examples

```
# The `small_table` dataset in the
# package will be used to validate that
# column values are not 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,
  segments = 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()`.
columns	The column (or a set of columns, provided as a character vector) to which this validation should be applied.
preconditions	An optional expression for mutating the input table before proceeding with the validation. This can either be provided as a one-sided R formula using a leading ~ (e.g., `~ . %>% dplyr::mutate(col = col + 10)` or as a function (e.g., `function(x) dplyr::mutate(x,col = col + 10)`. See the *Preconditions* section for more information.
segments	An optional expression or set of expressions (held in a list) that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on. See the *Segments* section for more details on this.
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 validation 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 validation function will be operating directly on data (no *agent* involvement), then any step with `active = FALSE` will simply pass the data through with no validation whatsoever. Aside from a logical vector, a one-sided R formula using a leading `~` can be used with `.` (serving as the input data table) to evaluate to a single logical value. With this approach, the **pointblank** function `has_columns()` can be used to determine whether to make a validation step active on the basis of one or more columns existing in the table (e.g., `~ . %>% has_columns(vars(d,e))`). The default for `active` 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.

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.

Column Names

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()`.

Preconditions

Providing expressions as `preconditions` means **pointblank** will preprocess the target table during interrogation as a preparatory step. It might happen that a particular validation requires a calculated column, some filtering of rows, or the addition of columns via a join, etc. Especially for an *agent*-based report this can be advantageous since we can develop a large validation plan with a single target table and make minor adjustments to it, as needed, along the way.

The table mutation is totally isolated in scope to the validation step(s) where `preconditions` is used. Using **dplyr** code is suggested here since the statements can be translated to SQL if necessary (i.e., if the target table resides in a database). 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_b = col_a + 10)`). Alternatively, a function could instead be supplied (e.g., `function(x) dplyr::mutate(x,col_b = col_a + 10)`).

Segments

By using the `segments` argument, it's possible to define a particular validation with segments (or row slices) of the target table. An optional expression or set of expressions that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on.

As an example of the first type of expression that can be used, `vars(a_column)` will segment the target table in however many unique values are present in the column called `a_column`. This is great if every unique value in a particular column (like different locations, or different dates) requires it's own repeating validation.

With a formula, we can be more selective with which column values should be used for segmentation. Using `a_column ~ c("group_1","group_2")` will attempt to obtain two segments where one is a slice of data where the value `"group_1"` exists in the column named `"a_column"`, and, the other is a slice where `"group_2"` exists in the same column. Each group of rows resolved from the formula will result in a separate validation step.

If there are multiple `columns` specified then the potential number of validation steps will be `m` columns multiplied by `n` segments resolved.

Segmentation will always occur after `preconditions` (i.e., statements that mutate the target table), if any, are applied. With this type of one-two combo, it's possible to generate labels for segmentation using an expression for `preconditions` and refer to those labels in `segments` without having to generate a separate version of the target table.

Actions

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).

Briefs

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.

YAML

A **pointblank** agent can be written to YAML with `yaml_write()` and the resulting YAML can be used to regenerate an agent (with `yaml_read_agent()`) or interrogate the target table (via `yaml_agent_interrogate()`). When `col_vals_not_null()` is represented in YAML (under the top-level `steps` key as a list member), the syntax closely follows the signature of the validation

function. Here is an example of how a complex call of `col_vals_not_null()` as a validation step is expressed in R code and in the corresponding YAML representation.

```
# R statement
agent %>%
  col_vals_not_null(
    vars(a),
    preconditions = ~ . %>% dplyr::filter(a < 10),
    segments = b ~ c("group_1", "group_2"),
    actions = action_levels(warn_at = 0.1, stop_at = 0.2),
    label = "The `col_vals_not_null()` step.",
    active = FALSE
  )

# YAML representation
steps:
- col_vals_not_null:
    columns: vars(a)
    preconditions: ~. %>% dplyr::filter(a < 10)
    segments: b ~ c("group_1", "group_2")
    actions:
      warn_fraction: 0.1
      stop_fraction: 0.2
    label: The `col_vals_not_null()` step.
    active: false
```

In practice, both of these will often be shorter as only the `columns` argument requires a value. Arguments with default values won't be written to YAML when using `yaml_write()` (though it is acceptable to include them with their default when generating the YAML by other means). It is also possible to preview the transformation of an agent to YAML without any writing to disk by using the `yaml_agent_string()` function.

Function ID

2-16

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_decreasing()`, `col_vals_equal()`, `col_vals_expr()`, `col_vals_gte()`, `col_vals_gt()`, `col_vals_in_set()`, `col_vals_increasing()`, `col_vals_lte()`, `col_vals_lt()`, `col_vals_make_set()`, `col_vals_make_subset()`, `col_vals_not_between()`, `col_vals_not_equal()`, `col_vals_not_in_set()`, `col_vals_null()`, `col_vals_regex()`, `col_vals_within_spec()`, `conjointly()`, `row_count_match()`, `rows_complete()`, `rows_distinct()`, `serially()`, `specially()`, `tbl_match()`

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,
  segments = 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` An optional expression for mutating the input table before proceeding with the validation. This can either be provided as a one-sided R formula using a leading `~` (e.g., `~ . %>% dplyr::mutate(col = col + 10)` or as a function (e.g., `function(x) dplyr::mutate(x,col = col + 10)`. See the *Preconditions* section for more information.

`segments` An optional expression or set of expressions (held in a list) that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on. See the *Segments* section for more details on this.

`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 validation 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 validation function will be operating directly on data (no *agent* involvement), then any step with `active = FALSE` will simply pass the data through with no validation whatsoever. Aside from a logical vector, a one-sided R formula using a leading `~` can be used with `.` (serving as the input data table) to evaluate to a single logical value. With this approach, the **pointblank** function `has_columns()` can be used to determine whether to make a validation step active on the basis of one or more columns existing in the table (e.g., `~ . %>% has_columns(vars(d,e))`). The default for `active` 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.

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.

Column Names

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().

Preconditions

Providing expressions as preconditions means **pointblank** will preprocess the target table during interrogation as a preparatory step. It might happen that a particular validation requires a calculated column, some filtering of rows, or the addition of columns via a join, etc. Especially for an *agent*-based report this can be advantageous since we can develop a large validation plan with a single target table and make minor adjustments to it, as needed, along the way.

The table mutation is totally isolated in scope to the validation step(s) where preconditions is used. Using **dplyr** code is suggested here since the statements can be translated to SQL if necessary (i.e., if the target table resides in a database). 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_b = col_a + 10)). Alternatively, a function could instead be supplied (e.g., function(x) dplyr::mutate(x,col_b = col_a + 10)).

Segments

By using the segments argument, it's possible to define a particular validation with segments (or row slices) of the target table. An optional expression or set of expressions that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on.

As an example of the first type of expression that can be used, vars(a_column) will segment the target table in however many unique values are present in the column called a_column. This is great if every unique value in a particular column (like different locations, or different dates) requires it's own repeating validation.

With a formula, we can be more selective with which column values should be used for segmentation. Using `a_column ~ c("group_1","group_2")` will attempt to obtain two segments where one is a slice of data where the value `"group_1"` exists in the column named `"a_column"`, and, the other is a slice where `"group_2"` exists in the same column. Each group of rows resolved from the formula will result in a separate validation step.

If there are multiple `columns` specified then the potential number of validation steps will be `m` columns multiplied by `n` segments resolved.

Segmentation will always occur after `preconditions` (i.e., statements that mutate the target table), if any, are applied. With this type of one-two combo, it's possible to generate labels for segmentation using an expression for `preconditions` and refer to those labels in `segments` without having to generate a separate version of the target table.

Actions

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).

Briefs

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.

YAML

A **pointblank** agent can be written to YAML with `yaml_write()` and the resulting YAML can be used to regenerate an agent (with `yaml_read_agent()`) or interrogate the target table (via `yaml_agent_interrogate()`). When `col_vals_null()` is represented in YAML (under the top-level `steps` key as a list member), the syntax closely follows the signature of the validation function. Here is an example of how a complex call of `col_vals_null()` as a validation step is expressed in R code and in the corresponding YAML representation.

```
# R statement
agent %>%
  col_vals_null(
    vars(a),
    preconditions = ~ . %>% dplyr::filter(a < 10),
    segments = b ~ c("group_1", "group_2"),
    actions = action_levels(warn_at = 0.1, stop_at = 0.2),
    label = "The `col_vals_null()` step.",
    active = FALSE
```

```
  )

# YAML representation
steps:
- col_vals_null:
    columns: vars(a)
    preconditions: ~. %>% dplyr::filter(a < 10)
    segments: b ~ c("group_1", "group_2")
    actions:
      warn_fraction: 0.1
      stop_fraction: 0.2
    label: The `col_vals_null()` step.
    active: false
```

In practice, both of these will often be shorter as only the `columns` argument requires a value. Arguments with default values won't be written to YAML when using `yaml_write()` (though it is acceptable to include them with their default when generating the YAML by other means). It is also possible to preview the transformation of an agent to YAML without any writing to disk by using the `yaml_agent_string()` function.

Function ID

2-15

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_decreasing()`, `col_vals_equal()`, `col_vals_expr()`, `col_vals_gte()`, `col_vals_gt()`, `col_vals_in_set()`, `col_vals_increasing()`, `col_vals_lte()`, `col_vals_lt()`, `col_vals_make_set()`, `col_vals_make_subset()`, `col_vals_not_between()`, `col_vals_not_equal()`, `col_vals_not_in_set()`, `col_vals_not_null()`, `col_vals_regex()`, `col_vals_within_spec()`, `conjointly()`, `row_count_match()`, `rows_complete()`, `rows_distinct()`, `serially()`, `specially()`, `tbl_match()`

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 correspond to a `regex` matching 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_regex(
  x,
  columns,
  regex,
  na_pass = FALSE,
  preconditions = NULL,
  segments = 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 regular expression pattern to test for a match to the target column. Any regex matches to values in the target columns will pass validation.
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	An optional expression for mutating the input table before proceeding with the validation. This can either be provided as a one-sided R formula using a leading ~ (e.g., ~ . %>% dplyr::mutate(col = col + 10) or as a function (e.g., function(x) dplyr::mutate(x,col = col + 10). See the *Preconditions* section for more information.
segments	An optional expression or set of expressions (held in a list) that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on. See the *Segments* section for more details on this.
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 validation 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 validation function will be operating directly on data (no *agent* involvement), then any step with active = FALSE will simply pass the data

through with no validation whatsoever. Aside from a logical vector, a one-sided R formula using a leading ~ can be used with . (serving as the input data table) to evaluate to a single logical value. With this approach, the **pointblank** function has_columns() can be used to determine whether to make a validation step active on the basis of one or more columns existing in the table (e.g., ~ . %>% has_columns(vars(d,e))). The default for active 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.

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.

Column Names

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().

Missing Values

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.

Preconditions

Providing expressions as preconditions means **pointblank** will preprocess the target table during interrogation as a preparatory step. It might happen that a particular validation requires a calculated column, some filtering of rows, or the addition of columns via a join, etc. Especially for an *agent*-based report this can be advantageous since we can develop a large validation plan with a single target table and make minor adjustments to it, as needed, along the way.

The table mutation is totally isolated in scope to the validation step(s) where preconditions is used. Using **dplyr** code is suggested here since the statements can be translated to SQL if necessary (i.e., if the target table resides in a database). 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_b = col_a + 10)`). Alternatively, a function could instead be supplied (e.g., `function(x) dplyr::mutate(x,col_b = col_a + 10)`).

Segments

By using the `segments` argument, it's possible to define a particular validation with segments (or row slices) of the target table. An optional expression or set of expressions that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on.

As an example of the first type of expression that can be used, `vars(a_column)` will segment the target table in however many unique values are present in the column called `a_column`. This is great if every unique value in a particular column (like different locations, or different dates) requires it's own repeating validation.

With a formula, we can be more selective with which column values should be used for segmentation. Using `a_column ~ c("group_1","group_2")` will attempt to obtain two segments where one is a slice of data where the value `"group_1"` exists in the column named `"a_column"`, and, the other is a slice where `"group_2"` exists in the same column. Each group of rows resolved from the formula will result in a separate validation step.

If there are multiple `columns` specified then the potential number of validation steps will be `m` columns multiplied by `n` segments resolved.

Segmentation will always occur after `preconditions` (i.e., statements that mutate the target table), if any, are applied. With this type of one-two combo, it's possible to generate labels for segmentation using an expression for `preconditions` and refer to those labels in `segments` without having to generate a separate version of the target table.

Actions

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).

Briefs

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.

YAML

A **pointblank** agent can be written to YAML with `yaml_write()` and the resulting YAML can be used to regenerate an agent (with `yaml_read_agent()`) or interrogate the target table (via

yaml_agent_interrogate()). When col_vals_regex() is represented in YAML (under the top-level steps key as a list member), the syntax closely follows the signature of the validation function. Here is an example of how a complex call of col_vals_regex() as a validation step is expressed in R code and in the corresponding YAML representation.

```
# R statement
agent %>%
  col_vals_regex(
    columns = vars(a),
    regex = "[0-9]-[a-z]{3}-[0-9]{3}",
    na_pass = TRUE,
    preconditions = ~ . %>% dplyr::filter(a < 10),
    segments = b ~ c("group_1", "group_2"),
    actions = action_levels(warn_at = 0.1, stop_at = 0.2),
    label = "The `col_vals_regex()` step.",
    active = FALSE
  )

# YAML representation
steps:
- col_vals_regex:
    columns: vars(a)
    regex: '[0-9]-[a-z]{3}-[0-9]{3}'
    na_pass: true
    preconditions: ~. %>% dplyr::filter(a < 10)
    segments: b ~ c("group_1", "group_2")
    actions:
      warn_fraction: 0.1
      stop_fraction: 0.2
    label: The `col_vals_regex()` step.
    active: false
```

In practice, both of these will often be shorter as only the columns and regex arguments require values. Arguments with default values won't be written to YAML when using yaml_write() (though it is acceptable to include them with their default when generating the YAML by other means). It is also possible to preview the transformation of an agent to YAML without any writing to disk by using the yaml_agent_string() function.

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_numeric(), col_is_posix(), col_schema_match(), col_vals_between(), col_vals_decreasing(), col_vals_equal(), col_vals_expr(), col_vals_gte(), col_vals_gt(), col_vals_in_set(), col_vals_increasing(), col_vals_lte(), col_vals_lt(), col_vals_make_set(), col_vals_make_subset(), col_vals_not_between(), col_vals_not_equal(),

col_vals_not_in_set(), col_vals_not_null(), col_vals_null(), col_vals_within_spec(), conjointly(), row_count_match(), rows_complete(), rows_distinct(), serially(), specially(), tbl_match()

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
  )
```

`col_vals_within_spec` *Do values in column data fit within a specification?*

Description

The `col_vals_within_spec()` validation function, the `expect_col_vals_within_spec()` expectation function, and the `test_col_vals_within_spec()` test function all check whether column values in a table correspond to a specification (`spec`) type (details of which are available in the *Specifications* section). 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_within_spec(
  x,
  columns,
  spec,
  na_pass = FALSE,
  preconditions = NULL,
  segments = NULL,
  actions = NULL,
  step_id = NULL,
  label = NULL,
  brief = NULL,
```

```
  active = TRUE
)

expect_col_vals_within_spec(
  object,
  columns,
  spec,
  na_pass = FALSE,
  preconditions = NULL,
  threshold = 1
)

test_col_vals_within_spec(
  object,
  columns,
  spec,
  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.
spec	A specification string. Examples are `"email"`, `"url"`, and `"postal[USA]"`. All options are explained in the *Specifications* section.
na_pass	Should any encountered `NA` values be considered as passing test units? This is by default `FALSE`. Set to `TRUE` to give `NA`s a pass.
preconditions	An optional expression for mutating the input table before proceeding with the validation. This can either be provided as a one-sided R formula using a leading `~` (e.g., `~ . %>% dplyr::mutate(col = col + 10)` or as a function (e.g., `function(x) dplyr::mutate(x,col = col + 10)`. See the *Preconditions* section for more information.
segments	An optional expression or set of expressions (held in a list) that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on. See the *Segments* section for more details on this.
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 validation 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 validation function will be operating directly on data (no *agent* involvement), then any step with `active = FALSE` will simply pass the data through with no validation whatsoever. Aside from a logical vector, a one-sided R formula using a leading `~` can be used with `.` (serving as the input data table) to evaluate to a single logical value. With this approach, the **pointblank** function `has_columns()` can be used to determine whether to make a validation step active on the basis of one or more columns existing in the table (e.g., `~ . %>% has_columns(vars(d,e))`). The default for `active` 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.

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.

Specifications

A specification type must be used with the `spec` argument. This is a character-based keyword that corresponds to the type of data in the specified `columns`. The following keywords can be used:

- `"isbn"`: The International Standard Book Number (ISBN) is a unique numerical identifier for books, pamphletes, educational kits, microforms, and digital/electronic publications. The specification has been formalized in ISO 2108. This keyword can be used to validate 10- or 13-digit ISBNs.
- `"VIN"`: A vehicle identification number (VIN) is a unique code (which includes a serial number) used by the automotive industry to identify individual motor vehicles, motorcycles, scooters, and mopeds as stipulated by ISO 3779 and ISO 4030.
- `"postal_code[<country_code>]"`: A postal code (also known as postcodes, PIN, or ZIP codes, depending on region) is a series of letters, digits, or both (sometimes including spaces/punctuation) included in a postal address to aid in sorting mail. Because the coding varies by country, a country code in either the 2- (ISO 3166-1 alpha-2) or 3-letter (ISO 3166-1 alpha-3) formats needs to be supplied along with the keywords (e.g., for postal codes in Germany, `"postal_code[DE]"` or `"postal_code[DEU]"` can be used). The keyword alias `"zip"` can be used for US ZIP codes.
- `"credit_card"`: A credit card number can be validated and this check works across a large variety of credit type issuers (where card numbers are allocated in accordance with ISO/IEC 7812). Numbers can be of various lengths (typically, they are of 14-19 digits) and the key validation performed here is the usage of the Luhn algorithm.
- `"iban[<country_code>]"`: The International Bank Account Number (IBAN) is a system of identifying bank accounts across different countries for the purpose of improving cross-border transactions. IBAN values are validated through conversion to integer values and performing a basic mod-97 operation (as described in ISO 7064) on them. Because the length and coding varies by country, a country code in either the 2- (ISO 3166-1 alpha-2) or 3-letter (ISO 3166-1 alpha-3) formats needs to be supplied along with the keywords (e.g., for IBANs in Germany, `"iban[DE]"` or `"iban[DEU]"` can be used).
- `"swift"`: Business Identifier Codes (also known as SWIFT-BIC, BIC, or SWIFT code) are defined in a standard format as described by ISO 9362. These codes are unique identifiers for both financial and non-financial institutions. SWIFT stands for the Society for Worldwide Interbank Financial Telecommunication. These numbers are used when transferring money between banks, especially important for international wire transfers.
- `"phone"`, `"email"`, `"url"`, `"ipv4"`, `"ipv6"`, `"mac"`: Phone numbers, email addresses, Internet URLs, IPv4 or IPv6 addresses, and MAC addresses can be validated with their respective keywords. These validations use regex-based matching to determine validity.

Only a single `spec` value should be provided per function call.

Column Names

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()`.

Missing Values

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.

Preconditions

Providing expressions as preconditions means **pointblank** will preprocess the target table during interrogation as a preparatory step. It might happen that a particular validation requires a calculated column, some filtering of rows, or the addition of columns via a join, etc. Especially for an *agent*-based report this can be advantageous since we can develop a large validation plan with a single target table and make minor adjustments to it, as needed, along the way.

The table mutation is totally isolated in scope to the validation step(s) where preconditions is used. Using **dplyr** code is suggested here since the statements can be translated to SQL if necessary (i.e., if the target table resides in a database). 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_b = col_a + 10)). Alternatively, a function could instead be supplied (e.g., function(x) dplyr::mutate(x,col_b = col_a + 10)).

Segments

By using the segments argument, it's possible to define a particular validation with segments (or row slices) of the target table. An optional expression or set of expressions that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on.

As an example of the first type of expression that can be used, vars(a_column) will segment the target table in however many unique values are present in the column called a_column. This is great if every unique value in a particular column (like different locations, or different dates) requires it's own repeating validation.

With a formula, we can be more selective with which column values should be used for segmentation. Using a_column ~ c("group_1","group_2") will attempt to obtain two segments where one is a slice of data where the value "group_1" exists in the column named "a_column", and, the other is a slice where "group_2" exists in the same column. Each group of rows resolved from the formula will result in a separate validation step.

If there are multiple columns specified then the potential number of validation steps will be m columns multiplied by n segments resolved.

Segmentation will always occur after preconditions (i.e., statements that mutate the target table), if any, are applied. With this type of one-two combo, it's possible to generate labels for segmentation using an expression for preconditions and refer to those labels in segments without having to generate a separate version of the target table.

Actions

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).

Briefs

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.

YAML

A **pointblank** agent can be written to YAML with yaml_write() and the resulting YAML can be used to regenerate an agent (with yaml_read_agent()) or interrogate the target table (via yaml_agent_interrogate()). When col_vals_within_spec() is represented in YAML (under the top-level steps key as a list member), the syntax closely follows the signature of the validation function. Here is an example of how a complex call of col_vals_within_spec() as a validation step is expressed in R code and in the corresponding YAML representation.

```
# R statement
agent %>%
  col_vals_within_spec(
    columns = vars(a),
    spec = "email",
    na_pass = TRUE,
    preconditions = ~ . %>% dplyr::filter(b < 10),
    segments = b ~ c("group_1", "group_2"),
    actions = action_levels(warn_at = 0.1, stop_at = 0.2),
    label = "The `col_vals_within_spec()` step.",
    active = FALSE
  )

# YAML representation
steps:
- col_vals_within_spec:
    columns: vars(a)
    spec: email
    na_pass: true
    preconditions: ~. %>% dplyr::filter(b < 10)
    segments: b ~ c("group_1", "group_2")
    actions:
      warn_fraction: 0.1
      stop_fraction: 0.2
    label: The `col_vals_within_spec()` step.
    active: false
```

In practice, both of these will often be shorter as only the columns and spec arguments require values. Arguments with default values won't be written to YAML when using yaml_write() (though it is acceptable to include them with their default when generating the YAML by other means). It

is also possible to preview the transformation of an agent to YAML without any writing to disk by using the yaml_agent_string() function.

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_is_posix(), col_schema_match(), col_vals_between(), col_vals_decreasing(), col_vals_equal(), col_vals_expr(), col_vals_gte(), col_vals_gt(), col_vals_in_set(), col_vals_increasing(), col_vals_lte(), col_vals_lt(), col_vals_make_set(), col_vals_make_subset(), col_vals_not_between(), col_vals_not_equal(), col_vals_not_in_set(), col_vals_not_null(), col_vals_null(), col_vals_regex(), conjointly(), row_count_match(), rows_complete(), rows_distinct(), serially(), specially(), tbl_match()

Examples

```
# The `specifications` dataset in the
# package has columns of character data
# that correspond to each of the
# specifications that can be tested;
# the following examples will validate
# that the `email_addresses` column
# has 5 correct values (this is true if
# we get a subset of the data: the first
# five rows)
spec_slice <- specifications[1:5, ]

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

# Validate that all values in the column
# `email_addresses` are correct
agent <-
  create_agent(spec_slice) %>%
  col_vals_within_spec(
    vars(email_addresses),
    spec = "email"
  ) %>%
  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
spec_slice %>%
  col_vals_within_spec(
    vars(email_addresses),
    spec = "email"
  ) %>%
  dplyr::select(email_addresses)

# 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_within_spec(
  spec_slice,
  vars(email_addresses),
  spec = "email"
)

# D: Using the test function

# With the `test_*()` form, we should
# get a single logical value returned
# to us
spec_slice %>%
  test_col_vals_within_spec(
    vars(email_addresses),
    spec = "email"
  )
```

`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. We can use validation functions that validate row units (the col_vals_*() series), check for column existence (`col_exists()`), or validate column type (the col_is_*() series).

Because of the imposed constraint on the allowed validation functions, the ensemble of test units are either comprised rows of the table (after any common `preconditions` have been applied) or are single test units (for those functions that validate columns).

Each of the functions used in a `conjointly()` validation step (composed using 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 adjacent values in 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,
  segments = 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 functions that validate

row units (the col_vals_*() series), column existence (col_exists()), or column type (the col_is_*() series). An example of this is ~ col_vals_gte(., vars(a), 5.5), ~ col_vals_not_null(

`.list` Allows for the use of a list as an input alternative to ....

`preconditions` An optional expression for mutating the input table before proceeding with the validation. This can either be provided as a one-sided R formula using a leading ~ (e.g., `~ . %>% dplyr::mutate(col = col + 10)` or as a function (e.g., `function(x) dplyr::mutate(x,col = col + 10)`. See the *Preconditions* section for more information.

`segments` An optional expression or set of expressions (held in a list) that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on. See the *Segments* section for more details on this.

`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 validation 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 validation function will be operating directly on data (no *agent* involvement), then any step with active = FALSE will simply pass the data through with no validation whatsoever. Aside from a logical vector, a one-sided R formula using a leading ~ can be used with . (serving as the input data table) to evaluate to a single logical value. With this approach, the **pointblank** function has_columns() can be used to determine whether to make a validation step active on the basis of one or more columns existing in the table (e.g., `~ . %>% has_columns(vars(d,e))`). The default for active 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.

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.

Column Names

If providing multiple column names in any of the supplied validation steps, 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()`.

Preconditions

Providing expressions as `preconditions` means **pointblank** will preprocess the target table during interrogation as a preparatory step. It might happen that a particular validation requires a calculated column, some filtering of rows, or the addition of columns via a join, etc. Especially for an *agent*-based report this can be advantageous since we can develop a large validation plan with a single target table and make minor adjustments to it, as needed, along the way.

