Data validation toolkit for assessing and monitoring data quality.

© 2024–2025 Posit Software, PBC

Table of Contents

1 Validation Plan

This article provides a quick overview of the data validation features in Pointblank. It introduces the key concepts and shows examples of the main functionality, giving you a foundation for using the library effectively.

Later articles in the **User Guide** will expand on each section covered here, providing more explanations and examples.

1.1 Validation Methods

Pointblank's core functionality revolves around validation steps, which are individual checks that verify different aspects of your data. These steps are created by calling validation methods from the `Validate` class. When combined they create a comprehensive validation plan for your data.

Here's an example of a validation that incorporates three different validation methods:

```
import pointblank as pb
import polars as pl

(
    pb.Validate(
        data=pb.load_dataset(dataset="small_table", tbl_type="polars"),
        label="Three different validation methods."
    )
    .col_vals_gt(columns="a", value=0)
    .rows_distinct()
    .col_exists(columns="date")
    .interrogate()
)
```

Pointblank Validation

Three different validation methods.

POLARS

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_gt()	a	0	○→	✓	13	13 1.00	0 0.00	—	—	—	—
2	rows_distinct()	ALL COLUMNS	—	○→	✓	13	9 0.69	2 0.15	—	—	—	CSV
3	col_exists()	date	—	○→	✓	1	1 1.00	0 0.00	—	—	—	—

This example showcases how you can combine different types of validations in a single validation plan:

- a column value validation with `Validate.col_vals_gt()`
- a row-based validation with `Validate.rows_distinct()`
- a table structure validation with `Validate.col_exists()`

Most validation methods share common parameters that enhance their flexibility and power. These shared parameters (overviewed in the next few sections) create a consistent interface across all validation steps while allowing you to customize validation behavior for specific needs.

1.2 Column Selection Patterns

You can apply the same validation logic to multiple columns at once through use of column selection patterns (used in the `columns=` parameter). This reduces repetitive code and makes your validation plans more maintainable:

```python
import narwhals.selectors as nws

# Map validations across multiple columns
(
    pb.Validate(
        data=pb.load_dataset(dataset="small_table", tbl_type="polars"),
        label="Applying column mapping in `columns`."
    )

    # Apply validation rules to multiple columns ---
    .col_vals_not_null(
        columns=["a", "b", "c"]
    )

    # Apply to numeric columns only with a Narwhals selector ---
    .col_vals_gt(
        columns=nws.numeric(),
        value=0
    )
    .interrogate()
)
```

Pointblank Validation

Applying column mapping in `columns`.

POLARS

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_not_null()	a	—	○→	✓	13	13 1.00	0 0.00	—	—	—	—
2	col_vals_not_null()	b	—	○→	✓	13	13 1.00	0 0.00	—	—	—	—
3	col_vals_not_null()	c	—	○→	✓	13	11 0.85	2 0.15	—	—	—	CSV
4	col_vals_gt()	a	0	○→	✓	13	13 1.00	0 0.00	—	—	—	—
5	col_vals_gt()	c	0	○→	✓	13	11 0.85	2 0.15	—	—	—	CSV
6	col_vals_gt()	d	0	○→	✓	13	13 1.00	0 0.00	—	—	—	—

This technique is particularly valuable when working with wide datasets containing many similarly-structured columns or when applying standard quality checks across an entire table. It also ensures consistency in how validation rules are applied across related data columns.

1.3 Preprocessing

Preprocessing (with the `pre=` parameter) allows you to transform or modify your data before applying validation checks, enabling you to validate derived or modified data without altering the original dataset:

```python
import polars as pl

# Define preprocessing functions for `pre=` parameters
def double_column_a(df):
    return df.with_columns(pl.col("a") * 2)

def square_column_c(df):
    return df.with_columns(pl.col("c").pow(2))

(
    pb.Validate(
        data=pb.load_dataset(dataset="small_table", tbl_type="polars"),
        label="Preprocessing validation steps via `pre=`."
    )
    .col_vals_gt(
        columns="a", value=5,

        # Apply transformation before validation ---
        pre=double_column_a  # Double values before checking
    )
    .col_vals_lt(
        columns="c", value=100,

        # Apply more complex transformation ---
        pre=square_column_c  # Square values before checking
    )
    .interrogate()
)
```

Pointblank Validation

Preprocessing validation steps via `pre=`.

POLARS

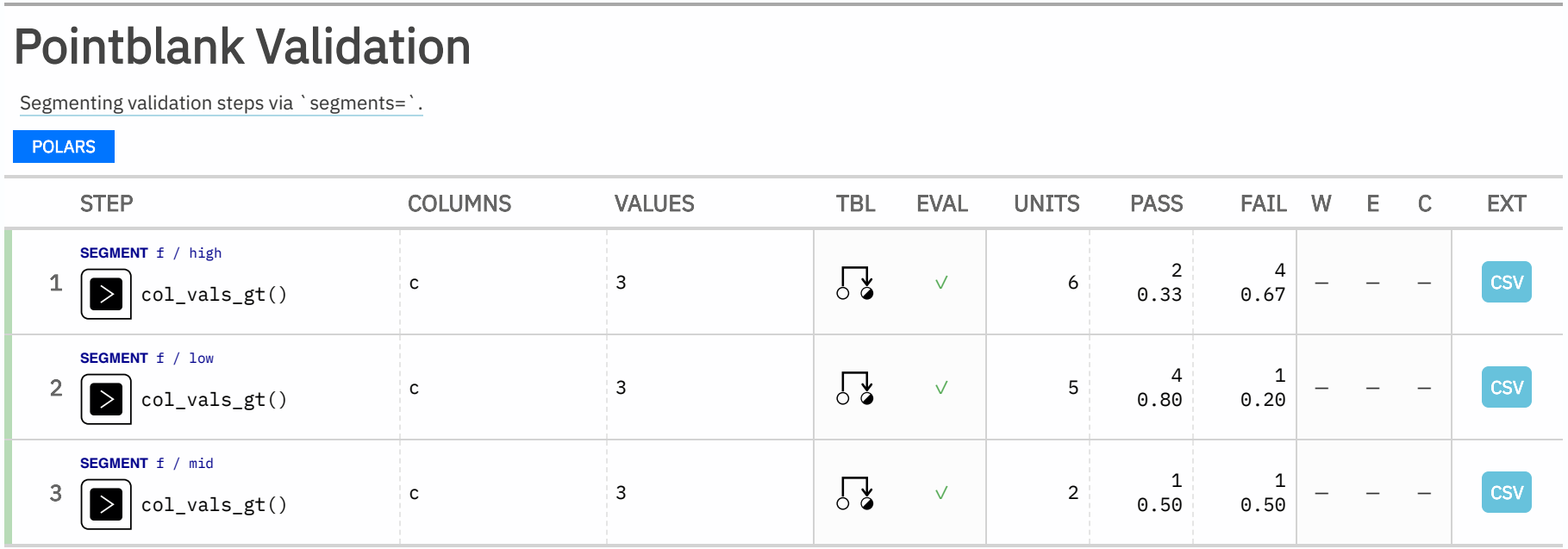
	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_gt()	a	5		✓	13	9 0.69	4 0.31	—	—	—	CSV
2	col_vals_lt()	c	100		✓	13	11 0.85	2 0.15	—	—	—	CSV

Preprocessing enables validation of transformed data without modifying your original dataset, making it ideal for checking derived metrics, or validating normalized values. This approach keeps your validation code clean while allowing for sophisticated data quality checks on calculated results.

1.4 Segmentation

Segmentation (through the `segments=` parameter) allows you to validate data across different groups, enabling you to identify segment-specific quality issues that might be hidden in aggregate analyses:

```
(
    pb.Validate(
        data=pb.load_dataset(dataset="small_table", tbl_type="polars"),
        label="Segmenting validation steps via `segments=`."
    )
    .col_vals_gt(
        columns="c", value=3,

        # Split into steps by categorical values in column 'f' ---
        segments="f"
    )
    .interrogate()
)
```

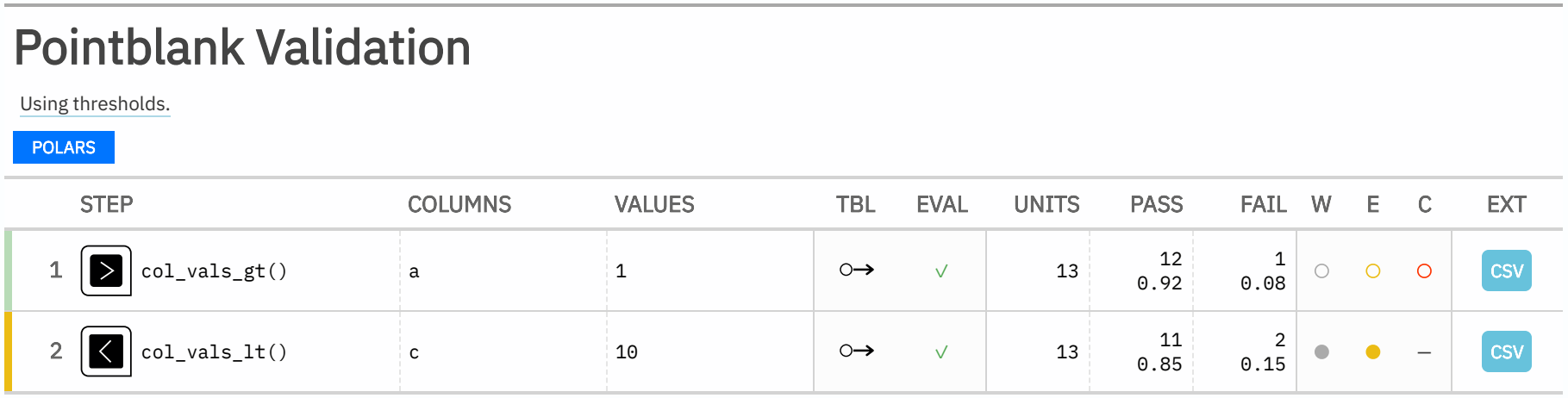


Segmentation is powerful for detecting patterns of quality issues that may exist only in specific data subsets, such as certain time periods, categories, or geographical regions. It helps ensure that all significant segments of your data meet quality standards, not just the data as a whole.

1.5 Thresholds

Thresholds (set through the `thresholds=` parameter) let you set acceptable levels of failure before triggering warnings, errors, or critical notifications for individual validation steps:

```python
(
    pb.Validate(
        data=pb.load_dataset(dataset="small_table", tbl_type="polars"),
        label="Using thresholds."
    )

    # Add validation steps with different thresholds ---
    .col_vals_gt(
        columns="a", value=1,
        thresholds=pb.Thresholds(warning=0.1, error=0.2, critical=0.3)
    )

    # Add another step with stricter thresholds ---
    .col_vals_lt(
        columns="c", value=10,
        thresholds=pb.Thresholds(warning=0.05, error=0.1)
    )
    .interrogate()
)
```



Pointblank Validation

Using thresholds.

POLARS

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_gt()	a	1	○→	✓	13	12 0.92	1 0.08	○	○	○	CSV
2	col_vals_lt()	c	10	○→	✓	13	11 0.85	2 0.15	●	●	—	CSV

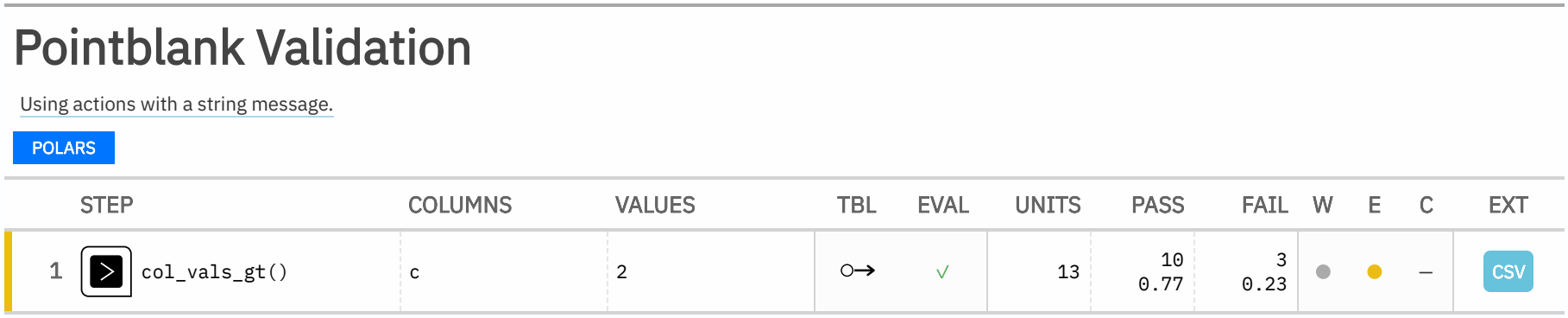
Thresholds provide a nuanced way to monitor data quality, allowing you to set different severity levels based on the importance of each validation and your organization's tolerance for specific types of data issues.

1.6 Actions

Actions (which can be configured in the `actions=` parameter) allow you to define specific responses when validation thresholds are crossed. You can use simple string messages or custom functions for more complex behavior:

```
# Example 1: Action with a string message ---

(
    pb.Validate(
        data=pb.load_dataset(dataset="small_table", tbl_type="polars"),
        label="Using actions with a string message."
    )
    .col_vals_gt(
        columns="c", value=2,
        thresholds=pb.Thresholds(warning=0.1, error=0.2),

        # Add a print-to-console action for the 'warning' threshold ---
        actions=pb.Actions(
            warning="WARNING: Values below `{value}` detected in column 'c'."
        )
    )
    .interrogate()
)
```

```
WARNING: Values below `2` detected in column 'c'.
```



```python
# Example 2: Action with a callable function ---

def custom_action():
    from datetime import datetime
    print(f"Data quality issue found ({datetime.now()}).")

(
    pb.Validate(
        data=pb.load_dataset(dataset="small_table", tbl_type="polars"),
        label="Using actions with a callable function."
    )
    .col_vals_gt(
        columns="a", value=5,
        thresholds=pb.Thresholds(warning=0.1, error=0.2),

        # Apply the function to the 'error' threshold ---
        actions=pb.Actions(error=custom_action)
    )
    .interrogate()
)
```

```
Data quality issue found (2025-11-02 19:38:13.303886).
```

Pointblank Validation

Using actions with a callable function.

POLARS

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_gt()	a	5	○→	✓	13	3 0.23	10 0.77	●	●	–	CSV

With custom action functions, you can implement sophisticated responses like sending notifications or logging to external systems.

1.7 Briefs

Briefs (which can be set through the `brief=` parameter) allow you to customize descriptions associated with validation steps, making validation results more understandable to stakeholders. Briefs can be either automatically generated by setting `brief=True` or defined as custom messages for more specific explanations:

```
(
    pb.Validate(
        data=pb.load_dataset(dataset="small_table", tbl_type="polars"),
        label="Using `brief=` for displaying brief messages."
    )
    .col_vals_gt(
        columns="a", value=0,

        # Use `True` for automatic generation of briefs ---
        brief=True
    )
    .col_exists(
        columns=["date", "date_time"],

        # Add a custom brief for this validation step ---
        brief="Verify required date columns exist for time-series analysis"
    )
    .interrogate()
)
```

Pointblank Validation

Using `brief=` for displaying brief messages.

POLARS

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_gt() Expect that values in a should be > 0.	a	0	○→	√	13	13 1.00	0 0.00	—	—	—	—
2	col_exists() Verify required date columns exist for time-series analysis	date	—	○→	√	1	1 1.00	0 0.00	—	—	—	—
3	col_exists() Verify required date columns exist for time-series analysis	date_time	—	○→	√	1	1 1.00	0 0.00	—	—	—	—

Briefs make validation results more meaningful by providing context about why each check matters. They're particularly valuable in shared reports where stakeholders from various disciplines need to understand validation results in domain-specific terms.

1.8 Getting More Information

Each validation step can be further customized and has additional options. See these pages for more information:

- Validation Methods: A closer look at the more common validation methods
- Column Selection Patterns: Techniques for targeting specific columns
- Preprocessing: Transform data before validation
- Segmentation: Apply validations to specific segments of your data
- Thresholds: Set quality standards and trigger severity levels
- Actions: Respond to threshold exceedances with notifications or custom functions
- Briefs: Add context to validation steps

1.9 Conclusion

Validation steps are the building blocks of data validation in Pointblank. By combining steps from different categories and leveraging common features like thresholds, actions, and preprocessing, you can create comprehensive data quality checks tailored to your specific needs.

The next sections of this guide will dive deeper into each of these topics, providing detailed explanations and examples.

Pointblank provides a comprehensive suite of validation methods to verify different aspects of your data. Each method creates a validation step that becomes part of your validation plan.

These validation methods cover everything from checking column values against thresholds to validating the table structure and detecting duplicates. Combined into validation steps, they form the foundation of your data quality workflow.

Pointblank provides over 25 validation methods to handle diverse data quality requirements. These are grouped into three main categories:

1. Column Value Validations
2. Row-based Validations
3. Table Structure Validations
4. AI-Powered Validations

Within each of these categories, we'll walk through several examples showing how each validation method creates steps in your validation plan.

And we'll use the `small_table` dataset for all of our examples. Here's a preview of it:

POLARS | ROWS 13 | COLUMNS 8

	date_time *Datetime*	date *Date*	a *Int64*	b *String*	c *Int64*	d *Float64*	e *Boolean*	f *String*
1	2016-01-04 11:00:00	2016-01-04	2	1-bcd-345	3	3423.29	True	high
2	2016-01-04 00:32:00	2016-01-04	3	5-egh-163	8	9999.99	True	low
3	2016-01-05 13:32:00	2016-01-05	6	8-kdg-938	3	2343.23	True	high
4	2016-01-06 17:23:00	2016-01-06	2	5-jdo-903	None	3892.4	False	mid
5	2016-01-09 12:36:00	2016-01-09	8	3-ldm-038	7	283.94	True	low
6	2016-01-11 06:15:00	2016-01-11	4	2-dhe-923	4	3291.03	True	mid
7	2016-01-15 18:46:00	2016-01-15	7	1-knw-093	3	843.34	True	high
8	2016-01-17 11:27:00	2016-01-17	4	5-boe-639	2	1035.64	False	low
9	2016-01-20 04:30:00	2016-01-20	3	5-bce-642	9	837.93	False	high
10	2016-01-20 04:30:00	2016-01-20	3	5-bce-642	9	837.93	False	high
11	2016-01-26 20:07:00	2016-01-26	4	2-dmx-010	7	833.98	True	low
12	2016-01-28 02:51:00	2016-01-28	2	7-dmx-010	8	108.34	False	low
13	2016-01-30 11:23:00	2016-01-30	1	3-dka-303	None	2230.09	True	high

1.10 Validation Methods to Validation Steps

In Pointblank, validation *methods* become validation *steps* when you add them to a validation plan. Each method creates a distinct step that performs a specific check on your data.