The table mutation is totally isolated in scope to the validation step(s) where `preconditions` is used. Using **dplyr** code is suggested here since the statements can be translated to SQL if necessary (i.e., if the target table resides in a database). 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_b = col_a + 10)`). Alternatively, a function could instead be supplied (e.g., `function(x) dplyr::mutate(x,col_b = col_a + 10)`).

Segments

By using the `segments` argument, it's possible to define a particular validation with segments (or row slices) of the target table. An optional expression or set of expressions that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on.

As an example of the first type of expression that can be used, `vars(a_column)` will segment the target table in however many unique values are present in the column called `a_column`. This is great if every unique value in a particular column (like different locations, or different dates) requires it's own repeating validation.

With a formula, we can be more selective with which column values should be used for segmentation. Using `a_column ~ c("group_1","group_2")` will attempt to obtain two segments where

one is a slice of data where the value "group_1" exists in the column named "a_column", and, the other is a slice where "group_2" exists in the same column. Each group of rows resolved from the formula will result in a separate validation step.

If there are multiple columns specified then the potential number of validation steps will be m columns multiplied by n segments resolved.

Segmentation will always occur after preconditions (i.e., statements that mutate the target table), if any, are applied. With this type of one-two combo, it's possible to generate labels for segmentation using an expression for preconditions and refer to those labels in segments without having to generate a separate version of the target table.

Actions

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).

Briefs

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.

YAML

A **pointblank** agent can be written to YAML with yaml_write() and the resulting YAML can be used to regenerate an agent (with yaml_read_agent()) or interrogate the target table (via yaml_agent_interrogate()). When conjointly() is represented in YAML (under the top-level steps key as a list member), the syntax closely follows the signature of the validation function. Here is an example of how a complex call of conjointly() as a validation step is expressed in R code and in the corresponding YAML representation.

```
# R statement
agent %>%
  conjointly(
    ~ col_vals_lt(., vars(a), 8),
    ~ col_vals_gt(., vars(c), vars(a)),
    ~ col_vals_not_null(., vars(b)),
    preconditions = ~ . %>% dplyr::filter(a < 10),
    segments = b ~ c("group_1", "group_2"),
    actions = action_levels(warn_at = 0.1, stop_at = 0.2),
    label = "The `conjointly()` step.",
    active = FALSE
```

```
  )

# YAML representation
steps:
- conjointly:
    fns:
    - ~col_vals_lt(., vars(a), 8)
    - ~col_vals_gt(., vars(c), vars(a))
    - ~col_vals_not_null(., vars(b))
    preconditions: ~. %>% dplyr::filter(a < 10)
    segments: b ~ c("group_1", "group_2")
    actions:
      warn_fraction: 0.1
      stop_fraction: 0.2
    label: The `conjointly()` step.
    active: false
```

In practice, both of these will often be shorter as only the expressions for validation steps are necessary. Arguments with default values won't be written to YAML when using yaml_write() (though it is acceptable to include them with their default when generating the YAML by other means). It is also possible to preview the transformation of an agent to YAML without any writing to disk by using the yaml_agent_string() function.

Function ID

2-33

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_decreasing(), col_vals_equal(), col_vals_expr(), col_vals_gte(), col_vals_gt(), col_vals_in_set(), col_vals_increasing(), col_vals_lte(), col_vals_lt(), col_vals_make_set(), col_vals_make_subset(), col_vals_not_between(), col_vals_not_equal(), col_vals_not_in_set(), col_vals_not_null(), col_vals_null(), col_vals_regex(), col_vals_within_spec(), row_count_match(), rows_complete(), rows_distinct(), serially(), specially(), tbl_match()

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 `8`, (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), value = 8),
    ~ col_vals_gt(., vars(c), value = 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), value = 8),
    ~ col_vals_gt(., vars(c), value = 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), value = 8),
  ~ col_vals_gt(., vars(c), value = 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), value = 8),
    ~ col_vals_gt(., vars(c), value = 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()` 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()` function, where there are more reporting options.

Usage

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

Arguments

tbl	The input table. This can be a data frame, a tibble, a tbl_dbi object, or a tbl_spark object. Alternatively, an expression can be supplied to serve as instructions on how to retrieve the target table at interrogation-time. There are two ways to specify an association to a target table: (1) as a table-prep formula, which is a right-hand side (RHS) formula expression (e.g., ~ { <table reading code>}), or (2) as a function (e.g., function() { <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().
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 expressions that should be invoked at the end of an interrogation. Each expression should be in the form of a one-sided R formula, so overall this construction should be used: end_fns = list(~ <R statements>, ~ <R statements>, ...). An example of a function included in **pointblank** that can be sensibly used here is email_blast(), which sends an email of the validation report (based on a 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 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"), Turkish ("tr"), Chinese ("zh"), Russian ("ru"), Polish ("pl"), Danish ("da"), Swedish ("sv"), and Dutch ("nl").

`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").

`read_fn` The `read_fn` argument is deprecated. Instead, supply a table-prep formula or function to `tbl`.

Value

A `ptblank_agent` object.

Data Products Obtained from an Agent

A very detailed list object, known as an 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.

The target data can be split into pieces that represent the 'pass' and 'fail' portions with the `get_sundered_data()` function. A primary requirement is an agent that has had `interrogate()` called on it. In addition, the validation steps considered for this data splitting need to be those that operate on values down a column (e.g., the col_vals_*() functions or `conjointly()`). With these in-consideration validation steps, rows with no failing test units across all validation steps comprise the 'pass' data piece, and rows with at least one failing test unit across the same series of validations constitute the 'fail' piece.

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.

While printing an *agent* will display the *agent* report in the Viewer, we can alternatively use the `get_agent_report()` to take advantage of other options (e.g., overriding the language, modifying the arrangement of report rows, etc.), and to return the report as independent objects. For example, with the `display_table = TRUE` option (the default), `get_agent_report()` will return a **gt** table object (`"gt_tbl"`). If `display_table` is set to `FALSE`, we'll get a data frame back instead.

YAML

A **pointblank** agent can be written to YAML with `yaml_write()` and the resulting YAML can be used to regenerate an agent (with `yaml_read_agent()`) or interrogate the target table (via `yaml_agent_interrogate()`). Here is an example of how a complex call of `create_agent()` is expressed in R code and in the corresponding YAML representation.

```
# R statement
create_agent(
```

```
  tbl = ~ small_table,
  tbl_name = "small_table",
  label = "An example.",
  actions = action_levels(
    warn_at = 0.10,
    stop_at = 0.25,
    notify_at = 0.35
  ),
  end_fns = list(
    ~ beepr::beep(2),
    ~ Sys.sleep(1)
  ),
  embed_report = TRUE,
  lang = "fr",
  locale = "fr_CA"
)

# YAML representation
type: agent
tbl: ~small_table
tbl_name: small_table
label: An example.
lang: fr
locale: fr_CA
actions:
  warn_fraction: 0.1
stop_fraction: 0.25
notify_fraction: 0.35
end_fns:
- ~beepr::beep(2)
- ~Sys.sleep(1)
embed_report: true
```

In practice, this block of YAML will be shorter since arguments with default values won't be written to YAML when using `yaml_write()` (though it is acceptable to include them with their default when generating the YAML by other means). The only requirement for writing the YAML representation of an *agent* is having `tbl` specified as table-prep formula.

What typically follows this chunk of YAML is a `steps` part, and that corresponds to the addition of validation steps via validation functions. Help articles for each validation function have a *YAML* section that describes how a given validation function is translated to YAML.

Should you need to preview the transformation of an *agent* to YAML (without any committing anything to disk), use the `yaml_agent_string()` function. If you already have a `.yml` file that holds an *agent*, you can get a glimpse of the R expressions that are used to regenerate that agent with `yaml_agent_show_exprs()`.

Writing an Agent to Disk

An *agent* object can be written to disk with the `x_write_disk()` function. This can be useful for keeping a history of validations and generating views of data quality over time. Agents are stored

in the serialized RDS format and can be easily retrieved with the `x_read_disk()` function.

It's recommended that table-prep formulas are supplied to the `tbl` argument of `create_agent()`. In this way, when an *agent* is read from disk through `x_read_disk()`, it can be reused to access the target table (which may change, hence the need to use an expression for this).

Combining Several Agents in a *multiagent* Object

Multiple *agent* objects can be part of a *multiagent* object, and two functions can be used for this: `create_multiagent()` and `read_disk_multiagent()`. By gathering multiple agents that have performed interrogations in the past, we can get a *multiagent* report showing how data quality evolved over time. This use case is interesting for data quality monitoring and management, and, the reporting (which can be customized with `get_multiagent_report()`) is robust against changes in validation steps for a given target table.

Figures

Function ID

1-2

See Also

Other Planning and Prep: `action_levels()`, `create_informant()`, `db_tbl()`, `draft_validation()`, `file_tbl()`, `scan_data()`, `tbl_get()`, `tbl_source()`, `tbl_store()`, `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;
# we also provide a target table and we'll
# use `pointblank::small_table`
agent <-
  create_agent(
    tbl = pointblank::small_table,
    tbl_name = "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),
    regex = "[0-9]-[a-z]{3}-[0-9]{3}"
  ) %>%
  rows_distinct() %>%
  col_vals_gt(vars(d), value = 100) %>%
  col_vals_lte(vars(c), value = 5) %>%
  col_vals_equal(
    vars(d), value = 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 object?
# Print it from an R Markdown code
# chunk, use it in a **blastula** email,
# put it in a webpage, etc.

# 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 any information that is useful for describing a particular data table. The *informant* object created by the `create_informant()` function takes information-focused functions: `info_columns()`, `info_tabular()`, `info_section()`, and `info_snippet()`.

The info_*() series of functions allows for a progressive build up of information about the target table. The `info_columns()` and `info_tabular()` functions facilitate the entry of *info text* that concerns the table columns and the table proper; the `info_section()` function allows for the creation of arbitrary sections that can have multiple subsections full of additional *info text*. The system allows for dynamic values culled from the target table by way of `info_snippet()`, for getting named text extracts from queries, and the use of {<snippet_name>} in the *info text*. To make the use of `info_snippet()` more convenient for common queries, a set of snip_*() functions are provided in the package (`snip_list()`, `snip_stats()`, `snip_lowest()`, and `snip_highest()`) though you are free to use your own expressions.

Because snippets need to query the target table to return fragments of *info text*, the `incorporate()` function needs to be used to initiate this action. This is also necessary for the *informant* to update other metadata elements such as row and column counts. Once the incorporation process is

complete, snippets and other metadata will be updated. Calling the *informant* itself will result in a reporting table. This reporting can also be accessed with the `get_informant_report()` function, where there are more reporting options.

Usage

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

Arguments

`tbl`	The input table. This can be a data frame, a tibble, a `tbl_dbi` object, or a `tbl_spark` object. Alternatively, an expression can be supplied to serve as instructions on how to retrieve the target table at incorporation-time. There are two ways to specify an association to a target table: (1) as a table-prep formula, which is a right-hand side (RHS) formula expression (e.g., ~ { <table reading code>}), or (2) as a function (e.g., function() { <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.
`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).
`agent`	A pointblank *agent* object. The table from this object can be extracted and used in the new informant instead of supplying a table in `tbl`.
`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"`), Turkish (`"tr"`), Chinese (`"zh"`), Russian (`"ru"`), Polish (`"pl"`), Danish (`"da"`), Swedish (`"sv"`), and Dutch (`"nl"`).
`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"`).
`read_fn`	The `read_fn` argument is deprecated. Instead, supply a table-prep formula or function to `tbl`.

Value

A `ptblank_informant` object.

YAML

A **pointblank** informant can be written to YAML with `yaml_write()` and the resulting YAML can be used to regenerate an informant (with `yaml_read_informant()`) or perform the 'incorporate' action using the target table (via `yaml_informant_incorporate()`). Here is an example of how a complex call of `create_informant()` is expressed in R code and in the corresponding YAML representation.

```
# R statement
create_informant(
  tbl = ~ small_table,
  tbl_name = "small_table",
  label = "An example.",
  lang = "fr",
  locale = "fr_CA"
)

# YAML representation
type: informant
tbl: ~small_table
tbl_name: small_table
info_label: An example.
lang: fr
locale: fr_CA
table:
  name: small_table
  _columns: 8
  _rows: 13.0
  _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
```

The generated YAML includes some top-level keys where `type` and `tbl` are mandatory, and, two metadata sections: `table` and `columns`. Keys that begin with an underscore character are those that are updated whenever `incorporate()` is called on an *informant*. The `table` metadata section can have multiple subsections with *info text*. The `columns` metadata section can similarly have have multiple subsections, so long as they are children to each of the column keys (in the above YAML example, `date_time` and `date` are column keys and they match the table's column names). Additional sections can be added but they must have key names on the top level that don't duplicate the default set (i.e., `type`, `table`, `columns`, etc. are treated as reserved keys).

Writing an Informant to Disk

An *informant* object can be written to disk with the `x_write_disk()` function. Informants are stored in the serialized RDS format and can be easily retrieved with the `x_read_disk()` function.

It's recommended that table-prep formulas are supplied to the `tbl` argument of `create_informant()`. In this way, when an *informant* is read from disk through `x_read_disk()`, it can be reused to access the target table (which may changed, hence the need to use an expression for this).

Figures

Function ID

1-3

See Also

Other Planning and Prep: `action_levels()`, `create_agent()`, `db_tbl()`, `draft_validation()`, `file_tbl()`, `scan_data()`, `tbl_get()`, `tbl_source()`, `tbl_store()`, `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(
    tbl = pointblank::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)
```

`create_multiagent` *Create a* **pointblank** multiagent *object*

Description

Multiple *agents* can be part of a single object called the *multiagent*. This can be useful when gathering multiple agents that have performed interrogations in the past (perhaps saved to disk with `x_write_disk()`). When be part of a *multiagent*, we can get a report that shows how data quality evolved over time. This can be of interest when it's important to monitor data quality and even the evolution of the validation plan itself. The reporting table, generated by printing a `ptblank_multiagent` object or by using the `get_multiagent_report()` function, is, by default, organized by the interrogation time and it automatically recognizes which validation steps are equivalent across interrogations.

Usage

```
create_multiagent(..., lang = NULL, locale = NULL)
```

Arguments

`...`	One or more **pointblank** agent objects.
`lang`	The language to use for any reporting that will be generated from the *multiagent*. (e.g., individual *agent reports*, *multiagent reports*, etc.). By default, `NULL` will create English (`"en"`) text. Other options include French (`"fr"`), German (`"de"`), Italian (`"it"`), Spanish (`"es"`), Portuguese (`"pt"`), Turkish (`"tr"`), Chinese (`"zh"`), Russian (`"ru"`), Polish (`"pl"`), Danish (`"da"`), Swedish (`"sv"`), and Dutch (`"nl"`).
`locale`	An optional locale ID to use for formatting values in the reporting outputs 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_multiagent` object.

Figures

Function ID

10-1

See Also

Other The multiagent: `get_multiagent_report()`, `read_disk_multiagent()`

Examples

```
if (interactive()) {

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

# To set failure limits and signal
# conditions, we designate proportional
# failure thresholds to the `warn`, `stop`,
# and `notify` states using `action_levels()`
al <-
  action_levels(
    warn_at = 0.05,
    stop_at = 0.10,
    notify_at = 0.20
  )

# We will create four different agents
# and have slightly different validation
# steps in each of them; in the first,
# `agent_1`, eight different validation
# steps are created and the agent will
# interrogate the `small_table`
agent_1 <-
  create_agent(
    tbl = small_table,
    label = "An example.",
    actions = al
  ) %>%
  col_vals_gt(
    vars(date_time),
    value = vars(date),
    na_pass = TRUE
  ) %>%
  col_vals_gt(
    vars(b),
```

```
    value = vars(g),
    na_pass = TRUE
  ) %>%
  rows_distinct() %>%
  col_vals_equal(
    vars(d),
    value = vars(d),
    na_pass = TRUE
  ) %>%
  col_vals_between(
    vars(c),
    left = vars(a), right = vars(d)
  ) %>%
  col_vals_not_between(
    vars(c),
    left = 10, right = 20,
    na_pass = TRUE
  ) %>%
  rows_distinct(vars(d, e, f)) %>%
  col_is_integer(vars(a)) %>%
  interrogate()

# The second agent, `agent_2`, retains
# all of the steps of `agent_1` and adds
# two more (the last of which is inactive)
agent_2 <-
  agent_1 %>%
  col_exists(vars(date, date_time)) %>%
  col_vals_regex(
    vars(b),
    regex = "[0-9]-[a-z]{3}-[0-9]{3}",
    active = FALSE
  ) %>%
  interrogate()

# The third agent, `agent_3`, adds a single
# validation step, removes the fifth one,
# and deactivates the first
agent_3 <-
  agent_2 %>%
  col_vals_in_set(
    vars(f),
    set = c("low", "mid", "high")
  ) %>%
  remove_steps(i = 5) %>%
  deactivate_steps(i = 1) %>%
  interrogate()

# The fourth and final agent, `agent_4`,
# reactivates steps 1 and 10, and removes
# the sixth step
agent_4 <-
  agent_3 %>%
```

```
  activate_steps(i = 1) %>%
  activate_steps(i = 10) %>%
  remove_steps(i = 6) %>%
  interrogate()

# While all the agents are slightly
# different from each other, we can still
# get a combined report of them by
# creating a 'multiagent'
multiagent <-
  create_multiagent(
    agent_1, agent_2, agent_3, agent_4
  )

# Calling `multiagent` in the console
# prints the multiagent report; but we
# can get a `gt_tbl` object with the
# `get_multiagent_report()` function
report <- get_multiagent_report(multiagent)

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). You can use `db_tbl()` as the basis for obtaining a database table for the `tbl` parameter in `create_agent()` or `create_informant()`. Another great option is supplying a table-prep formula involving `db_tbl()` to `tbl_store()` so that you have access to database tables though single names via a table store.

The username and password are supplied though environment variable names. If desired, values for the username and password can be supplied directly by enclosing such values in `I()`.

Usage

```
db_tbl(
  table,
  dbname,
  dbtype,
  host = NULL,
  port = NULL,
  user = NULL,
```

```
  password = NULL
)
```

Arguments

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 chosen as either "sqlite" or "duckdb", (2) the dbname was is set to ":memory:", and (3) the object supplied to table is a data frame or a tibble object.
dbname	The database name.
dbtype	Either an appropriate driver function (e.g., RPostgres::Postgres()) or a short-name for the database type. Valid names are: "postgresql", "postgres", or "pgsql" (PostgreSQL, using the RPostgres::Postgres() driver function); "mysql" (MySQL, using RMySQL::MySQL()); "duckdb" (DuckDB, using duckdb::duckdb()); and "sqlite" (SQLite, using RSQLite::SQLite()).
host, port	The database host and optional port number.
user, password	The environment variables used to access the username and password for the database.

Value

A tbl_dbi object.

Function ID

1-6

See Also

Other Planning and Prep: action_levels(), create_agent(), create_informant(), draft_validation(), file_tbl(), scan_data(), tbl_get(), tbl_source(), tbl_store(), validate_rmd()

Examples

```
# You can use an in-memory database
# table and supply an in-memory table
# to it too:
small_table_duckdb <-
  db_tbl(
    table = small_table,
    dbname = ":memory:",
    dbtype = "duckdb"
  )

if (interactive()) {

# It's also possible to obtain a remote
# file and shove it into an in-memory
```

```
# database; use the all-powerful
# `file_tbl()` + `db_tbl()` combo
all_revenue_large_duckdb <-
  db_tbl(
    table = file_tbl(
      file = from_github(
        file = "all_revenue_large.rds",
        repo = "rich-iannone/intendo",
        subdir = "data-large"
      )
    ),
    dbname = ":memory:",
    dbtype = "duckdb"
  )

# For remote databases, it's much the
# same; here's an example that accesses
# the `rna` table (in the RNA Central
# public database) using `db_tbl()`
rna_db_tbl <-
  db_tbl(
    table = "rna",
    dbname = "pfmegrnargs",
    dbtype = "postgres",
    host = "hh-pgsql-public.ebi.ac.uk",
    port = 5432,
    user = I("reader"),
    password = I("NWDMCE5xdipIjRrp")
  )

# Using `I()` for the user name and
# password means that you're passing in
# the actual values but, normally, you
# would want use the names of environment
# variables (envvars) to securely access
# the appropriate username and password
# values when connecting to a DB:
example_db_tbl <-
  db_tbl(
    table = "<table_name>",
    dbname = "<database_name>",
    dbtype = "<database_type_shortname>",
    host = "<connection_url>",
    port = "<connection_port>",
    user = "<DB_USER_NAME>",
    password = "<DB_PASSWORD>"
  )

# Environment variables can be created
# by editing the user `.Renviron` file and
# the `usethis::edit_r_environ()` function
# makes this pretty easy to do
```

```
# Storing table-prep formulas in a table
# store makes it easier to work with DB
# tables in pointblank; here's how to
# generate a table store with two named
# entries for table preparations
tbls <-
  tbl_store(
    small_table_duck ~ db_tbl(
      table = pointblank::small_table,
      dbname = ":memory:",
      dbtype = "duckdb"
    ),
    small_high_duck ~ db_tbl(
      table = pointblank::small_table,
      dbname = ":memory:",
      dbtype = "duckdb"
    ) %>%
      dplyr::filter(f == "high")
  )

# Now it's easy to access either of these
# tables (the second is a mutated version)
# via the `tbl_get()` function
tbl_get("small_table_duck", store = tbls)
tbl_get("small_high_duck", store = tbls)

# The table-prep formulas in `tbls`
# could also be used in functions with
# the `tbl` argument; this is thanks
# to the `tbl_source()` function
agent <-
  create_agent(
    tbl = ~ tbl_source(
      "small_table_duck",
      store = tbls
    )
  )

informant <-
  create_informant(
    tbl = ~ tbl_source(
      "small_high_duck",
      store = tbls
    )
  )

}
```

`deactivate_steps` *Deactivate one or more of an* agent*'s validation steps*

Description

Should the deactivation of one or more validation steps be necessary after creation of the validation plan for an *agent*, the `deactivate_steps()` function will be helpful for that. This has the same effect as using the `active = FALSE` option (`active` is an argument in all validation functions) for the selected validation steps. Please note that this directly edits the validation step, wiping out any function that may have been defined for whether the step should be active or not.

Usage

```
deactivate_steps(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 definition.

Value

A `ptblank_agent` object.

Function ID

9-6

See Also

For the opposite behavior, use the `activate_steps()` function.

Other Object Ops: `activate_steps()`, `export_report()`, `remove_steps()`, `set_tbl()`, `x_read_disk()`, `x_write_disk()`

Examples

```
# Create an agent that has the
# `small_table` object as the
# target table, add a few
# validation steps, and then use
# `interrogate()`
agent_1 <-
  create_agent(
    tbl = small_table,
    tbl_name = "small_table",
    label = "An example."
  ) %>%
  col_exists(vars(date)) %>%
  col_vals_regex(
    vars(b),
    regex = "[0-9]-[a-z]{3}-[0-9]"
  ) %>%
  interrogate()
```

```
# The second validation step is
# now being reconsidered and may
# be either phased out or improved
# upon; in the interim period it
# was decided that the step should
# be deactivated for now
agent_2 <-
  agent_1 %>%
  deactivate_steps(i = 2) %>%
  interrogate()
```

`draft_validation` *Draft a starter* **pointblank** *validation .R/.Rmd file with a data table*

Description

Generate a draft validation plan in a new .R or .Rmd file using an input data table. Using this workflow, the data table will be scanned to learn about its column data and a set of starter validation steps (constituting a validation plan) will be written. It's best to use a data extract that contains at least 1000 rows and is relatively free of spurious data.

Once in the file, it's possible to tweak the validation steps to better fit the expectations to the particular domain. While column inference is used to generate reasonable validation plans, it is difficult to infer the acceptable values without domain expertise. However, using `draft_validation()` could get you started on floor 10 of tackling data quality issues and is in any case better than starting with an empty code editor view.

Usage

```
draft_validation(
  tbl,
  tbl_name = NULL,
  file_name = tbl_name,
  path = NULL,
  lang = NULL,
  output_type = c("R", "Rmd"),
  add_comments = TRUE,
  overwrite = FALSE,
  quiet = FALSE
)
```

Arguments

`tbl`	The input table. This can be a data frame, tibble, a `tbl_dbi` object, or a `tbl_spark` object.

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().
file_name	An optional name for the .R or .Rmd file. This should be a name without an extension. By default, this is taken from the tbl_name but if nothing is supplied for that, the name will contain the text "draft_validation_" followed by the current date and time.
path	A path can be specified here if there shouldn't be an attempt to place the generated file in the working directory.
lang	The language to use when creating comments for the automatically- generated validation steps. By default, NULL will create English ("en") text. Other options include French ("fr"), German ("de"), Italian ("it"), Spanish ("es"), Portuguese ("pt"), Turkish ("tr"), Chinese ("zh"), Russian ("ru"), Polish ("pl"), Danish ("da"), Swedish ("sv"), and Dutch ("nl").
output_type	An option for choosing what type of output should be generated. By default, this is an .R script ("R") but this could alternatively be an R Markdown document ("Rmd").
add_comments	Should there be comments that explain the features of the validation plan in the generated document? By default, this is TRUE.
overwrite	Should a file of the same name be overwritten? By default, this is FALSE.
quiet	Should the function *not* inform when the file is written? By default this is FALSE.

Value

Invisibly returns TRUE if the file has been written.

Function ID

1-11

See Also

Other Planning and Prep: action_levels(), create_agent(), create_informant(), db_tbl(), file_tbl(), scan_data(), tbl_get(), tbl_source(), tbl_store(), validate_rmd()

Examples

```
if (interactive()) {

# Draft validation plan for the
# `dplyr::storms` dataset
draft_validation(tbl = dplyr::storms)

}
```

`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 (i.e., to send multiple email messages at the granularity of different validation steps exceeding failure thresholds).

To better get a handle on emailing with `email_blast()`, the analogous `email_create()` function can be used with a **pointblank** agent object or an x-list obtained from using the `get_agent_x_list()` function.

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 internally 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 of 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 information on how to use these functions.
`msg_subject`	The subject line of the email message.
`msg_header, msg_body, msg_footer`	Content for the header, body, and footer components of the HTML email message.

send_condition An expression that should evaluate to a logical vector of length 1. If evaluated as 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 explored 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).

YAML

A **pointblank** agent can be written to YAML with yaml_write() and the resulting YAML can be used to regenerate an agent (with yaml_read_agent()) or interrogate the target table (via yaml_agent_interrogate()). Here is an example of how the use of email_blast() inside the end_fns argument of create_agent() is expressed in R code and in the corresponding YAML representation.

```
# R statement
create_agent(
  tbl = ~ small_table,
  tbl_name = "small_table",
  label = "An example.",
  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 = "smtp2go"
      ),
    )
  )
) %>%
  col_vals_gt(vars(a), 1) %>%
  col_vals_lt(vars(a), 7)

# YAML representation
type: agent
tbl: ~small_table
tbl_name: small_table
label: An example.
lang: en
locale: en
actions:
  warn_count: 1.0
  notify_count: 2.0
end_fns: ~email_blast(x, to = "joe_public@example.com",
  from = "pb_notif@example.com", msg_subject = "Table Validation",
```

```
  credentials = blastula::creds_key(id = "smtp2go"),
  )
embed_report: true
steps:
  - col_vals_gt:
    columns: vars(a)
    value: 1.0
  - col_vals_lt:
    columns: vars(a)
    value: 7.0
```

Function ID

4-1

See Also

Other Emailing: `email_create()`, `stock_msg_body()`, `stock_msg_footer()`

Examples

```
# 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
  )

if (interactive()) {

# Validate that values in column
# `a` from `small_tbl` are always > 1
# and that they are always < 7; 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 = small_table,
    tbl_name = "small_table",
    label = "An example.",
    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 = "smtp2go"
        ),
      )
    )
  ) %>%
  col_vals_gt(vars(a), value = 1) %>%
  col_vals_lt(vars(a), value = 7) %>%
  interrogate()

}

# The above example was intentionally
# not run because email credentials
# aren't available and the `to`
# and `from` email addresses are
# nonexistent

# To get a blastula email object
# instead of eagerly sending the
# message, we can use the
# `email_create()` function
email_object <-
  create_agent(
    tbl = small_table,
    tbl_name = "small_table",
    label = "An example.",
    actions = al
  ) %>%
  col_vals_gt(vars(a), value = 5) %>%
  col_vals_lt(vars(a), value = 7) %>%
  interrogate() %>%
  email_create()
```

`email_create` *Create an email object from a* **pointblank** agent *or* informant

---

Description

The `email_create()` function produces an email message object that could be sent using the **blastula** 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()` function), or a **pointblank** *informant*. In all cases, 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

```
if (interactive()) {

# 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 = small_table,
    tbl_name = "small_table",
    label = "An example.",
    actions = al
  ) %>%
  col_vals_gt(vars(a), value = 1) %>%
  col_vals_lt(vars(a), value = 7) %>%
  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 = small_table,
    tbl_name = "small_table",
    label = "An example.",
    actions = al
  ) %>%
  col_vals_gt(vars(a), value = 5) %>%
  col_vals_lt(vars(b), value = 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 = small_table,
    tbl_name = "small_table",
    label = "An example."
  ) %>%
  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()

}
```

`export_report` *Export an* agent*,* informant*,* multiagent*, or table scan to HTML*

Description

The *agent*, *informant*, *multiagent*, and the table scan object can be easily written as HTML with `export_report()`. Furthermore, any report objects from the *agent*, *informant*, and *multiagent* (generated using `get_agent_report()`, `get_informant_report()`, and `get_multiagent_report()`) can be provided here for HTML export. Each HTML document written to disk is self-contained and easily viewable in a web browser.

Usage

```
export_report(x, filename, path = NULL, quiet = FALSE)
```

Arguments

`x`	An *agent* object of class `ptblank_agent`, an *informant* of class `ptblank_informant`, a *multiagent* of class `ptblank_multiagent`, a table scan of class `ptblank_tbl_scan`, or, customized reporting objects (`ptblank_agent_report`, `ptblank_informant_report`, `ptblank_multiagent_report.wide`, `ptblank_multiagent_report.long`).
`filename`	The filename to create on disk for the HTML export of the object provided. It's recommended that the extension `".html"` is included.
`path`	An optional path to which the file should be saved (this is automatically combined with `filename`).
`quiet`	Should the function *not* inform when the file is written? By default this is `FALSE`.

Value

Invisibly returns `TRUE` if the file has been written.

Function ID

9-3

See Also

Other Object Ops: activate_steps(), deactivate_steps(), remove_steps(), set_tbl(), x_read_disk(), x_write_disk()

Examples

```
if (interactive()) {

# A: Writing an agent report as HTML

# Let's go through the process of (1)
# developing an agent with a validation
# plan (to be used for the data quality
# analysis of the `small_table` dataset),
# (2) interrogating the agent with the
# `interrogate()` function, and (3) writing
# the agent and all its intel to a file

# Creating an `action_levels` object is a
# common workflow step when creating a
# pointblank agent; we designate 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 `tbl`
agent <-
  create_agent(
    tbl = ~ small_table,
    tbl_name = "small_table",
    label = "`export_report()`",
    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 `interrogate()`
agent <-
  agent %>%
  col_exists(vars(date, date_time)) %>%
  col_vals_regex(
    vars(b), regex = "[0-9]-[a-z]{3}-[0-9]{3}"
```

```
  ) %>%
  rows_distinct() %>%
  col_vals_gt(vars(d), value = 100) %>%
  col_vals_lte(vars(c), value = 5) %>%
  interrogate()

# The agent report can be written to an
# HTML file with `export_report()`
export_report(
  agent,
  filename = "agent-small_table.html"
)

# If you're consistently writing agent
# reports when periodically checking data,
# we could make use of `affix_date()` or
# `affix_datetime()` depending on the
# granularity you need; here's an example
# that writes the file with the format:
# 'agent-small_table-YYYY-mm-dd_HH-MM-SS.html'
export_report(
  agent,
  filename = affix_datetime(
    "agent-small_table.html"
  )
)

# B: Writing an informant report as HTML

# Let's go through the process of (1)
# creating an informant object that
# minimally describes the `small_table`
# dataset, (2) ensuring that data is
# captured from the target table using
# the `incorporate()` function, and (3)
# writing the informant report to HTML

# Create a pointblank `informant`
# object with `create_informant()`
# and the `small_table` dataset;
# `incorporate()` so that info snippets
# are integrated into the text
informant <-
  create_informant(
    tbl = ~ small_table,
    tbl_name = "small_table",
    label = "`export_report()`"
  ) %>%
  info_snippet(
    snippet_name = "high_a",
    fn = snip_highest(column = "a")
  ) %>%
  info_snippet(
```

```
    snippet_name = "low_a",
    fn = snip_lowest(column = "a")
  ) %>%
  info_columns(
    columns = vars(a),
    info = "From {low_a} to {high_a}."
  ) %>%
  info_columns(
    columns = starts_with("date"),
    info = "Time-based values."
  ) %>%
  info_columns(
    columns = "date",
    info = "The date part of `date_time`."
  ) %>%
  incorporate()

# The informant report can be written
# to an HTML file with `export_report()`;
# let's do this with `affix_date()` so
# the filename has a datestamp
export_report(
  informant,
  filename = affix_date(
    "informant-small_table.html"
  )
)

# C: Writing a table scan as HTML

# We can get an report that describes all
# of the data in the `storms` dataset
tbl_scan <- scan_data(tbl = dplyr::storms)

# The table scan object can be written
# to an HTML file with `export_report()`
export_report(
  tbl_scan,
  filename = "tbl_scan-storms.html"
)

}
```

`file_tbl` *Get a table from a local or remote file*

Description

If your target table is in a file, stored either locally or remotely, the `file_tbl()` function can make it possible to access it in a single function call. Compatible file types for this function are: CSV

(.csv), TSV (.tsv), RDA (.rda), and RDS (.rds) files. This function generates an in-memory tbl_df object, which can be used as a target table for create_agent() and create_informant(). Another great option is supplying a table-prep formula involving file_tbl() to tbl_store() so that you have access to tables based on flat files though single names via a table store.

In the remote data use case, we can specify a URL starting with http://, https://, etc., and ending with the file containing the data table. If data files are available in a GitHub repository then we can use the from_github() function to specify the name and location of the table data in a repository.

Usage

```
file_tbl(file, type = NULL, ..., keep = FALSE, verify = TRUE)
```

Arguments

file	The complete file path leading to a compatible data table either in the user system or at a http://, https://, ftp://, or ftps:// URL. For a file hosted in a GitHub repository, a call to the from_github() function can be used here.
type	The file type. This is normally inferred by file extension and is by default NULL to indicate that the extension will dictate the type of file reading that is performed internally. However, if there is no extension (and valid extensions are .csv, .tsv, .rda, and .rds), we can provide the type as either of csv, tsv, rda, or rds.
...	Options passed to **readr**'s read_csv() or read_tsv() function. Both functions have the same arguments and one or the other will be used internally based on the file extension or an explicit value given to type.
keep	In the case of a downloaded file, should it be stored in the working directory (keep = TRUE) or should it be downloaded to a temporary directory? By default, this is FALSE.
verify	If TRUE (the default) then a verification of the data object having the data.frame class will be carried out.

Value

A tbl_df object.

Function ID

1-7

See Also

Other Planning and Prep: action_levels(), create_agent(), create_informant(), db_tbl(), draft_validation(), scan_data(), tbl_get(), tbl_source(), tbl_store(), validate_rmd()

Examples

```
# A local CSV file can be obtained as
# a tbl object by supplying a path to
# the file and some CSV reading options
# (the ones used by `readr::read_csv()`)
# to the `file_tbl()` function; for
# this example we could obtain a path
# to a CSV file in the pointblank
# package with `system.file()`:
csv_path <-
  system.file(
    "data_files", "small_table.csv",
    package = "pointblank"
  )

# Then use that path in `file_tbl()`
# with the option to specify the column
# types in that CSV
tbl <-
  file_tbl(
    file = csv_path,
    col_types = "TDdcddlc"
  )

# Now that we have a `tbl` object that
# is a tibble, it can be introduced to
# `create_agent()` for validation
agent <- create_agent(tbl = tbl)

# A different strategy is to provide
# the data-reading function call
# directly to `create_agent()`:
agent <-
  create_agent(
    tbl = ~ file_tbl(
      file = system.file(
        "data_files", "small_table.csv",
        package = "pointblank"
      ),
      col_types = "TDdcddlc"
    )
  ) %>%
  col_vals_gt(vars(a), value = 0)

# All of the file-reading instructions
# are encapsulated in the `tbl`
# expression so the agent will always
# obtain the most recent version of
# the table (and the logic can be
# translated to YAML, for later use)

if (interactive()) {
```

```
# A CSV can be obtained from a public
# GitHub repo by using the `from_github()`
# helper function; let's create an agent
# a supply a table-prep formula that
# gets the same CSV file from the GitHub
# repository for the pointblank package
agent <-
  create_agent(
    tbl = ~ file_tbl(
      file = from_github(
        file = "inst/data_files/small_table.csv",
        repo = "rich-iannone/pointblank"
      ),
      col_types = "TDdcddlc"
    )
  ) %>%
  col_vals_gt(vars(a), value = 0) %>%
  interrogate()

# This interrogated the data that was
# obtained from the remote source file,
# and, there's nothing to clean up (by
# default, the downloaded file goes into
# a system temp directory)

# Storing table-prep formulas in a table
# store makes it easier to work with
# tabular data originating from files;
# here's how to generate a table store
# with two named entries for table
# preparations
tbls <-
  tbl_store(
    small_table_file ~ file_tbl(
      file = system.file(
        "data_files", "small_table.csv",
        package = "pointblank"
      ),
      col_types = "TDdcddlc"
    ),
    small_high_file ~ file_tbl(
      file = system.file(
        "data_files", "small_table.csv",
        package = "pointblank"
      ),
      col_types = "TDdcddlc"
    ) %>%
      dplyr::filter(f == "high")
  )

# Now it's easy to access either of these
# tables (the second is a mutated version)
```

```
# via the `tbl_get()` function
tbl_get("small_table_file", store = tbls)
tbl_get("small_high_file", store = tbls)

# The table-prep formulas in `tbls`
# could also be used in functions with
# the `tbl` argument; this is thanks
# to the `tbl_source()` function
agent <-
  create_agent(
    tbl = ~ tbl_source(
      "small_table_file",
      store = tbls
    )
  )

informant <-
  create_informant(
    tbl = ~ tbl_source(
      "small_high_file",
      store = tbls
    )
  )

}
```

`from_github` *Specify a file for download from GitHub*

Description

The `from_github()` function is helpful for generating a valid URL that points to a data file in a public GitHub repository. This function can be used in the `file` argument of the `file_tbl()` function or anywhere else where GitHub URLs for raw user content are needed.

Usage

```
from_github(file, repo, subdir = NULL, default_branch = "main")
```

Arguments

`file`	The name of the file to target in a GitHub repository. This can be a path leading to and including the file. This is combined with any path given in `subdir`.
`repo`	The GitHub repository address in the format username/repo[/subdir][@ref|#pull|@*release].
`subdir`	A path string representing a subdirectory in the GitHub repository. This is combined with any path components included in `file`.
`default_branch`	The name of the default branch for the repo. This is usually `"main"` (the default used here).

Value

A character vector of length 1 that contains a URL.

Function ID

13-6

See Also

Other Utility and Helper Functions: `affix_datetime()`, `affix_date()`, `col_schema()`, `has_columns()`, `stop_if_not()`

Examples

```
# A valid URL to a data file in GitHub can be
# obtained from the HEAD of the default branch
# from_github(
#   file = "inst/data_files/small_table.csv",
#   repo = "rich-iannone/pointblank"
# )

# The path to the file location can be supplied
# fully or partially to `subdir`
# from_github(
#   file = "small_table.csv",
#   repo = "rich-iannone/pointblank",
#   subdir = "inst/data_files"
# )

# We can use the first call in combination with
# `file_tbl()` and `create_agent()`; this
# supplies a table-prep formula that gets
# a CSV file from the GitHub repository for the
# pointblank package
# agent <-
#   create_agent(
#     tbl = ~ file_tbl(
#       file = from_github(
#         file = "inst/data_files/small_table.csv",
#         repo = "rich-iannone/pointblank"
#       ),
#       col_types = "TDdcddlc"
#     )
#   ) %>%
#   col_vals_gt(vars(a), 0) %>%
#   interrogate()

# The `from_github()` helper function is
# pretty powerful and can get at lots of
# different files in a repository

# A data file from GitHub can be obtained from
```

```
# a commit at release time
# from_github(
#   file = "inst/extdata/small_table.csv",
#   repo = "rich-iannone/pointblank@v0.2.1"
# )

# A file may also be obtained from a repo at the
# point in time of a specific commit (partial or
# full SHA-1 hash for the commit can be used)
# from_github(
#   file = "data-raw/small_table.csv",
#   repo = "rich-iannone/pointblank@e04a71"
# )

# A file may also be obtained from an
# *open* pull request
# from_github(
#   file = "data-raw/small_table.csv",
#   repo = "rich-iannone/pointblank#248"
# )
```

`game_revenue` *A table with game revenue data*

Description

This table is a subset of the `sj_all_revenue` table from the **intendo** data package. It's the first 2,000 rows from that table where revenue records range from `2015-01-01` to `2015-01-21`.

Usage

```
game_revenue
```

Format

A tibble with 2,000 rows and 11 variables:

player_id A `character` column with unique identifiers for each user/player.

session_id A `character` column that contains unique identifiers for each player session.

session_start A date-time column that indicates when the session (containing the revenue event) started.

time A date-time column that indicates exactly when the player purchase (or revenue event) occurred.

item_type A `character` column that provides the class of the item purchased.

item_name A `character` column that provides the name of the item purchased.

item_revenue A `numeric` column with the revenue amounts per item purchased.

session_duration A `numeric` column that states the length of the session (in minutes) for which the purchase occurred.

start_day A `Date` column that provides the date of first login for the player making a purchase.

acquisition A `character` column that provides the method of acquisition for the player.

country A `character` column that provides the probable country of residence for the player.

Function ID

14-4

See Also

Other Datasets: `game_revenue_info`, `small_table_sqlite()`, `small_table`, `specifications`

Examples

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

---

game_revenue_info *A table with metadata for the* `game_revenue` *dataset*

Description

This table contains metadata for the `game_revenue` table. The first column (named `column`) provides the column names from `game_revenue`. The second column (`info`) contains descriptions for each of the columns in that dataset. This table is in the correct format for use in the `info_columns_from_tbl()` function.

Usage

```
game_revenue_info
```

Format

A tibble with 11 rows and 2 variables:

column A `character` column with unique identifiers for each user/player.

info A `character` column that contains unique identifiers for each player session.

Function ID

14-5

See Also

Other Datasets: game_revenue, small_table_sqlite(), small_table, specifications

Examples

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

`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, etc.
- TBL: indicates whether any there were any changes to the target table just prior to interrogation. A rightward arrow from a small circle indicates that there was no mutation of the table. An arrow from a circle to a purple square indicates that preconditions were used to modify the target table. An arrow from a circle to a half-filled circle indicates that the target table has been segmented.
- EVAL: a symbol that denotes the success of interrogation evaluation for each step. A checkmark indicates no issues with evaluation. A warning sign indicates that a warning occurred during evaluation. An explosion symbol indicates that evaluation failed due to an error. Hover over the symbol for details on each condition.
- UNITS: the total number of test units for the validation step
- PASS: on top is the absolute number of *passing* test units and below that is the fraction of *passing* test units over the total number of test units.
- FAIL: on top is the absolute number of *failing* test units and below that is the fraction of *failing* test units over the total number of test units.

- 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 to download data extracts as CSV files for row-based validation steps having **failing** test units. Buttons only appear when there is data to collect.

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.

The `ptblank_agent_report` can be exported to a standalone HTML document with the `export_report()` function.

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 success of interrogation evaluation for each step. A value of `"OK"` indicates no issues with evaluation. The `"WARNING"` value indicates a warning occurred during evaluation. The `"ERROR"` VALUES indicates that evaluation failed due to an error. With `"W+E"` both warnings and an error occurred during evaluation.
- units: the total number of test units for the validation step.
- n_pass: the number of *passing* test units.
- f_pass: the fraction of *passing* test units.
- W, S, N: logical value stating whether the `warn`, `stop`, or `notify` states were entered. Will be `NA` for states that are unset.
- extract: an integer value that indicates the number of rows available in a data extract. Will be `NA` if no extract is available.

Usage

```
get_agent_report(
  agent,
  arrange_by = c("i", "severity"),
  keep = c("all", "fail_states"),
  display_table = TRUE,
  size = "standard",
  title = ":default:",
  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).
title	Options for customizing the title of the report. The default is the keyword ":default:" which produces generic title text that refers to the **pointblank** package in the language governed by the lang option. Another keyword option is ":tbl_name:", and that presents the name of the table as the title for the report. If no title is wanted, then the ":none:" keyword option can be used. Aside from keyword options, text can be provided for the title and glue::glue() calls can be used to construct the text string. If providing text, it will be interpreted as Markdown text and transformed internally to HTML. To circumvent such a transformation, use text in I() to explicitly state that the supplied text should not be transformed.
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"), Turkish ("tr"), Chinese ("zh"), Russian ("ru"), Polish ("pl"), Danish ("da"), Swedish ("sv"), and Dutch ("nl"). This lang option will override any previously set language setting (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 ptblank_agent_report object if display_table = TRUE or a tibble if display_table = FALSE.

Function ID

6-2

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), value = 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 = 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.

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

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.

Value

A list object.

Function ID

8-1

See Also

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

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), value = 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 (i.e., initiating with `create_agent()`), after interrogation with `interrogate()`, we can extract the row data that didn't pass row-based validation steps with the `get_data_extracts()` function. There is one discrete extract per row-based validation step and the amount of data available in a particular extract depends on both the fraction of test units that didn't pass the validation step and the level of sampling or explicit collection from that set of units. These extracts can be collected programmatically through `get_data_extracts()` but they may also be downloaded as CSV files from the HTML report generated by the agent's print method or through the use of `get_agent_report()`.

The availability of data extracts for each row-based validation step 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 those validation functions of the form col_vals_*() and also include `conjointly()` and `rows_distinct()`. Only functions from that combined set of validation functions can yield 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 by **pointblank** 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

8-2

See Also

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

Examples

```
# Create a series of two validation
# steps focused on testing row values
# for part of the `small_table` object;
# `interrogate()` immediately
agent <-
  create_agent(
    tbl = small_table %>%
      dplyr::select(a:f),
    label = "`get_data_extracts()`"
  ) %>%
  col_vals_gt(vars(d), value = 1000) %>%
  col_vals_between(
    vars(c),
    left = vars(a), right = vars(d),
    na_pass = TRUE
  ) %>%
  interrogate()

# Using `get_data_extracts()` with its
# defaults returns of a list of tables,
```

```
# where each table is named after the
# validation step that has an extract
# available
agent %>% get_data_extracts()

# We can get an extract for a specific
# step by specifying it in the `i`
# argument; let's get the failing rows
# from the first validation step
# (`col_vals_gt`)
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 depends 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",
  title = ":default:",
  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).
`title`	Options for customizing the title of the report. The default is the keyword `":default:"` which produces generic title text that refers to the **pointblank** package in the language governed by the `lang` option. Another keyword option is `":tbl_name:"`, and that presents the name of the table as the title for the report. If no title is wanted, then the `":none:"` keyword option can be used. Aside from keyword options, text can be provided for the title and `glue::glue()` calls can be used to construct the text string. If providing text, it will be interpreted as Markdown text and transformed internally to HTML. To circumvent such a transformation, use text in `I()` to explicitly state that the supplied text should not be transformed.

`lang` The language to use for the *information report*. By default, `NULL` will create English (`"en"`) text. Other options include French (`"fr"`), German (`"de"`), Italian (`"it"`), Spanish (`"es"`), Portuguese (`"pt"`), Turkish (`"tr"`), Chinese (`"zh"`), Russian (`"ru"`), Polish (`"pl"`), Danish (`"da"`), Swedish (`"sv"`), and Dutch (`"nl"`). This `lang` option will override any previously set language setting (e.g., by the `create_informant()` call).

`locale` An optional locale ID to use for formatting values in the *information 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_informant()` call).

Value

A **gt** table object.

Function ID

7-2

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_multiagent_report` *Get a summary report using multiple agents*

Description

We can get an informative summary table from a collective of agents by using the `get_multiagent_report()` function. Information from multiple agent can be provided in three very forms: (1) the *Long Display* (stacked reports), (2) the *Wide Display* (a comparison report), (3) as a tibble with packed columns.

Usage

```
get_multiagent_report(
  multiagent,
  display_table = TRUE,
  display_mode = c("long", "wide"),
  title = ":default:",
  lang = NULL,
  locale = NULL
)
```

Arguments

`multiagent`	A multiagent object of class `ptblank_multiagent`.
`display_table`	Should a display table be generated? If `TRUE` (the default) a display table for the report will be shown in the Viewer. If `FALSE` then a tibble will be returned.
`display_mode`	If we are getting a display table, should the agent data be presented in a `"long"` or `"wide"` form? The default is `"long"` but when comparing multiple runs where the target table is the same it might be preferable to choose `"wide"`.
`title`	Options for customizing the title of the report when `display_table = TRUE`. The default is the keyword `":default:"` which produces generic title text. If no title is wanted, then the `":none:"` keyword option can be used. Another keyword option is `":tbl_name:"`, and that presents the name of the table as the title for the report (this can only be used when `display_mode = "long"`). Aside from keyword options, text can be provided for the title and `glue::glue()` calls can be used to construct the text string. If providing text, it will be interpreted as Markdown text and transformed internally to HTML. To circumvent such a transformation, use text in `I()` to explicitly state that the supplied text should not be transformed.
`lang`	The language to use for the long or wide report forms. By default, `NULL` will preserve any language set in the component reports. The following options will force the same language across all component reports: English (`"en"`), French (`"fr"`), German (`"de"`), Italian (`"it"`), Spanish (`"es"`), Portuguese (`"pt"`), Turkish (`"tr"`), Chinese (`"zh"`), Russian (`"ru"`), Polish (`"pl"`), Danish (`"da"`), Swedish (`"sv"`), and Dutch (`"nl"`).

`locale` An optional locale ID to use for formatting values in the long or wide report forms (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 values.

Value

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

The Long Display

When displayed as `"long"` the multiagent report will stack individual agent reports in a single document in the order of the agents in the multiagent object.

Each validation plan (possibly with interrogation info) will be provided and the output for each is equivalent to calling `get_agent_report()` on each of the agents within the multiagent object.

The Wide Display

When displayed as `"wide"` the multiagent report will show data from individual agents as columns, with rows standing as validation steps common across the agents.

Each validation step is represented with an icon (standing in for the name of the validation function) and the associated SHA1 hash. This is a highly trustworthy way for ascertaining which validation steps are effectively identical across interrogations. This way of organizing the report is beneficial because different agents may have used different steps and we want to track the validation results where the validation step doesn't change but the target table does (i.e., new rows are added, existing rows are updated, etc.).

The single table from this display mode will have the following columns:

- STEP: the SHA1 hash for the validation step, possibly shared among several interrogations.
- *subsequent columns*: each column beyond `STEP` represents a separate interrogation from an *agent* object. The time stamp for the completion of each interrogation is shown as the column label.

Function ID

10-3

See Also

Other The multiagent: `create_multiagent()`, `read_disk_multiagent()`

Examples