Here's a simple example showing how three validation methods create three validation steps:

```python
import pointblank as pb

(
    pb.Validate(data=pb.load_dataset(dataset="small_table", tbl_type="polars"))

    # Step 1: Check that values in column `a` are greater than 2 ---
    .col_vals_gt(columns="a", value=2, brief="Values in 'a' must exceed 2.")

    # Step 2: Check that column 'date' exists in the table ---
    .col_exists(columns="date", brief="Column 'date' must exist.")

    # Step 3: Check that the table has exactly 13 rows ---
    .row_count_match(count=13, brief="Table should have exactly 13 rows.")
    .interrogate()
)
```

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_gt() Values in 'a' must exceed 2.	a	2	○→	√	13	9 0.69	4 0.31	—	—	—	CSV
2	col_exists() Column 'date' must exist.	date	—	○→	√	1	1 1.00	0 0.00	—	—	—	—
3	row_count_match() Table should have exactly 13 rows.	—	13	○→	√	1	1 1.00	0 0.00	—	—	—	—

Each validation method produces one step in the validation report above. When combined, these steps form a complete validation plan that systematically checks different aspects of your data quality.

1.11 Common Arguments

Most validation methods in Pointblank share a set of common arguments that provide consistency and flexibility across different validation types:

- `columns=`: specifies which column(s) to validate (used in column-based validations)

- `pre=`: allows data transformation before validation
- `segments=`: enables validation across different data subsets
- `thresholds=`: sets acceptable failure thresholds
- `actions=`: defines actions to take when validations fail
- `brief=`: provides a description of what the validation is checking
- `active=`: determines if the validation step should be executed (default is `True`)
- `na_pass=`: controls how missing values are handled (only for column value validation methods)

For column validation methods, the `na_pass=` parameter determines whether missing values (Null/None/NA) should pass validation (this parameter is covered in a later section).

These arguments follow a consistent pattern across validation methods, so you don't need to memorize different parameter sets for each function. This systematic approach makes Pointblank more intuitive to work with as you build increasingly complex validation plans.

We'll cover most of these common arguments in their own dedicated sections later in the **User Guide**, as some of them represent a deeper topic worthy of focused attention.

1.12 1. Column Value Validations

These methods check individual values within columns against specific criteria:

- **Comparison checks** (`Validate.col_vals_gt()`, `Validate.col_vals_lt()`, etc.) for comparing values to thresholds or other columns
- **Range checks** (`Validate.col_vals_between()`, `Validate.col_vals_outside()`) for verifying that values fall within or outside specific ranges
- **Set membership checks** (`Validate.col_vals_in_set()`, `Validate.col_vals_not_in_set()`) for validating values against predefined sets
- **Null value checks** (`Validate.col_vals_null()`, `Validate.col_vals_not_null()`) for testing presence or absence of null values
- **Pattern matching checks** (`Validate.col_vals_regex()`, `Validate.col_vals_within_spec()`) for validating text patterns with regular expressions or against standard specifications

- **Trending value checks** (`Validate.col_vals_increasing()`, `Validate.col_vals_decreasing()`) for verifying that values increase or decrease as you move down the rows
- **Custom expression checks** (`Validate.col_vals_expr()`) for complex validations using custom expressions

Now let's look at some key examples from select categories of column value validations.

1.12.1 Comparison Checks

Let's start with a simple example of how `Validate.col_vals_gt()` might be used to check if the values in a column are greater than a specified value.

```
(
    pb.Validate(data=pb.load_dataset(dataset="small_table", tbl_type="polars"))
    .col_vals_gt(columns="a", value=5)
    .interrogate()
)
```

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_gt()	a	5	○→	✓	13	3 0.23	10 0.77	—	—	—	CSV

If you're checking data in a column that contains Null/`None`/`NA` values and you'd like to disregard those values (i.e., let them pass validation), you can use `na_pass=True`. The following example checks values in column `c` of `small_table`, which contains two `None` values:

```
(
    pb.Validate(data=pb.load_dataset(dataset="small_table", tbl_type="polars"))
    .col_vals_le(columns="c", value=10, na_pass=True)
    .interrogate()
)
```

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_le()	c	10	○→	✓	13	13 1.00	0 0.00	—	—	—	—

In the above validation table, we see that all test units passed. If we didn't use `na_pass=True` there would be 2 failing test units, one for each `None` value in the `c` column.

It's possible to check against column values against values in an adjacent column. To do this, supply the `value=` argument with the column name within the `col()` helper function. Here's an example of that:

```
(
    pb.Validate(data=pb.load_dataset(dataset="small_table", tbl_type="polars"))
    .col_vals_lt(columns="a", value=pb.col("c"))
    .interrogate()
)
```

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_lt()	a	c	○→	✓	13	6 0.46	7 0.54	—	—	—	CSV

This validation checks that values in column `a` are less than values in column `c`.

1.12.2 Checking of Missing Values

A very common thing to validate is that there are no Null/NA/missing values in a column. The `Validate.col_vals_not_null()` method checks for the presence of missing values:

```
(
    pb.Validate(data=pb.load_dataset(dataset="small_table", tbl_type="polars"))
    .col_vals_not_null(columns="a")
    .interrogate()
)
```

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_not_null()	a	—	○→	✓	13	13 1.00	0 0.00	—	—	—	—

Column `a` has no missing values and the above validation proves this.

1.12.3 Checking Strings with Regexes

A regular expression (regex) validation via the `Validate.col_vals_regex()` validation method checks if values in a column match a specified pattern. Here's an example with two validation steps, each checking text values in a column:

```
(
    pb.Validate(data=pb.load_dataset(dataset="small_table", tbl_type="polars"))
    .col_vals_regex(columns="b", pattern=r"^\d-[a-z]{3}-\d{3}$")
    .col_vals_regex(columns="f", pattern=r"high|low|mid")
    .interrogate()
)
```

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_regex()	b	^\d-[a-z]{3}-\d…	○→	✓	13	13 1.00	0 0.00	—	—	—	—
2	col_vals_regex()	f	high|low|mid	○→	✓	13	13 1.00	0 0.00	—	—	—	—

1.12.4 Checking Strings Against Specifications

The `Validate.col_vals_within_spec()` method validates column values against common data specifications like email addresses, URLs, postal codes, credit card numbers, ISBNs, VINs, and IBANs. This is particularly useful when you need to validate that text data conforms to standard formats:

```python
import polars as pl

# Create a sample table with various data types
sample_data = pl.DataFrame({
    "isbn": ["978-0-306-40615-7", "0-306-40615-2", "invalid"],
    "email": ["test@example.com", "user@domain.co.uk", "not-an-email"],
    "zip": ["12345", "90210", "invalid"]
})

(
    pb.Validate(data=sample_data)
    .col_vals_within_spec(columns="isbn", spec="isbn")
    .col_vals_within_spec(columns="email", spec="email")
    .col_vals_within_spec(columns="zip", spec="postal_code[US]")
    .interrogate()
)
```

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	 col_vals_within_spec()	isbn	isbn	○→	✓	3	2 0.67	1 0.33	—	—	—	CSV
2	 col_vals_within_spec()	email	email	○→	✓	3	2 0.67	1 0.33	—	—	—	CSV
3	 col_vals_within_spec()	zip	postal_code[US]	○→	✓	3	2 0.67	1 0.33	—	—	—	CSV

1.12.5 Checking for Trending Values

The `Validate.col_vals_increasing()` and `Validate.col_vals_decreasing()` validation methods check whether column values are increasing or decreasing as you move down the rows. These are useful for validating time series data, sequential identifiers, or any data where you expect monotonic trends:

```python
import polars as pl

# Create a sample table with increasing and decreasing values
trend_data = pl.DataFrame({
    "id": [1, 2, 3, 4, 5],
    "temperature": [20, 22, 25, 28, 30],
    "countdown": [100, 80, 60, 40, 20]
})

(
    pb.Validate(data=trend_data)
    .col_vals_increasing(columns="id")
    .col_vals_increasing(columns="temperature")
    .col_vals_decreasing(columns="countdown")
    .interrogate()
)
```

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	 col_vals_increasing()	id		○→	✓	5	5 1.00	0 0.00	—	—	—	—
2	 col_vals_increasing()	temperature		○→	✓	5	5 1.00	0 0.00	—	—	—	—
3	 col_vals_decreasing()	countdown		○→	✓	5	5 1.00	0 0.00	—	—	—	—

The `allow_stationary=` parameter lets you control whether consecutive identical values should pass validation. By default, stationary values (e.g., `[1, 2, 2, 3]`) will fail the increasing check, but setting `allow_stationary=True` will allow them to pass.

1.12.6 Handling Missing Values with `na_pass=`

When validating columns containing Null/None/NA values, you can control how these missing values are treated with the `na_pass=` parameter:

```
(
    pb.Validate(data=pb.load_dataset(dataset="small_table", tbl_type="polars"))
    .col_vals_le(columns="c", value=10, na_pass=True)
    .interrogate()
)
```

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_le()	c	10	○→	✓	13	13 1.00	0 0.00	—	—	—	—

In the above example, column `c` contains two `None` values, but all test units pass because we set `na_pass=True`. Without this setting, those two values would fail the validation.

In summary, `na_pass=` works like this:

- `na_pass=True`: missing values pass validation regardless of the condition being tested
- `na_pass=False` (the default): missing values fail validation

1.13 2. Row-based Validations

Row-based validations focus on examining properties that span across entire rows rather than individual columns. These are essential for detecting issues that can't be found by looking at columns in isolation:

- `Validate.rows_distinct()`: ensures no duplicate rows exist in the table
- `Validate.rows_complete()`: verifies that no rows contain any missing values

These row-level validations are particularly valuable for ensuring data integrity and completeness at the record level, which is crucial for many analytical and operational data applications.

1.13.1 Checking Row Distinctness

Here's an example where we check for duplicate rows with `Validate.rows_distinct()`:

```
(
    pb.Validate(data=pb.load_dataset(dataset="small_table", tbl_type="polars"))
    .rows_distinct()
    .interrogate()
)
```

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	 rows_distinct()	ALL COLUMNS	—	○→	√	13	9 0.69	2 0.15	—	—	—	CSV

We can also adapt the `Validate.rows_distinct()` check to use a single column or a subset of columns. To do that, we need to use the `columns_subset=` parameter. Here's an example of that:

```
(
    pb.Validate(data=pb.load_dataset(dataset="small_table", tbl_type="polars"))
    .rows_distinct(columns_subset="b")
    .interrogate()
)
```

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	 rows_distinct()	b	—	○→	√	13	11 0.85	2 0.15	—	—	—	CSV

1.13.2 Checking Row Completeness

Another important validation is checking for complete rows: rows that have no missing values across all columns or a specified subset of columns. The `Validate.rows_complete()` validation method performs this check.

Here's an example checking if all rows in the table are complete (have no missing values in any column):

```
(
    pb.Validate(data=pb.load_dataset(dataset="small_table", tbl_type="polars"))
    .rows_complete()
    .interrogate()
)
```

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	 rows_complete()	ALL COLUMNS	—	○→	√	13	11 0.85	2 0.15	—	—	—	CSV

As the report indicates, there are some incomplete rows in the table.

1.14 3. Table Structure Validations

Table structure validations ensure that the overall architecture of your data meets expectations. These structural checks form a foundation for more detailed data quality assessments:

- `Validate.col_exists()` : verifies a column exists in the table
- `Validate.col_schema_match()` : ensures table matches a defined schema
- `Validate.col_count_match()` : confirms the table has the expected number of columns
- `Validate.row_count_match()` : verifies the table has the expected number of rows
- `Validate.tbl_match()` : validates that the target table matches a comparison table

These structural validations provide essential checks on the fundamental organization of your data tables, ensuring they have the expected dimensions and components needed for reliable data analysis.

1.14.1 Checking Column Presence

If you need to check for the presence of individual columns, the `Validate.col_exists()` validation method is useful. In this example, we check whether the `date` column is present in the table:

```
(
    pb.Validate(data=pb.load_dataset(dataset="small_table", tbl_type="polars"))
    .col_exists(columns="date")
    .interrogate()
)
```

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_exists()	date	—	○→	✓	1	1 1.00	0 0.00	—	—	—	—

That column is present, so the single test unit of this validation step is a passing one.

1.14.2 Checking the Table Schema

For deeper checks of table structure, a schema validation can be performed with the `Validate.col_schema_match()` validation method, where the goal is to check whether the structure of a table matches an expected schema. To define an expected table schema, we need to use the `Schema` class. Here is a simple example that (1) prepares a schema consisting of column names, (2) uses that `schema` object in a `Validate.col_schema_match()` validation step:

```
schema = pb.Schema(columns=["date_time", "date", "a", "b", "c", "d", "e", "f"])

(
    pb.Validate(data=pb.load_dataset(dataset="small_table", tbl_type="polars"))
    .col_schema_match(schema=schema)
    .interrogate()
)
```

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_schema_match()	—	SCHEMA	○→	✓	1	1 1.00	0 0.00	—	—	—	—

The `Validate.col_schema_match()` validation step will only have a single test unit (signifying pass or fail). We can see in the above validation report that the column schema validation passed.

More often, a schema will be defined using column names and column types. We can do that by using a list of tuples in the `columns=` parameter of `Schema`. Here's an example of that approach in action:

```
schema = pb.Schema(
    columns=[
        ("date_time", "Datetime(time_unit='us', time_zone=None)"),
        ("date", "Date"),
        ("a", "Int64"),
        ("b", "String"),
        ("c", "Int64"),
        ("d", "Float64"),
        ("e", "Boolean"),
        ("f", "String"),
    ]
)

(
    pb.Validate(data=pb.load_dataset(dataset="small_table", tbl_type="polars"))
    .col_schema_match(schema=schema)
    .interrogate()
)
```

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_schema_match()	—	SCHEMA	○→	✓	1	1 1.00	0 0.00	—	—	—	—

The `Validate.col_schema_match()` validation method has several boolean parameters for making the checks less stringent:

- `complete=`: requires exact column matching (all expected columns must exist, no extra columns allowed)
- `in_order=`: enforces that columns appear in the same order as defined in the schema
- `case_sensitive_colnames=`: column names must match with exact letter case
- `case_sensitive_dtypes=`: data type strings must match with exact letter case

These parameters all default to `True`, providing strict schema validation. Setting any to `False` relaxes the validation requirements, making the checks more flexible when exact matching isn't necessary or practical for your use case.

1.14.3 Comparing Tables with `tbl_match()`

The `Validate.tbl_match()` validation method provides a comprehensive way to verify that two tables are identical. It performs a progressive series of checks, from least to most stringent:

1. Column count match
2. Row count match
3. Schema match (loose - case-insensitive, any order)
4. Schema match (order - columns in correct order)
5. Schema match (exact - case-sensitive, correct order)
6. Data match (cell-by-cell comparison)

This progressive approach helps identify exactly where tables differ. Here's an example comparing the `small_table` dataset with itself:

```
(
    pb.Validate(data=pb.load_dataset(dataset="small_table", tbl_type="polars"))
    .tbl_match(tbl_compare=pb.load_dataset(dataset="small_table", tbl_type="polars"))
    .interrogate()
)
```

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	tbl_match()	None	EXTERNAL TABLE	○→	✓	1	1 1.00	0 0.00	—	—	—	—

This validation method is especially useful for:

- Verifying that data transformations preserve expected properties
- Comparing production data against a golden dataset
- Ensuring data consistency across different environments
- Validating that imported data matches source data

1.14.4 Checking Counts of Row and Columns

Row and column count validations check the number of rows and columns in a table.

Using `Validate.row_count_match()` checks whether the number of rows in a table matches a specified count.

```
(
    pb.Validate(data=pb.load_dataset(dataset="small_table", tbl_type="polars"))
    .row_count_match(count=13)
    .interrogate()
)
```

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	 row_count_match()	—	13	○→	✓	1	1 1.00	0 0.00	—	—	—	—

The `Validate.col_count_match()` validation method checks if the number of columns in a table matches a specified count.

```
(
    pb.Validate(data=pb.load_dataset(dataset="small_table", tbl_type="polars"))
    .col_count_match(count=8)
    .interrogate()
)
```

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	 col_count_match()	—	8	○→	✓	1	1 1.00	0 0.00	—	—	—	—

Expectations on column and row counts can be useful in certain situations and they align nicely with schema checks.

1.15 4. AI-Powered Validations

AI-powered validations use Large Language Models (LLMs) to validate data based on natural language criteria. This opens up new possibilities for complex validation rules that are difficult to express with traditional programmatic methods.

1.15.1 Validating with Natural Language Prompts

The `Validate.prompt()` validation method allows you to describe validation criteria in plain language. The LLM interprets your prompt and evaluates each row, producing pass/fail results just like other Pointblank validation methods.

This is particularly useful for:

- Semantic checks (e.g., "descriptions should mention a product name")
- Context-dependent validation (e.g., "prices should be reasonable for the product category")
- Subjective quality assessments (e.g., "comments should be professional and constructive")
- Complex rules that would require extensive regex patterns or custom functions

Here's a simple example that validates whether text descriptions contain specific information:

```
import polars as pl

# Create sample data with product descriptions
products = pl.DataFrame({
    "product": ["Widget A", "Gadget B", "Tool C"],
    "description": [
        "High-quality widget made in USA",
        "Innovative gadget with warranty",
        "Professional tool"
    ],
    "price": [29.99, 49.99, 19.99]
})

# Validate that descriptions mention quality or features
(
    pb.Validate(data=products)
    .prompt(
        prompt="Each description should mention either quality, features, or warranty",
        columns_subset=["description"],
        model="anthropic:claude-sonnet-4-5"
    )
    .interrogate()
)
```

The `columns_subset=` parameter lets you specify which columns to include in the validation, improving performance and reducing API costs by only sending relevant data to the LLM.

Note: To use `Validate.prompt()`, you need to have the appropriate API credentials configured for your chosen LLM provider (Anthropic, OpenAI, Ollama, or AWS Bedrock).

1.16 Conclusion

In this article, we've explored the various types of validation methods that Pointblank offers for ensuring data quality. These methods provide a framework for validating column values, checking row properties, verifying table structures, and even using AI for complex semantic validations. By combining these validation methods into comprehensive plans, you can systematically test your data against business rules and quality expectations. And this all helps to ensure your data remains reliable and trustworthy.

Data validation often requires working with columns in flexible ways. Pointblank offers two powerful approaches:

1. Applying validation rules across multiple columns: validate many columns with a single rule
2. Comparing values between columns: create validations that compare values across different columns

This guide covers both approaches in detail with practical examples.

1.17 Part 1: Applying Rules Across Multiple Columns

Many of Pointblank's validation methods perform column-level checks. These methods provide the `columns=` parameter, which accepts not just a single column name but multiple columns through various selection methods.

Why is this useful? Often you'll want to perform the same validation check (e.g., checking that numerical values are all positive) across multiple columns. Rather than defining the same rules multiple times, you can map the validation across those columns in a single step.

Let's explore this using the `game_revenue` dataset:

POLARS | ROWS 2,000 | COLUMNS 11

	player_id *String*	session_id *String*	session_start *Datetime*	time *Datetime*	item_type *String*	item_name *String*
1	ECPANOIXLZHF896	ECPANOIXLZHF896-eol2j8bs	2015-01-01 01:31:03+00:00	2015-01-01 01:31:27+00:00	iap	offer2
2	ECPANOIXLZHF896	ECPANOIXLZHF896-eol2j8bs	2015-01-01 01:31:03+00:00	2015-01-01 01:36:57+00:00	iap	gems3
3	ECPANOIXLZHF896	ECPANOIXLZHF896-eol2j8bs	2015-01-01 01:31:03+00:00	2015-01-01 01:37:45+00:00	iap	gold7
4	ECPANOIXLZHF896	ECPANOIXLZHF896-eol2j8bs	2015-01-01 01:31:03+00:00	2015-01-01 01:42:33+00:00	ad	ad_20sec
5	ECPANOIXLZHF896	ECPANOIXLZHF896-hdu9jkls	2015-01-01 11:50:02+00:00	2015-01-01 11:55:20+00:00	ad	ad_5sec
1996	NAOJRDMCSEBI281	NAOJRDMCSEBI281-j2vs9ilp	2015-01-21 01:57:50+00:00	2015-01-21 02:02:50+00:00	ad	ad_survey
1997	NAOJRDMCSEBI281	NAOJRDMCSEBI281-j2vs9ilp	2015-01-21 01:57:50+00:00	2015-01-21 02:22:14+00:00	ad	ad_survey
1998	RMOSWHJGELCI675	RMOSWHJGELCI675-vbhcsmtr	2015-01-21 02:39:48+00:00	2015-01-21 02:40:00+00:00	ad	ad_5sec
1999	RMOSWHJGELCI675	RMOSWHJGELCI675-vbhcsmtr	2015-01-21 02:39:48+00:00	2015-01-21 02:47:12+00:00	iap	offer5
2000	GJCXNTWEBIPQ369	GJCXNTWEBIPQ369-9elq67md	2015-01-21 03:59:23+00:00	2015-01-21 04:06:29+00:00	ad	ad_5sec

1.17.1 Using a List of Column Names

The simplest way to validate multiple columns is to provide a list to the `columns=` parameter. In the `game_revenue` dataset, we have two columns with numerical data: `item_revenue` and `session_duration`. If we expect all values in both columns to be greater than `0`, we can write:

```
import pointblank as pb

(
    pb.Validate(data=pb.load_dataset("game_revenue"))
    .col_vals_gt(
        columns=["item_revenue", "session_duration"],
        value=0
    )
    .interrogate()
)
```

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_gt()	item_revenue	0	○→	√	2000	2000 1.00	0 0.00	—	—	—	—
2	col_vals_gt()	session_duration	0	○→	√	2000	2000 1.00	0 0.00	—	—	—	—

The validation report shows two validation steps were created from a single method call! All validation parameters are shared across all generated steps, including thresholds and briefs:

```
(
    pb.Validate(data=pb.load_dataset("game_revenue"))
    .col_vals_gt(
        columns=["item_revenue", "session_duration"],
        value=0,
        thresholds=(0.1, 0.2, 0.3),
        brief="`{col}` must be greater than zero."
    )
    .interrogate()
)
```

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_gt() item_revenue must be greater than zero.	item_revenue	0	○→	√	2000	2000 1.00	0 0.00	○	○	○	—
2	col_vals_gt() session_duration must be greater than zero.	session_duration	0	○→	√	2000	2000 1.00	0 0.00	○	○	○	—

In this example, you can see that the validation report displays customized briefs for each column ("`item_revenue` must be greater than zero." and "`session_duration` must be greater than zero."), automatically substituting the column name using the `{col}` placeholder in the brief template. This feature is particularly helpful when reviewing reports, as it provides clear, human-readable descriptions of what

each validation step is checking. When working with multiple columns through a single validation call, these dynamically generated briefs make your validation reports more understandable for both technical and non-technical stakeholders.

1.17.2 Using Pointblank's Column Selectors

For more advanced column selection, Pointblank provides selector functions that resolve columns based on:

- text patterns in column names
- column position
- column data type

Two common selectors, `starts_with()` and `ends_with()`, resolve columns based on text patterns in column names.

The `game_revenue` dataset has three columns starting with "item": `item_type`, `item_name`, and `item_revenue`. Let's check that these columns contain no missing values:

```
(
    pb.Validate(data=pb.load_dataset("game_revenue"))
    .col_vals_not_null(columns=pb.starts_with("item"))
    .interrogate()
)
```

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_not_null()	item_type	—	○→	√	2000	2000 1.00	0 0.00	—	—	—	—
2	col_vals_not_null()	item_name	—	○→	√	2000	2000 1.00	0 0.00	—	—	—	—
3	col_vals_not_null()	item_revenue	—	○→	√	2000	2000 1.00	0 0.00	—	—	—	—

Three validation steps were automatically created because three columns matched the pattern.

The complete list of column selectors includes:

- `starts_with()`
- `ends_with()`
- `contains()`
- `matches()`
- `everything()`
- `first_n()`
- `last_n()`

1.17.3 Combining Column Selectors

Column selectors can be combined for more powerful selection. To do this, use the `col()` helper function with logical operators:

- `&` (*and*)
- `|` (*or*)
- `-` (*difference*)
- `~` (*not*)

For example, to select all columns except the first four:

```
col_selection = pb.col(pb.everything() - pb.first_n(4))

(
    pb.Validate(data=pb.load_dataset("game_revenue"))
    .col_vals_not_null(
        columns=col_selection,
        thresholds=(1, 0.05, 0.1)
    )
    .interrogate()
)
```

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_not_null()	item_type	–	○→	✓	2000	2000 1.00	0 0.00	○	○	○	–
2	col_vals_not_null()	item_name	–	○→	✓	2000	2000 1.00	0 0.00	○	○	○	–
3	col_vals_not_null()	item_revenue	–	○→	✓	2000	2000 1.00	0 0.00	○	○	○	–
4	col_vals_not_null()	session_duration	–	○→	✓	2000	2000 1.00	0 0.00	○	○	○	–
5	col_vals_not_null()	start_day	–	○→	✓	2000	2000 1.00	0 0.00	○	○	○	–
6	col_vals_not_null()	acquisition	–	○→	✓	2000	2000 1.00	0 0.00	○	○	○	–
7	col_vals_not_null()	country	–	○→	✓	2000	2000 1.00	0 0.00	○	○	○	–

This selects every column except the first four, resulting in seven validation steps.

1.17.4 Narwhals Selectors

Pointblank also supports column selectors from the Narwhals library, which include:

- `matches()`
- `by_dtype()`
- `boolean()`
- `categorical()`
- `datetime()`
- `numeric()`
- `string()`

Here's an example selecting all numeric columns:

```
import narwhals.selectors as ncs

(
    pb.Validate(data=pb.load_dataset("game_revenue"))
    .col_vals_gt(
        columns=ncs.numeric(),
        value=0
    )
    .interrogate()
)
```

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_gt()	item_revenue	0	○→	✓	2000	2000 1.00	0 0.00	—	—	—	—
2	col_vals_gt()	session_duration	0	○→	✓	2000	2000 1.00	0 0.00	—	—	—	—

And selecting all string columns matching “item_”:

```
(
    pb.Validate(data=pb.load_dataset("game_revenue"))
    .col_vals_not_null(columns=pb.col(ncs.string() & ncs.matches("item_")))
    .interrogate()
)
```

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_not_null()	item_type	—	○→	✓	2000	2000 1.00	0 0.00	—	—	—	—
2	col_vals_not_null()	item_name	—	○→	✓	2000	2000 1.00	0 0.00	—	—	—	—

This example demonstrates the power of combining Narwhals selectors with logical operators. By using `ncs.string()` to select string columns and then filtering with `ncs.matches("item_")`, we can precisely target text columns with specific naming patterns. This type of

targeted selection is particularly valuable when working with wide datasets that have consistent column naming conventions, allowing you to apply appropriate validation rules to logically grouped columns without explicitly listing each one.

1.17.5 Caveats for Using Column Selectors

While column selectors are powerful, there are some caveats. If a selector doesn't match any columns, the validation won't fail but will show an 'explosion' in the report:

```
(
    pb.Validate(data=pb.load_dataset("game_revenue"))
    .col_vals_not_null(columns=pb.starts_with("items"))
    .col_vals_gt(columns="item_revenue", value=0)
    .interrogate()
)
```

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_not_null()	StartsWith(text…	–	○→	💥	–	–	–	–	–	–	–
2	col_vals_gt()	item_revenue	0	○→	✓	2000	2000 1.00	0 0.00	–	–	–	–

Notice that although there was a problem with Step 1 (that should be addressed), the interrogation did move on to Step 2 without complication.

To mitigate uncertainty, include validation steps that check for the existence of key columns with `Validate.col_exists()` or verify the schema with `Validate.col_schema_match()`.

1.18 Part 2: Comparing Values Between Columns

Sometimes you need to compare values across different columns rather than against fixed values. Pointblank enables this through the `col()` helper function.

Let's look at examples using the `small_table` dataset:

POLARS | ROWS 13 | COLUMNS 8

	date_time *Datetime*	date *Date*	a *Int64*	b *String*	c *Int64*	d *Float64*	e *Boolean*	f *String*
1	2016-01-04 11:00:00	2016-01-04	2	1-bcd-345	3	3423.29	True	high
2	2016-01-04 00:32:00	2016-01-04	3	5-egh-163	8	9999.99	True	low
3	2016-01-05 13:32:00	2016-01-05	6	8-kdg-938	3	2343.23	True	high
4	2016-01-06 17:23:00	2016-01-06	2	5-jdo-903	None	3892.4	False	mid
5	2016-01-09 12:36:00	2016-01-09	8	3-ldm-038	7	283.94	True	low
6	2016-01-11 06:15:00	2016-01-11	4	2-dhe-923	4	3291.03	True	mid
7	2016-01-15 18:46:00	2016-01-15	7	1-knw-093	3	843.34	True	high
8	2016-01-17 11:27:00	2016-01-17	4	5-boe-639	2	1035.64	False	low
9	2016-01-20 04:30:00	2016-01-20	3	5-bce-642	9	837.93	False	high
10	2016-01-20 04:30:00	2016-01-20	3	5-bce-642	9	837.93	False	high
11	2016-01-26 20:07:00	2016-01-26	4	2-dmx-010	7	833.98	True	low
12	2016-01-28 02:51:00	2016-01-28	2	7-dmx-010	8	108.34	False	low
13	2016-01-30 11:23:00	2016-01-30	1	3-dka-303	None	2230.09	True	high

1.18.1 Using `col()` to Specify a Comparison Column

While we typically use validation methods to compare column values against fixed values:

```
...
.col_vals_gt(columns="a", value=2, ...)
...
```

We can also compare values between columns by using `col()` in the `value=` parameter:

```
...
.col_vals_gt(columns="a", value=pb.col("x"), ...)
...
```

This checks that each value in column `a` is greater than the corresponding value in column `x`. Here's a concrete example:

```
(
    pb.Validate(data=pb.load_dataset("small_table"))
    .col_vals_gt(
        columns="d",
        value=pb.col("c")
    )
    .interrogate()
)
```

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_gt()	d	c	○→	✓	13	11 0.85	2 0.15	—	—	—	CSV

Notice that the validation report shows both column names (`d` and `c`). There are two failing test units because of missing values in column `c`. When comparing across columns, missing values in either column can cause failures.

To handle missing values, use `na_pass=True`:

```
(
    pb.Validate(data=pb.load_dataset("small_table"))
    .col_vals_gt(
        columns="d",
        value=pb.col("c"),
        na_pass=True
    )
    .interrogate()
)
```

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_gt()	d	c	○→	✓	13	13 1.00	0 0.00	—	—	—	—

Now all tests pass.

The following validation methods accept a `col()` expression in their `value=` parameter:

- `Validate.col_vals_gt()`
- `Validate.col_vals_lt()`
- `Validate.col_vals_ge()`
- `Validate.col_vals_le()`
- `Validate.col_vals_eq()`
- `Validate.col_vals_ne()`

1.18.2 Using `col()` in Range Checks

For range validations via `Validate.col_vals_between()` and `Validate.col_vals_outside()` you can use a mix of column references and fixed values:

```
(
    pb.Validate(data=pb.load_dataset("small_table"))
    .col_vals_between(
        columns="d",
        left=pb.col("c"),
        right=10_000,
        na_pass=True
    )
    .interrogate()
)
```

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_between()	d	[c, 10000]	○→	✓	13	13 1.00	0 0.00	—	—	—	—

The validation report shows the range as `[c, 10000]`, indicating that the lower bound comes from column `c` while the upper bound is fixed at `10000`.

1.19 Advanced Examples: Combining Both Approaches

The true power comes from combining both approaches: validating multiple columns and using cross-column comparisons:

```
validation = (
    pb.Validate(data=pb.load_dataset("small_table"))
    .col_vals_gt(
        columns=["c", "d"],
        value=pb.col("a"),
        na_pass=True
    )
    .interrogate()
)

validation
```

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_gt()	c	a	○→	√	13	8 0.62	5 0.38	—	—	—	CSV
2	col_vals_gt()	d	a	○→	√	13	13 1.00	0 0.00	—	—	—	—

This creates validation steps checking that values in both columns `d` and `e` are greater than their corresponding values in column `a`.

1.20 Conclusion

Pointblank provides flexible approaches to working with columns:

1. Column selection: validate multiple columns with a single validation rule
2. Cross-column comparison: compare values between columns

These capabilities allow you to:

- write more concise validation code
- apply consistent validation rules across similar columns
- create dynamic validations that check relationships between columns
- build comprehensive data quality checks with minimal code

By getting familiar with these techniques, you can create more elegant and powerful validation plans while also reducing repetition in your code.

While the available validation methods can do a lot for you, there's likewise a lot of things you *can't* easily do with them. What if you wanted to validate that

- string lengths in a column are less than 10 characters?
- the median of values in a column is less than the median of values in another column?
- there are at least three instances of every categorical value in a column?

These constitute more sophisticated validation requirements, yet such examinations are quite prevalent in practice. Rather than expanding our library to encompass every conceivable validation scenario (a pursuit that would yield an unwieldy and potentially infinite collection) we instead employ a more elegant approach. By transforming the table under examination through judicious preprocessing and exposing key metrics, we may subsequently employ the existing collection of validation methods. This compositional strategy affords us considerable analytical power while maintaining conceptual clarity and implementation parsimony.

Central to this approach is the idea of composability. Pointblank makes it easy to safely transform the target table for a given validation via the `pre=` argument. Any computed columns are available for the (short) lifetime of the validation step during interrogation. This composability means:

1. we can validate on different forms of the initial dataset (e.g., validating on aggregate forms, validating on calculated columns, etc.)
2. there's no need to start an entirely new validation process for each transformed version of the data (i.e., one tabular report could be produced instead of several)

This compositional paradigm allows us to use data transformation effectively within our validation workflows, maintaining both flexibility and clarity in our data quality assessments.

1.21 Transforming Data with Lambda Functions

Now, through examples, let's look at the process of performing the validations mentioned above. We'll use the `small_table` dataset for all of the examples. Here it is in its entirety:

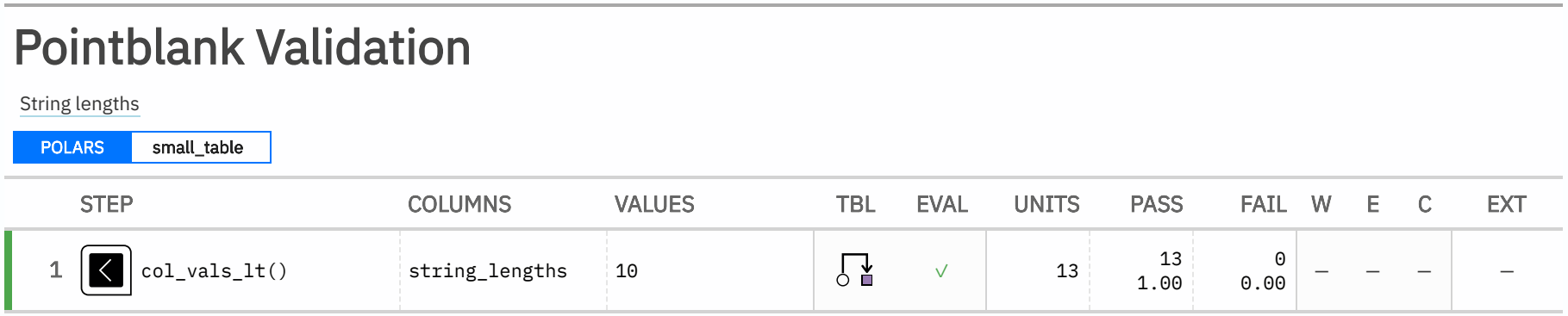
POLARS ROWS 13 COLUMNS 8

	date_time *Datetime*	date *Date*	a *Int64*	b *String*	c *Int64*	d *Float64*	e *Boolean*	f *String*
1	2016-01-04 11:00:00	2016-01-04	2	1-bcd-345	3	3423.29	True	high
2	2016-01-04 00:32:00	2016-01-04	3	5-egh-163	8	9999.99	True	low
3	2016-01-05 13:32:00	2016-01-05	6	8-kdg-938	3	2343.23	True	high
4	2016-01-06 17:23:00	2016-01-06	2	5-jdo-903	None	3892.4	False	mid
5	2016-01-09 12:36:00	2016-01-09	8	3-ldm-038	7	283.94	True	low
6	2016-01-11 06:15:00	2016-01-11	4	2-dhe-923	4	3291.03	True	mid
7	2016-01-15 18:46:00	2016-01-15	7	1-knw-093	3	843.34	True	high
8	2016-01-17 11:27:00	2016-01-17	4	5-boe-639	2	1035.64	False	low
9	2016-01-20 04:30:00	2016-01-20	3	5-bce-642	9	837.93	False	high
10	2016-01-20 04:30:00	2016-01-20	3	5-bce-642	9	837.93	False	high
11	2016-01-26 20:07:00	2016-01-26	4	2-dmx-010	7	833.98	True	low
12	2016-01-28 02:51:00	2016-01-28	2	7-dmx-010	8	108.34	False	low
13	2016-01-30 11:23:00	2016-01-30	1	3-dka-303	None	2230.09	True	high

In getting to grips with the basics, we'll try to validate that string lengths in the `b` column are less than 10 characters. We can't directly use the `Validate.col_vals_lt()` validation method with that column because it is meant to be used with a column of numeric values. Let's just give that method what it needs and create a column with string lengths!