```
if (interactive()) {

# Let's walk through several theoretical
# data quality analyses of an extremely
```

```
# small table; that table is called
# `small_table` and we can find it as a
# dataset in this package
small_table

# To set failure limits and signal
# conditions, we designate proportional
# failure thresholds to the `warn`, `stop`,
# and `notify` states using `action_levels()`
al <-
  action_levels(
    warn_at = 0.05,
    stop_at = 0.10,
    notify_at = 0.20
  )

# We will create four different agents
# and have slightly different validation
# steps in each of them; in the first,
# `agent_1`, eight different validation
# steps are created and the agent will
# interrogate the `small_table`
agent_1 <-
  create_agent(
    tbl = small_table,
    tbl_name = "small_table",
    label = "`get_multiagent_report()`",
    actions = al
  ) %>%
  col_vals_gt(
    vars(date_time),
    value = vars(date),
    na_pass = TRUE
  ) %>%
  col_vals_gt(
    vars(b),
    value = vars(g),
    na_pass = TRUE
  ) %>%
  rows_distinct() %>%
  col_vals_equal(
    vars(d),
    value = vars(d),
    na_pass = TRUE
  ) %>%
  col_vals_between(
    vars(c),
    left = vars(a), right = vars(d)
  ) %>%
  col_vals_not_between(
    vars(c),
    left = 10, right = 20,
    na_pass = TRUE
```

```
  ) %>%
  rows_distinct(vars(d, e, f)) %>%
  col_is_integer(vars(a)) %>%
  interrogate()

# The second agent, `agent_2`, retains
# all of the steps of `agent_1` and adds
# two more (the last of which is inactive)
agent_2 <-
  agent_1 %>%
  col_exists(vars(date, date_time)) %>%
  col_vals_regex(
    vars(b),
    regex = "[0-9]-[a-z]{3}-[0-9]{3}",
    active = FALSE
  ) %>%
  interrogate()

# The third agent, `agent_3`, adds a single
# validation step, removes the fifth one,
# and deactivates the first
agent_3 <-
  agent_2 %>%
  col_vals_in_set(
    vars(f),
    set = c("low", "mid", "high")
  ) %>%
  remove_steps(i = 5) %>%
  deactivate_steps(i = 1) %>%
  interrogate()

# The fourth and final agent, `agent_4`,
# reactivates steps 1 and 10, and removes
# the sixth step
agent_4 <-
  agent_3 %>%
  activate_steps(i = 1) %>%
  activate_steps(i = 10) %>%
  remove_steps(i = 6) %>%
  interrogate()

# While all the agents are slightly
# different from each other, we can still
# get a combined report of them by
# creating a 'multiagent'
multiagent <-
  create_multiagent(
    agent_1, agent_2, agent_3, agent_4
  )

# Calling `multiagent` in the console
# prints the multiagent report; but we
# can use some non-default option with
```

```
# the `get_multiagent_report()` function

# By default, `get_multiagent_report()`
# gives you a tall report with agent
# reports being stacked
report_1 <-
  get_multiagent_report(multiagent)

# We can modify the title with that's
# more suitable or use a keyword like
# `:tbl_name:` to give us the target
# table name in each section
report_2 <-
  get_multiagent_report(
    multiagent,
    title = ":tbl_name:"
  )

# We can opt for a wide display of
# the reporting info, and this is
# great when reporting on multiple
# validations of the same target
# table
report_3 <-
  get_multiagent_report(
    multiagent,
    display_mode = "wide"
  )
}
```

`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). As a final option, we can have emit all the data with a new column (called `.pb_combined`) which labels each row as passing or failing across validation steps. These labels are `"pass"` and `"fail"` by default but their values can be easily customized.

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 validation 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 together, we can count on this single column or grouping of columns to distinguish 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 considered, 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

8-3

See Also

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

Examples

```
# Create a series of two validation
# steps focused on testing row values
# for part of the `small_table` object;
# `interrogate()` immediately
agent <-
  create_agent(
    tbl = small_table %>%
      dplyr::select(a:f),
    label = "`get_sundered_data()`"
  ) %>%
  col_vals_gt(vars(d), value = 1000) %>%
  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 both
# validation steps (the default piece);
# this yields 5 of 13 total rows
agent %>% get_sundered_data()

# Get the complementary data piece:
# all of those rows that failed either
# of the two validation steps;
# this yields 8 of 13 total rows
agent %>%
  get_sundered_data(type = "fail")

# We can get all of the input data
# returned with a flag column (called
# `.pb_combined`); this is done by
# using `type = "combined"` and that
# rightmost column will contain `"pass"`
# and `"fail"` values
agent %>%
  get_sundered_data(type = "combined")

# We can change the `"pass"` or `"fail"`
# text values to another type of coding
# with the `pass_fail` argument; one
# possibility is TRUE/FALSE
agent %>%
  get_sundered_data(
    type = "combined",
    pass_fail = c(TRUE, FALSE)
  )

# ...and using `0` and `1` might be
```

```
# worthwhile in some situations
agent %>%
  get_sundered_data(
    type = "combined",
    pass_fail = 0:1
  )
```

`get_tt_param` *Get a parameter value from a summary table*

Description

The `get_tt_param()` function can help you to obtain a single parameter value from a summary table generated by the tt_*() functions `tt_summary_stats()`, `tt_string_info()`, `tt_tbl_dims()`, or `tt_tbl_colnames()`. The following parameters are to be used depending on the input `tbl`:

- from `tt_summary_stats()`: `"min"`, `"p05"`, `"q_1"`, `"med"`, `"q_3"`, `"p95"`, `"max"`, `"iqr"`, `"range"`
- from `tt_string_info()`: `"length_mean"`, `"length_min"`, `"length_max"`
- from `tt_tbl_dims()`: `"rows"`, `"columns"`
- from `tt_tbl_colnames()`: any integer present in the `.param.` column

The `tt_summary_stats()` and `tt_string_info()` functions will generate summary tables with columns that mirror the numeric and character columns in their input tables, respectively. For that reason, a column name must be supplied to the `column` argument in `get_tt_param()`.

Usage

```
get_tt_param(tbl, param, column = NULL)
```

Arguments

`tbl`	A summary table generated by either of the `tt_summary_stats()`, `tt_string_info()`, `tt_tbl_dims()`, or `tt_tbl_colnames()` functions.
`param`	The parameter name associated to the value that is to be gotten. These parameter names are always available in the first column (`.param.`) of a summary table obtained by `tt_summary_stats()`, `tt_string_info()`, `tt_tbl_dims()`, or `tt_tbl_colnames()`.
`column`	The column in the summary table for which the data value should be obtained. This must be supplied for summary tables generated by `tt_summary_stats()` and `tt_string_info()` (the `tt_tbl_dims()` and `tt_tbl_colnames()` functions will always generate a two-column summary table).

Function ID

12-7

See Also

Other Table Transformers: `tt_string_info()`, `tt_summary_stats()`, `tt_tbl_colnames()`, `tt_tbl_dims()`, `tt_time_shift()`, `tt_time_slice()`

Examples

```
# Get summary statistics for the
# first quarter of the `game_revenue`
# dataset that's included in the package
stat_tbl <-
  game_revenue %>%
  tt_time_slice(slice_point = 0.25) %>%
  tt_summary_stats()

# Based on player behavior for the first
# quarter of the year, test whether the
# maximum session duration during the
# rest of the year is never higher
game_revenue %>%
  tt_time_slice(
    slice_point = 0.25,
    keep = "right"
  ) %>%
  test_col_vals_lte(
    columns = vars(session_duration),
    value = get_tt_param(
      tbl = stat_tbl,
      param = "max",
      column = "session_duration"
    )
  )
```

`has_columns` *Determine if one or more columns exist in a table*

Description

This utility function can help you easily determine whether a column of a specified name is present in a table object. This function works well enough on a table object but it can also be used as part of a formula in any validation function's `active` argument. Using `active = ~ . %>% has_columns("column_1")` means that the validation step will be inactive if the target table doesn't contain a column named `column_1`. We can also use multiple columns in `vars()` so having `active = ~ . %>% has_columns(vars(column_1,column_` in a validation step will make it inactive at `interrogate()` time unless the columns `column_1` and `column_2` are both present.

Usage

```
has_columns(x, columns)
```

Arguments

x	The table object.
columns	One or more column names that are to be checked for existence in the table x.

Value

A length-1 logical vector.

Function ID

13-2

See Also

Other Utility and Helper Functions: affix_datetime(), affix_date(), col_schema(), from_github(), stop_if_not()

Examples

```
# The `small_table` dataset in the
# package has the columns `date_time`,
# `date`, and the `a` through `f`
# columns
small_table

# With `has_columns()` we can check for
# column existence by using it directly
# on the table; a column name can be
# verified as present by using it in
# double quotes
small_table %>% has_columns("date")

# Multiple column names can be supplied;
# this is `TRUE` because both columns are
# present in `small_table`
small_table %>% has_columns(c("a", "b"))

# It's possible to supply column names
# in `vars()` as well
small_table %>% has_columns(vars(a, b))

# Because column `h` isn't present, this
# returns `FALSE` (all specified columns
# need to be present to obtain `TRUE`)
small_table %>% has_columns(vars(a, h))

# The `has_columns()` function can be
# useful in expressions that involve the
# target table, especially if it is
# uncertain that the table will contain
# a column that's involved in a validation
```

```
# In the following agent-based validation,
# the first two steps will be 'active'
# because all columns checked for in the
# expressions are present; the third step
# is inactive because column `j` isn't
# there (without the `active` statement we
# would get an evaluation failure in the
# agent report)
agent <-
  create_agent(
    tbl = small_table,
    tbl_name = "small_table"
  ) %>%
  col_vals_gt(
    vars(c), value = vars(a),
    active = ~ . %>% has_columns(vars(a, c))
  ) %>%
  col_vals_lt(
    vars(h), value = vars(d),
    preconditions = ~ . %>% dplyr::mutate(h = d - a),
    active = ~ . %>% has_columns(vars(a, d))
  ) %>%
  col_is_character(
    vars(j),
    active = ~ . %>% has_columns("j")
  ) %>%
  interrogate()
```

`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

7-1

See Also

Other Incorporate and Report: get_informant_report()

Examples

```
if (interactive()) {

# 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(
    tbl = ~ 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 subsection 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.

Value

A `ptblank_informant` object.

Info Text

The *info text* that's used for any of the info_*() functions readily accepts Markdown formatting, and, there are a few *Text Tricks* that can be used to spice up the presentation. 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-style text by using: (1) double parentheses around text for a rectangular border as in ((label text)), or (2) triple parentheses around text for a rounded-rectangular border like (((label text))).

CSS style rules 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."`

The are quite a few CSS style rules that can be used to great effect. Here are a few you might like:

- color: <a color value>; (text color)
- background-color: <a color value>; (the text's background color)
- `text-decoration: (overline | line-through | underline);`
- `text-transform: (uppercase | lowercase | capitalize);`
- letter-spacing: <a +/- length value>;
- word-spacing: <a +/- length value>;
- `font-style: (normal | italic | oblique);`
- `font-weight: (normal | bold | 100-900);`
- `font-variant: (normal | bold | 100-900);`
- border: <a color value> <a length value> (solid | dashed | dotted);

In the above examples, 'length value' refers to a CSS length which can be expressed in different units of measure (e.g., 12px, 1em, etc.). Some lengths can be expressed as positive or negative values (e.g., for `letter-spacing`). Color values can be expressed in a few ways, the most common being in the form of hexadecimal color values or as CSS color names.

YAML

A **pointblank** informant can be written to YAML with `yaml_write()` and the resulting YAML can be used to regenerate an informant (with `yaml_read_informant()`) or perform the 'incorporate' action using the target table (via `yaml_informant_incorporate()`). The way that information on table columns is represented in YAML works like this: *info text* goes into subsections of YAML keys named for the columns, which are themselves part of the top-level `columns` key. Here is an example of how several calls of `info_columns()` are expressed in R code and how the result corresponds to the YAML representation.

```
# R statement
informant %>%
  info_columns(
    columns = "date_time",
    info = "*info text* 1."
  ) %>%
  info_columns(
    columns = "date",
    info = "*info text* 2."
  ) %>%
  info_columns(
    columns = "item_count",
    info = "*info text* 3. Statistics: {snippet_1}."
  ) %>%
  info_columns(
    columns = vars(date, date_time),
    info = "UTC time."
  )

# YAML representation
columns:
  date_time:
    _type: POSIXct, POSIXt
    info: '*info text* 1. UTC time.'
  date:
    _type: Date
    info: '*info text* 2. UTC time.'
  item_count:
    _type: integer
    info: '*info text* 3. Statistics: {snippet_1}.'
```

Subsections represented as column names are automatically generated when creating an informant. Within these, there can be multiple subsections used for holding *info text* on each column. The subsections used across the different columns needn't be the same either, the only commonality that should be enforced is the presence of the _type key (automatically updated at every `incorporate()` invocation).

It's safest to use single quotation marks around any *info text* if directly editing it in a YAML file. Note that Markdown formatting and *info snippet* placeholders (shown here as `{snippet_1}`, see `info_snippet()` for more information) are preserved in the YAML. The Markdown to HTML

conversion is done when printing an informant (or invoking get_informant_report() on an *informant*) and the processing of snippets (generation and insertion) is done when using the incorporate() function. Thus, the source text is always maintained in the YAML representation and is never written in processed form.

Figures

Function ID

3-2

See Also

Other Information Functions: info_columns_from_tbl(), info_section(), info_snippet(), info_tabular(), snip_highest(), snip_list(), snip_lowest(), snip_stats()

Examples

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

# We can add *info text* to describe
# the columns in the table with multiple
# `info_columns()` calls; the *info text*
# calls are additive to existing content
# in subsections
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

if (interactive()) {

# 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(
    filename = "informant.yml"
  )

}
```

`info_columns_from_tbl` *Add column information from another data table*

Description

The `info_columns_from_tbl()` function is a wrapper around the `info_columns()` function and is useful if you wish to apply *info text* to columns where that information already exists in a data frame (or in some form that can readily be coaxed into a data frame). The form of the input `tbl` (the one that contains column metadata) has a few basic requirements:

- the data frame must have two columns
- both columns must be of class `character`
- the first column should contain column names and the second should contain the *info text*

Each column that matches across tables (i.e., the `tbl` and the target table of the informant) will have a new entry for the `"info"` property. Empty or missing info text will be pruned from `tbl`.

Usage

```
info_columns_from_tbl(x, tbl, .add = TRUE)
```

Arguments

x	An informant object of class ptblank_informant.
tbl	The two-column data frame which contains metadata about the target table in the informant object.
.add	Should new text be added to existing text? This is TRUE by default; setting to FALSE replaces any existing text for the "info" property.

Value

A ptblank_informant object.

Function ID

3-3

See Also

The info_columns() function, which allows for manual entry of *info text*.

Other Information Functions: info_columns(), info_section(), info_snippet(), info_tabular(), snip_highest(), snip_list(), snip_lowest(), snip_stats()

Examples

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

# We can add *info text* to describe
# the columns in the table by using
# information in another table; the
# `game_revenue_info` dataset contains
# metadata for `game_revenue`

game_revenue_info

# The `info_columns_from_tbl()`
# function takes a table object
# where the first column has the
# column names and the second
# contains the *info text*
informant <-
  informant %>%
```

```
  info_columns_from_tbl(
    tbl = game_revenue_info
  )

# We can continue to add more *info
# text* since the process is additive;
# the `info_columns_from_tbl()`
# function populates the `info`
# subsection
informant <-
  informant %>%
  info_columns(
    columns = "item_revenue",
    info = "Revenue reported in USD."
  ) %>%
  info_columns(
    columns = "acquisition",
    `top list` = "{top5_aq}"
  ) %>%
  info_snippet(
    snippet_name = "top5_aq",
    fn = snip_list(column = "acquisition")
  ) %>%
  incorporate()
```

`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*).

Value

A `ptblank_informant` object.

Info Text

The *info text* that's used for any of the info_*() functions readily accepts Markdown formatting, and, there are a few *Text Tricks* that can be used to spice up the presentation. 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-style text by using: (1) double parentheses around text for a rectangular border as in ((label text)), or (2) triple parentheses around text for a rounded-rectangular border like (((label text))).

CSS style rules 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."`

The are quite a few CSS style rules that can be used to great effect. Here are a few you might like:

- color: <a color value>; (text color)
- background-color: <a color value>; (the text's background color)
- `text-decoration: (overline | line-through | underline);`
- `text-transform: (uppercase | lowercase | capitalize);`
- letter-spacing: <a +/- length value>;
- word-spacing: <a +/- length value>;
- `font-style: (normal | italic | oblique);`
- `font-weight: (normal | bold | 100-900);`
- `font-variant: (normal | bold | 100-900);`
- border: <a color value> <a length value> (solid | dashed | dotted);

In the above examples, 'length value' refers to a CSS length which can be expressed in different units of measure (e.g., 12px, 1em, etc.). Some lengths can be expressed as positive or negative values (e.g., for `letter-spacing`). Color values can be expressed in a few ways, the most common being in the form of hexadecimal color values or as CSS color names.

YAML

A **pointblank** informant can be written to YAML with `yaml_write()` and the resulting YAML can be used to regenerate an informant (with `yaml_read_informant()`) or perform the 'incorporate' action using the target table (via `yaml_informant_incorporate()`). Extra sections (i.e., neither the `table` nor the `columns` sections) can be generated and filled with *info text* by using one or more calls of `info_section()`. This is how it is expressed in both R code and in the YAML representation.

```
# R statement
informant %>%
  info_section(
    section_name = "History",
    Changes = "
- Change 1
- Change 2
- Change 3",
    `Last Update` = "(2020-10-23) at 3:28 PM."
  ) %>%
  info_section(
    section_name = "Additional Notes",
    `Notes 1` = "Notes with a {snippet}.",
    `Notes 2` = "**Bold notes**."
  )

# YAML representation
History:
  Changes: |2-

    - Change 1
    - Change 2
    - Change 3
  Last Update: (2020-10-23) at 3:28 PM.
Additional Notes:
  Notes 1: Notes with a {snippet}.
  Notes 2: '**Bold notes**.'
```

Subsections represented as column names are automatically generated when creating an informant. Within each of the top-level sections (i.e., `History` and Additional Notes) there can be multiple subsections used for holding *info text*.

It's safest to use single quotation marks around any *info text* if directly editing it in a YAML file. Note that Markdown formatting and *info snippet* placeholders (shown here as `{snippet}`, see `info_snippet()` for more information) are preserved in the YAML. The Markdown to HTML conversion is done when printing an informant (or invoking `get_informant_report()` on an *informant*) and the processing of snippets (generation and insertion) is done when using the `incorporate()` function. Thus, the source text is always maintained in the YAML representation and is never written in processed form.

Figures

Function ID

3-4

See Also

Other Information Functions: info_columns_from_tbl(), info_columns(), info_snippet(), info_tabular(), snip_highest(), snip_list(), snip_lowest(), snip_stats()

Examples

```
# Create a pointblank `informant`
# object with `create_informant()`;
# we can specify a `tbl` with the
# `~` followed by a statement that
# gets the `small_table` dataset
informant <-
  create_informant(
    tbl = ~ 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()`"
  ) %>%
  incorporate()

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

if (interactive()) {

# 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(
    filename = "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. The snippet definition is supplied either with a formula, or, with a **pointblank**-supplied snip_*() 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 other info_*() functions (`info_tabular()`, `info_columns()`, and `info_section()`). Use curly braces with just 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 result of the snippet formula into *info text* defined by an info_*() function.
`fn`	A formula that obtains a snippet of data from the target table. It's best to use a leading dot (`.`) that stands for the table itself and use pipes to construct a series of operations to be performed on the table (e.g., `~ . %>% dplyr::pull(column_2) %>% max(na.rm = TRUE)`). So long as the result is a length-1 vector, it'll likely be valid for insertion into some info text. Alternatively, a snip_*() function can be used here (these functions always return a formula that's suitable for all types of data sources).

Value

A `ptblank_informant` object.

Snip functions provided in pointblank

For convenience, there are several snip_*() functions provided in the package that work on column data from the *informant*'s target table. These are:

- `snip_list()`: get a list of column categories
- `snip_stats()`: get an inline statistical summary

- snip_lowest(): get the lowest value from a column
- snip_highest() : get the highest value from a column

As it's understood what the target table is, only the column in each of these functions is necessary for obtaining the resultant text.

YAML

A **pointblank** informant can be written to YAML with yaml_write() and the resulting YAML can be used to regenerate an informant (with yaml_read_informant()) or perform the 'incorporate' action using the target table (via yaml_informant_incorporate()). Snippets are stored in the YAML representation and here is is how they are expressed in both R code and in the YAML output (showing both the meta_snippets and columns keys to demonstrate their relationship here).

```
# R statement
informant %>%
  info_columns(
    columns = "date_time",
    `Latest Date` = "The latest date is {latest_date}."
  ) %>%
  info_snippet(
    snippet_name = "latest_date",
    fn = ~ . %>% dplyr::pull(date) %>% max(na.rm = TRUE)
  ) %>%
  incorporate()

# YAML representation
meta_snippets:
  latest_date: ~. %>% dplyr::pull(date) %>% max(na.rm = TRUE)
...
columns:
  date_time:
    _type: POSIXct, POSIXt
    Latest Date: The latest date is {latest_date}.
  date:
    _type: Date
  item_count:
    _type: integer
```

Figures

Function ID

3-5

See Also

Other Information Functions: info_columns_from_tbl(), info_columns(), info_section(), info_tabular(), snip_highest(), snip_list(), snip_lowest(), snip_stats()

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(
    tbl = ~ test_table,
    tbl_name = "test_table",
    label = "An example."
  ) %>%
  info_snippet(
    snippet_name = "row_count",
    fn = ~ . %>% nrow()
  ) %>%
  info_snippet(
    snippet_name = "max_a",
    fn = snip_highest(column = "a")
  ) %>%
  info_columns(
    columns = vars(a),
    info = "In the range of 1 to {max_a}. (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 subsection 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*).

Value

A `ptblank_informant` object.

Info Text

The *info text* that's used for any of the info_*() functions readily accepts Markdown formatting, and, there are a few *Text Tricks* that can be used to spice up the presentation. 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-style text by using: (1) double parentheses around text for a rectangular border as in ((label text)), or (2) triple parentheses around text for a rounded-rectangular border like (((label text))).

CSS style rules 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."`

The are quite a few CSS style rules that can be used to great effect. Here are a few you might like:

- color: <a color value>; (text color)
- background-color: <a color value>; (the text's background color)
- `text-decoration: (overline | line-through | underline);`
- `text-transform: (uppercase | lowercase | capitalize);`
- letter-spacing: <a +/- length value>;
- word-spacing: <a +/- length value>;
- `font-style: (normal | italic | oblique);`
- `font-weight: (normal | bold | 100-900);`
- `font-variant: (normal | bold | 100-900);`
- border: <a color value> <a length value> (solid | dashed | dotted);

In the above examples, 'length value' refers to a CSS length which can be expressed in different units of measure (e.g., 12px, 1em, etc.). Some lengths can be expressed as positive or negative values (e.g., for `letter-spacing`). Color values can be expressed in a few ways, the most common being in the form of hexadecimal color values or as CSS color names.

YAML

A **pointblank** informant can be written to YAML with `yaml_write()` and the resulting YAML can be used to regenerate an informant (with `yaml_read_informant()`) or perform the 'incorporate' action using the target table (via `yaml_informant_incorporate()`). When `info_tabular()` is represented in YAML, *info text* goes into subsections of the top-level `table` key. Here is an example of how a call of `info_tabular()` is expressed in R code and in the corresponding YAML representation.

```
# R statement
informant %>%
  info_tabular(
    section_1 = "*info text* 1.",
    `section 2` = "*info text* 2 and {snippet_1}"
  )

# YAML representation
table:
  _columns: 23
  _rows: 205.0
  _type: tbl_df
  section_1: '*info text* 1.'
  section 2: '*info text* 2 and {snippet_1}'
```

Subsection titles as defined in `info_tabular()` can be set in backticks if they are not syntactically correct as an argument name without them (e.g., when using spaces, hyphens, etc.).

It's safest to use single quotation marks around any *info text* if directly editing it in a YAML file. Note that Markdown formatting and *info snippet* placeholders (shown here as `{snippet_1}`, see `info_snippet()` for more information) are preserved in the YAML. The Markdown to HTML conversion is done when printing an informant (or invoking `get_informant_report()` on an *informant*) and the processing of snippets (generation and insertion) is done when using the `incorporate()` function. Thus, the source text is always maintained in the YAML representation and is never written in processed form.

Figures

Function ID

3-1

See Also

Other Information Functions: `info_columns_from_tbl()`, `info_columns()`, `info_section()`, `info_snippet()`, `snip_highest()`, `snip_list()`, `snip_lowest()`, `snip_stats()`

Examples

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

# We can add *info text* to describe
# the table with `info_tabular()`
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
```

```
if (interactive()) {

# 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(
    filename = "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

6-1

See Also

Other Interrogate and Report: `get_agent_report()`

Examples

```
if (interactive()) {

# 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), value = 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.

Value

Nothing is returned however log files may be written in very specific conditions.

Function ID

5-1

Examples

```
# We can create an `action_levels`
# object that has a threshold for
# the `warn` state, and, an
# associated function that should
# be invoked whenever the `warn`
# state is entered. Here, the
# function call with `log4r_step()`
# will be invoked whenever there
# is one failing test unit. It's
# important to match things up here;
# notice that `warn_at` is given a
# threshold and the list of functions
# given to `fns` has a `warn` component
al <-
  action_levels(
    warn_at = 1,
    fns = list(
      warn = ~ log4r_step(
        x, append_to = "example_log"
      )
    )
  )

# Printing `al` will show us the
# settings for the
# `action_levels` object:
al

# Let's create an agent with
# `small_table` as the target
# table, apply the `action_levels`
# object created above as `al`,
# add two validation steps, and
# then `interrogate()` the data
agent <-
  create_agent(
    tbl = ~ small_table,
    tbl_name = "small_table",
    actions = al
  ) %>%
  col_vals_gt(vars(d), 300) %>%
  col_vals_in_set(
    vars(f), c("low", "high")
  ) %>%
  interrogate()

# From the agent report, we can
# see that both steps have yielded
# warnings upon interrogation
# (i.e., filled yellow circles
# in the `W` column).
```

```
# We can see this more directly
# by inspecting the `warn`
# component of the agent's x-list:
get_agent_x_list(agent)$warn

# Upon entering the `warn` state
# in each validation step during
# interrogation, the `log4r_step()`
# function call was invoked! This
# will generate an `"example_log"`
# file in the working directory
# and log entries will be appended
# to the file

if (file.exists("example_log")) {
  file.remove("example_log")
}
```

`read_disk_multiagent` *Read* **pointblank** agents *stored on disk as a* multiagent

---

Description

An *agent* or *informant* can be written to disk with the `x_write_disk()` function. While useful for later retrieving the stored agent with `x_read_disk()` it's also possible to read a series of on-disk agents with the `read_disk_multiagent()` function, which creates a `ptblank_multiagent` object. A *multiagent* object can also be generated via the `create_multiagent()` function but is less convenient to use if one is just using agents that have been previous written to disk.

Usage

```
read_disk_multiagent(filenames = NULL, pattern = NULL, path = NULL)
```

Arguments

`filenames`	The names of files (holding *agent* objects) that were previously written by `x_write_disk()`.
`pattern`	A regex pattern for accessing saved-to-disk *agent* files located in a directory (specified in the `path` argument).
`path`	A path to a collection of files. This is either optional in the case that files are specified in `filenames` (the `path` combined with all `filenames`), or, required when providing a `pattern` for file names.

Value

A `ptblank_multiagent` object.

Function ID

10-2

See Also

Other The multiagent: `create_multiagent()`, `get_multiagent_report()`

`remove_steps` *Remove one or more of an* agent*'s validation steps*

Description

Validation steps can be removed from an *agent* object through use of the `remove_steps()` function. This is useful, for instance, when getting an agent from disk (via the `x_read_disk()` function) and omitting one or more steps from the *agent*'s validation plan. Please note that when removing validation steps all stored data extracts will be removed from the *agent*.

Usage

```
remove_steps(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 definition. If `NULL` (the default) then step removal won't occur by index.

Value

A `ptblank_agent` object.

A `ptblank_agent` object.

Function ID

9-7

See Also

Instead of removal, the `deactivate_steps()` function will simply change the `active` status of one or more validation steps to `FALSE` (and `activate_steps()` will do the opposite).

Other Object Ops: `activate_steps()`, `deactivate_steps()`, `export_report()`, `set_tbl()`, `x_read_disk()`, `x_write_disk()`

Examples

```
# Create an agent that has the
# `small_table` object as the
# target table, add a few
# validation steps, and then use
# `interrogate()`
agent_1 <-
  create_agent(
    tbl = small_table,
    tbl_name = "small_table",
    label = "An example."
  ) %>%
  col_exists(vars(date)) %>%
  col_vals_regex(
    vars(b), regex = "[0-9]-[a-z]{3}-[0-9]"
  ) %>%
  interrogate()

# The second validation step has
# been determined to be unneeded and
# is to be removed; this can be done
# by using `remove_steps()` with the
# agent object
agent_2 <-
  agent_1 %>%
  remove_steps(i = 2) %>%
  interrogate()
```

---

rows_complete *Are row data complete?*

Description

The `rows_complete()` validation function, the `expect_rows_complete()` expectation function, and the `test_rows_complete()` test function all check whether rows contain any `NA`/`NULL` values (optionally constrained to a selection of specified `columns`). 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).

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

Usage

```
rows_complete(
```

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

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

test_rows_complete(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	An optional expression for mutating the input table before proceeding with the validation. This can either be provided as a one-sided R formula using a leading ~ (e.g., ~ . %>% dplyr::mutate(col = col + 10) or as a function (e.g., function(x) dplyr::mutate(x,col = col + 10). See the *Preconditions* section for more information.
segments	An optional expression or set of expressions (held in a list) that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on. See the *Segments* section for more details on this.
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 validation 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 validation function will be operating directly on data (no *agent* involvement), then any step with `active = FALSE` will simply pass the data through with no validation whatsoever. Aside from a logical vector, a one-sided R formula using a leading `~` can be used with `.` (serving as the input data table) to evaluate to a single logical value. With this approach, the **pointblank** function `has_columns()` can be used to determine whether to make a validation step active on the basis of one or more columns existing in the table (e.g., `~ . %>% has_columns(vars(d,e))`). The default for `active` 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.

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.

Preconditions

Providing expressions as `preconditions` means **pointblank** will preprocess the target table during interrogation as a preparatory step. It might happen that a particular validation requires a calculated column, some filtering of rows, or the addition of columns via a join, etc. Especially for an *agent*-based report this can be advantageous since we can develop a large validation plan with a single target table and make minor adjustments to it, as needed, along the way.

The table mutation is totally isolated in scope to the validation step(s) where `preconditions` is used. Using **dplyr** code is suggested here since the statements can be translated to SQL if necessary

(i.e., if the target table resides in a database). 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_b = col_a + 10)`). Alternatively, a function could instead be supplied (e.g., `function(x) dplyr::mutate(x,col_b = col_a + 10)`).

Segments

By using the `segments` argument, it's possible to define a particular validation with segments (or row slices) of the target table. An optional expression or set of expressions that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on.

As an example of the first type of expression that can be used, `vars(a_column)` will segment the target table in however many unique values are present in the column called `a_column`. This is great if every unique value in a particular column (like different locations, or different dates) requires it's own repeating validation.

With a formula, we can be more selective with which column values should be used for segmentation. Using `a_column ~ c("group_1","group_2")` will attempt to obtain two segments where one is a slice of data where the value `"group_1"` exists in the column named `"a_column"`, and, the other is a slice where `"group_2"` exists in the same column. Each group of rows resolved from the formula will result in a separate validation step.

Segmentation will always occur after `preconditions` (i.e., statements that mutate the target table), if any, are applied. With this type of one-two combo, it's possible to generate labels for segmentation using an expression for `preconditions` and refer to those labels in `segments` without having to generate a separate version of the target table.

Actions

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. 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).

Briefs

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.

YAML

A **pointblank** agent can be written to YAML with `yaml_write()` and the resulting YAML can be used to regenerate an agent (with `yaml_read_agent()`) or interrogate the target table (via

yaml_agent_interrogate()). When rows_complete() is represented in YAML (under the top-level steps key as a list member), the syntax closely follows the signature of the validation function. Here is an example of how a complex call of rows_complete() as a validation step is expressed in R code and in the corresponding YAML representation.

```
# R statement
agent %>%
  rows_complete(
    columns = vars(a, b),
    preconditions = ~ . %>% dplyr::filter(a < 10),
    segments = b ~ c("group_1", "group_2"),
    actions = action_levels(warn_at = 0.1, stop_at = 0.2),
    label = "The `rows_complete()` step.",
    active = FALSE
  )

# YAML representation
steps:
- rows_complete:
    columns: vars(a, b)
    preconditions: ~. %>% dplyr::filter(a < 10)
    segments: b ~ c("group_1", "group_2")
    actions:
      warn_fraction: 0.1
      stop_fraction: 0.2
    label: The `rows_complete()` step.
    active: false
```

In practice, both of these will often be shorter. A value for columns is only necessary if checking for unique values across a subset of columns. Arguments with default values won't be written to YAML when using yaml_write() (though it is acceptable to include them with their default when generating the YAML by other means). It is also possible to preview the transformation of an agent to YAML without any writing to disk by using the yaml_agent_string() function.

Function ID

2-21

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_decreasing(), col_vals_equal(), col_vals_expr(), col_vals_gte(), col_vals_gt(), col_vals_in_set(), col_vals_increasing(), col_vals_lte(), col_vals_lt(), col_vals_make_set(), col_vals_make_subset(), col_vals_not_between(), col_vals_not_equal(), col_vals_not_in_set(), col_vals_not_null(), col_vals_null(), col_vals_regex(), col_vals_within_spec(), conjointly(), row_count_match(), rows_distinct(), serially(), specially(), tbl_match()

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)
  )

tbl

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

# Validate that when considering only
# data in columns `a` and `b`, there
# are only complete rows (i.e., all
# rows have no `NA` values)
agent <-
  create_agent(tbl = tbl) %>%
  rows_complete(vars(a, b)) %>%
  interrogate()

# Determine if this validation passed
# by using `all_passed()`
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 %>%
  rows_complete(vars(a, b)) %>%
  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_rows_complete(
  tbl, vars(a, b)
)

# D: Using the test function

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

---

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).

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

Usage

```
rows_distinct(
  x,
  columns = NULL,
  preconditions = NULL,
  segments = 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	An optional expression for mutating the input table before proceeding with the validation. This can either be provided as a one-sided R formula using a leading ~ (e.g., ~ . %>% dplyr::mutate(col = col + 10) or as a function (e.g., function(x) dplyr::mutate(x,col = col + 10). See the *Preconditions* section for more information.
segments	An optional expression or set of expressions (held in a list) that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on. See the *Segments* section for more details on this.
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 validation 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 validation function will be operating directly on data (no *agent*

involvement), then any step with `active = FALSE` will simply pass the data through with no validation whatsoever. Aside from a logical vector, a one-sided R formula using a leading `~` can be used with `.` (serving as the input data table) to evaluate to a single logical value. With this approach, the **pointblank** function `has_columns()` can be used to determine whether to make a validation step active on the basis of one or more columns existing in the table (e.g., `~ . %>% has_columns(vars(d,e))`). The default for `active` 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.

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.

Preconditions

Providing expressions as `preconditions` means **pointblank** will preprocess the target table during interrogation as a preparatory step. It might happen that a particular validation requires a calculated column, some filtering of rows, or the addition of columns via a join, etc. Especially for an *agent*-based report this can be advantageous since we can develop a large validation plan with a single target table and make minor adjustments to it, as needed, along the way.

The table mutation is totally isolated in scope to the validation step(s) where `preconditions` is used. Using **dplyr** code is suggested here since the statements can be translated to SQL if necessary (i.e., if the target table resides in a database). 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_b = col_a + 10)`). Alternatively, a function could instead be supplied (e.g., `function(x) dplyr::mutate(x,col_b = col_a + 10)`).

Segments

By using the `segments` argument, it's possible to define a particular validation with segments (or row slices) of the target table. An optional expression or set of expressions that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on.

As an example of the first type of expression that can be used, `vars(a_column)` will segment the target table in however many unique values are present in the column called `a_column`. This is great

if every unique value in a particular column (like different locations, or different dates) requires it's own repeating validation.

With a formula, we can be more selective with which column values should be used for segmentation. Using `a_column ~ c("group_1","group_2")` will attempt to obtain two segments where one is a slice of data where the value `"group_1"` exists in the column named `"a_column"`, and, the other is a slice where `"group_2"` exists in the same column. Each group of rows resolved from the formula will result in a separate validation step.

Segmentation will always occur after `preconditions` (i.e., statements that mutate the target table), if any, are applied. With this type of one-two combo, it's possible to generate labels for segmentation using an expression for `preconditions` and refer to those labels in `segments` without having to generate a separate version of the target table.

Actions

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. 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).

Briefs

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.

YAML

A **pointblank** agent can be written to YAML with `yaml_write()` and the resulting YAML can be used to regenerate an agent (with `yaml_read_agent()`) or interrogate the target table (via `yaml_agent_interrogate()`). When `rows_distinct()` is represented in YAML (under the top-level `steps` key as a list member), the syntax closely follows the signature of the validation function. Here is an example of how a complex call of `rows_distinct()` as a validation step is expressed in R code and in the corresponding YAML representation.

```
# R statement
agent %>%
  rows_distinct(
    columns = vars(a, b),
    preconditions = ~ . %>% dplyr::filter(a < 10),
    segments = b ~ c("group_1", "group_2"),
    actions = action_levels(warn_at = 0.1, stop_at = 0.2),
    label = "The `rows_distinct()` step.",
    active = FALSE
```

```
  )

# YAML representation
steps:
- rows_distinct:
    columns: vars(a, b)
    preconditions: ~. %>% dplyr::filter(a < 10)
    segments: b ~ c("group_1", "group_2")
    actions:
      warn_fraction: 0.1
      stop_fraction: 0.2
    label: The `rows_distinct()` step.
    active: false
```

In practice, both of these will often be shorter. A value for `columns` is only necessary if checking for unique values across a subset of columns. Arguments with default values won't be written to YAML when using `yaml_write()` (though it is acceptable to include them with their default when generating the YAML by other means). It is also possible to preview the transformation of an agent to YAML without any writing to disk by using the `yaml_agent_string()` function.

Function ID

2-20

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_decreasing()`, `col_vals_equal()`, `col_vals_expr()`, `col_vals_gte()`, `col_vals_gt()`, `col_vals_in_set()`, `col_vals_increasing()`, `col_vals_lte()`, `col_vals_lt()`, `col_vals_make_set()`, `col_vals_make_subset()`, `col_vals_not_between()`, `col_vals_not_equal()`, `col_vals_not_in_set()`, `col_vals_not_null()`, `col_vals_null()`, `col_vals_regex()`, `col_vals_within_spec()`, `conjointly()`, `row_count_match()`, `rows_complete()`, `serially()`, `specially()`, `tbl_match()`

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)
  )

tbl

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

```
# 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 this validation passed
# by using `all_passed()`
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 %>%
  rows_distinct(vars(a, b)) %>%
  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_rows_distinct(
  tbl, vars(a, b)
)

# D: Using the test function

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

`row_count_match` *Does the row count match that of a different table?*

Description

The `row_count_match()` validation function, the `expect_row_count_match()` expectation function, and the `test_row_count_match()` test function all check whether the row count in the target table matches that of a comparison 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, there is a single test unit that hinges on whether the row counts for the two tables are the same (after any `preconditions` have been applied).

Usage

```
row_count_match(
  x,
  tbl_compare,
  preconditions = NULL,
  segments = NULL,
  actions = NULL,
  step_id = NULL,
  label = NULL,
  brief = NULL,
  active = TRUE
)

expect_row_count_match(
  object,
  tbl_compare,
  preconditions = NULL,
  threshold = 1
)

test_row_count_match(object, tbl_compare, 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()`.
`tbl_compare`	A table to compare against the target table in terms of row count values. This can either be a table object, a table-prep formula.This can be a table object such as a data frame, a tibble, a `tbl_dbi` object, or a `tbl_spark` object. Alternatively, a table-prep formula (~ <table reading code>) or a function (function() <table reading code>) can be used to lazily read in the table at interrogation time.

`preconditions` An optional expression for mutating the input table before proceeding with the validation. This can either be provided as a one-sided R formula using a leading `~` (e.g., `~ . %>% dplyr::mutate(col = col + 10)` or as a function (e.g., `function(x) dplyr::mutate(x,col = col + 10)`. See the *Preconditions* section for more information.

`segments` An optional expression or set of expressions (held in a list) that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on. See the *Segments* section for more details on this.

`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 validation 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 validation function will be operating directly on data (no *agent* involvement), then any step with `active = FALSE` will simply pass the data through with no validation whatsoever. Aside from a logical vector, a one-sided R formula using a leading `~` can be used with `.` (serving as the input data table) to evaluate to a single logical value. With this approach, the **pointblank** function `has_columns()` can be used to determine whether to make a validation step active on the basis of one or more columns existing in the table (e.g., `~ . %>% has_columns(vars(d,e))`). The default for `active` 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.

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.

Preconditions

Providing expressions as `preconditions` means **pointblank** will preprocess the target table during interrogation as a preparatory step. It might happen that this particular validation requires some operation on the target table before the row count comparison takes place. Using `preconditions` can be useful at times since since we can develop a large validation plan with a single target table and make minor adjustments to it, as needed, along the way.

The table mutation is totally isolated in scope to the validation step(s) where `preconditions` is used. Using **dplyr** code is suggested here since the statements can be translated to SQL if necessary (i.e., if the target table resides in a database). 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. Alternatively, a function could instead be supplied.

Segments

By using the `segments` argument, it's possible to define a particular validation with segments (or row slices) of the target table. An optional expression or set of expressions that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on.

As an example of the first type of expression that can be used, `vars(a_column)` will segment the target table in however many unique values are present in the column called `a_column`. This is great if every unique value in a particular column (like different locations, or different dates) requires it's own repeating validation.

With a formula, we can be more selective with which column values should be used for segmentation. Using `a_column ~ c("group_1","group_2")` will attempt to obtain two segments where one is a slice of data where the value `"group_1"` exists in the column named `"a_column"`, and, the other is a slice where `"group_2"` exists in the same column. Each group of rows resolved from the formula will result in a separate validation step.

Segmentation will always occur after `preconditions` (i.e., statements that mutate the target table), if any, are applied. With this type of one-two combo, it's possible to generate labels for segmentation using an expression for `preconditions` and refer to those labels in `segments` without having to generate a separate version of the target table.

Actions

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).

Briefs

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.

YAML

A **pointblank** agent can be written to YAML with `yaml_write()` and the resulting YAML can be used to regenerate an agent (with `yaml_read_agent()`) or interrogate the target table (via `yaml_agent_interrogate()`). When `row_count_match()` is represented in YAML (under the top-level `steps` key as a list member), the syntax closely follows the signature of the validation function. Here is an example of how a complex call of `row_count_match()` as a validation step is expressed in R code and in the corresponding YAML representation.

```
# R statement
agent %>%
  row_count_match(
    tbl_compare = ~ file_tbl(
      file = from_github(
        file = "all_revenue_large.rds",
        repo = "rich-iannone/intendo",
        subdir = "data-large"
        )
      ),
    preconditions = ~ . %>% dplyr::filter(a < 10),
    segments = b ~ c("group_1", "group_2"),
    actions = action_levels(warn_at = 0.1, stop_at = 0.2),
    label = "The `row_count_match()` step.",
    active = FALSE
  )

# YAML representation
steps:
- row_count_match:
    tbl_compare: ~ file_tbl(
      file = from_github(
        file = "all_revenue_large.rds",
```

```
      repo = "rich-iannone/intendo",
      subdir = "data-large"
      )
    )
  preconditions: ~. %>% dplyr::filter(a < 10)
  segments: b ~ c("group_1", "group_2")
  actions:
    warn_fraction: 0.1
    stop_fraction: 0.2
  label: The `row_count_match()` step.
  active: false
```

In practice, both of these will often be shorter. Arguments with default values won't be written to YAML when using `yaml_write()` (though it is acceptable to include them with their default when generating the YAML by other means). It is also possible to preview the transformation of an agent to YAML without any writing to disk by using the `yaml_agent_string()` function.

Function ID

2-31

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_decreasing()`, `col_vals_equal()`, `col_vals_expr()`, `col_vals_gte()`, `col_vals_gt()`, `col_vals_in_set()`, `col_vals_increasing()`, `col_vals_lte()`, `col_vals_lt()`, `col_vals_make_set()`, `col_vals_make_subset()`, `col_vals_not_between()`, `col_vals_not_equal()`, `col_vals_not_in_set()`, `col_vals_not_null()`, `col_vals_null()`, `col_vals_regex()`, `col_vals_within_spec()`, `conjointly()`, `rows_complete()`, `rows_distinct()`, `serially()`, `specially()`, `tbl_match()`

Examples

```
# Create a simple table with three
# columns and four rows of values
tbl <-
  dplyr::tibble(
    a = c(5, 7, 6, 5),
    b = c(7, 1, 0, 0),
    c = c(1, 1, 1, 3)
  )

tbl

# Create a second table which is
# quite different but has the
# same number of rows as `tbl`
tbl_2 <-
  dplyr::tibble(
    e = c("a", NA, "a", "c"),
    f = c(2.6, 1.2, 0, NA)
```

```
  )

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

# Validate that the count of rows
# in the target table (`tbl`) matches
# that of the comparison table
# (`tbl_2`)
agent <-
  create_agent(tbl = tbl) %>%
  row_count_match(tbl_compare = tbl_2) %>%
  interrogate()

# Determine if this validation passed
# by using `all_passed()`
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 %>%
  row_count_match(tbl_compare = tbl_2)

# 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_row_count_match(
  tbl, tbl_compare = tbl_2
)

# D: Using the test function

# With the `test_*()` form, we should
# get a single logical value returned
# to us
tbl %>%
  row_count_match(
    tbl_compare = tbl_2
```

```
)
```

`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,
  width = NULL,
  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`.

`width` An optional fixed width (in pixels) for the HTML report. By default, no fixed width is applied.

`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"`), Turkish (`"tr"`), Chinese (`"zh"`), Russian (`"ru"`), Polish (`"pl"`), Danish (`"da"`), Swedish (`"sv"`), and Dutch (`"nl"`).

`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"`).

Figures

Function ID

1-1

See Also

Other Planning and Prep: `action_levels()`, `create_agent()`, `create_informant()`, `db_tbl()`, `draft_validation()`, `file_tbl()`, `tbl_get()`, `tbl_source()`, `tbl_store()`, `validate_rmd()`

Examples

```
if (interactive()) {

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

}
```

`serially` *Run several tests and a final validation in a serial manner*

Description

The `serially()` validation function allows for a series of tests to run in sequence before either culminating in a final validation step or simply exiting the series. This construction allows for pre-testing that may make sense before a validation step. For example, there may be situations where it's vital to check a column type before performing a validation on the same column (since having the wrong type can result in an evaluation error for the subsequent validation). Another serial workflow might entail having a bundle of checks in a prescribed order and, if all pass, then the goal of this testing has been achieved (e.g., checking if a table matches another through a series of increasingly specific tests).

A series as specified inside `serially()` is composed with a listing of calls, and we would draw upon test functions (**T**) to describe tests and optionally provide a finalizing call with a validation function (**V**). The following constraints apply:

- there must be at least one test function in the series (**T** -> **V** is good, **V** is *not*)
- there can only be one validation function call, **V**; it's optional but, if included, it must be placed at the end (**T** -> **T** -> **V** is good, these sequences are bad: (1) **T** -> **V** -> **T**, (2) **T** -> **T** -> **V** -> **V**)
- a validation function call (**V**), if included, mustn't itself yield multiple validation steps (this may happen when providing multiple `columns` or any `segments`)

Here's an example of how to arrange expressions:

```
~ test_col_exists(., columns = vars(count)),
~ test_col_is_numeric(., columns = vars(count)),
~ col_vals_gt(., columns = vars(count), value = 2)
```

This series concentrates on the column called `count` and first checks whether the column exists, then checks if that column is numeric, and then finally validates whether all values in the column are greater than `2`.

Note that in the above listing of calls, the `.` stands in for the target table and is always necessary here. Also important is that all test_*() functions have a `threshold` argument that is set to `1` by default. Should you need to bump up the threshold value it can be changed to a different integer value (as an absolute threshold of failing test units) or a decimal value between `0` and `1` (serving as a fractional threshold of failing test units).

Usage

```
serially(
  x,
  ...,
  .list = list2(...),
  preconditions = NULL,
```

```
  actions = NULL,
  step_id = NULL,
  label = NULL,
  brief = NULL,
  active = TRUE
)

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

test_serially(
  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 test_*() function calls (e.g., test_col_vals_between(), etc.) arranged in sequence of intended interrogation order. Typically, validations up until the final one would have some threshold value set (default is 1) for short circuiting within the series. A finishing validation function call (e.g., col_vals_increasing(), etc.) can optionally be inserted at the end of the series, serving as a validation step that only undergoes interrogation if the prior tests adequately pass. An example of this is ~ test_column_exists(., vars(a)), ~ col_vals_not_null(., vars(a))).
.list	Allows for the use of a list as an input alternative to ....
preconditions	An optional expression for mutating the input table before proceeding with the validation. This can either be provided as a one-sided R formula using a leading ~ (e.g., ~ . %>% dplyr::mutate(col = col + 10) or as a function (e.g., function(x) dplyr::mutate(x,col = col + 10). See the *Preconditions* section for more information.
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 validation 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 validation function will be operating directly on data (no *agent* involvement), then any step with `active = FALSE` will simply pass the data through with no validation whatsoever. Aside from a logical vector, a one-sided R formula using a leading `~` can be used with `.` (serving as the input data table) to evaluate to a single logical value. With this approach, the **pointblank** function `has_columns()` can be used to determine whether to make a validation step active on the basis of one or more columns existing in the table (e.g., `~ . %>% has_columns(vars(d,e))`). The default for `active` 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.

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.

Column Names

If providing multiple column names in any of the supplied validation steps, 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()`.

Preconditions

Providing expressions as `preconditions` means **pointblank** will preprocess the target table during interrogation as a preparatory step. It might happen that a particular validation requires a calculated column, some filtering of rows, or the addition of columns via a join, etc. Especially for an *agent*-based report this can be advantageous since we can develop a large validation plan with a single target table and make minor adjustments to it, as needed, along the way.

The table mutation is totally isolated in scope to the validation step(s) where `preconditions` is used. Using **dplyr** code is suggested here since the statements can be translated to SQL if necessary (i.e., if the target table resides in a database). 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_b = col_a + 10)`). Alternatively, a function could instead be supplied (e.g., `function(x) dplyr::mutate(x,col_b = col_a + 10)`).

Actions

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).

Briefs

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.

YAML

A **pointblank** agent can be written to YAML with `yaml_write()` and the resulting YAML can be used to regenerate an agent (with `yaml_read_agent()`) or interrogate the target table (via `yaml_agent_interrogate()`). When `serially()` is represented in YAML (under the top-level `steps` key as a list member), the syntax closely follows the signature of the validation function. Here is an example of how a complex call of `serially()` as a validation step is expressed in R code and in the corresponding YAML representation.