The target table is a Polars DataFrame so we'll provide a function that uses the Polars API to add in that numeric column:

```python
import polars as pl

# Define a preprocessing function that gets string lengths from column `b`
def add_string_length_column(df):
    return df.with_columns(string_lengths=pl.col("b").str.len_chars())

(
    pb.Validate(
        data=pb.load_dataset(dataset="small_table", tbl_type="polars"),
        tbl_name="small_table",
        label="String lengths"
    )
    .col_vals_lt(

        # The generated column, via `pre=` (see below) ---
        columns="string_lengths",

        # The string length value to be less than ---
        value=10,

        # The preprocessing function that modifies the table ---
        pre=add_string_length_column
    )
    .interrogate()
)
```



Pointblank Validation

String lengths

POLARS | small_table

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_lt()	string_lengths	10		✓	13	13 1.00	0 0.00	—	—	—	—

The validation was successfully constructed and we can see from the validation report table that all strings in `b` had lengths less than 10 characters. Also note that the icon under the `TBL` column is no longer a rightward-facing arrow, but one that is indicative of a transformation taking place.

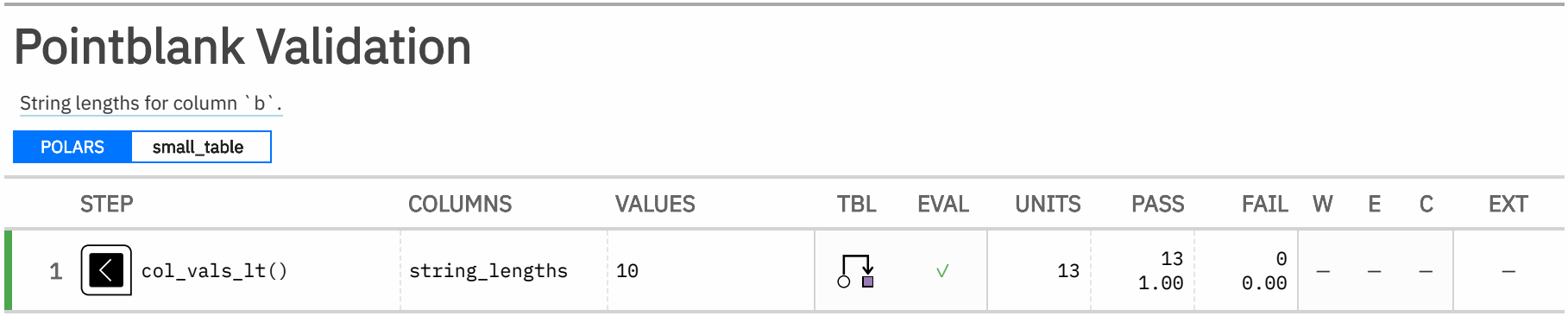
Let's examine the transformation approach more closely. In the previous example, we're not directly testing the `b` column itself. Instead, we're validating the `string_lengths` column that was generated by the lambda function provided to `pre=`. The Polars API's `with_columns()` method does the heavy lifting, creating numerical values that represent each string's length in the original column.

That transformation occurs only during interrogation and only for that validation step. Any prior or subsequent steps would normally use the as-provided `small_table`. Having the possibility for data transformation being isolated at the step level means that you don't have to generate separate validation plans for each form of the data, you're free to fluidly transform the target table as necessary for perform validations on different representations of the data.

1.22 Using Custom Functions for Preprocessing

While lambda functions work well for simple transformations, custom named functions can make your validation code more organized and reusable, especially for complex preprocessing logic. Let's implement the same string length validation using a dedicated function:

```
def add_string_lengths(df):
    # This generates string length from a column `b`; the new column with
    # the values is called `string_lengths` (will be placed as the last column)
    return df.with_columns(string_lengths=pl.col("b").str.len_chars())

(
    pb.Validate(
        data=pb.load_dataset(dataset="small_table", tbl_type="polars"),
        tbl_name="small_table",
        label="String lengths for column `b`."
    )
    .col_vals_lt(

        # Use of a column selector function to select the last column ---
        columns=pb.last_n(1),

        # The string length to be less than ---
        value=10,

        # Custom function for generating string lengths in a new column ---
        pre=add_string_lengths
    )
    .interrogate()
)
```

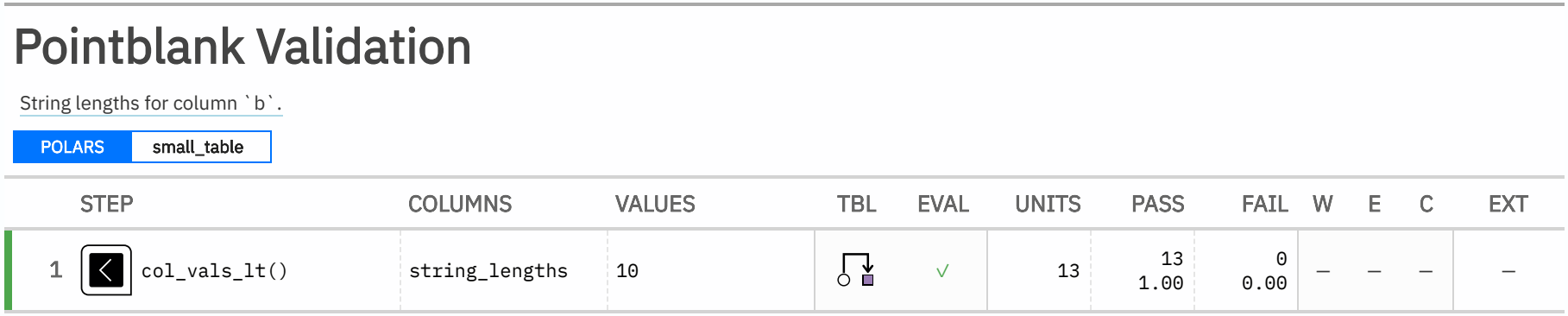


The column-generating logic was placed in the `add_string_lengths()` function, which is then passed to `pre=`. Notice we're using `pb.last_n(1)` in the `columns` parameter. This is a convenient column selector that targets the last column in the DataFrame, which in our case is the newly created `string_lengths` column. This saves us from having to explicitly write out the column name, making our code more adaptable if column names change. Despite not specifying the name directly, you'll still see the actual column name (`string_lengths`) displayed in the validation report.

1.23 Creating Parameterized Preprocessing Functions

So far we've used simple functions and lambdas, but sometimes you may want to create more flexible preprocessing functions that can be configured with parameters. Let's create a reusable function that can calculate string lengths for any column:

```python
def string_length_calculator(column_name):
    """Returns a preprocessing function that calculates string lengths for the specified column."""
    def preprocessor(df):
        return df.with_columns(string_lengths=pl.col(column_name).str.len_chars())
    return preprocessor

# Validate string lengths in column b
(
    pb.Validate(
        data=pb.load_dataset(dataset="small_table", tbl_type="polars"),
        tbl_name="small_table",
        label="String lengths for column `b`."
    )
    .col_vals_lt(
        columns=pb.last_n(1),
        value=10,
        pre=string_length_calculator(column_name="b")
    )
    .interrogate()
)
```



Pointblank Validation

String lengths for column `b`.

POLARS | small_table

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_lt()	string_lengths	10		✓	13	13 1.00	0 0.00	—	—	—	—

This pattern is called a *function factory*, which is a function that creates and returns another function. The outer function (`string_length_calculator()`) accepts parameters that customize the behavior of the returned preprocessing function. The inner function (`preprocessor()`) is what actually gets called during validation.

This approach offers several benefits as it:

- creates reusable, configurable preprocessing functions

- keeps your validation code DRY
- allows you to separate configuration from implementation
- enables easy application of the same transformation to different columns

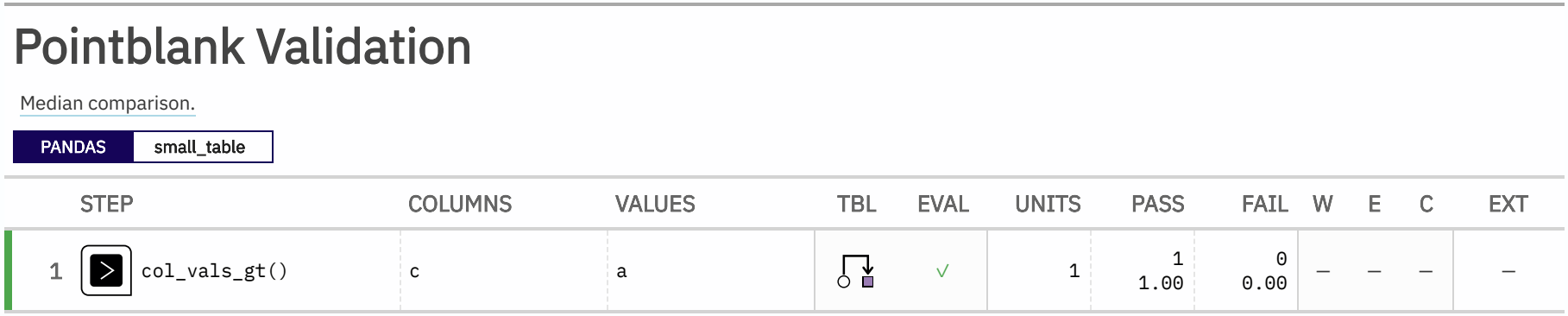
You could extend this pattern to create even more sophisticated preprocessing functions with multiple parameters, default values, and complex logic.

1.24 Using Narwhals to Preprocess Many Types of DataFrames

In this previous example we used a Polars table. You might have a situation where you perform data validation variously on Pandas and Polars DataFrames. This is where Narwhals becomes handy: it provides a single, consistent API that works across multiple DataFrame types, eliminating the need to learn and switch between different APIs depending on your data source.

Let's obtain `small_table` as a Pandas DataFrame. We'll construct a validation step to verify that the median of column `c` is greater than the median in column `a`.

```
import narwhals as nw

# Define preprocessing function using Narwhals for cross-backend compatibility
def get_median_columns_c_and_a(df):
    return nw.from_native(df).select(nw.median("c"), nw.median("a"))

(
    pb.Validate(
        data=pb.load_dataset(dataset="small_table", tbl_type="pandas"),
        tbl_name="small_table",
        label="Median comparison.",
    )
    .col_vals_gt(
        columns="c",
        value=pb.col("a"),

        # Using Narwhals to modify the table; generates table with columns `c` and `a` ---
        pre=get_median_columns_c_and_a
    )
    .interrogate()
)
```



The goal is to check that the median value of `c` is greater than the corresponding median of column `a`, which we set up through the `columns=` and `value=` parameters in the `Validate.col_vals_gt()` method.

There's a bit to unpack here so let's look at at the lambda function first. Narwhals can translate a Pandas DataFrame to a Narwhals DataFrame with its `from_native()` function. After that initiating step, you're free to use the Narwhals API (which is modeled on a subset of the Polars API) to do the necessary data transformation. In this case, we are getting the medians of the `c` and `a` columns and ending up with a one-row, two-column table.

We should note that the transformed table is, perhaps surprisingly, a Narwhals DataFrame (we didn't have to go back to a Pandas DataFrame by using `.to_native()`). Pointblank is able to work directly with the Narwhals DataFrame for validation purposes, which makes the workflow more concise.

One more thing to note: Pointblank provides a convenient syntactic sugar for working with Narwhals. If you name the lambda parameter `dfn` instead of `df`, the system automatically applies `nw.from_native()` to the input DataFrame first. This lets you write more concise code without having to explicitly convert the DataFrame to a Narwhals format.

1.25 Swapping in a Totally Different DataFrame

Sometimes data validation requires looking at completely transformed versions of your data (such as aggregated summaries, pivoted views, or even reference tables). While this approach goes against the typical paradigm of validating a single *target table*, there are legitimate use cases where you might need to validate properties that only emerge after significant transformations.

Let's now try to prepare the final validation scenario, checking that there are at least three instances of every categorical value in column `f` (which contains string values in the set of `"low"`, `"mid"`, and `"high"`). This time, we'll prepare the transformed table (transformed by

Polars expressions) outside of the Pointblank code.

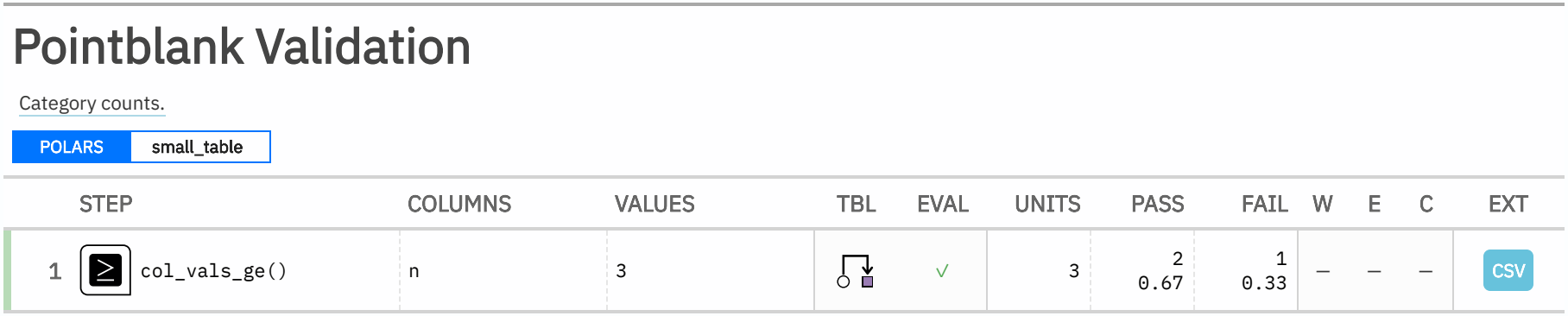
```
data_original = pb.load_dataset(dataset="small_table", tbl_type="polars")
data_transformed = data_original.group_by("f").len(name="n")

data_transformed
```

shape: (3, 2)

f	n
str	u32
"mid"	2
"high"	6
"low"	5

Then, we'll plug in the `data_transformed` DataFrame with a preprocessing function:

```
# Define preprocessing function to use the transformed data
def use_transformed_data(df):
    return data_transformed

(
    pb.Validate(
        data=data_original,
        tbl_name="small_table",
        label="Category counts.",
    )
    .col_vals_ge(
        columns="n",
        value=3,
        pre=use_transformed_data
    )
    .interrogate()
)
```



We can see from the validation report table that there are three test units. This corresponds to a row for each of the categorical value counts. From the report, we find that two of the three test units are passing test units (turns out there are only two instances of `"mid"` in column `f`).

Note that the swapped-in table can be any table type that Pointblank supports, like a Polars DataFrame (as shown here), a Pandas DataFrame, a Narwhals DataFrame, or any other compatible format. This flexibility allows you to validate properties of your data that might only be apparent after significant reshaping or aggregation.

1.26 Conclusion

The preprocessing capabilities in Pointblank provide the power and flexibility for validating complex data properties beyond what's directly possible with the standard validation methods. Through the `pre=` parameter, you can:

- transform your data on-the-fly with computed columns
- generate aggregated metrics to validate statistical properties
- work seamlessly across different DataFrame types using Narwhals
- swap in completely different tables when validating properties that emerge only after transformation

By combining these preprocessing techniques with Pointblank's validation methods, you can create comprehensive data quality checks that address virtually any validation scenario without needing an endless library of specialized validation functions. This composable approach keeps your validation code concise while allowing you to verify even the most complex data quality requirements.

Remember that preprocessing happens just for the specific validation step, keeping your validation plan organized and maintaining the integrity of your original data throughout the rest of the validation process.

When validating data, you often need to analyze specific subsets or segments of your data separately. Maybe you want to ensure that data quality meets standards in each geographic region, for each product category, or across different time periods. This is where the `segments=` argument can be useful.

Data segmentation lets you split a validation step into multiple segments, with each segment receiving its own validation step. Rather than validating an entire table at once, you could instead validate different partitions separately and get separate results for each.

The `segments=` argument is available in many validation methods; typically it's in those methods that check values within rows, and those methods that examine entire rows (`Validate.rows_distinct()`, `Validate.rows_complete()`). When you use it, Pointblank will:

1. split your data according to your segmentation criteria
2. run the validation separately on each segment
3. report results individually for each segment

Let's explore how to use the `segments=` argument through a few practical examples.

1.27 Basic Segmentation by Column Values

The simplest way to segment data is by the unique values in a column. For the upcoming example, we'll use the `small_table` dataset, which contains a categorical-value column called `f`.

First, let's preview the dataset:

```
table = pb.load_dataset()

pb.preview(table)
```

POLARS | ROWS 13 | COLUMNS 8

	date_time *Datetime*	date *Date*	a *Int64*	b *String*	c *Int64*	d *Float64*	e *Boolean*	f *String*
1	2016-01-04 11:00:00	2016-01-04	2	1-bcd-345	3	3423.29	True	high
2	2016-01-04 00:32:00	2016-01-04	3	5-egh-163	8	9999.99	True	low
3	2016-01-05 13:32:00	2016-01-05	6	8-kdg-938	3	2343.23	True	high
4	2016-01-06 17:23:00	2016-01-06	2	5-jdo-903	None	3892.4	False	mid
5	2016-01-09 12:36:00	2016-01-09	8	3-ldm-038	7	283.94	True	low
9	2016-01-20 04:30:00	2016-01-20	3	5-bce-642	9	837.93	False	high
10	2016-01-20 04:30:00	2016-01-20	3	5-bce-642	9	837.93	False	high
11	2016-01-26 20:07:00	2016-01-26	4	2-dmx-010	7	833.98	True	low
12	2016-01-28 02:51:00	2016-01-28	2	7-dmx-010	8	108.34	False	low
13	2016-01-30 11:23:00	2016-01-30	1	3-dka-303	None	2230.09	True	high

Now, let's validate that values in column `d` are greater than `100`, but we'll also segment the validation by the categorical values in column `f`:

```
validation_1 = (
    pb.Validate(
        data=pb.load_dataset(),
        tbl_name="small_table",
        label="Segmented validation by category"
    )
    .col_vals_gt(
        columns="d", value=100,

        # Segment by unique values in column `f` ---
        segments="f"
    )
    .interrogate()
)

validation_1
```

Pointblank Validation

Segmented validation by category

POLARS | small_table

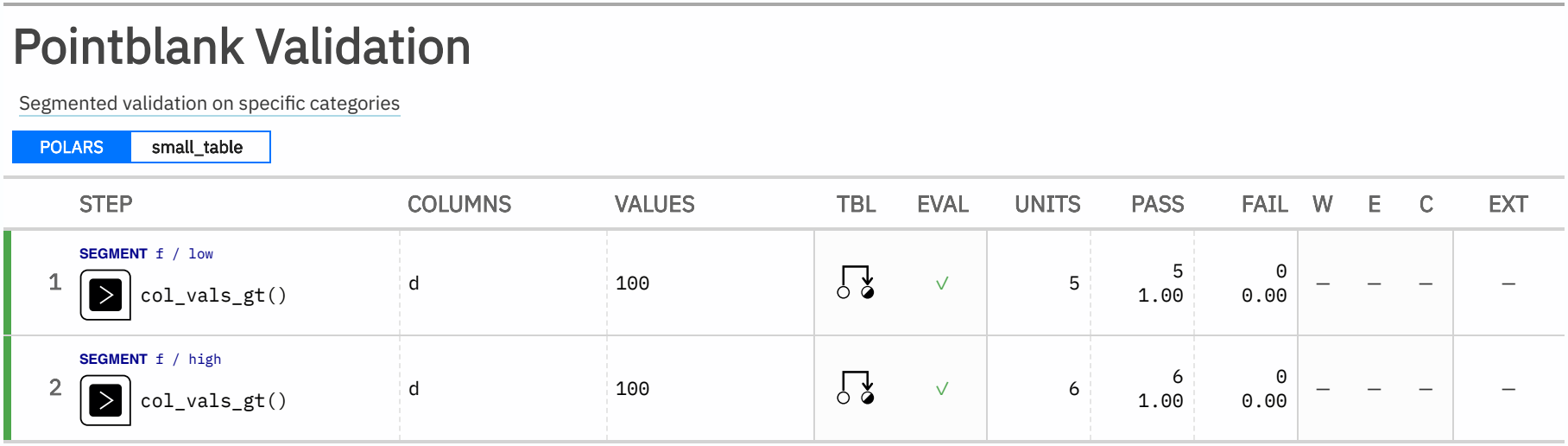
	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	SEGMENT f / high col_vals_gt()	d	100		✓	6	6 1.00	0 0.00	—	—	—	—
2	SEGMENT f / low col_vals_gt()	d	100		✓	5	5 1.00	0 0.00	—	—	—	—
3	SEGMENT f / mid col_vals_gt()	d	100		✓	2	2 1.00	0 0.00	—	—	—	—

In the validation report, notice that instead of a single validation step, we have multiple steps: one for each unique value in the `f` column. The segmentation is clearly indicated in the `STEP` column with labels like `SEGMENT f / high`, making it easy to identify which segment each validation result belongs to. This clear labeling helps when reviewing reports, especially with complex validations that use multiple segmentation criteria.