```
# R statement
agent %>%
  serially(
    ~ col_vals_lt(., vars(a), 8),
```

```
    ~ col_vals_gt(., vars(c), vars(a)),
    ~ col_vals_not_null(., vars(b)),
    preconditions = ~ . %>% dplyr::filter(a < 10),
    actions = action_levels(warn_at = 0.1, stop_at = 0.2),
    label = "The `serially()` step.",
    active = FALSE
  )

# YAML representation
steps:
- serially:
    fns:
    - ~col_vals_lt(., vars(a), 8)
    - ~col_vals_gt(., vars(c), vars(a))
    - ~col_vals_not_null(., vars(b))
    preconditions: ~. %>% dplyr::filter(a < 10)
    actions:
      warn_fraction: 0.1
      stop_fraction: 0.2
    label: The `serially()` step.
    active: false
```

In practice, both of these will often be shorter as only the expressions for validation steps are necessary. Arguments with default values won't be written to YAML when using `yaml_write()` (though it is acceptable to include them with their default when generating the YAML by other means). It is also possible to preview the transformation of an agent to YAML without any writing to disk by using the `yaml_agent_string()` function.

Function ID

2-34

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_decreasing()`, `col_vals_equal()`, `col_vals_expr()`, `col_vals_gte()`, `col_vals_gt()`, `col_vals_in_set()`, `col_vals_increasing()`, `col_vals_lte()`, `col_vals_lt()`, `col_vals_make_set()`, `col_vals_make_subset()`, `col_vals_not_between()`, `col_vals_not_equal()`, `col_vals_not_in_set()`, `col_vals_not_null()`, `col_vals_null()`, `col_vals_regex()`, `col_vals_within_spec()`, `conjointly()`, `row_count_match()`, `rows_complete()`, `rows_distinct()`, `specially()`, `tbl_match()`

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(6, 4, 9),
    c = c(1, 2, 3)
  )

tbl

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

# The `serially()` function can be set
# up to perform a series of tests and
# then perform a validation (only if
# all tests pass); here, we are going
# to (1) test whether columns `a` and
# `b` are numeric, (2) check that both
# don't have any `NA` values, and (3)
# perform a finalizing validation that
# checks whether values in `b` are
# greater than values in `a`
agent_1 <-
  create_agent(tbl = tbl) %>%
  serially(
    ~ test_col_is_numeric(., vars(a, b)),
    ~ test_col_vals_not_null(., vars(a, b)),
    ~ col_vals_gt(., vars(b), vars(a))
    ) %>%
  interrogate()

# Determine if this validation
# had no failing test units (there are
# 4 tests and a final validation)
all_passed(agent_1)

# Calling `agent` in the console
# prints the agent's report; but we
# can get a `gt_tbl` object directly
# with `get_agent_report(agent_1)`

# What's going on? All four of the tests
# passed and so the final validation
# occurred; there were no failing test
# units in that either!

# The final validation is optional; here
# is a different agent where only the
# serial tests are performed
agent_2 <-
  create_agent(tbl = tbl) %>%
  serially(
    ~ test_col_is_numeric(., vars(a, b)),
    ~ test_col_vals_not_null(., vars(a, b))
```

```
  ) %>%
  interrogate()

# Everything is good here too:
all_passed(agent_2)

# 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 %>%
  serially(
    ~ test_col_is_numeric(., vars(a, b)),
    ~ test_col_vals_not_null(., vars(a, b)),
    ~ col_vals_gt(., vars(b), vars(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_serially(
  tbl,
  ~ test_col_is_numeric(., vars(a, b)),
  ~ test_col_vals_not_null(., vars(a, b)),
  ~ col_vals_gt(., vars(b), vars(a))
)

# D: Using the test function

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

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

Description

Setting a data table to an *agent* or an *informant* with `set_tbl()` replaces any associated table (a data frame, a tibble, objects of class `tbl_dbi` or `tbl_spark`).

Usage

```
set_tbl(x, tbl, tbl_name = NULL, label = NULL)
```

Arguments

`x`	An *agent* object of class `ptblank_agent`, or, an *informant* of class `ptblank_informant`.
`tbl`	The input table for the *agent* or the *informant*. This can be a data frame, a tibble, a `tbl_dbi` object, or a `tbl_spark` object. Alternatively, an expression can be supplied to serve as instructions on how to retrieve the target table at interrogation- or incorporation-time. There are two ways to specify an association to a target table: (1) as a table-prep formula, which is a right-hand side (RHS) formula expression (e.g., ~ { <table reading code>}), or (2) as a function (e.g., function() { <table reading code>}).
`tbl_name`	A optional name to assign to the new 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 validation plan. If no value is provided then any existing label will be retained.

Function ID

9-4

See Also

Other Object Ops: `activate_steps()`, `deactivate_steps()`, `export_report()`, `remove_steps()`, `x_read_disk()`, `x_write_disk()`

Examples

```
# Set 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
  )

# Create an agent that has
# `small_table` set as the target
# table via `tbl`; apply the actions,
# add some validation steps and then
# interrogate the data
```

```
agent_1 <-
  create_agent(
    tbl = small_table,
    tbl_name = "small_table",
    label = "An example.",
    actions = al
  ) %>%
  col_exists(vars(date, date_time)) %>%
  col_vals_regex(
    vars(b), "[0-9]-[a-z]{3}-[0-9]{3}"
  ) %>%
  rows_distinct() %>%
  interrogate()

# Replace the agent's association to
# `small_table` with a mutated version
# of it (one that removes duplicate rows);
# then, interrogate the new target table
agent_2 <-
  agent_1 %>%
  set_tbl(
    tbl = small_table %>% dplyr::distinct()
  ) %>%
  interrogate()
```

`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

14-1

See Also

Other Datasets: `game_revenue_info`, `game_revenue`, `small_table_sqlite()`, `specifications`

Examples

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

---

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

Description

The `small_table_sqlite()` function creates an 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

14-2

See Also

Other Datasets: `game_revenue_info`, `game_revenue`, `small_table`, `specifications`

Examples

```
# Use `small_table_sqlite()` to
# create an 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_highest() 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()'s fn argument.

Function ID

3-9

See Also

Other Information Functions: info_columns_from_tbl(), info_columns(), info_section(), info_snippet(), info_tabular(), snip_list(), snip_lowest(), snip_stats()

Examples

```
# Generate an informant object, add
# a snippet with `info_snippet()`
# and `snip_highest()` (giving us a
# method to get the highest value in
# column `a`); define a location for
# the snippet result in `{ }` and
# then `incorporate()` the snippet
# into the info text
informant <-
  create_informant(
    tbl = ~ small_table,
    tbl_name = "small_table",
    label = "An example."
  ) %>%
  info_columns(
    columns = "a",
    `Highest Value` = "Highest value is {highest_a}."
  ) %>%
```

```
  info_snippet(
    snippet_name = "highest_a",
    fn = snip_highest(column = "a")
  ) %>%
  incorporate()

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

`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()` 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,
  sorting = c("inorder", "infreq", "inseq"),
  reverse = FALSE,
  sep = ",",
  and_or = NULL,
  oxford = TRUE,
  as_code = TRUE,
  quot_str = NULL,
  lang = NULL
)
```

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`. By default, the limit is `5`.
`sorting`	A keyword used to designate the type of sorting to use for the list. The three options are `"inorder"` (the default), `"infreq"`, and `"inseq"`. With `"inorder"`, distinct items are listed in the order in which they first appear. Using `"infreq"` orders the items by the decreasing frequency of each item. The `"inseq"` option applies an alphanumeric sorting to the distinct list items.
`reverse`	An option to reverse the ordering of list items. By default, this is `FALSE` but using `TRUE` will reverse the items before applying the `limit`.
`sep`	The separator to use between list items. By default, this is a comma.

and_or	The type of conjunction to use between the final and penultimate list items (should the item length be below the limit value). If NULL (the default) is used, then the 'and' conjunction will be used. Alternatively, the following keywords can be used: "and", "or", or an empty string (for no conjunction at all).
oxford	Whether to use an Oxford comma under certain conditions. By default, this is TRUE.
as_code	Should each list item appear in a 'code font' (i.e., as monospaced text)? By default this is TRUE. Using FALSE keeps all list items in the same font as the rest of the information report.
quot_str	An option for whether list items should be set in double quotes. If NULL (the default), the quotation marks are mainly associated with list items derived from character or factor values; numbers, dates, and logical values won't have quotation marks. We can explicitly use quotations (or not) with either TRUE or FALSE here.
lang	The language to use for any joining words (from the and_or option) or additional words in the generated list string. By default, NULL will use whichever lang setting is available in the parent *informant* object (this is settable in the create_informant() lang argument). If specified here as an override, the language options are English ("en"), French ("fr"), German ("de"), Italian ("it"), Spanish ("es"), Portuguese ("pt"), Turkish ("tr"), Chinese ("zh"), Russian ("ru"), Polish ("pl"), Danish ("da"), Swedish ("sv"), and Dutch ("nl").

Value

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

Function ID

3-6

See Also

Other Information Functions: info_columns_from_tbl(), info_columns(), info_section(), info_snippet(), info_tabular(), snip_highest(), snip_lowest(), snip_stats()

Examples

```
# Generate an informant object, add
# a snippet with `info_snippet()`
# and `snip_list()` (giving us a
# method to get a distinct list of
# column values for column `f`);
# define a location for the snippet
# result in `{ }` and then
# `incorporate()` the snippet into
# the info text
informant <-
  create_informant(
    tbl = ~ small_table,
```

```
    tbl_name = "small_table",
    label = "An example."
  ) %>%
  info_columns(
    columns = "f",
    `Items` = "This column contains {values_f}."
  ) %>%
  info_snippet(
    snippet_name = "values_f",
    fn = snip_list(column = "f")
  ) %>%
  incorporate()

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

`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()` 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()`'s `fn` argument.

Function ID

3-8

See Also

Other Information Functions: `info_columns_from_tbl()`, `info_columns()`, `info_section()`, `info_snippet()`, `info_tabular()`, `snip_highest()`, `snip_list()`, `snip_stats()`

Examples

```
# Generate an informant object, add
# a snippet with `info_snippet()`
# and `snip_lowest()` (giving us a
# method to get the lowest value in
# column `a`); define a location for
# the snippet result in `{ }` and
# then `incorporate()` the snippet
# into the info text
informant <-
  create_informant(
    tbl = ~ small_table,
    tbl_name = "small_table",
    label = "An example."
  ) %>%
  info_columns(
    columns = "a",
    `Lowest Value` = "Lowest value is {lowest_a}."
  ) %>%
  info_snippet(
    snippet_name = "lowest_a",
    fn = snip_lowest(column = "a")
  ) %>%
  incorporate()

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

`snip_stats` *A* `fn` *for* `info_snippet()`*: get an inline statistical summary*

Description

The `snip_stats()` function can be used as an `info_snippet()` function (i.e., provided to `fn`) to produce a five- or seven-number statistical summary. This inline summary works well within a paragraph of text and can help in describing the distribution of numerical values in a column.

For a given column, three different types of inline statistical summaries can be provided:

1. a five-number summary (`"5num"`): minimum, Q1, median, Q3, maximum
2. a seven-number summary (`"7num"`): P2, P9, Q1, median, Q3, P91, P98
3. Bowley's seven-figure summary (`"bowley"`): minimum, P10, Q1, median, Q3, P90, maximum

Usage

```
snip_stats(column, type = c("5num", "7num", "bowley"))
```

Arguments

column	The name of the column that contains the target values.
type	The type of summary. By default, the "5num" keyword is used to generate a five-number summary. Two other options provide seven-number summaries: "7num" and "bowley".

Value

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

Function ID

3-7

See Also

Other Information Functions: info_columns_from_tbl(), info_columns(), info_section(), info_snippet(), info_tabular(), snip_highest(), snip_list(), snip_lowest()

Examples

```
# Generate an informant object, add
# a snippet with `info_snippet()`
# and `snip_stats()` (giving us a
# method to get some summary stats for
# column `a`); define a location for
# the snippet result in `{ }` and
# then `incorporate()` the snippet
# into the info text
informant <-
  create_informant(
    tbl = ~ small_table,
    tbl_name = "small_table",
    label = "An example."
  ) %>%
  info_columns(
    columns = "a",
    `Stats` = "Stats (fivenum): {stats_a}."
  ) %>%
  info_snippet(
    snippet_name = "stats_a",
    fn = snip_stats(column = "a")
  ) %>%
  incorporate()

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

specially *Perform a specialized validation with a user-defined function*

Description

The `specially()` validation function allows for custom validation with a function that *you* provide. The major proviso for the provided function is that it must either return a logical vector or a table where the final column is logical. The function will operate on the table object, or, because you can do whatever you like, it could also operate on other types of objects. To do this, you can transform the input table in `preconditions` or inject an entirely different object there. During interrogation, there won't be any checks to ensure that the data is a table object.

Usage

```
specially(
  x,
  fn,
  preconditions = NULL,
  actions = NULL,
  step_id = NULL,
  label = NULL,
  brief = NULL,
  active = TRUE
)

expect_specially(object, fn, preconditions = NULL, threshold = 1)

test_specially(object, fn, 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()`.
`fn`	A function that performs the specialized validation on the data. It must either return a logical vector or a table where the last column is a logical column.
`preconditions`	An optional expression for mutating the input table before proceeding with the validation. This can either be provided as a one-sided R formula using a leading `~` (e.g., `~ . %>% dplyr::mutate(col = col + 10)` or as a function (e.g., `function(x) dplyr::mutate(x,col = col + 10)`. See the *Preconditions* section for more information.
`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 validation 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 validation function will be operating directly on data (no *agent* involvement), then any step with `active = FALSE` will simply pass the data through with no validation whatsoever. Aside from a logical vector, a one-sided R formula using a leading `~` can be used with `.` (serving as the input data table) to evaluate to a single logical value. With this approach, the **pointblank** function `has_columns()` can be used to determine whether to make a validation step active on the basis of one or more columns existing in the table (e.g., `~ . %>% has_columns(vars(d,e))`). The default for `active` 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.

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.

Preconditions

Providing expressions as `preconditions` means **pointblank** will preprocess the target table during interrogation as a preparatory step. It might happen that a particular validation requires a calculated

column, some filtering of rows, or the addition of columns via a join, etc. Especially for an *agent*-based report this can be advantageous since we can develop a large validation plan with a single target table and make minor adjustments to it, as needed, along the way. Within `specially()`, because this function is special, there won't be internal checking as to whether the `preconditions`-based output is a table.

The table mutation is totally isolated in scope to the validation step(s) where `preconditions` is used. Using **dplyr** code is suggested here since the statements can be translated to SQL if necessary (i.e., if the target table resides in a database). 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_b = col_a + 10)`). Alternatively, a function could instead be supplied (e.g., `function(x) dplyr::mutate(x,col_b = col_a + 10)`).

Actions

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).

Briefs

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.

YAML

A **pointblank** agent can be written to YAML with `yaml_write()` and the resulting YAML can be used to regenerate an agent (with `yaml_read_agent()`) or interrogate the target table (via `yaml_agent_interrogate()`). When `specially()` is represented in YAML (under the top-level `steps` key as a list member), the syntax closely follows the signature of the validation function. Here is an example of how a complex call of `specially()` as a validation step is expressed in R code and in the corresponding YAML representation.

```
# R statement
agent %>%
  specially(
    fn = function(x) { ... },
    preconditions = ~ . %>% dplyr::filter(a < 10),
    actions = action_levels(warn_at = 0.1, stop_at = 0.2),
    label = "The `specially()` step.",
    active = FALSE
  )
```

```
# YAML representation
steps:
- specially:
    fn: function(x) { ... }
    preconditions: ~. %>% dplyr::filter(a < 10)
    actions:
      warn_fraction: 0.1
      stop_fraction: 0.2
    label: The `specially()` step.
    active: false
```

In practice, both of these will often be shorter as only the expressions for validation steps are necessary. Arguments with default values won't be written to YAML when using `yaml_write()` (though it is acceptable to include them with their default when generating the YAML by other means). It is also possible to preview the transformation of an agent to YAML without any writing to disk by using the `yaml_agent_string()` function.

Function ID

2-35

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_decreasing()`, `col_vals_equal()`, `col_vals_expr()`, `col_vals_gte()`, `col_vals_gt()`, `col_vals_in_set()`, `col_vals_increasing()`, `col_vals_lte()`, `col_vals_lt()`, `col_vals_make_set()`, `col_vals_make_subset()`, `col_vals_not_between()`, `col_vals_not_equal()`, `col_vals_not_in_set()`, `col_vals_not_null()`, `col_vals_null()`, `col_vals_regex()`, `col_vals_within_spec()`, `conjointly()`, `row_count_match()`, `rows_complete()`, `rows_distinct()`, `serially()`, `tbl_match()`

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 that the target table has
# exactly three rows; this single
# validation with `specially()` has
# 1 test unit since the function
# executed on `x` (the target table)
# results in a logical vector with a
# length of 1
agent <-
  create_agent(tbl = tbl) %>%
    specially(
      fn = function(x) nrow(x) == 3
    ) %>%
    interrogate()

# Determine if this validation
# had no failing test units (there
# is 1 test unit)
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 %>%
  specially(
      fn = function(x) nrow(x) == 3
   )

# 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_specially(
  tbl,
  fn = function(x) nrow(x) == 3
)

# D: Using the test function

# With the `test_*()` form, we should
# get a single logical value returned
```

```
# to us
tbl %>%
  test_specially(
    fn = function(x) nrow(x) == 3
  )

# Variations

# We can do more complex things with
# `specially()` and its variants

tbl %>% test_specially(
  fn = function(x) {
    inherits(x, "data.frame")
  }
)

# Check that the number of rows in the
# target table is less than `small_table`
tbl %>% test_specially(
  fn = function(x) {
    nrow(x) < nrow(small_table)
  }
)

# Check that all numbers across all
# numeric column are less than `10`
tbl %>% test_specially(
  fn = function(x) {
    (x %>%
       dplyr::select(where(is.numeric)) %>%
       unlist()
    ) < 10
  }
)

# Check that all values in column
# `c` are greater than b and greater
# than `a` (in each row) and always
# less than 10; this creates a table
# with the new column `d` which is
# a logical column (that is used as
# the evaluation of test units)
tbl %>% test_specially(
  fn = function(x) {
    x %>%
      dplyr::mutate(
        d = c > b & c > a & c < 10
      )
  }
)

# Check that the `game_revenue`
```

```
# table (which is not the target
# table) has exactly 2000 rows
tbl %>% test_specially(
  fn = function(x) {
    nrow(game_revenue) == 2000
  }
)
```

`specifications` *A table containing data pertaining to various specifications*

Description

The `specifications` dataset is useful for testing the `col_vals_within_spec()`, `test_col_vals_within_spec()`, and `expect_col_vals_within_spec()` functions. For each column, holding character values for different specifications, rows 1-5 contain valid values, the 6th row is an NA value, and the final two values (rows 7 and 8) are invalid. Different specification (`spec`) keywords apply to each of columns when validating with any of the aforementioned functions.

Usage

```
specifications
```

Format

A tibble with 8 rows and 12 variables:

isbn_numbers ISBN-13 numbers; can be validated with the `"isbn"` specification.

vin_numbers VIN numbers (identifiers for motor vehicles); can be validated with the `"vin"` specification.

zip_codes Postal codes for the U.S.; can be validated with the `"postal[USA]"` specification or its `"zip"` alias.

credit_card_numbers Credit card numbers; can be validated with the `"credit_card"` specification or the `"cc"` alias.

iban_austria IBAN numbers for Austrian accounts; can be validated with the `"iban[AUT]"` specification.

swift_numbers Swift-BIC numbers; can be validated with the `"swift"` specification.

phone_numbers Phone numbers; can be validated with the `"phone"` specification.

email_addresses Email addresses; can be validated with the `"email"` specification.

urls URLs; can be validated with the `"url"` specification.

ipv4_addresses IPv4 addresses; can be validated with the `"ipv4"` specification

ipv6_addresses IPv6 addresses; can be validated with the `"ipv6"` specification

mac_addresses MAC addresses; can be validated with the `"mac"` specification

Function ID

14-3

See Also

Other Datasets: game_revenue_info, game_revenue, small_table_sqlite(), small_table

Examples

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

---

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 argument 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()` 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_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`.

Function ID

13-5

See Also

Other Utility and Helper Functions: `affix_datetime()`, `affix_date()`, `col_schema()`, `from_github()`, `has_columns()`

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 < 20:25 - 18)
```

`tbl_get` *Obtain a materialized table via a table store*

Description

The `tbl_get()` function gives us the means to materialize a table that has an entry in a table store (i.e., has a table-prep formula with a unique name). The table store that is used for this can be in the form of a `tbl_store` object (created with the `tbl_store()` function) or an on-disk YAML representation of a table store (created by using `yaml_write()` with a `tbl_store` object).

Should you want a table-prep formula from a table store to use as a value for `tbl` (in `create_agent()`, `create_informant()`, or `set_tbl()`), then have a look at the `tbl_source()` function.

Usage

```
tbl_get(tbl, store = NULL)
```

Arguments

tbl	The table to retrieve from a table store. This table could be identified by its name (e.g., tbl = "large_table") or by supplying a reference using a subset (with $) of the tbl_store object (e.g., tbl = store$large_table). If using the latter method then nothing needs to be supplied to store.
store	Either a table store object created by the tbl_store() function or a path to a table store YAML file created by yaml_write().

Value

A table object.

Function ID

1-10

See Also

Other Planning and Prep: action_levels(), create_agent(), create_informant(), db_tbl(), draft_validation(), file_tbl(), scan_data(), tbl_source(), tbl_store(), validate_rmd()

Examples

```
if (interactive()) {

# Define a `tbl_store` object by adding
# table-prep formulas in `tbl_store()`
tbls <-
  tbl_store(
    small_table_duck ~ db_tbl(
      table = small_table,
      dbname = ":memory:",
      dbtype = "duckdb"
    ),
    ~ db_tbl(
      table = "rna",
      dbname = "pfmegrnargs",
      dbtype = "postgres",
      host = "hh-pgsql-public.ebi.ac.uk",
      port = 5432,
      user = I("reader"),
      password = I("NWDMCE5xdipIjRrp")
    ),
    all_revenue ~ db_tbl(
      table = file_tbl(
        file = from_github(
          file = "all_revenue_large.rds",
          repo = "rich-iannone/intendo",
          subdir = "data-large"
        )
      ),
```

```
      dbname = ":memory:",
      dbtype = "duckdb"
    ),
    sml_table ~ pointblank::small_table
  )

# Once this object is available, you can
# check that the table of interest is
# produced to your specification
tbl_get(
  tbl = "small_table_duck",
  store = tbls
)

# An alternative method for getting the
# same table materialized is by using `$`
# to get the formula of choice from `tbls`
tbls$small_table_duck %>% tbl_get()

}
```

`tbl_match` *Does the target table match a comparison table?*

Description

The `tbl_match()` validation function, the `expect_tbl_match()` expectation function, and the `test_tbl_match()` test function all check whether the target table's composition matches that of a comparison 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, there is a single test unit that hinges on whether the two tables are the same (after any `preconditions` have been applied).

Usage

```
tbl_match(
  x,
  tbl_compare,
  preconditions = NULL,
  segments = NULL,
  actions = NULL,
  step_id = NULL,
  label = NULL,
  brief = NULL,
  active = TRUE
)
```

```
expect_tbl_match(object, tbl_compare, preconditions = NULL, threshold = 1)

test_tbl_match(object, tbl_compare, 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().
tbl_compare	A table to compare against the target table. This can either be a table object, a table-prep formula. This can be a table object such as a data frame, a tibble, a tbl_dbi object, or a tbl_spark object. Alternatively, a table-prep formula (~ <table reading code>) or a function (function() <table reading code>) can be used to lazily read in the table at interrogation time.
preconditions	An optional expression for mutating the input table before proceeding with the validation. This can either be provided as a one-sided R formula using a leading ~ (e.g., ~ . %>% dplyr::mutate(col = col + 10) or as a function (e.g., function(x) dplyr::mutate(x,col = col + 10). See the *Preconditions* section for more information.
segments	An optional expression or set of expressions (held in a list) that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on. See the *Segments* section for more details on this.
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 validation 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 validation function will be operating directly on data (no *agent* involvement), then any step with `active = FALSE` will simply pass the data through with no validation whatsoever. Aside from a logical vector, a one-sided R formula using a leading `~` can be used with `.` (serving as the input data table) to evaluate to a single logical value. With this approach, the **pointblank** function `has_columns()` can be used to determine whether to make a validation step active on the basis of one or more columns existing in the table (e.g., `~ . %>% has_columns(vars(d,e))`). The default for `active` 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.

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.

Preconditions

Providing expressions as `preconditions` means **pointblank** will preprocess the target table during interrogation as a preparatory step. It might happen that this particular validation requires some operation on the target table before the comparison takes place. Using `preconditions` can be useful at times since since we can develop a large validation plan with a single target table and make minor adjustments to it, as needed, along the way.

The table mutation is totally isolated in scope to the validation step(s) where `preconditions` is used. Using **dplyr** code is suggested here since the statements can be translated to SQL if necessary (i.e., if the target table resides in a database). 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. Alternatively, a function could instead be supplied.

Segments

By using the `segments` argument, it's possible to define a particular validation with segments (or row slices) of the target table. An optional expression or set of expressions that serve to segment the target table by column values. Each expression can be given in one of two ways: (1) as column names, or (2) as a two-sided formula where the LHS holds a column name and the RHS contains the column values to segment on.

As an example of the first type of expression that can be used, `vars(a_column)` will segment the target table in however many unique values are present in the column called `a_column`. This is great

if every unique value in a particular column (like different locations, or different dates) requires it's own repeating validation.

With a formula, we can be more selective with which column values should be used for segmentation. Using `a_column ~ c("group_1","group_2")` will attempt to obtain two segments where one is a slice of data where the value `"group_1"` exists in the column named `"a_column"`, and, the other is a slice where `"group_2"` exists in the same column. Each group of rows resolved from the formula will result in a separate validation step.

Segmentation will always occur after `preconditions` (i.e., statements that mutate the target table), if any, are applied. With this type of one-two combo, it's possible to generate labels for segmentation using an expression for `preconditions` and refer to those labels in `segments` without having to generate a separate version of the target table.

Actions

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).

Briefs

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.

YAML

A **pointblank** agent can be written to YAML with `yaml_write()` and the resulting YAML can be used to regenerate an agent (with `yaml_read_agent()`) or interrogate the target table (via `yaml_agent_interrogate()`). When `tbl_match()` is represented in YAML (under the top-level `steps` key as a list member), the syntax closely follows the signature of the validation function. Here is an example of how a complex call of `tbl_match()` as a validation step is expressed in R code and in the corresponding YAML representation.

```
# R statement
agent %>%
  tbl_match(
    tbl_compare = ~ file_tbl(
      file = from_github(
        file = "all_revenue_large.rds",
        repo = "rich-iannone/intendo",
        subdir = "data-large"
        )
      ),
    preconditions = ~ . %>% dplyr::filter(a < 10),
```

```
    segments = b ~ c("group_1", "group_2"),
    actions = action_levels(warn_at = 0.1, stop_at = 0.2),
    label = "The `tbl_match()` step.",
    active = FALSE
  )

# YAML representation
steps:
- tbl_match:
    tbl_compare: ~ file_tbl(
      file = from_github(
        file = "all_revenue_large.rds",
        repo = "rich-iannone/intendo",
        subdir = "data-large"
        )
      )
    preconditions: ~. %>% dplyr::filter(a < 10)
    segments: b ~ c("group_1", "group_2")
    actions:
      warn_fraction: 0.1
      stop_fraction: 0.2
    label: The `tbl_match()` step.
    active: false
```

In practice, both of these will often be shorter. Arguments with default values won't be written to YAML when using `yaml_write()` (though it is acceptable to include them with their default when generating the YAML by other means). It is also possible to preview the transformation of an agent to YAML without any writing to disk by using the `yaml_agent_string()` function.

Function ID

2-32

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_decreasing()`, `col_vals_equal()`, `col_vals_expr()`, `col_vals_gte()`, `col_vals_gt()`, `col_vals_in_set()`, `col_vals_increasing()`, `col_vals_lte()`, `col_vals_lt()`, `col_vals_make_set()`, `col_vals_make_subset()`, `col_vals_not_between()`, `col_vals_not_equal()`, `col_vals_not_in_set()`, `col_vals_not_null()`, `col_vals_null()`, `col_vals_regex()`, `col_vals_within_spec()`, `conjointly()`, `row_count_match()`, `rows_complete()`, `rows_distinct()`, `serially()`, `specially()`

Examples

```
# Create a simple table with three
# columns and four rows of values
tbl <-
  dplyr::tibble(
    a = c(5, 7, 6, 5),
```

```
    b = c(7, 1, 0, 0),
    c = c(1, 1, 1, 3)
  )

tbl

# Create a second table which is
# the same as `tbl`
tbl_2 <-
  dplyr::tibble(
    a = c(5, 7, 6, 5),
    b = c(7, 1, 0, 0),
    c = c(1, 1, 1, 3)
  )

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

# Validate that the target table
# (`tbl`) and the comparison table
# (`tbl_2`) are equivalent in terms
# of content
agent <-
  create_agent(tbl = tbl) %>%
  tbl_match(tbl_compare = tbl_2) %>%
  interrogate()

# Determine if this validation passed
# by using `all_passed()`
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 %>%
  tbl_match(tbl_compare = tbl_2)

# 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_tbl_match(
  tbl, tbl_compare = tbl_2
)

# D: Using the test function

# With the `test_*()` form, we should
# get a single logical value returned
# to us
tbl %>%
  tbl_match(
    tbl_compare = tbl_2
  )
```

`tbl_source` *Obtain a table-prep formula from a table store*

Description

The `tbl_source()` function provides a convenient means to access a table-prep formula from either a `tbl_store` object or a table store YAML file (which can be created with the `yaml_write()` function). A call to `tbl_source()` is most useful as an input to the `tbl` argument of `create_agent()`, `create_informant()`, or `set_tbl()`.

Should you need to obtain the table itself (that is generated via the table-prep formula), then the `tbl_get()` function should be used for that.

Usage

```
tbl_source(tbl, store = NULL)
```

Arguments

`tbl`	The table name associated with a table-prep formula. This is part of the table `store`. This table could be identified by its name (e.g., `tbl = "large_table"`) or by supplying a reference using a subset (with `$`) of the `tbl_store` object (e.g., `tbl = store$large_table`). If using the latter method then nothing needs to be supplied to `store`.
`store`	Either a table store object created by the `tbl_store()` function or a path to a table store YAML file created by `yaml_write()`.

Value

A table-prep formula.

Function ID

1-9

See Also

Other Planning and Prep: action_levels(), create_agent(), create_informant(), db_tbl(), draft_validation(), file_tbl(), scan_data(), tbl_get(), tbl_store(), validate_rmd()

Examples

```
if (interactive()) {

# Let's create a `tbl_store` object by
# giving two table-prep formulas to
# `tbl_store()`
tbls <-
  tbl_store(
    small_table_duck ~ db_tbl(
      table = small_table,
      dbname = ":memory:",
      dbtype = "duckdb"
    ),
    sml_table ~ pointblank::small_table
  )

# We can pass a table-prep formula
# to `create_agent()` and interrogate
# the table shortly thereafter
agent <-
  create_agent(
    tbl = ~ tbl_source("sml_table", tbls),
    label = "An example that uses a table store.",
    actions = action_levels(warn_at = 0.10)
  ) %>%
  col_exists(vars(date, date_time)) %>%
  interrogate()

# Both the `tbl_store` object and the
# `agent` can be transformed to YAML with
# the `yaml_write()` function

# This writes the `tbl_store.yml` file
# by default (but a different name
# could be used)
yaml_write(tbls)

# Let's modify the agent's target
# to point to the table labeled as
# `"sml_table"` in the YAML
# representation of the `tbl_store`
agent <-
  agent %>%
  set_tbl(
    ~ tbl_source(
        tbl = "sml_table",
        store = "tbl_store.yml"
```

```
      )
  )

# Then we can write agent to a YAML
# file (writes to `agent-sml_table.yml`
# by default)
yaml_write(agent)

# Now that both are in this on-disk format
# an interrogation can be done by accessing
# the agent YAML
agent <-
  yaml_agent_interrogate(
    filename = "agent-sml_table.yml"
  )

}
```

`tbl_store` *Define a store of tables with table-prep formulas: a table store*

Description

It can be useful to set up all the data sources you need and just draw from them when necessary. This upfront configuration with `tbl_store()` lets us define the methods for obtaining tabular data from mixed sources (e.g., database tables, tables generated from flat files, etc.) and provide names for these data preparation procedures. Then we have a convenient way to access the materialized tables with `tbl_get()`, or, the table-prep formulas with `tbl_source()`. Table-prep formulas can be as simple as getting a table from a location, or, it can involve as much mutation as is necessary (imagine procuring several mutated variations of the same source table, generating a table from multiple sources, or pre-filtering a database table according to the system time). Another nice aspect of organizing table-prep formulas in a single object is supplying it to the `tbl` argument of `create_agent()` or `create_informant()` via `$` notation (e.g, create_agent(tbl = <tbl_store>$<name>)) or with `tbl_source()` (e.g., create_agent(tbl = ~ tbl_source("<name>", <tbl_store>))).

Usage

```
tbl_store(..., .list = list2(...))
```

Arguments

`...`	Expressions that contain table-prep formulas and table names for data retrieval. Two-sided formulas (e.g, <LHS> ~ <RHS>) are to be used, where the left-hand side is a given name and the right-hand is the portion that is is used to obtain the table.
`.list`	Allows for the use of a list as an input alternative to `...`.

Value

A `tbl_store` object that contains table-prep formulas.

YAML

A **pointblank** table store can be written to YAML with `yaml_write()` and the resulting YAML can be used in several ways. The ideal scenario is to have pointblank agents and informants also in YAML form. This way the agent and informant can refer to the table store YAML (via `tbl_source()`), and, the processing of both agents and informants can be performed with `yaml_agent_interrogate()` and `yaml_informant_incorporate()`. With the following R code, a table store with two table-prep formulas is generated and written to YAML (if no filename is given then the YAML is written to `"tbl_store.yml"`).

```
# R statement for generating the "tbl_store.yml" file
tbl_store(
  tbl_duckdb ~ db_tbl(small_table, dbname = ":memory:", dbtype = "duckdb"),
  sml_table_high ~ small_table %>% dplyr::filter(f == "high")
) %>%
  yaml_write()

# YAML representation ("tbl_store.yml")
tbls:
  tbl_duckdb: ~ db_tbl(small_table, dbname = ":memory:", dbtype = "duckdb")
  sml_table_high: ~ small_table %>% dplyr::filter(f == "high")
```

This is useful when you want to get fresh pulls of prepared data from a source materialized in an R session (with the `tbl_get()` function. For example, the `sml_table_high` table can be obtained by using `tbl_get("sml_table_high","tbl_store.yml")`. To get an agent to check this prepared data periodically, then the following example with `tbl_source()` will be useful:

```
# Generate agent that checks `sml_table_high`, write it to YAML
create_agent(
  tbl = ~ tbl_source("sml_table_high", "tbl_store.yml"),
  label = "An example that uses a table store.",
  actions = action_levels(warn_at = 0.10)
) %>%
  col_exists(vars(date, date_time)) %>%
  write_yaml()

# YAML representation ("agent-sml_table_high.yml")
tbl: ~ tbl_source("sml_table_high", "tbl_store.yml")
tbl_name: sml_table_high
label: An example that uses a table store.
actions:
  warn_fraction: 0.1
locale: en
steps:
  - col_exists:
    columns: vars(date, date_time)
```

Now, whenever the sml_table_high table needs to be validated, it can be done with yaml_agent_interrogate() (e.g., yaml_agent_interrogate("agent-sml_table_high.yml")).

Function ID

1-8

See Also

Other Planning and Prep: action_levels(), create_agent(), create_informant(), db_tbl(), draft_validation(), file_tbl(), scan_data(), tbl_get(), tbl_source(), validate_rmd()

Examples

```
if (interactive()) {

# Define a `tbl_store` object by adding
# table-prep formulas inside the
# `tbl_store()` call
tbls <-
  tbl_store(
    small_table_duck ~ db_tbl(
      table = small_table,
      dbname = ":memory:",
      dbtype = "duckdb"
    ),
    ~ db_tbl(
      table = "rna",
      dbname = "pfmegrnargs",
      dbtype = "postgres",
      host = "hh-pgsql-public.ebi.ac.uk",
      port = 5432,
      user = I("reader"),
      password = I("NWDMCE5xdipIjRrp")
    ),
    all_revenue ~ db_tbl(
      table = file_tbl(
        file = from_github(
          file = "all_revenue_large.rds",
          repo = "rich-iannone/intendo",
          subdir = "data-large"
        )
      ),
      dbname = ":memory:",
      dbtype = "duckdb"
    ),
    sml_table ~ pointblank::small_table
  )

# Once this object is available, you
# can check that the table of interest
# is produced to your specification with
```

```
# the `tbl_get()` function
tbl_get(
  tbl = "small_table_duck",
  store = tbls
)

# Another simpler way to get the same
# table materialized is by using `$` to
# get the entry of choice for `tbl_get()`
tbls$small_table_duck %>% tbl_get()

# Creating an agent is easy when all
# table-prep formulas are encapsulated
# in a `tbl_store` object; use `$`
# notation to pass the appropriate
# procedure for reading a table to the
# `tbl` argument
agent_1 <-
  create_agent(
    tbl = tbls$small_table_duck
  )

# There are other ways to use the
# table store to assign a target table
# to an agent, like using the
# `tbl_source()` function
agent_2 <-
  create_agent(
    tbl = ~ tbl_source(
      tbl = "small_table_duck",
      store = tbls
      )
  )

# The table store can be moved to
# YAML with `yaml_write` and the
# `tbl_source()` call could then
# refer to that on-disk table store;
# let's do that YAML conversion
yaml_write(tbls)

# The above writes the `tbl_store.yml`
# file (by not providing a `filename`
# this default filename is chosen);
# next, modify the `tbl_source()`
# so that `store` refer to the YAML
# file
agent_3 <-
  create_agent(
    tbl = ~ tbl_source(
      tbl = "small_table_duck",
      store = "tbl_store.yml"
    )
```

```
  )
}
```

`tt_string_info` *Table Transformer: obtain a summary table for string columns*

Description

With any table object, you can produce a summary table that is scoped to string-based columns. The output summary table will have a leading column called `".param."` with labels for each of the three rows, each corresponding to the following pieces of information pertaining to string length:

1. Mean String Length (`"length_mean"`)
2. Minimum String Length (`"length_min"`)
3. Maximum String Length (`"length_max"`)

Only string data from the input table will generate columns in the output table. Column names from the input will be used in the output, preserving order as well.

Usage

```
tt_string_info(tbl)
```

Arguments

`tbl`	A table object to be used as input for the transformation. This can be a data frame, a tibble, a `tbl_dbi` object, or a `tbl_spark` object.

Value

A `tibble` object.

Function ID

12-2

See Also

Other Table Transformers: `get_tt_param()`, `tt_summary_stats()`, `tt_tbl_colnames()`, `tt_tbl_dims()`, `tt_time_shift()`, `tt_time_slice()`

Examples

```
# Get string information for the
# string-based columns in the
# `game_revenue` dataset
tt_string_info(game_revenue)

# Ensure that `player_id` and
# `session_id` values always have
# the same number of characters
# throughout the table
tt_string_info(game_revenue) %>%
  col_vals_equal(
    columns = vars(player_id),
    value = 15
  ) %>%
  col_vals_equal(
    columns = vars(session_id),
    value = 24
  )

# Check that the maximum string
# length in column `f` of the
# `small_table` dataset is no
# greater than `4`
tt_string_info(small_table) %>%
  test_col_vals_lte(
    columns = vars(f),
    value = 4
  )
```

`tt_summary_stats` *Table Transformer: obtain a summary stats table for numeric columns*

Description

With any table object, you can produce a summary table that is scoped to the numeric column values. The output summary table will have a leading column called `".param."` with labels for each of the nine rows, each corresponding to the following summary statistics:

1. Minimum (`"min"`)
2. 5th Percentile (`"p05"`)
3. 1st Quartile (`"q_1"`)
4. Median (`"med"`)
5. 3rd Quartile (`"q_3"`)
6. 95th Percentile (`"p95"`)
7. Maximum (`"max"`)

8. Interquartile Range (`"iqr"`)
9. Range (`"range"`)

Only numerical data from the input table will generate columns in the output table. Column names from the input will be used in the output, preserving order as well.

Usage

```
tt_summary_stats(tbl)
```

Arguments

`tbl`	A table object to be used as input for the transformation. This can be a data frame, a tibble, a `tbl_dbi` object, or a `tbl_spark` object.

Value

A `tibble` object.

Function ID

12-1

See Also

Other Table Transformers: `get_tt_param()`, `tt_string_info()`, `tt_tbl_colnames()`, `tt_tbl_dims()`, `tt_time_shift()`, `tt_time_slice()`

Examples

```
# Get summary statistics for the
# `game_revenue` dataset that is
# included in the package
tt_summary_stats(game_revenue)

# Ensure that the maximum revenue
# for individual purchases in the
# `game_revenue` table is less than
# $150
tt_summary_stats(game_revenue) %>%
  col_vals_lt(
    columns = vars(item_revenue),
    value = 150,
    segments = .param. ~ "max"
  )

# For in-app purchases in the
# `game_revenue` table, check that
# median revenue is somewhere
# between $8 and $12
game_revenue %>%
  dplyr::filter(item_type == "iap") %>%
```

```
  tt_summary_stats() %>%
  col_vals_between(
    columns = vars(item_revenue),
    left = 8, right = 12,
    segments = .param. ~ "med"
  )

# While performing validations of the
# `game_revenue` table with an agent
# we can include the same revenue
# check by using `tt_summary_stats()`
# in the `preconditions` argument (this
# will transform the target table for
# the validation step); we also need
# to get just a segment of that table
# (the row with the median values)
agent <-
  create_agent(
    tbl = game_revenue,
    tbl_name = "game_revenue",
    label = "An example.",
    actions = action_levels(
      warn_at = 0.10,
      stop_at = 0.25,
      notify_at = 0.35
    )
  ) %>%
  rows_complete() %>%
  rows_distinct() %>%
  col_vals_between(
    columns = vars(item_revenue),
    left = 8, right = 12,
    preconditions = ~ . %>%
      dplyr::filter(item_type == "iap") %>%
      tt_summary_stats(),
    segments = .param. ~ "med"
  ) %>%
  interrogate()

# This should all pass but let's check:
all_passed(agent)
```

`tt_tbl_colnames` *Table Transformer: get a table's column names*

Description

With any table object, you can produce a summary table that contains table's column names. The output summary table will have two columns and as many rows as there are columns in the input

table. The first column is the ".param." column, which is an integer-based column containing the indices of the columns from the input table. The second column, "value", contains the column names from the input table.

Usage

```
tt_tbl_colnames(tbl)
```

Arguments

tbl	A table object to be used as input for the transformation. This can be a data frame, a tibble, a tbl_dbi object, or a tbl_spark object.

Value

A tibble object.

Function ID

12-4

See Also

Other Table Transformers: get_tt_param(), tt_string_info(), tt_summary_stats(), tt_tbl_dims(), tt_time_shift(), tt_time_slice()

Examples

```
# Get the column names of the
# `game_revenue` dataset that's
# included in the package
tt_tbl_colnames(game_revenue)

# This output table is useful when
# you want to validate the
# column names of the table; here,
# we check that `game_revenue` has
# certain column names present
tt_tbl_colnames(game_revenue) %>%
  test_col_vals_make_subset(
    columns = vars(value),
    set = c("acquisition", "country")
  )

# We can check to see whether the
# column names in the `specifications`
# table are all less than 15
# characters in length
specifications %>%
  tt_tbl_colnames() %>%
  tt_string_info() %>%
  test_col_vals_lt(
```

```
  columns = vars(value),
  value = 15
)
```

`tt_tbl_dims` *Table Transformer: get the dimensions of a table*

Description

With any table object, you can produce a summary table that contains nothing more than the table's dimensions: the number of rows and the number of columns. The output summary table will have two columns and two rows. The first is the `".param."` column with the labels `"rows"` and `"columns"`; the second column, `"value"`, contains the row and column counts.

Usage

```
tt_tbl_dims(tbl)
```

Arguments

`tbl`	A table object to be used as input for the transformation. This can be a data frame, a tibble, a `tbl_dbi` object, or a `tbl_spark` object.

Value

A `tibble` object.

Function ID

12-3

See Also

Other Table Transformers: `get_tt_param()`, `tt_string_info()`, `tt_summary_stats()`, `tt_tbl_colnames()`, `tt_time_shift()`, `tt_time_slice()`

Examples

```
# Get the dimensions of the
# `game_revenue` dataset that's
# included in the package
tt_tbl_dims(game_revenue)

# This output table is useful when
# you want to validate the
# dimensions of the table; here,
# we check that `game_revenue` has
# at least 1500 rows
```

```
tt_tbl_dims(game_revenue) %>%
  dplyr::filter(.param. == "rows") %>%
  test_col_vals_gt(
    columns = vars(value),
    value = 1500
  )

# We can check `small_table` for
# an exact number of columns (`8`)
tt_tbl_dims(small_table) %>%
  dplyr::filter(.param. == "columns") %>%
  test_col_vals_equal(
    columns = vars(value),
    value = 8
  )
```

`tt_time_shift` *Table Transformer: shift the times of a table*

Description

With any table object containing date or date-time columns, these values can be precisely shifted with `tt_time_shift()` and specification of the time shift. We can either provide a string with the time shift components and the shift direction (like `"-4y 10d"`) or a `difftime` object (which can be created via **lubridate** expressions or by using the `base::difftime()` function).

Usage

```
tt_time_shift(tbl, time_shift = "0y 0m 0d 0H 0M 0S")
```

Arguments

`tbl`	A table object to be used as input for the transformation. This can be a data frame, a tibble, a `tbl_dbi` object, or a `tbl_spark` object.
`time_shift`	Either a character-based representation that specifies the time difference by which all time values in time-based columns will be shifted, or, a `difftime` object. The character string is constructed in the format `"0y 0m 0d 0H 0M 0S"` and individual time components can be omitted (i.e., `"1y 5d"` is a valid specification of shifting time values ahead one year and five days). Adding a `"-"` at the beginning of the string (e.g., `"-2y"`) will shift time values back.

Details

The `time_shift` specification cannot have a higher time granularity than the least granular time column in the input table. Put in simpler terms, if there are any date-based based columns (or just a single date-based column) then the time shifting can only be in terms of years, months, and days. Using a `time_shift` specification of `"20d 6H"` in the presence of any dates will result in a truncation to `"20d"`. Similarly, a `difftime` object will be altered in the same circumstances, however, the object will resolved to an exact number of days through rounding.

Value

A data frame, a tibble, a `tbl_dbi` object, or a `tbl_spark` object depending on what was provided as `tbl`.

Function ID

12-5

See Also

Other Table Transformers: `get_tt_param()`, `tt_string_info()`, `tt_summary_stats()`, `tt_tbl_colnames()`, `tt_tbl_dims()`, `tt_time_slice()`

Examples

```
# With the `game_revenue` dataset,
# which has entries in the first
# 21 days of 2015, move all of the
# date and date-time values to the
# beginning of 2021
tt_time_shift(
  tbl = game_revenue,
  time_shift = "6y"
)

# Keeping only the `date_time` and
# `a`-`f` columns of `small_table`,
# shift the times back 2 days and
# 12 hours
small_table %>%
  dplyr::select(-date) %>%
  tt_time_shift("-2d 12H")
```

`tt_time_slice` *Table Transformer: slice a table with a slice point on a time column*

Description

With any table object containing date, date-time columns, or a mixture thereof, any one of those columns can be used to effectively slice the data table in two with a `slice_point`: and you get to choose which of those slices you want to keep. The slice point can be defined in several ways. One method involves using a decimal value between `0` and `1`, which defines the slice point as the time instant somewhere between the earliest time value (at `0`) and the latest time value (at `1`). Another way of defining the slice point is by supplying a time value, and the following input types are accepted: (1) an ISO 8601 formatted time string (as a date or a date-time), (2) a `POSIXct` time, or (3) a `Date` object.

Usage

```
tt_time_slice(
  tbl,
  time_column = NULL,
  slice_point = 0,
  keep = c("left", "right"),
  arrange = FALSE
)
```

Arguments

tbl	A table object to be used as input for the transformation. This can be a data frame, a tibble, a tbl_dbi object, or a tbl_spark object.
time_column	The time-based column that will be used as a basis for the slicing. If no time column is provided then the first one found will be used.
slice_point	The location on the time_column where the slicing will occur. This can either be a decimal value from 0 to 1, an ISO 8601 formatted time string (as a date or a date-time), a POSIXct time, or a Date object.
keep	Which slice should be kept? The "left" side (the default) contains data rows that are earlier than the slice_point and the "right" side will have rows that are later.
arrange	Should the slice be arranged by the time_column? This may be useful if the input tbl isn't ordered by the time_column. By default, this is FALSE and the original ordering is retained.

Details

There is the option to arrange the table by the date or date-time values in the time_column. This ordering is always done in an ascending manner. Any NA/NULL values in the time_column will result in the corresponding rows can being removed (no matter which slice is retained).

Value

A data frame, a tibble, a tbl_dbi object, or a tbl_spark object depending on what was provided as tbl.

Function ID

12-6

See Also

Other Table Transformers: get_tt_param(), tt_string_info(), tt_summary_stats(), tt_tbl_colnames(), tt_tbl_dims(), tt_time_shift()

Examples

```
# With the `game_revenue` dataset,
# which has entries in the first
# 21 days of 2015, elect to get all
# of the records where the `time`
# values are strictly for the first
# 15 days of 2015
tt_time_slice(
  tbl = game_revenue,
  time_column = "time",
  slice_point = "2015-01-16"
)

# Omit the first 25% of records
# from `small_table` on the basis
# of a timeline that begins at
# `2016-01-04 11:00:00` and
# ends at `2016-01-30 11:23:00`
small_table %>%
  tt_time_slice(
    slice_point = 0.25,
    keep = "right"
  )
```

---

validate_rmd *Perform* **pointblank** *validation testing within R Markdown documents*

Description

The `validate_rmd()` function sets up a framework for validation testing within specialized validation code chunks inside an R Markdown document. To enable this functionality, `validate_rmd()` should be called early within an R Markdown document code chunk (preferably in the `setup` chunk) to signal that validation should occur within specific code chunks. The validation code chunks require the `validate = TRUE` option to be 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 a centered status button either indicates success or the number of overall failures. Clicking the button reveals the otherwise hidden validation statements and their error messages (if any).

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(), create_agent(), create_informant(), db_tbl(), draft_validation(), file_tbl(), scan_data(), tbl_get(), tbl_source(), tbl_store()

---

write_testthat_file *Transform a* **pointblank** *agent to a* **testthat** *test file*

Description

With a **pointblank** *agent*, we can write a **testthat** test file and opt to place it in the testthat/tests if it is available in the project path (we can specify an alternate path as well). This works by transforming the validation steps to a series of expect_*() calls inside individual testthat::test_that() statements.

A major requirement for using write_testthat_file() on an agent is the presence of an expression that can retrieve the target table. Typically, we might supply a table-prep formula, which is a formula that can be invoked to obtain the target table (e.g., tbl = ~ pointblank::small_table). This user-supplied statement will be used by write_testthat_file() to generate a table-loading statement at the top of the new **testthat** test file so that the target table is available for each of the testthat::test_that() statements that follow. If an *agent* was not created using a table-prep formula set for the tbl, it can be modified via the set_tbl() function.

Thresholds will be obtained from those applied for the stop state. This can be set up for a **pointblank** *agent* by passing an action_levels object to the actions argument of create_agent() or the same argument of any included validation function. If stop thresholds are not available, then a threshold value of 1 will be used for each generated expect_*() statement in the resulting **testthat** test file.

There is no requirement that the **agent** first undergo interrogation with interrogate(). However, it may be useful as a dry run to interactively perform an interrogation on the target data before generating the **testthat** test file.

Usage