1.28 Segmenting on Specific Values

Sometimes you don't want to segment on all unique values in a column, but only on specific ones of interest. You can do this by providing a tuple with the column name and a list of values:

```
validation_2 = (
    pb.Validate(
        data=pb.load_dataset(),
        tbl_name="small_table",
        label="Segmented validation on specific categories"
    )
    .col_vals_gt(
        columns="d",
        value=100,
        segments=("f", ["low", "high"])  # Only segment on "low" and "high" values in column `f`
    )
    .interrogate()
)

validation_2
```



In this example, we only create validation steps for the `"low"` and `"high"` segments, ignoring any rows with `f` equal to `"mid"`.

1.29 Multiple Segmentation Criteria

For more complex segmentation, you can provide a list of columns or column-value tuples. This creates segments based on combinations of criteria:

```
validation_3 = (
    pb.Validate(
        data=pb.load_dataset(),
        tbl_name="small_table",
        label="Multiple segmentation criteria"
    )
    .col_vals_gt(
        columns="d",
        value=100,

        # Segment by values in `f` AND specific values in `a` ---
        segments=["f", ("a", [1, 2])]
    )
    .interrogate()
)

validation_3
```

Pointblank Validation

Multiple segmentation criteria

POLARS | small_table

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	SEGMENT f / high col_vals_gt()	d	100		✓	6	6 1.00	0 0.00	—	—	—	—
2	SEGMENT f / low col_vals_gt()	d	100		✓	5	5 1.00	0 0.00	—	—	—	—
3	SEGMENT f / mid col_vals_gt()	d	100		✓	2	2 1.00	0 0.00	—	—	—	—
4	SEGMENT a / 1 col_vals_gt()	d	100		✓	1	1 1.00	0 0.00	—	—	—	—
5	SEGMENT a / 2 col_vals_gt()	d	100		✓	3	3 1.00	0 0.00	—	—	—	—

This creates validation steps for each combination of values in column `f` and the specified values in column `a`.

1.30 Segmentation with Preprocessing

You can combine segmentation with preprocessing for powerful and flexible validations. All preprocessing is applied before segmentation occurs, which means you can create derived columns to segment on:

```
import polars as pl

# Define preprocessing function for creating a categorical column
def add_d_category_column(df):
    return df.with_columns(
        d_category=pl.when(pl.col("d") > 150).then(pl.lit("high")).otherwise(pl.lit("low"))
    )

validation_4 = (
    pb.Validate(
        data=pb.load_dataset(tbl_type="polars"),
        tbl_name="small_table",
        label="Segmentation with preprocessing",
    )
    .col_vals_gt(
        columns="d", value=100,

        # Create a column containing categorical values ---
        pre=add_d_category_column,

        # Segment by the computed column `d_category` generated via `pre=` ---
        segments="d_category",
    )
    .interrogate()
)

validation_4
```

Pointblank Validation

Segmentation with preprocessing

POLARS | small_table

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	SEGMENT d_category / high col_vals_gt()	d	100		✓	12	12 1.00	0 0.00	—	—	—	—
2	SEGMENT d_category / low col_vals_gt()	d	100		✓	1	1 1.00	0 0.00	—	—	—	—

In this example, we first create a derived column `d_category` based on whether `d` is greater than `150`. Then, we segment our validation based on this derived column by using `segments="d_category"`.

1.31 When to Use Segmentation

Segmentation is particularly useful when:

1. Data quality standards vary by group: different regions, product lines, or customer segments might have different acceptable thresholds
2. Identifying problem areas: segmentation helps pinpoint exactly where data quality issues exist, rather than just knowing that some issue exists somewhere in the data
3. Generating detailed reports: by segmenting, you get more granular reporting that can be shared with different stakeholders responsible for different parts of the data
4. Tracking improvements over time: segmented validations make it easier to see if data quality is improving in specific areas that were previously problematic

By using segmentation strategically in these scenarios, you can transform your data validation from a simple pass/fail system into a much more nuanced diagnostic tool that provides actionable insights about data quality across different dimensions. This targeted approach not only helps identify issues more precisely but also enables more effective communication of data quality metrics to relevant stakeholders.

1.32 Segmentation vs. Multiple Validation Steps

So why use segmentation instead of just creating separate validation steps for each segment using filtering in the `pre=` argument? Well, segmentation offers several nice advantages:

1. Conciseness: you define your validation logic once, not repeatedly for each segment
2. Consistency: we can be certain that the same validation is applied uniformly across segments
3. Clarity: the validation report will clearly organize results by segment (with extra labeling)
4. Convenience: there's no need to manually extract and filter subsets of your data

Segmentation can end of simplifying your validation code while also providing more structured and informative reporting about different portions of your data.

1.33 Practical Example: Validating Sales Data by Region and Product Type

Let's see a more realistic example where we validate sales data segmented by both region and product type:

```
import pandas as pd
import numpy as np

# Create a sample sales dataset
np.random.seed(123)

# Create a simple sales dataset
sales_data = pd.DataFrame({
    "region": np.random.choice(["North", "South", "East", "West"], 100),
    "product_type": np.random.choice(["Electronics", "Clothing", "Food"], 100),
    "units_sold": np.random.randint(5, 100, 100),
    "revenue": np.random.uniform(100, 10000, 100),
    "cost": np.random.uniform(50, 5000, 100)
})

# Calculate profit
sales_data["profit"] = sales_data["revenue"] - sales_data["cost"]
sales_data["profit_margin"] = sales_data["profit"] / sales_data["revenue"]

# Preview the dataset
pb.preview(sales_data)
```

PANDAS ROWS 100 COLUMNS 7

	region *object*	product_type *object*	units_sold *int64*	revenue *float64*	cost *float64*	profit *float64*	profit_margin *float64*
1	East	Clothing	55	8428.654356103547	1363.5197435071943	7065.134612596353	0.8382280627607168
2	South	Electronics	7	6589.7066024003025	3824.069456121553	2765.6371462787497	0.41969048292246663
3	East	Food	23	4680.5819759229435	4122.54515636936	558.0368195535839	0.11922381071929566
4	East	Clothing	51	5693.611988153584	1797.3122335569797	3896.2997545966045	0.6843282897927435
5	North	Clothing	50	4296.763518753258	4872.448283639371	-575.684764886113	-0.13398102138354426
96	West	Clothing	85	6551.261354681658	936.7119894981438	5614.549365183515	0.8570180704470368
97	South	Electronics	29	9543.579639173184	2779.779531480257	6763.800107692927	0.7087277901396456
98	East	Food	20	4822.302251263769	2833.48720726181	1988.815044001959	0.41242023837903463
99	North	Clothing	54	8801.046116310079	2185.8559620190636	6615.1901542910155	0.7516368016788095
100	North	Clothing	85	7942.857049695305	1834.7969383843642	6108.060111310941	0.7690003827458094

Now, let's validate that profit margins are above 20% across different regions and product types:

```
validation_5 = (
    pb.Validate(
        data=sales_data,
        tbl_name="sales_data",
        label="Sales data validation by region and product"
    )
    .col_vals_gt(
        columns="profit_margin",
        value=0.2,
        segments=["region", "product_type"],
        brief="Profit margin > 20% check"
    )
    .interrogate()
)

validation_5
```

Pointblank Validation

Sales data validation by region and product

PANDAS | sales_data

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	**SEGMENT** region / East col_vals_gt() Profit margin > 20% check	profit_margin	0.2		√	30	20 0.67	10 0.33	—	—	—	CSV
2	**SEGMENT** region / North col_vals_gt() Profit margin > 20% check	profit_margin	0.2		√	25	17 0.68	8 0.32	—	—	—	CSV
3	**SEGMENT** region / South col_vals_gt() Profit margin > 20% check	profit_margin	0.2		√	21	18 0.86	3 0.14	—	—	—	CSV
4	**SEGMENT** region / West col_vals_gt() Profit margin > 20% check	profit_margin	0.2		√	24	16 0.67	8 0.33	—	—	—	CSV
5	**SEGMENT** product_type / Clothing col_vals_gt() Profit margin > 20% check	profit_margin	0.2		√	38	28 0.74	10 0.26	—	—	—	CSV
6	**SEGMENT** product_type / Electronics col_vals_gt() Profit margin > 20% check	profit_margin	0.2		√	33	21 0.64	12 0.36	—	—	—	CSV
7	**SEGMENT** product_type / Food col_vals_gt() Profit margin > 20% check	profit_margin	0.2		√	29	22 0.76	7 0.24	—	—	—	CSV

This validation gives us a detailed breakdown of profit margin performance across the different regions and product types, making it easy to identify areas that need attention.

1.34 Best Practices for Segmentation

Effective data segmentation requires thoughtful planning about how to divide your data in ways that make sense for your validation needs. When implementing segmentation in your data validation workflow, consider these key principles:

1. Choose meaningful segments: select segmentation columns that align with your business logic and organizational structure
2. Use preprocessing when needed: if your raw data doesn't have good segmentation columns, create them through preprocessing (with the `pre=` argument)
3. Combine with actions: for critical segments, define segment-specific actions using the `actions=` parameter to respond to validation failures.

By implementing these best practices, you'll create more targeted, maintainable, and actionable data validations. Segmentation becomes most powerful when it aligns with natural divisions in your data and analytical processes, allowing for more precise identification of quality issues while maintaining a unified validation framework.

1.35 Conclusion

Data segmentation can make your validations more targeted and informative. By dividing your data into meaningful segments, you can identify quality issues with greater precision, apply appropriate validation standards to different parts of your data, and generate more actionable reports.

The `segments=` parameter transforms validation from a monolithic process into a granular assessment of data quality across various dimensions of your dataset. Whether you're dealing with regional differences, product categories, time periods, or any other meaningful divisions in your data, segmentation makes it possible to validate each portion according to its specific requirements while maintaining the simplicity of a unified validation framework.

Thresholds are a key concept in Pointblank that allow you to define acceptable limits for failing validation tests. Rather than a simple pass/fail model, thresholds enable you to signal failure at different severity levels ('warning', 'error', and 'critical'), giving you fine-grained control over how data quality issues are reported and handled.

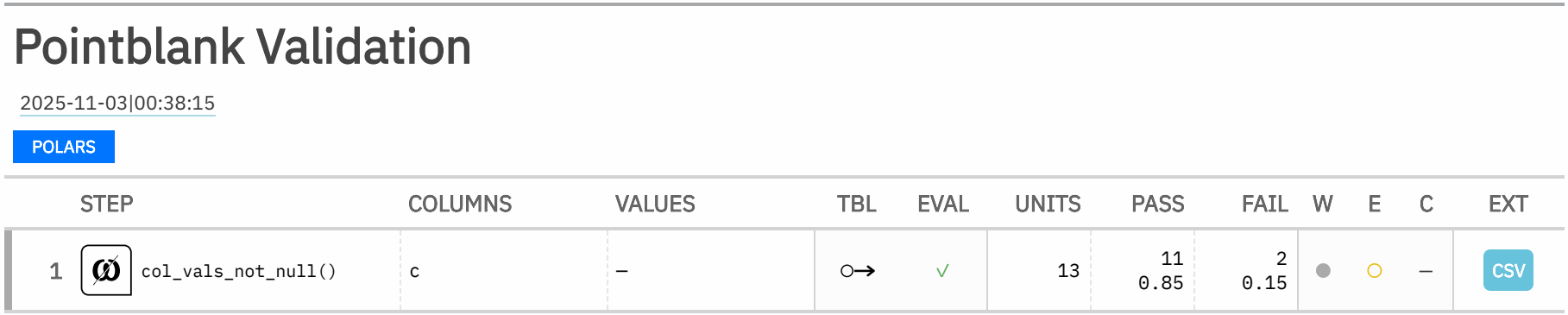
When used with actions (covered in the next section), thresholds create a robust system for responding to data quality issues based on their severity. This approach allows you to:

- set different tolerance levels for different types of validation checks
- escalate responses based on the severity of data quality issues
- configure different notification strategies for different threshold levels
- create a more nuanced data validation workflow than simple pass/fail tests

1.36 A Simple Example

Let's start with a basic example that demonstrates how thresholds work in practice:

```
import pointblank as pb

(
    pb.Validate(data=pb.load_dataset(dataset="small_table", tbl_type="polars"))
    .col_vals_not_null(
        columns="c",

        # Set thresholds for the validation step ---
        thresholds=pb.Thresholds(warning=1, error=0.2)
    )
    .interrogate()
)
```



Pointblank Validation

2025-11-03|00:38:15

POLARS

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_not_null()	c	–	○→	✓	13	11 0.85	2 0.15	●	○	–	CSV

In this example, we're validating that column `c` contains no Null values. We've set:

- A 'warning' threshold of `1` (triggers when 1 or more values are Null)
- An 'error' threshold of `0.2` (triggers when 20% or more values are Null)

Looking at the results:

- the `FAIL` column shows that 2 test units have failed
- the `W` column (for 'warning') shows a filled gray circle, indicating the warning threshold has been exceeded
- the `E` column (for 'error') shows an open yellow circle, indicating the error threshold has not been exceeded
- the `C` column (for 'critical') shows a dash since we didn't set a critical threshold

1.37 Types of Threshold Values

Thresholds in Pointblank can be specified in two different ways:

1.37.1 Absolute Thresholds

Absolute thresholds are specified as integers and represent a fixed number of failing test units:

```
# Warning threshold of exactly 5 failing test units
thresholds_absolute = pb.Thresholds(warning=5)
```

With this configuration, the 'warning' threshold would be triggered if 5 or more test units fail.

1.37.2 Proportional Thresholds

Proportional thresholds are specified as decimals between 0 and 1, representing a percentage of the total test units:

```
# Error threshold of 10% of test units failing
thresholds_proportional = pb.Thresholds(error=0.1)
```

With this configuration, the 'error' threshold would be triggered if 10% or more of the test units fail.

1.37.3 Boolean Shorthand

For cases where you want to allow exactly 1 failing test unit, you can use `True` as a convenient shorthand:

```
# Critical threshold of exactly 1 failing test unit
thresholds_boolean = pb.Thresholds(critical=True)
```

This is equivalent to setting `critical=1` but provides a more intuitive way to express "allow at most one failure". This shorthand is particularly useful for strict validations where any failure beyond a single edge case should trigger immediate attention.

1.38 Understanding Severity Levels

The three threshold levels in Pointblank ('warning', 'error', and 'critical') are inspired by traditional logging levels used in software development. These names suggest a progression of severity:

- **'warning'** (level `30`): indicates potential issues that don't necessarily prevent normal operation
- **'error'** (level `40`): suggests more serious problems that might impact data quality
- **'critical'** (level `50`): represents the most severe issues that likely require immediate attention

These numerical values (`30`, `40`, `50`) are used internally by Pointblank when determining threshold hierarchy and can be accessed through the `{level_num}` field in action metadata (covered in the next **User Guide** article).

While these names imply certain severity levels, they're ultimately just convenient labels for different thresholds. You have complete flexibility in how you use them:

- you could use 'warning' for issues that should block a pipeline
- you might configure 'critical' for minor issues that just need documentation
- the 'error' level could trigger informational emails rather than actual error handling

The naming is primarily a suggestion to help organize your validation strategy. What matters most is how you configure actions for each threshold level to suit your specific data quality requirements.

1.39 Threshold Behavior

It's important to understand a few key behaviors of thresholds:

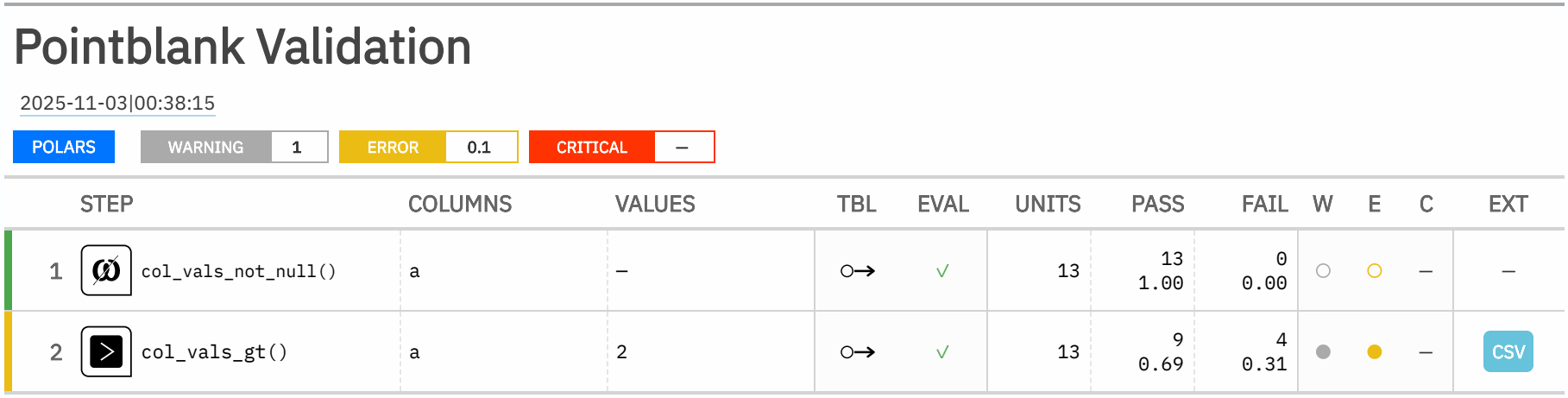
- thresholds are **inclusive**: a value equal to or exceeding the threshold will trigger the associated level
- thresholds can be **mixed**: you can use absolute values for some levels and proportional for others

- threshold levels are **hierarchical**: 'critical' is more severe than 'error', which is more severe than 'warning'
- when a test fails, **all** applicable threshold levels are marked in the report (though actions may only execute for the highest level by default)

1.40 Setting Global Thresholds

You can set thresholds globally for all validation steps in a workflow using the `thresholds=` parameter in `Validate`:

```
(
    pb.Validate(
        data=pb.load_dataset(dataset="small_table", tbl_type="polars"),

        # Setting thresholds for all validation steps ---
        thresholds=pb.Thresholds(warning=1, error=0.1)
    )
    .col_vals_not_null(columns="a")
    .col_vals_gt(columns="a", value=2)
    .interrogate()
)
```



Pointblank Validation

2025-11-03|00:38:15

POLARS | WARNING 1 | ERROR 0.1 | CRITICAL –

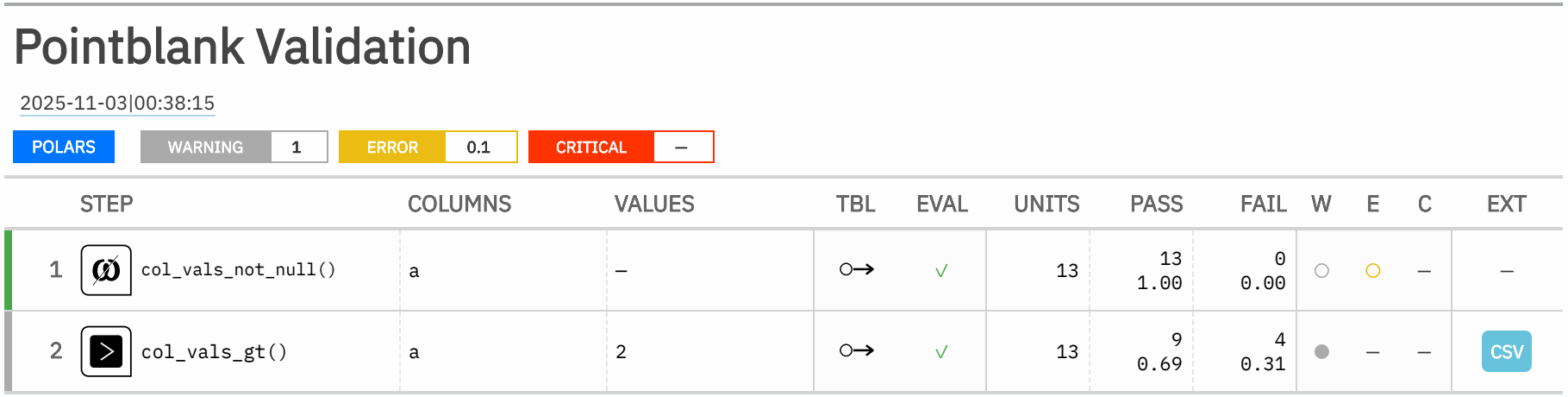
	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_not_null()	a	–	○→	√	13	13 1.00	0 0.00	○	○	–	–
2	col_vals_gt()	a	2	○→	√	13	9 0.69	4 0.31	●	●	–	CSV

With this approach, the same thresholds are applied to every validation step in the workflow.

1.41 Overriding Thresholds for Specific Steps

You can override global thresholds for specific validation steps by providing the `thresholds=` parameter in individual validation methods:

```
(
    pb.Validate(
        data=pb.load_dataset(dataset="small_table", tbl_type="polars"),

        # Setting global thresholds ---
        thresholds=pb.Thresholds(warning=1, error=0.1)
    )
    .col_vals_not_null(columns="a")
    .col_vals_gt(
        columns="a", value=2,

        # Step-specific threshold that overrides global ---
        thresholds=pb.Thresholds(warning=3)
    )
    .interrogate()
)
```



Pointblank Validation

2025-11-03|00:38:15

POLARS | WARNING 1 | ERROR 0.1 | CRITICAL —

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_not_null()	a	—	○→	√	13	13 1.00	0 0.00	○	○	—	—
2	col_vals_gt()	a	2	○→	√	13	9 0.69	4 0.31	●	—	—	CSV

In this example, the second validation step uses its own 'warning' threshold of `3`, overriding the global setting of `1`.

1.42 Ways to Define Thresholds

Pointblank offers multiple ways to define thresholds to accommodate different coding styles and requirements.

1.42.1 1. Using the `Thresholds` Class (Recommended)

The most explicit and flexible approach is using the `Thresholds` class:

```
# Set individual thresholds for different levels
thresholds_all_levels = pb.Thresholds(warning=0.05, error=0.1, critical=0.25)

# Set only specific levels
thresholds_error_only = pb.Thresholds(error=0.15)
```

This approach allows you to:

- set any combination of threshold levels
- use descriptive parameter names for clarity
- skip levels you don't need to set

1.42.2 2. Using a Tuple

For concise code, you can use a tuple where positions represent 'warning', 'error', and 'critical' levels in that order:

```
# (warning, error, critical)
thresholds_tuple = (1, 0.1, 0.25)

# Shorter tuples are also allowed
thresholds_tuple_warning = (3,)            # Only the 'warning' threshold
thresholds_tuple_warning_error = (3, 0.2)  # Both 'warning' and 'error' thresholds
```

While concise, this approach requires you to start with the 'warning' level and add levels in order.

1.42.3 3. Using a Dictionary

You can also use a dictionary with keys that match the threshold level names:

```
# Can use any combination of threshold levels
thresholds_dict = {"warning": 1, "critical": 0.15}
```

The dictionary must use the exact keys `"warning"`, `"error"`, and/or `"critical"`.

1.42.4 4. Using a Single Value

The simplest approach is using a single numeric value, which sets just the ‘warning’ threshold:

```
# Sets 'warning' threshold to `5`
thresholds_single = 5
```

This is equivalent to `pb.Thresholds(warning=5)`.

1.43 Thresholds and Validation Steps

Let’s look at a more complete validation workflow that demonstrates different threshold configurations:

```python
# Create a validation workflow with global and step-specific thresholds
(
    pb.Validate(
        data=pb.load_dataset(dataset="small_table", tbl_type="polars"),

        # Global thresholds applied to all steps unless overridden ---
        thresholds=pb.Thresholds(warning=0.05, error=0.1, critical=0.2)
    )

    # Step 1: Uses global thresholds ---
    .col_vals_not_null(columns="b")

    # Step 2: Overrides with step-specific thresholds ---
    .col_vals_gt(
        columns="a", value=2,
        thresholds=pb.Thresholds(warning=1, critical=0.3) # No 'error' threshold
    )

    # Step 3: Uses a simplified tuple notation ---
    .col_vals_not_null(columns="c", thresholds=(2, 0.15))

    .interrogate()
)
```

Pointblank Validation

2025-11-03|00:38:15

POLARS | WARNING 0.05 | ERROR 0.1 | CRITICAL 0.2

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_not_null()	b	–	○→	√	13	13 1.00	0 0.00	○	○	○	–
2	col_vals_gt()	a	2	○→	√	13	9 0.69	4 0.31	●	–	●	CSV
3	col_vals_not_null()	c	–	○→	√	13	11 0.85	2 0.15	●	●	–	CSV

1.44 Thresholds and Actions

While thresholds by themselves provide visual indicators of validation severity in reports, their real power emerges when combined with Actions. The Actions system (covered in the next article) allows you to specify what happens when a threshold is exceeded.

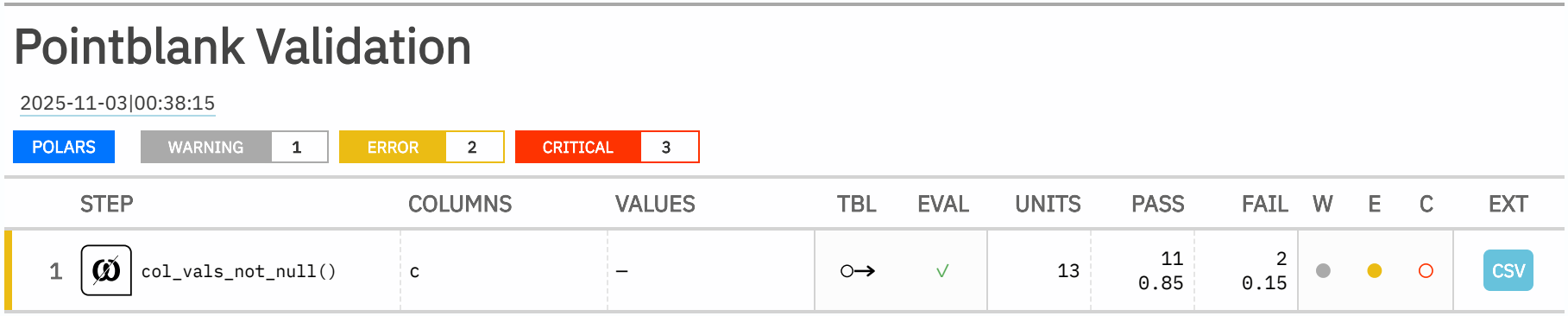
For example, you might configure:

- A 'warning' threshold that logs a message
- An 'error' threshold that sends an email notification
- A 'critical' threshold that blocks a data pipeline

Here's a simple preview of how thresholds and actions work together:

```
(
    pb.Validate(
        data=pb.load_dataset(dataset="small_table", tbl_type="polars"),

        # Define thresholds for all three severity levels ---
        thresholds=pb.Thresholds(warning=1, error=2, critical=3),

        # Define actions for different threshold levels ---
        actions=pb.Actions(
            warning="Warning: {step} has {FAIL} failing values",
            error="ERROR: Step {step} exceeded the 'error' threshold",
            critical="CRITICAL: Data quality issue in column {col}"
        )
    )
    .col_vals_not_null(columns="c")
    .interrogate()
)
```

```
ERROR: Step 1 exceeded the 'error' threshold
```



1.45 Conclusion

Thresholds are a powerful feature that transform Pointblank from a simple validation tool into a sophisticated data quality monitoring system. By setting appropriate thresholds, you can:

1. Define different severity levels for data quality issues
2. Customize tolerance levels for different types of validation checks
3. Create a more nuanced approach to data validation than binary pass/fail
4. Enable targeted actions based on the severity of issues detected

In the next article, we'll explore the Actions system in depth, showing you how to define automatic responses when thresholds are exceeded.

Actions transform data validation from passive reporting to active response by automatically executing code when quality issues arise. They bridge the gap between detection and intervention, enabling immediate notifications and comprehensive logging when thresholds are exceeded.

Whether you need simple console messages for interactive analysis or complex alerting for production pipelines, Actions provide the framework to make your validation workflows responsive. For example, when validating revenue values, you can configure immediate alerts if failures exceed acceptable thresholds, ensuring data issues are addressed promptly rather than discovered later.

In this article, we'll explore how to use Actions to respond to threshold violations during data validation, and Final Actions to execute code after all validation steps are complete, giving you powerful tools to monitor, alert, and report on your data's quality.

1.46 How Actions Work

Let's look at an example on how this works in practice. The following validation plan contains a single step (using `Validate.col_vals_gt()`) where the `thresholds=` and `actions=` parameters are set using `Thresholds` and `Actions` calls:

```
import pointblank as pb

(
    pb.Validate(data=pb.load_dataset(dataset="small_table"))
    .col_vals_gt(
        columns="c", value=2,
        thresholds=pb.Thresholds(warning=1, error=5),

        # Emit a console message when the warning threshold is exceeded ---
        actions=pb.Actions(warning="WARNING: failing test found.")
    )
    .interrogate()
)
```

```
WARNING: failing test found.
```

Pointblank Validation

2025-11-03|00:38:15

POLARS

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_gt()	c	2	○→	√	13	10 0.77	3 0.23	●	○	—	CSV

The code uses `thresholds=pb.Thresholds(warning=1, error=5)` to set a 'warning' threshold of `1` and an 'error' threshold of `5` failing test units. The results part of the validation table shows that:

- The `FAIL` column shows that 3 tests units have failed

- The `W` column (short for 'warning') shows a filled gray circle indicating it's reached its threshold level
- The `E` ('error') column shows an open yellow circle indicating it's below the threshold level

More importantly, the text `"WARNING: failing test found."` has been emitted. Here it appears above the validation table and that's because the action is executed eagerly during interrogation (before the report has even been generated).

So, an action is executed for a particular condition (e.g., 'warning') within a validation step if these three things are true:

1. there is a threshold set for that condition (either globally, or as part of that step)
2. there is an associated action set for the condition (again, either set globally or within the step)
3. during interrogation, the threshold value for the condition was exceeded by the number or proportion of failing test units

There is a lot of flexibility for setting both thresholds and actions and everything here is considered optional. Put another way, you can set various thresholds and various actions as needed and the interrogation phase will determine whether all the requirements are met for executing an action.

1.47 Defining Actions

Actions can be defined in several ways, providing flexibility for different notification needs.

1.47.1 Using String Messages

There are a few options in how to define the actions:

1. **String**: a message to be displayed in the console
2. **Callable**: a function to be called
3. **List of Strings/Callables**: for execution of multiple messages or functions

The actions are executed at interrogation time when the threshold level assigned to the action is exceeded by the number or proportion of failing test units. When providing a string, it will simply be printed to the console. A callable will also be executed at the time of interrogation. If providing a list of strings or callables, each item in the list will be executed in order. Such a list can contain a mix of strings and callables.

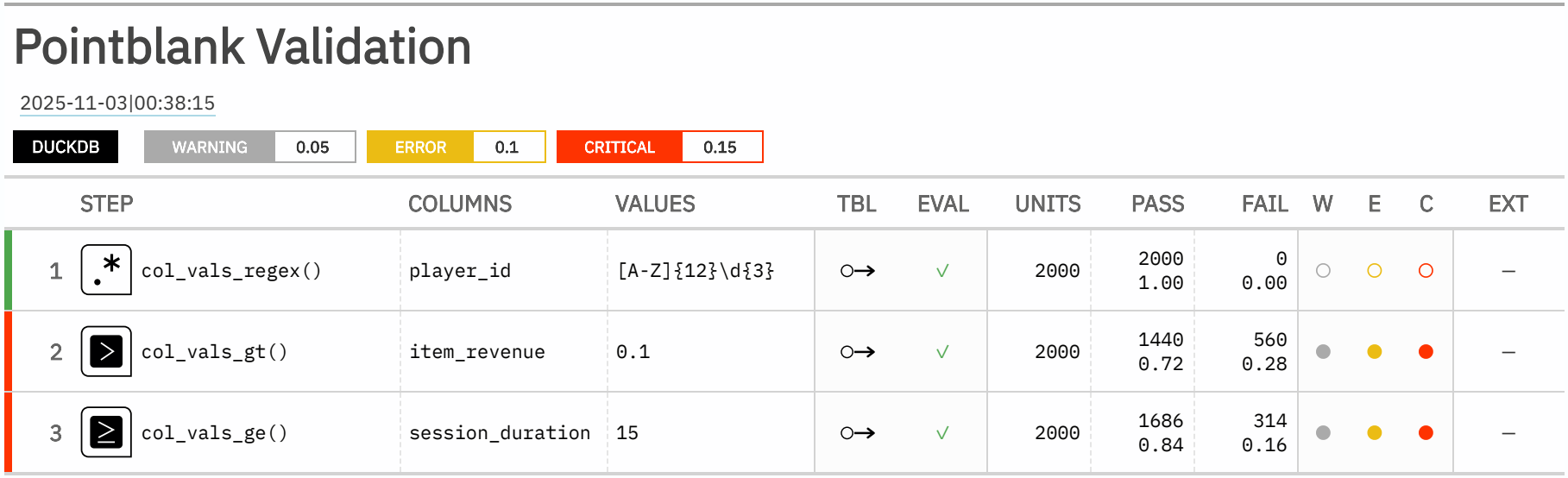
Displaying console messages may be a simple approach, but it is effective. And the strings don't have to be static, there are templating features that can be useful for constructing strings for a variety of situations. The following placeholders are available for use:

- `{type}`: The validation step type where the action is executed (e.g., 'col_vals_gt', etc.)
- `{level}`: The threshold level where the action is executed ('warning', 'error', or 'critical')
- `{step}` or `{i}`: The step number in the validation workflow where the action is executed
- `{col}` or `{column}`: The column name where the action is executed
- `{val}` or `{value}`: An associated value for the validation method
- `{time}`: A datetime value for when the action was executed

Here's an example where we prepare a console message with a number of value placeholders (`action_str`) and use it globally at `Actions(critical=)`:

```
action_str = "[{LEVEL}: {TYPE}]: Step {step} has failed validation. ({time})"

(
    pb.Validate(
        data=pb.load_dataset(dataset="game_revenue", tbl_type="duckdb"),
        thresholds=pb.Thresholds(warning=0.05, error=0.10, critical=0.15),

        # Use `action_str` for any critical thresholds exceeded ---
        actions=pb.Actions(critical=action_str),
    )
    .col_vals_regex(columns="player_id", pattern=r"[A-Z]{12}\d{3}")
    .col_vals_gt(columns="item_revenue", value=0.10)
    .col_vals_ge(columns="session_duration", value=15)
    .interrogate()
)
```

```
[CRITICAL: COL_VALS_GT]: Step 2 has failed validation. (2025-11-03 00:38:15.775766+00:00)
[CRITICAL: COL_VALS_GE]: Step 3 has failed validation. (2025-11-03 00:38:15.794886+00:00)
```



What we get here are two messages in the console, corresponding to critical failures in steps 2 and 3. The placeholders were replaced with the correct text for the context. Note that some of the resulting text is capitalized (e.g., `"CRITICAL"`, `"COL_VALS_GT"`, etc.) and this is because we capitalized the placeholder text itself. Have a look at the documentation article of `Actions` for more details on this.