```
write_testthat_file(
  agent,
  name = NULL,
  path = NULL,
  overwrite = FALSE,
```

```
  skips = NULL,
  quiet = FALSE
)
```

Arguments

agent	An agent object of class ptblank_agent.
name	An optional name for for the **testthat** test file. This should be a name without extension and without the leading "test-" text. If nothing is supplied, the name will be derived from the tbl_name in the agent. If that's not present, a generic name will be used.
path	A path can be specified here if there shouldn't be an attempt to place the file in testthat/tests.
overwrite	Should a **testthat** file of the same name be overwritten? By default, this is FALSE.
skips	This is an optional vector of test-skipping keywords modeled after the **testthat** skip_on_*() functions. The following keywords can be used to include skip_on_*() statements: "cran" (testthat::skip_on_cran()), "travis" (testthat::skip_on_travi "appveyor" (testthat::skip_on_appveyor()), "ci" (testthat::skip_on_ci()), "covr" (testthat::skip_on_covr()), "bioc" (testthat::skip_on_bioc()). There are keywords for skipping tests on certain operating systems and all of them will insert a specific testthat::skip_on_os() call. These are "windows" (skip_on_os("windows")), "mac" (skip_on_os("mac")), "linux" (skip_on_os("linux")), and "solaris" (skip_on_os("solaris")). These calls will be placed at the top of the generated **testthat** test file.
quiet	Should the function *not* inform when the file is written? By default this is FALSE.

Details

Tests for inactive validation steps will be skipped with a clear message indicating that the reason for skipping was due to the test not being active. Any inactive validation steps can be forced into an active state by using the activate_steps() on an *agent* (the opposite is possible with the deactivate_steps() function).

The **testthat** package comes with a series of skip_on_*() functions which conveniently cause the test file to be skipped entirely if certain conditions are met. We can quickly set any number of these at the top of the **testthat** test file by supplying keywords as a vector to the skips option of write_testthat_file(). For instance, setting skips = c("cran", "windows) will add the **testthat** skip_on_cran() and skip_on_os("windows") statements, meaning that the generated test file won't run on a CRAN system or if the system OS is Windows.

Here is an example of **testthat** test file output:

```
test-small_table.R

# Generated by pointblank

tbl <- small_table

test_that("column `date_time` exists", {
```

```
  expect_col_exists(
    tbl,
    columns = vars(date_time),
    threshold = 1
  )
})

test_that("values in `c` should be <= `5`", {

  expect_col_vals_lte(
    tbl,
    columns = vars(c),
    value = 5,
    threshold = 0.25
  )
})
```

This was generated by the following set of statements:

```
library(pointblank)

agent <-
  create_agent(
    tbl = ~ small_table,
    actions = action_levels(stop_at = 0.25)
  ) %>%
  col_exists(vars(date_time)) %>%
  col_vals_lte(vars(c), value = 5)

write_testthat_file(
  agent = agent,
  name = "small_table",
  path = "."
)
```

Value

Invisibly returns TRUE if the **testthat** file has been written.

Function ID

8-5

See Also

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

Examples

```
if (interactive()) {

# Creating an `action_levels` object is a
# common workflow step when creating a
# pointblank agent; we designate 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
  )

# A pointblank `agent` object is now
# created and the `al` object is provided
# to the agent; the static thresholds
# provided by `al` make reports a bit
# more useful after interrogation
agent <-
  create_agent(
    tbl = ~ small_table,
    label = "An example.",
    actions = al
  ) %>%
  col_exists(vars(date, date_time)) %>%
  col_vals_regex(
    vars(b),
    regex = "[0-9]-[a-z]{3}-[0-9]{3}"
  ) %>%
  col_vals_gt(vars(d), value = 100) %>%
  col_vals_lte(vars(c), value = 5) %>%
  interrogate()

# This agent and all of the checks can
# be transformed into a testthat file
# with `write_testthat_file()`; the `stop`
# thresholds will be ported over
write_testthat_file(
  agent = agent,
  name = "small_table",
  path = "."
)

# The above code will generate a file with
# the name `test-small_table.R`; the path
# was specified with `"."` but, by default,
# the function will place the file in the
# `tests/testthat` folder if it's available

# An agent on disk as a YAML file can be
```

```
# made into a testthat file; the
# 'agent-small_table.yml' file is
# available in the package through
# `system.file()`
yml_file <-
  system.file(
    "yaml", "agent-small_table.yml",
    package = "pointblank"
  )

# Writing the testthat file into the
# working directory is much the same
# as before but we're reading the
# agent from disk this time
write_testthat_file(
  agent = yaml_read_agent(yml_file),
  name = "from_agent_yaml",
  path = "."
)

}
```

`x_read_disk` *Read an* agent*,* informant*,* multiagent*, or table scan from disk*

Description

An *agent*, *informant*, *multiagent*, or table scan that has been written to disk (with `x_write_disk()`) can be read back into memory with the `x_read_disk()` function. For an *agent* or an *informant* object that 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(filename, path = NULL, quiet = FALSE)
```

Arguments

`filename`	The name of a file that was previously written by `x_write_disk()`.
`path`	An optional path to the file (combined with `filename`).
`quiet`	Should the function *not* inform when the file is read? By default this is `FALSE`.

Details

Should a written-to-disk *agent* or *informant* possess a table-prep formula or a specific in-memory tablewe 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.

Value

Either a `ptblank_agent`, `ptblank_informant`, or a `ptblank_tbl_scan` object.

Function ID

9-2

See Also

Other Object Ops: `activate_steps()`, `deactivate_steps()`, `export_report()`, `remove_steps()`, `set_tbl()`, `x_write_disk()`

Examples

```
if (interactive()) {

# A: Reading an agent from disk

# The process of developing an agent
# and writing it to disk with the
# `x_write_disk()` function is explained
# in that function's documentation;
# but suppose we have such a written file
# that's named "agent-small_table.rds",
# we could read that to a new agent
# object with `x_read_disk()`
agent <-
  x_read_disk("agent-small_table.rds")

# B: Reading an informant from disk

# If there is an informant written
# to disk via `x_write_disk()` and it's
# named "informant-small_table.rds",
# we could read that to a new informant
# object with `x_read_disk()`
informant <-
  x_read_disk("informant-small_table.rds")

# C: Reading a multiagent from disk

# The process of creating a multiagent
```

```
# and writing it to disk with the
# `x_write_disk()` function is shown
# in that function's documentation;
# but should we have such a written file
# called "multiagent-small_table.rds",
# we could read that to a new multiagent
# object with `x_read_disk()`
multiagent <-
  x_read_disk("multiagent-small_table.rds")

# D: Reading a table scan from disk

# If there is a table scan written
# to disk via `x_write_disk()` and it's
# named "tbl_scan-storms.rds", we could
# read it back into R with `x_read_disk()`
tbl_scan <-
  x_read_disk("tbl_scan-storms.rds")

}
```

`x_write_disk` *Write an* agent*,* informant*,* multiagent*, or table scan to disk*

Description

Writing an *agent*, *informant*, *multiagent*, or even a table scan to disk with `x_write_disk()` can be useful for keeping data validation 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 (not applicable to any table scan since they never hold a table object). 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,
  quiet = FALSE
)
```

Arguments

`x`	An *agent* object of class `ptblank_agent`, an *informant* of class `ptblank_informant`, or an table scan of class `ptblank_tbl_scan`.

filename	The filename to create on disk for the agent, informant, or table scan.
path	An optional path to which the file should be saved (this is automatically combined 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 table, ...)). 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. Only applies to *agent* objects. By default, this is FALSE (i.e., extract data is removed).
quiet	Should the function *not* inform when the file is written? By default this is FALSE.

Details

It is recommended to set up a table-prep formula so that the *agent* and *informant* can access refreshed data after being read from disk through x_read_disk(). This can be done initially with the tbl argument of create_agent()/create_informant() by passing in a table-prep formula or a function that can obtain the target table when invoked. Alternatively, we can use the set_tbl() with a similarly crafted tbl expression to ensure that an *agent* or *informant* can retrieve a table at a later time.

Value

Invisibly returns TRUE if the file has been written.

Function ID

9-1

See Also

Other Object Ops: activate_steps(), deactivate_steps(), export_report(), remove_steps(), set_tbl(), x_read_disk()

Examples

```
if (interactive()) {

# A: Writing an `agent` to disk

# Let's go through the process of (1)
# developing an agent with a validation
# plan (to be used for the data quality
# analysis of the `small_table` dataset),
# (2) interrogating the agent with the
# `interrogate()` function, and (3) writing
# the agent and all its intel to a file

# Creating an `action_levels` object is a
# common workflow step when creating a
```

```
# pointblank agent; we designate 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 `tbl`
agent <-
  create_agent(
    tbl = ~ small_table,
    tbl_name = "small_table",
    label = "`x_write_disk()`",
    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 `interrogate()`
agent <-
  agent %>%
  col_exists(vars(date, date_time)) %>%
  col_vals_regex(
    vars(b), regex = "[0-9]-[a-z]{3}-[0-9]{3}"
  ) %>%
  rows_distinct() %>%
  col_vals_gt(vars(d), value = 100) %>%
  col_vals_lte(vars(c), value = 5) %>%
  interrogate()

# The `agent` can be written to a file with
# the `x_write_disk()` function
x_write_disk(
  agent,
  filename = "agent-small_table.rds"
)

# We can read the file back as an agent
# with the `x_read_disk()` function and
# we'll get all of the intel along with the
# restored agent

# If you're consistently writing agent
# reports when periodically checking data,
# we could make use of the `affix_date()`
```

```
# or `affix_datetime()` depending on the
# granularity you need; here's an example
# that writes the file with the format:
# 'agent-small_table-YYYY-mm-dd_HH-MM-SS.rds'
x_write_disk(
  agent,
  filename = affix_datetime(
    "agent-small_table.rds"
  )
)

# B: Writing an `informant` to disk

# Let's go through the process of (1)
# creating an informant object that
# minimally describes the `small_table`
# dataset, (2) ensuring that data is
# captured from the target table using
# the `incorporate()` function, and (3)
# writing the informant to a file

# Create a pointblank `informant`
# object with `create_informant()`
# and the `small_table` dataset; use
# `incorporate()` so that info snippets
# are integrated into the text
informant <-
  create_informant(
    tbl = ~ small_table,
    tbl_name = "small_table",
    label = "`x_write_disk()`"
  ) %>%
  info_snippet(
    snippet_name = "high_a",
    fn = snip_highest(column = "a")
  ) %>%
  info_snippet(
    snippet_name = "low_a",
    fn = snip_lowest(column = "a")
  ) %>%
  info_columns(
    columns = vars(a),
    info = "From {low_a} to {high_a}."
  ) %>%
  info_columns(
    columns = starts_with("date"),
    info = "Time-based values."
  ) %>%
  info_columns(
    columns = "date",
    info = "The date part of `date_time`."
  ) %>%
  incorporate()
```

```
# The `informant` can be written to a
# file with `x_write_disk()`; let's do
# this with `affix_date()` so that the
# filename has a datestamp
x_write_disk(
  informant,
  filename = affix_date(
    "informant-small_table.rds"
  )
)

# We can read the file back into a
# new informant object (in the same
# state as when it was saved) by using
# `x_read_disk()`

# C: Writing a multiagent to disk

# Let's create one more pointblank
# agent object, provide it with some
# validation steps, and `interrogate()`
agent_b <-
  create_agent(
    tbl = ~ small_table,
    tbl_name = "small_table",
    label = "`x_write_disk()`",
    actions = al
  ) %>%
  col_vals_gt(
    vars(b), vars(g), na_pass = TRUE,
    label = "b > g"
  ) %>%
  col_is_character(
    vars(b, f),
    label = "Verifying character-type columns"
  ) %>%
  interrogate()

# Now we can combine the earlier `agent`
# object with the newer `agent_b` to
# create a `multiagent`
multiagent <-
  create_multiagent(agent, agent_b)

# The `multiagent` can be written to
# a file with the `x_write_disk()` function
x_write_disk(
  multiagent,
  filename = "multiagent-small_table.rds"
)

# We can read the file back as a multiagent
```

```
# with the `x_read_disk()` function and
# we'll get all of the constituent agents
# and their associated intel back as well

# D: Writing a table scan to disk

# We can get an report that describes all
# of the data in the `storms` dataset
tbl_scan <- scan_data(tbl = dplyr::storms)

# The table scan object can be written
# to a file with `x_write_disk()`
x_write_disk(
  tbl_scan,
  filename = "tbl_scan-storms.rds"
)

}
```

`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-prep formula 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(filename, path = NULL)
```

Arguments

`filename`	The name of the YAML file that contains fields related to an *agent*.
`path`	An optional path to the YAML file (combined with `filename`).

Value

A `ptblank_agent` object.

Function ID

11-4

See Also

Other pointblank YAML: yaml_agent_show_exprs(), yaml_agent_string(), yaml_exec(), yaml_informant_incorpor
yaml_read_agent(), yaml_read_informant(), yaml_write()

Examples

```
if (interactive()) {

# 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()`

# Creating an `action_levels` object is a
# common workflow step when creating a
# pointblank agent; we designate 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 `tbl`
# (a requirement for writing to YAML)
agent <-
  create_agent(
    tbl = ~ small_table,
    tbl_name = "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),
    regex = "[0-9]-[a-z]{3}-[0-9]{3}"
  ) %>%
  rows_distinct() %>%
  col_vals_gt(vars(d), value = 100) %>%
  col_vals_lte(vars(c), value = 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(
    "yaml", "agent-small_table.yml",
    package = "pointblank"
  )

# We can view the YAML file in the console
# with the `yaml_agent_string()` function
yaml_agent_string(filename = 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 `tbl`)
# through direct use of the YAML file
# with `yaml_agent_interrogate()`
agent <-
  yaml_agent_interrogate(filename = 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(filename = 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(filename, path = NULL)
```

Arguments

filename	The name of the YAML file that contains fields related to an *agent*.
path	An optional path to the YAML file (combined with filename).

Function ID

11-6

See Also

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

Examples

```
if (interactive()) {

# Let's create a validation plan for the
# data quality analysis of the `small_table`
# dataset; we need an agent and its
# table-prep formula enables retrieval
# of the target table
agent <-
  create_agent(
    tbl = ~ small_table,
    tbl_name = "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),
    regex = "[0-9]-[a-z]{3}-[0-9]{3}"
  ) %>%
  rows_distinct() %>%
  col_vals_gt(vars(d), value = 100) %>%
  col_vals_lte(vars(c), value = 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(
    "yaml", "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(filename = 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(filename = 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, filename = NULL, path = NULL, expanded = FALSE)
```

Arguments

`agent`	An *agent* object of class `ptblank_agent`. If an object is provided here, then `filename` must not be provided.
`filename`	The name of the YAML file that contains fields related to an *agent*. If a file name is provided here, then *agent* object must not be provided in `agent`.
`path`	An optional path to the YAML file (combined with `filename`).
`expanded`	Should the written validation expressions for an *agent* be expanded such that **tidyselect** and `vars()` expressions for columns are evaluated, yielding a validation function per column? By default, this is `FALSE` so expressions as written will be retained in the YAML representation.

Function ID

11-5

See Also

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

Examples

```
if (interactive()) {

# Let's create a validation plan for the
# data quality analysis of the `small_table`
# dataset; we need an agent and its
# table-prep formula enables retrieval
# of the target table
agent <-
  create_agent(
    tbl = ~ small_table,
    tbl_name = "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),
    regex = "[0-9]-[a-z]{3}-[0-9]{3}"
  ) %>%
  rows_distinct() %>%
  col_vals_gt(vars(d), value = 100) %>%
  col_vals_lte(vars(c), value = 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"
)

# There's a similar file in the package
# ('agent-small_table.yml') and it's
# accessible with `system.file()`
yml_file <-
  system.file(
    "yaml", "agent-small_table.yml",
    package = "pointblank"
  )

# The `yaml_agent_string()` function can
# be used with the YAML file as well,
# use the `filename` argument instead
yaml_agent_string(filename = yml_file)

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

}
```

---

yaml_exec *Execute all agent and informant YAML tasks*

Description

The yaml_exec() function takes all relevant **pointblank** YAML files in a directory and executes them. Execution involves interrogation of agents for YAML agents and incorporation of informants

for YAML informants. Under the hood, this uses `yaml_agent_interrogate()` and `yaml_informant_incorporate()` and then `x_write_disk()` to save the processed objects to an output directory. These written artifacts can be read in at any later time with the `x_read_disk()` function or the `read_disk_multiagent()` function. This is useful when data in the target tables are changing and the periodic testing of such tables is part of a data quality monitoring plan.

The output RDS files are named according to the object type processed, the target table, and the date-time of processing. For convenience and modularity, this setup is ideal when a table store YAML file (typically named `"tbl_store.yml"` and produced via the `tbl_store()` and `yaml_write()` workflow) is available in the directory, and when table-prep formulas are accessed by name through `tbl_source()`.

A typical directory of files set up for execution in this way might have the following contents:

- a `"tbl_store.yml"` file for holding table-prep formulas (created with `tbl_store()` and written to YAML with `yaml_write()`)
- one or more YAML *agent* files to validate tables (ideally using `tbl_source()`)
- one or more YAML *informant* files to provide refreshed metadata on tables (again, using `tbl_source()` to reference table preparations is ideal)
- an output folder (default is `"output"`) to save serialized versions of processed agents and informants

Minimal example files of the aforementioned types can be found in the **pointblank** package through the following `system.file()` calls:

- `system.file("yaml","agent-small_table.yml",package = "pointblank")`
- `system.file("yaml","informant-small_table.yml",package = "pointblank")`
- `system.file("yaml","tbl_store.yml",package = "pointblank")`

The directory itself can be accessed using `system.file("yaml",package = "pointblank")`.

Usage

```
yaml_exec(
  path = NULL,
  files = NULL,
  write_to_disk = TRUE,
  output_path = file.path(path, "output"),
  keep_tbl = FALSE,
  keep_extracts = FALSE
)
```

Arguments

`path`	The path that contains the YAML files for agents and informants.
`files`	A vector of YAML files to use in the execution workflow. By default, `yaml_exec()` will attempt to process every valid YAML file in `path` but supplying a vector here limits the scope to the specified files.

`write_to_disk` Should the execution workflow include a step that writes output files to disk? This internally calls `x_write_disk()` to write RDS files and uses the base filename of the agent/informant YAML file as part of the output filename, appending the date-time to the basename.

`output_path` The output path for any generated output files. By default, this will be a subdirectory of the provided `path` called `"output"`.

`keep_tbl, keep_extracts` For agents, the table may be kept if it is a data frame object (databases tables will never be pulled for storage) and *extracts*, collections of table rows that failed a validation step, may also be stored. By default, both of these options are set to `FALSE`.

Value

Invisibly returns a named vector of file paths for the input files that were processed; file output paths (for wherever writing occurred) are given as the names.

Function ID

11-8

See Also

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

Examples

```
if (interactive()) {

# The 'yaml' directory that is
# accessible in the package through
# `system.file()` contains the files
# 1. `agent-small_table.yml`
# 2. `informant-small_table.yml`
# 3. `tbl_store.yml`

# There are references in YAML files
# 1 & 2 to the table store YAML file,
# so, they all work together cohesively

# Let's process the agent and the
# informant YAML files with `yaml_exec()`;
# and we'll specify the working directory
# as the place where the output RDS files
# are written

output_dir <- getwd()

yaml_exec(
  path = system.file(
```

```
    "yaml", package = "pointblank"
  ),
  output = output_dir
)

# This generates two RDS files in the
# working directory: one for the agent
# and the other for the informant; each
# of them are automatically time-stamped
# so that periodic execution can be
# safely carried out without risk of
# overwriting

}
```

`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-prep formula 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(filename, path = NULL)
```

Arguments

`filename`	The name of the YAML file that contains fields related to an *informant*.
`path`	An optional path to the YAML file (combined with `filename`).

Value

A `ptblank_informant` object.

Function ID

11-7

See Also

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

Examples

```
if (interactive()) {

# 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
# to `tbl` with a table-prep formula
# (a requirement for writing to YAML)
informant <-
  create_informant(
    tbl = ~ 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(
    "yaml", "informant-small_table.yml",
    package = "pointblank"
  )

# We can incorporate the data (which
# is accessible through the table-prep
# formula) into the info text through
# direct use of the YAML file with
# `yaml_informant_incorporate()`
informant <-
  yaml_informant_incorporate(filename = 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(filename = 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-prep formula that is set with the `tbl` argument of `create_agent()`). 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(filename, path = NULL)
```

Arguments

`filename`	The name of the YAML file that contains fields related to an *agent*.
`path`	An optional path to the YAML file (combined with `filename`).

Value

A `ptblank_agent` object.

Function ID

11-2

See Also

Other pointblank YAML: `yaml_agent_interrogate()`, `yaml_agent_show_exprs()`, `yaml_agent_string()`, `yaml_exec()`, `yaml_informant_incorporate()`, `yaml_read_informant()`, `yaml_write()`

Examples

```
if (interactive()) {

# 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()`

# Creating an `action_levels` object is a
# common workflow step when creating a
# pointblank agent; we designate 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 `tbl` with
# a table-prep formula (a requirement
# for writing to YAML)
agent <-
  create_agent(
    tbl = ~ small_table,
    tbl_name = "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),
    regex = "[0-9]-[a-z]{3}-[0-9]{3}"
  ) %>%
  rows_distinct() %>%
  col_vals_gt(vars(d), value = 100) %>%
  col_vals_lte(vars(c), value = 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(
    "yaml", "agent-small_table.yml",
    package = "pointblank"
  )

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

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

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

class(agent)

# We can interrogate the data (which
# is accessible through the table-prep
# formula supplied initially) 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(
    filename = 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 information (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(filename, path = NULL)
```

Arguments

`filename`	The name of the YAML file that contains fields related to an *informant*.
`path`	An optional path to the YAML file (combined with `filename`).

Value

A `ptblank_informant` object.

Function ID

11-3

See Also

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

Examples

```
if (interactive()) {

# 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(
    "yaml", "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(filename = yml_file)

class(informant)

}
```

`yaml_write` *Write* **pointblank** *objects to YAML files*

Description

With `yaml_write()` we can take different **pointblank** objects (these are the `ptblank_agent`, `ptblank_informant`, and `tbl_store`) and write them to YAML. With an *agent*, for example, `yaml_write()` will write that everything that is needed to specify an *agent* and it's validation plan to a YAML file. With 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 an *agent* or an *informant* to YAML is that we need to have a table-prep formula specified (it's an R formula that is used to read the target table when `interrogate()` or `incorporate()` is called). This option can be set when using `create_agent()`/`create_informant()` or with `set_tbl()` (useful with an existing agent or informant object).

Usage

```
yaml_write(
  ...,
  .list = list2(...),
  filename = NULL,
  path = NULL,
  expanded = FALSE,
  quiet = FALSE
)
```

Arguments

`...`	Any mix of **pointblank** objects such as the *agent* (`ptblank_agent`), the *informant* (`ptblank_informant`), or the table store (`tbl_store`). The agent and informant can be combined into a single YAML file (so long as both objects refer to the same table). A table store cannot be combined with either an agent or an informant so it must undergo conversion alone.
`.list`	Allows for the use of a list as an input alternative to `...`.

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. If not provided then default names will be used (`"tbl_store.yml"`) for a table store and the other objects will get default naming to the effect of `"<object>-<tbl_name>.yml"`.
path	An optional path to which the YAML file should be saved (combined with `filename`).
expanded	Should the written validation expressions for an *agent* be expanded such that **tidyselect** and `vars()` expressions for columns are evaluated, yielding a validation function per column? By default, this is `FALSE` so expressions as written will be retained in the YAML representation.
quiet	Should the function *not* inform when the file is written? By default this is `FALSE`.

Value

Invisibly returns `TRUE` if the YAML file has been written.

Function ID

11-1

See Also

Other pointblank YAML: `yaml_agent_interrogate()`, `yaml_agent_show_exprs()`, `yaml_agent_string()`, `yaml_exec()`, `yaml_informant_incorporate()`, `yaml_read_agent()`, `yaml_read_informant()`

Examples

```
if (interactive()) {

# 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

# Creating an `action_levels` object is a
# common workflow step when creating a
# pointblank agent; we designate 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 `tbl`
# (a requirement for writing to YAML)
agent <-
  create_agent(
    tbl = ~ small_table,
    tbl_name = "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), regex = "[0-9]-[a-z]{3}-[0-9]{3}"
  ) %>%
  rows_distinct() %>%
  col_vals_gt(vars(d), value = 100) %>%
  col_vals_lte(vars(c), value = 5)

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

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

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

# The YAML can also be printed in the console
# by supplying the agent as the input
yaml_agent_string(agent = agent)

# At some later time, the YAML file can
# be read as a new agent with the
# `yaml_read_agent()` function
```

```
agent <-
  yaml_read_agent(filename = yml_file)

class(agent)

# We can interrogate the data (which
# is accessible through `tbl`)
# 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(filename = yml_file)

class(agent)

}
```

Index