1.47.2 Using Callable Functions

Aside from strings, any callable can be used as an action value. Here's an example where we use a custom function as part of an action:

```python
def duration_issue():
    from datetime import datetime
    print(f"Data quality issue found ({datetime.now()}).")

(
    pb.Validate(
        data=pb.load_dataset(dataset="game_revenue", tbl_type="duckdb"),
        thresholds=pb.Thresholds(warning=0.05, error=0.10, critical=0.15),
    )
    .col_vals_regex(columns="player_id", pattern=r"[A-Z]{12}\d{3}")
    .col_vals_gt(columns="item_revenue", value=0.05)
    .col_vals_gt(
        columns="session_duration", value=15,

        # Use the `duration_issue()` function as an action for this step ---
        actions=pb.Actions(warning=duration_issue),
    )
    .interrogate()
)
```

```
Data quality issue found (2025-11-02 19:38:15.956767).
```

Pointblank Validation

2025-11-03|00:38:15

DUCKDB | WARNING 0.05 | ERROR 0.1 | CRITICAL 0.15

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_regex()	player_id	[A-Z]{12}\d{3}	○→	√	2000	2000 1.00	0 0.00	○	○	○	—
2	col_vals_gt()	item_revenue	0.05	○→	√	2000	1701 0.85	299 0.15	●	●	○	—
3	col_vals_gt()	session_duration	15	○→	√	2000	1675 0.84	325 0.16	●	●	●	—

In this case, the 'warning' action is set to call the user's `dq_issue()` function. This action is only executed when the 'warning' threshold is exceeded in step 3. Because all three thresholds are exceeded in that step, the 'warning' action of executing the function occurs (resulting in a message being printed to the console).

This is an example where actions can be defined locally for an individual validation step. The global threshold setting applied to all three validation steps but the step-level action only applied to step 3. You are free to mix and match both threshold and action settings at the global level (i.e., set in the `Validate` call) or at the step level. The key thing to be aware of is that step-level settings of thresholds and actions take precedence.

1.48 Accessing Context in Actions

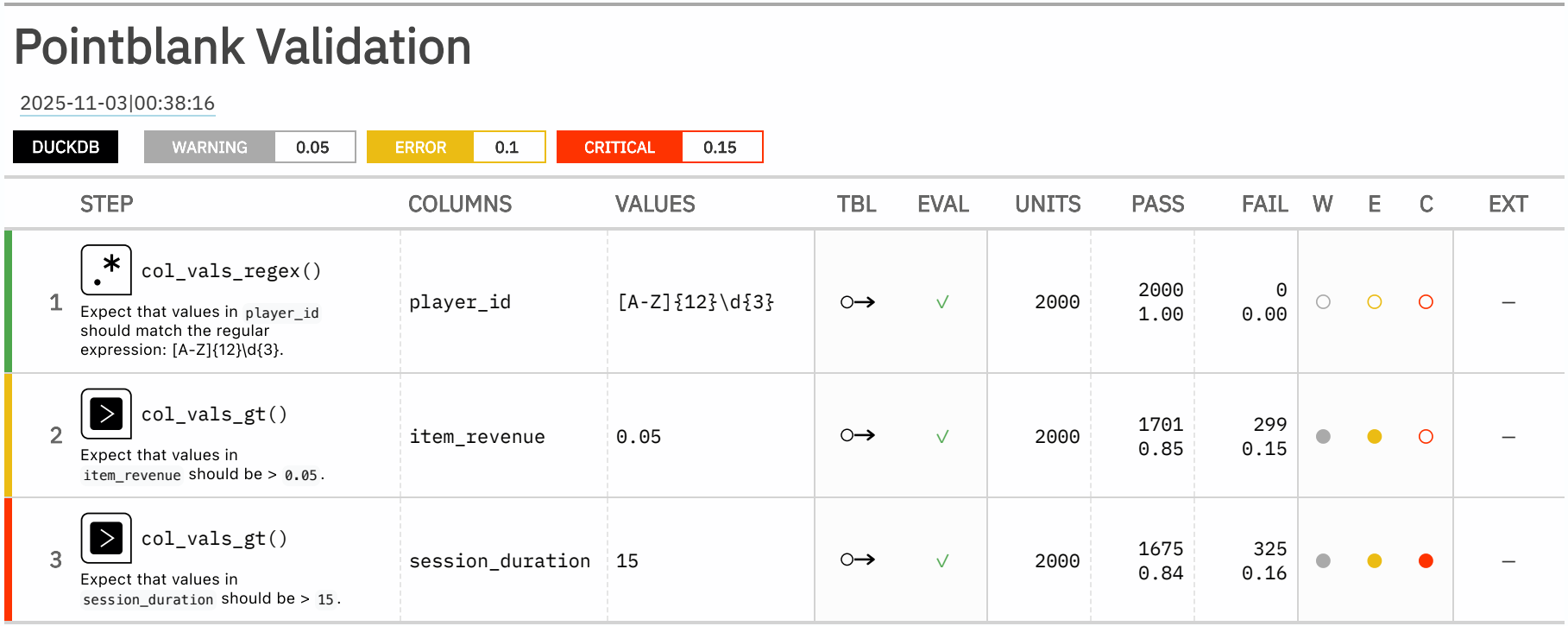
While string templates provide helpful placeholders to access information about validation steps, callable functions offer more flexibility through access to detailed metadata. When using functions as actions, you can retrieve comprehensive information about the validation context, allowing for complex logic and dynamic responses to validation issues.

1.48.1 Using `get_action_metadata()` in Callables

To access information about the validation step where an action was triggered, we can call `get_action_metadata()` in the body of a function to be used within `Actions`. This provides useful context about the validation step that triggered the action.

```
def print_problem():
    m = pb.get_action_metadata()
    print(f"{m['level']} ({m['level_num']}) for Step {m['step']}: {m['failure_text']}")

(
    pb.Validate(
        data=pb.load_dataset(dataset="game_revenue", tbl_type="duckdb"),
        thresholds=pb.Thresholds(warning=0.05, error=0.10, critical=0.15),

        # Use the `print_problem()` function as the action ---
        actions=pb.Actions(default=print_problem),
        brief=True,
    )
    .col_vals_regex(columns="player_id", pattern=r"[A-Z]{12}\d{3}")
    .col_vals_gt(columns="item_revenue", value=0.05)
    .col_vals_gt(columns="session_duration", value=15)
    .interrogate()
)
```

```
error (40) for Step 2: Exceedance of failed test units where values in `item_revenue` should have been > `0.05`.
critical (50) for Step 3: Exceedance of failed test units where values in `session_duration` should have been > `15`.
```



Pointblank Validation

2025-11-03|00:38:16

DUCKDB | WARNING 0.05 | ERROR 0.1 | CRITICAL 0.15

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_regex() Expect that values in player_id should match the regular expression: [A-Z]{12}\d{3}.	player_id	[A-Z]{12}\d{3}	○→	✓	2000	2000 1.00	0 0.00	○	○	○	–
2	col_vals_gt() Expect that values in item_revenue should be > 0.05.	item_revenue	0.05	○→	✓	2000	1701 0.85	299 0.15	●	●	○	–
3	col_vals_gt() Expect that values in session_duration should be > 15.	session_duration	15	○→	✓	2000	1675 0.84	325 0.16	●	●	●	–

In this example, we're creating a function called `print_problem()` that prints information about each validation step that fails. We then apply this function as the default action for all threshold levels using `actions=pb.Actions(default=print_problem)`. (Note that the `default=` and `highest_only=` parameters will be covered in more detail in following sections.)

We end up seeing two messages printed for failures in Steps 2 and 3. And though those steps had more than one threshold exceeded, only the most severe level in each yielded a console message (due to the default `highest_only=True` behavior).

By setting the action in `Validate(actions=)`, we applied it to all validation steps where thresholds are exceeded. This eliminates the need to set `actions=` at every validation step (though you can do this as a local override, even setting `actions=None` to disable globally set actions).

1.48.2 Available Metadata Fields

The dictionary returned by `get_action_metadata()` contains the following fields:

- `step`: The step number.
- `column`: The column name.
- `value`: The value being compared (only available in certain validation steps).
- `type`: The assertion type (e.g., `"col_vals_gt"`, etc.).
- `time`: The time the validation step was executed (in ISO format).
- `level`: The severity level (`"warning"`, `"error"`, or `"critical"`).
- `level_num`: The severity level as a numeric value (`30`, `40`, or `50`).
- `autobrief`: A localized and brief statement of the expectation for the step.
- `failure_text`: Localized text that explains how the validation step failed.

1.49 Customizing Action Behavior

The `Actions` class has two additional parameters that provide more control over how actions are executed:

1.49.1 Setting Default Actions with `default=`

Instead of specifying actions separately for each threshold level, you can use the `default=` parameter to set a common action for all levels:

```python
def log_all_issues():
    m = pb.get_action_metadata()
    print(f"[{m['level'].upper()}] Validation failed in step {m['step']} with level {m['level']}")

(
    pb.Validate(
        data=pb.load_dataset(dataset="game_revenue", tbl_type="duckdb"),
        thresholds=pb.Thresholds(warning=0.05, error=0.10, critical=0.15),

        # The `log_all_issues()` callable is set to every threshold ---
        actions=pb.Actions(default=log_all_issues),
    )
    .col_vals_regex(columns="player_id", pattern=r"[A-Z]{12}\d{3}")
    .col_vals_gt(columns="item_revenue", value=0.05)
    .col_vals_gt(columns="session_duration", value=15)
    .interrogate()
)
```

```
[ERROR] Validation failed in step 2 with level error
[CRITICAL] Validation failed in step 3 with level critical
```

Pointblank Validation

2025-11-03|00:38:16

DUCKDB | WARNING 0.05 | ERROR 0.1 | CRITICAL 0.15

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_regex()	player_id	[A-Z]{12}\d{3}	○→	✓	2000	2000 1.00	0 0.00	○	○	○	—
2	col_vals_gt()	item_revenue	0.05	○→	✓	2000	1701 0.85	299 0.15	●	●	○	—
3	col_vals_gt()	session_duration	15	○→	✓	2000	1675 0.84	325 0.16	●	●	●	—

The `default=` parameter sets the same action for all threshold levels. If you later specify an action for a specific level, it will override this default for that level only.

When using the `default=` parameter, be aware that your action (whether a string template or callable function) needs to work across all validation steps where thresholds might be exceeded. Not all validation methods provide the same context for string templates or in the metadata dictionary returned by `get_action_metadata()`.

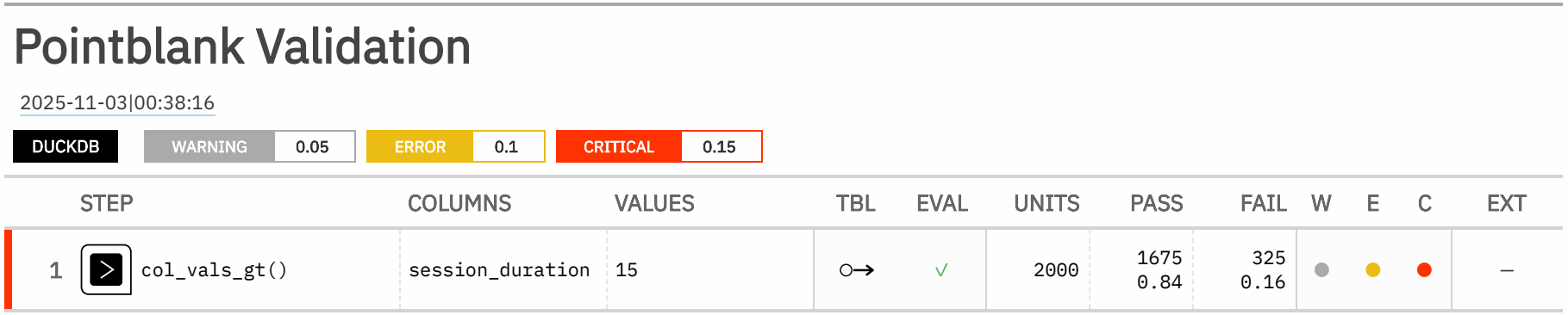
For example, some validation steps like `Validate.col_vals_gt()` provide a `value` field that can be accessed with `{value}` in string templates, while others like `Validate.col_exists()` don't have this concept. When creating default actions, either use only the universally available placeholders (`{step}`, `{level}`, `{type}`, and `{time}`), or include conditional logic in your callable functions to handle different validation types appropriately.

1.49.2 Controlling Action Execution with `highest_only=`

By default, Pointblank only executes the action for the most severe threshold level that's been exceeded. If you want actions for all exceeded thresholds to be executed, you can set `highest_only=False`:

```
(
    pb.Validate(
        data=pb.load_dataset(dataset="game_revenue", tbl_type="duckdb"),
        thresholds=pb.Thresholds(warning=0.05, error=0.10, critical=0.15),
        actions=pb.Actions(
            warning="Warning threshold exceeded in step {step}",
            error="Error threshold exceeded in step {step}",
            critical="Critical threshold exceeded in step {step}",

            # Execute all applicable actions ---
            highest_only=False
        ),
    )
    .col_vals_gt(columns="session_duration", value=15)
    .interrogate()
)
```

```
Critical threshold exceeded in step 1
Error threshold exceeded in step 1
Warning threshold exceeded in step 1
```



In this example, if all three thresholds are exceeded in a step, you'll see all three messages printed, rather than just the critical one.

The default behavior (`highest_only=True`) helps prevent notification fatigue by limiting the number of actions executed when multiple thresholds are exceeded in the same validation step. For example, if a validation step fails with 60% of rows not passing, it would exceed 'warning', 'error', and 'critical' thresholds simultaneously. With `highest_only=True`, only the critical action would execute.

You might want to set `highest_only=False` when:

- different threshold levels need to trigger different types of notifications (e.g., warnings to Slack, errors to email, critical to urgent notifications)
- you need comprehensive logging of all severity levels for audit purposes
- you're building a dashboard that displays counts of issues at each severity level

1.50 Using Multiple Actions for a Threshold

You can specify multiple actions to be executed for a single threshold level by providing a list:

```python
def send_notification():
    print("📧 Notification sent to data team")

def log_to_system():
    print("📝 Issue logged in system")

(
    pb.Validate(
        data=pb.load_dataset(dataset="game_revenue", tbl_type="duckdb"),
        thresholds=pb.Thresholds(critical=0.15),

        # Set multiple actions for the critical threshold exceedance ---
        actions=pb.Actions(
            critical=[
                "CRITICAL: Data validation failed",  # First action: display message
                send_notification,                   # Second action: call function
                log_to_system                        # Third action: call another function
            ]
        ),
    )
    .col_vals_gt(columns="session_duration", value=15)
    .interrogate()
)
```

```
CRITICAL: Data validation failed
📧 Notification sent to data team
📝 Issue logged in system
```

Pointblank Validation

2025-11-03|00:38:16

DUCKDB | WARNING — | ERROR — | CRITICAL 0.15

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	> col_vals_gt()	session_duration	15	○→	✓	2000	1675 0.84	325 0.16	—	—	●	—

When providing a list of actions, they will be executed in sequence when the threshold is exceeded. This allows you to combine different types of actions such as displaying messages, sending notifications, and logging events.

1.51 Final Actions

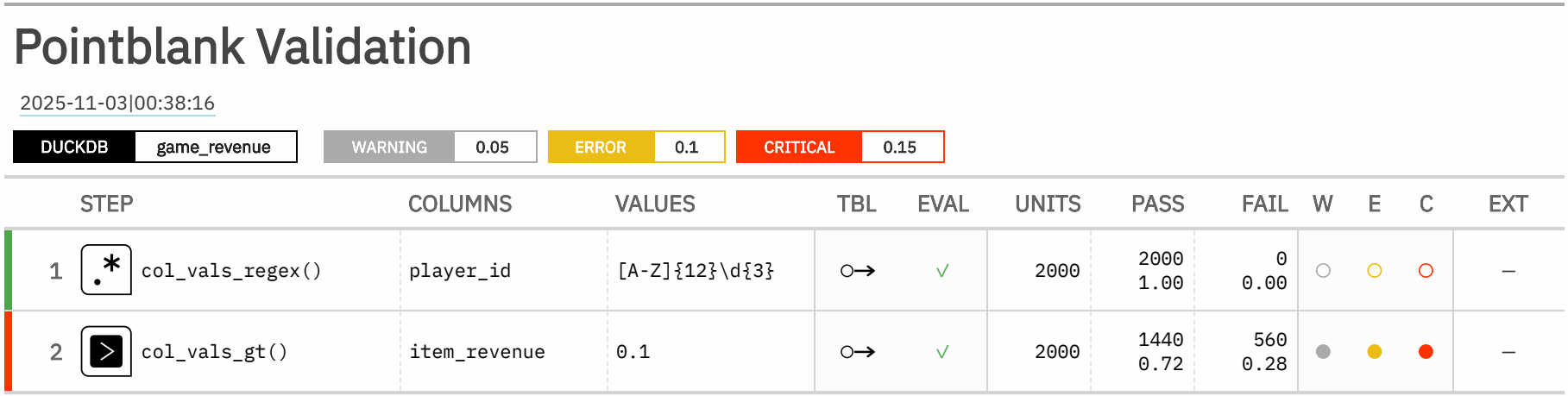
1.51.1 Creating Final Actions

When you need to execute actions after all validation steps are complete, Pointblank provides the `FinalActions` class. Unlike `Actions` which triggers on a per-step basis during the validation process, `FinalActions` executes after the entire validation is complete, giving you a way to respond to the overall validation results.

Here's how to use `FinalActions`:

```
def send_alert():
    summary = pb.get_validation_summary()
    if summary["highest_severity"] == "critical":
        print(f"ALERT: Critical validation failures found in `{summary['tbl_name']}`")

(
    pb.Validate(
        data=pb.load_dataset(dataset="game_revenue", tbl_type="duckdb"),
        tbl_name="game_revenue",
        thresholds=pb.Thresholds(warning=0.05, error=0.10, critical=0.15),

        # Set final actions to be executed after all interrogations ---
        final_actions=pb.FinalActions(
            "Validation complete.",  # 1. a string message
            send_alert               # 2. a callable function
        )
    )
    .col_vals_regex(columns="player_id", pattern=r"[A-Z]{12}\d{3}")
    .col_vals_gt(columns="item_revenue", value=0.10)
    .interrogate()
)
```

```
Validation complete.
ALERT: Critical validation failures found in `game_revenue`
```



Pointblank Validation

2025-11-03|00:38:16

DUCKDB | game_revenue

WARNING 0.05 | ERROR 0.1 | CRITICAL 0.15

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_regex()	player_id	[A-Z]{12}\d{3}	○→	✓	2000	2000 1.00	0 0.00	○	○	○	—
2	col_vals_gt()	item_revenue	0.1	○→	✓	2000	1440 0.72	560 0.28	●	●	●	—

In this example:

- We define the function `send_alert()` that checks the validation summary for critical failures
- We provide a simple string message `"Validation complete."` that will print to the console
- Both actions will execute in order after all validation steps have completed

Because the 'critical' threshold was exceeded in Step 2, we see the printed alert of `send_alert()` after the simple string message.

`FinalActions` accepts any number of actions as positional arguments. Each argument can be:

1. **String**: A message to be displayed in the console
2. **Callable**: A function to be called with no arguments
3. **List of Strings/Callables**: Multiple actions to execute in sequence

All actions will be executed in the order they are provided after all validation steps have completed.

1.51.2 Using `get_validation_summary()` in Final Actions

When creating a callable function to use with `FinalActions`, you can access information about the overall validation results using the `get_validation_summary()` function. This gives you a dictionary with comprehensive information about the validation:

```python
def comprehensive_report():
    summary = pb.get_validation_summary()
    print(f"Validation Report for {summary['tbl_name']}:")
    print(f"- Steps: {summary['n_steps']}")
    print(f"- Passing steps: {summary['n_passing_steps']}")
    print(f"- Failing steps: {summary['n_failing_steps']}")

    # Take additional actions based on results
    if summary["n_failing_steps"] > 0:

        # Create a Slack notification function ---
        notify = pb.send_slack_notification(
            webhook_url="https://hooks.slack.com/services/your/webhook/url",
            summary_msg="""
            🚨 *Validation Failure Alert*
            • Table: {tbl_name}
            • Failed Steps: {n_failing_steps} of {n_steps}
            • Highest Severity: {highest_severity}
            • Time: {time}
            """,
        )

        # Execute the notification function
        notify()

(
    pb.Validate(
        data=pb.load_dataset(dataset="game_revenue", tbl_type="duckdb"),
        tbl_name="game_revenue",
        final_actions=pb.FinalActions(comprehensive_report),
    )
    .col_vals_regex(columns="player_id", pattern=r"[A-Z]{12}\d{3}")
    .col_vals_gt(columns="item_revenue", value=0.05)
    .interrogate()
)
```

Pointblank Validation

2025-11-03|00:38:16

DUCKDB | game_revenue

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_regex()	player_id	[A-Z]{12}\d{3}	○→	✓	2000	2000 1.00	0 0.00	—	—	—	—
2	col_vals_gt()	item_revenue	0.05	○→	✓	2000	1701 0.85	299 0.15	—	—	—	—

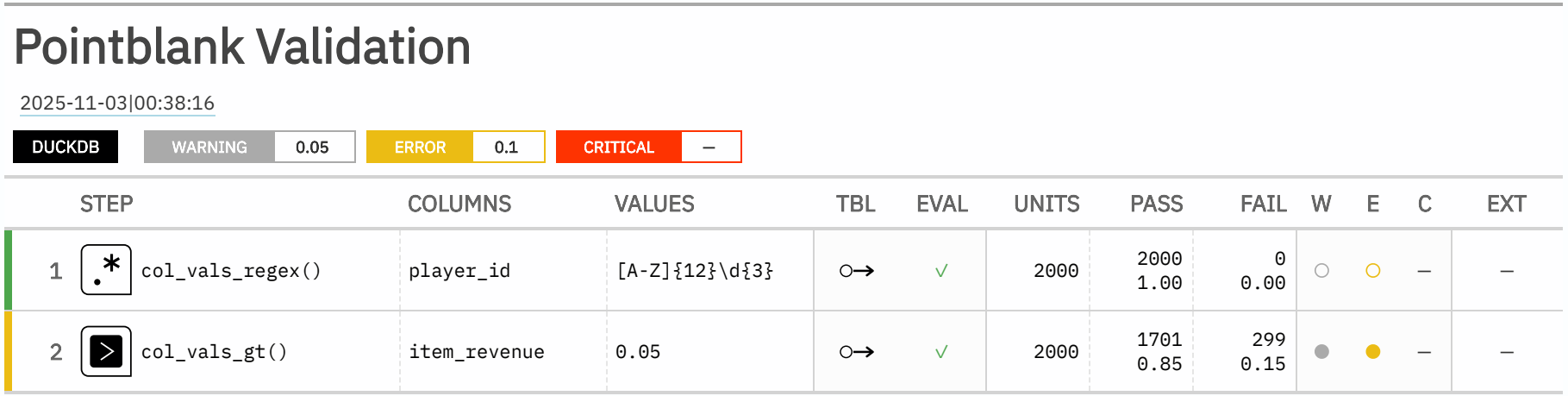
Here we used the `send_slack_notification()` function, which is available in Pointblank as a pre-built action. It can be used by itself in `final_actions=` but here it's integrated into the user's `comprehensive_report()` function to provide finer control with conditional logic.

1.51.3 Combining Step-level and Final Actions

You can use both `Actions` and `FinalActions` together for comprehensive validation control:

```python
def log_step_failure():
    m = pb.get_action_metadata()
    print(f"Step {m['step']} failed with {m['level']}")


def generate_summary():
    summary = pb.get_validation_summary()
    # Sum up total failed test units across all steps
    total_failed = sum(summary["dict_n_failed"].values())
    # Sum up total test units across all steps
    total_units = sum(summary["dict_n"].values())
    print(f"Validation complete: {total_failed} failures out of {total_units} tests")

(
    pb.Validate(
        data=pb.load_dataset(dataset="game_revenue", tbl_type="duckdb"),
        thresholds=pb.Thresholds(warning=0.05, error=0.10),

        # Set an action for each step (highest threshold exceeded) ---
        actions=pb.Actions(default=log_step_failure),

        # Set a final action to get a summary of the validation process ---
        final_actions=pb.FinalActions(generate_summary),
    )
    .col_vals_regex(columns="player_id", pattern=r"[A-Z]{12}\d{3}")
    .col_vals_gt(columns="item_revenue", value=0.05)
    .interrogate()
)
```

```
Step 2 failed with error
Validation complete: 299 failures out of 4000 tests
```



This approach allows you to:

1. log individual step failures during the validation process using `Actions`
2. generate a comprehensive report after all validation steps are complete using `FinalActions`

Using both action types gives you fine-grained control over when and how notifications and other actions are triggered in your validation workflow.

1.52 Conclusion

Actions provide a powerful mechanism for responding to data validation results in Pointblank. By combining threshold settings with appropriate actions, you can create sophisticated data quality workflows that:

- provide immediate feedback through console messages
- execute custom functions when validation thresholds are exceeded
- customize notifications based on severity levels
- generate comprehensive reports after validation is complete
- automate responses to data quality issues

The flexible design of `Actions` and `FinalActions` allows you to start simple with basic console messages and gradually build up to complex validation workflows with conditional logic, custom reporting, and integrations with other systems like Slack, email, or logging services.

When designing your validation strategy, consider leveraging both step-level actions for immediate responses and final actions for holistic reporting. This combination provides comprehensive control over your data validation process and helps ensure that data quality issues are detected, reported, and addressed efficiently.

When validating data with Pointblank, it's often helpful to have descriptive labels for each validation step. This is where *briefs* come in. A brief is a short description of what a validation step is checking and it appears in the `STEP` column of the validation report table. Briefs make your validation reports more readable and they help others understand what each step is verifying without needing to look at the code.

Briefs can be set in two ways:

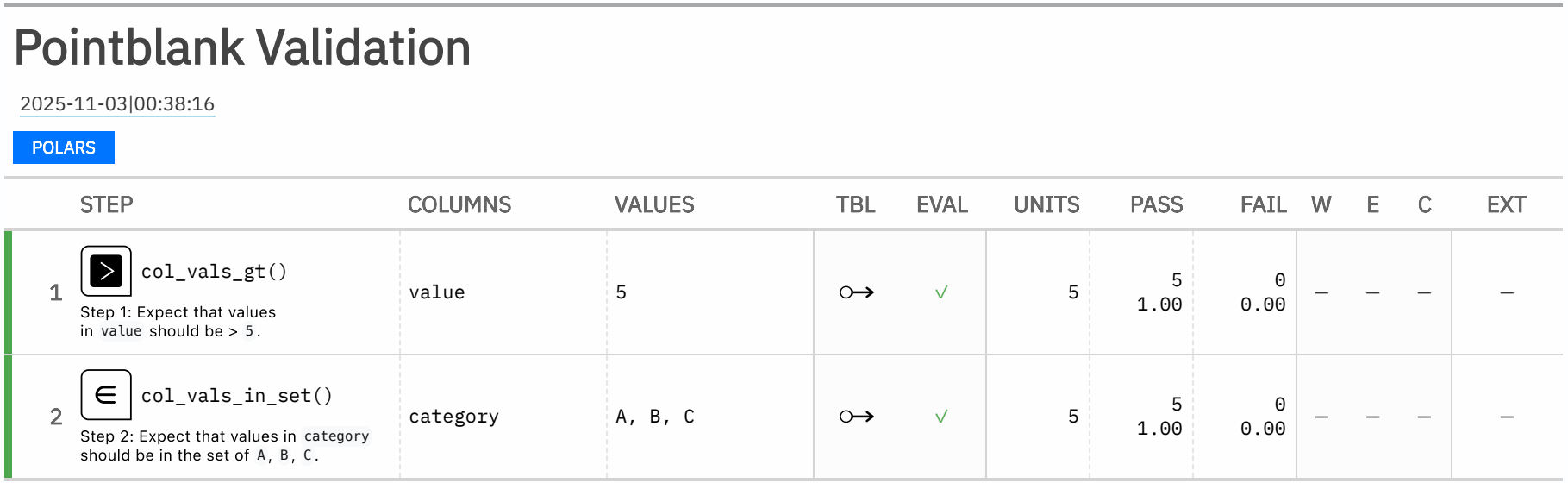
1. Globally: applied to all validation steps via the `brief=` parameter in `Validate`
2. Locally: set for individual validation steps via the `brief=` parameter in each validation method

Understanding these two approaches to adding briefs gives you flexibility in how you document your validation process. Global briefs provide consistency across all steps and save time when you want similar descriptions throughout, while step-level briefs allow for precise customization when specific validations need more detailed or unique explanations. In practice, many validation workflows will combine both approaches (i.e., setting a useful global brief template while overriding it for steps that require special attention).

1.53 Global Briefs

To set a global brief that applies to all validation steps, use the `Validate(brief=)` parameter when creating a `Validate` object:

```python
import pointblank as pb
import polars as pl

# Sample data
data = pl.DataFrame({
    "id": [1, 2, 3, 4, 5],
    "value": [10, 20, 30, 40, 50],
    "category": ["A", "B", "C", "A", "B"]
})

# Create a validation with a global brief
(
    pb.Validate(
        data=data,

        # Global brief template ---
        brief="Step {step}: {auto}"
    )
    .col_vals_gt(columns="value", value=5)
    .col_vals_in_set(columns="category", set=["A", "B", "C"])
    .interrogate()
)
```



Pointblank Validation

2025-11-03|00:38:16

POLARS

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_gt() Step 1: Expect that values in value should be > 5.	value	5	○→	✓	5	5 1.00	0 0.00	—	—	—	—
2	col_vals_in_set() Step 2: Expect that values in category should be in the set of A, B, C.	category	A, B, C	○→	✓	5	5 1.00	0 0.00	—	—	—	—

In this example, every validation step will have a brief description that follows the pattern `"Step X: [auto-generated description]"`.

This is a simple example of template-based briefs. Later in this guide, we'll explore the full range of templating elements available for creating custom brief descriptions that precisely communicate what each validation step is checking.

1.54 Step-level Briefs

You can also set briefs for individual validation steps:

```
(
    pb.Validate(data=data)
    .col_vals_gt(
        columns="value", value=5,
        brief="Check if values exceed minimum threshold of 5"
    )
    .col_vals_in_set(
        columns="category", set=["A", "B", "C"],
        brief="Verify categories are valid"
    )
    .interrogate()
)
```

Pointblank Validation

2025-11-03|00:38:16

POLARS

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_gt() Check if values exceed minimum threshold of 5	value	5	○→	√	5	5 1.00	0 0.00	—	—	—	—
2	col_vals_in_set() Verify categories are valid	category	A, B, C	○→	√	5	5 1.00	0 0.00	—	—	—	—

Local briefs override any global briefs that might be set.

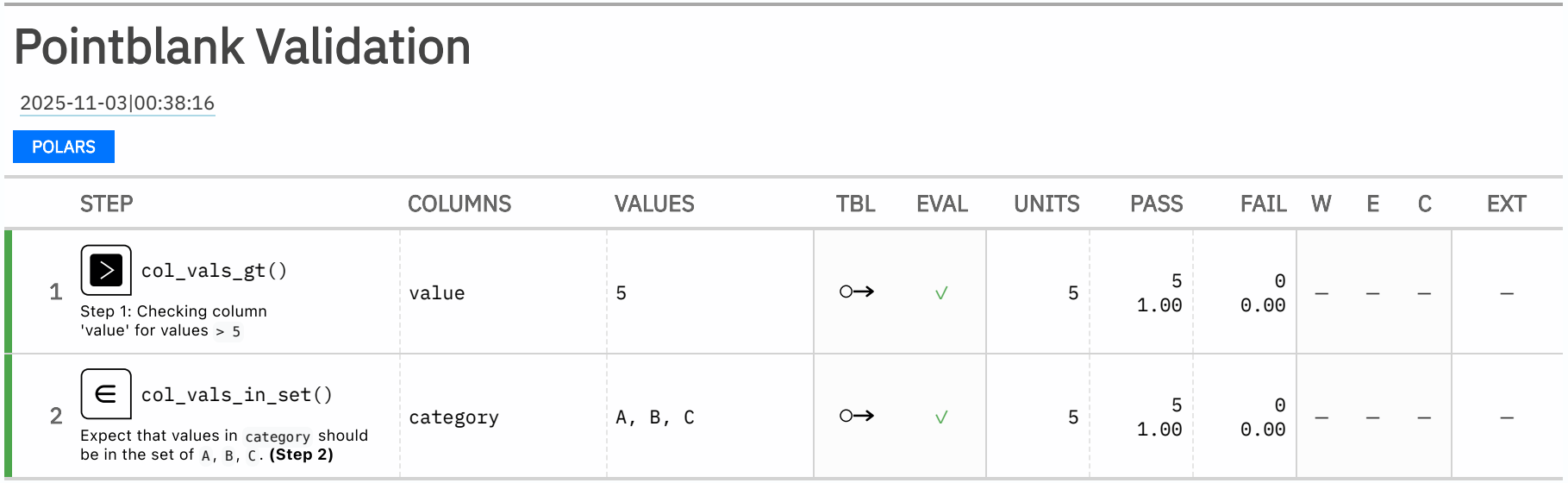
1.55 Brief Templating

Briefs support templating elements that get replaced with specific values:

- `{auto}`: an auto-generated description of the validation
- `{step}`: the step number in the validation plan
- `{col}`: the column name(s) being validated
- `{value}`: the comparison value used in the validation (when applicable)
- `{thresholds}`: a short summary of thresholds levels set (or unset) for the step
- `{segment}`, `{segment_column}`, `{segment_value}`: information on the step's segment

Here's how to use these templates:

```
(
    pb.Validate(data=data)
    .col_vals_gt(
        columns="value", value=5,
        brief="Step {step}: Checking column '{col}' for values `> 5`"
    )
    .col_vals_in_set(
        columns="category", set=["A", "B", "C"],
        brief="{auto} **(Step {step})**"
    )
    .interrogate()
)
```



Pointblank Validation

2025-11-03|00:38:16

POLARS

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_gt() Step 1: Checking column 'value' for values > 5	value	5	○→	✓	5	5 1.00	0 0.00	—	—	—	—
2	col_vals_in_set() Expect that values in category should be in the set of A, B, C. **(Step 2)**	category	A, B, C	○→	✓	5	5 1.00	0 0.00	—	—	—	—

These template elements make briefs highly flexible and customizable. You can combine multiple templating elements in a single brief to create descriptive yet concise validation step descriptions. The templates help maintain consistency across your validation reports while providing enough detail to understand what each step is checking.

Note that not all templating elements will be relevant for every validation step. For instance, `{value}` is only applicable to validation functions that hold a comparison value like `Validate.col_vals_gt()`. If you include a templating element that isn't relevant to a particular step, it will not be replaced with a corresponding value.

Briefs support the use of Markdown formatting, allowing you to add emphasis with **bold** or *italic* text, include `inline code` formatting, or other Markdown elements to make your briefs more visually distinctive and informative. This can be especially helpful when you want certain parts of your briefs to stand out in the validation report.

1.56 Automatic Briefs

If you want Pointblank to generate briefs for you automatically, you can set `brief=True`. Here, we'll make that setting at the global level (by using `Validate(brief=True)`):

```
(
    pb.Validate(
        data=data,

        # Setting for automatically generated briefs ---
        brief=True
    )
    .col_vals_gt(columns="value", value=5)
    .col_vals_in_set(columns="category", set=["A", "B", "C"])
    .interrogate()
)
```

Pointblank Validation

2025-11-03|00:38:16

POLARS

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_gt() Expect that values in value should be > 5.	value	5	○→	✓	5	5 1.00	0 0.00	—	—	—	—
2	col_vals_in_set() Expect that values in category should be in the set of A, B, C.	category	A, B, C	○→	✓	5	5 1.00	0 0.00	—	—	—	—

Automatic briefs are descriptive and include information about what's being validated, including the column names and the validation conditions.

1.57 Briefs Localized to a Specified Language

When using the `lang=` parameter in `Validate`, automatically generated briefs will be created in the specified language (along with other elements of the validation report table):

```
(
    pb.Validate(
        data=data,

        # Setting the language as Spanish ---
        lang="es",

        # Automatically generate all briefs in Spanish
        brief=True
    )
    .col_vals_gt(columns="value", value=5)
    .col_vals_in_set(columns="category", set=["A", "B", "C"])
    .interrogate()
)
```

Validación de Pointblank

2025-11-03|00:38:16

POLARS

	PASO	COLUMNAS	VALORES	TBL	EVAL	UNID.	PASA	FALLO	W	E	C	EXT
1	 col_vals_gt() Se espera que los valores en value sean > 5.	value	5	○→	✓	5	5 1,00	0 0,00	—	—	—	—
2	 col_vals_in_set() Se espera que los valores en category estén en el conjunto de A, B, C.	category	A, B, C	○→	✓	5	5 1,00	0 0,00	—	—	—	—

When using the `lang=` parameter in combination with the `{auto}` templating element, the auto-generated portion of the brief will also be translated to the specified language. This makes it possible to create fully localized validation reports where both custom text and auto-generated descriptions appear in the same language.

Pointblank supports several languages for localized briefs, including French (`"fr"`), German (`"de"`), Spanish (`"es"`), Italian (`"it"`), and Portuguese (`"pt"`). For the complete list of supported languages, refer to the `Validate` documentation.

1.58 Disabling Briefs

If you've set a global brief but want to disable it for specific validation steps, you can set `brief=False`:

```
(
    pb.Validate(
        data=data,

        # Global brief template ---
        brief="Step {step}: {auto}"
    )
    .col_vals_gt(columns="value", value=5)  # This step uses the global brief setting
    .col_vals_in_set(
        columns="category",
        set=["A", "B", "C"],

        # No brief for this step ---
        brief=False
    )
    .interrogate()
)
```

Pointblank Validation

2025-11-03|00:38:16

POLARS

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_gt() Step 1: Expect that values in `value` should be > `5`.	value	5	○→	✓	5	5 1.00	0 0.00	—	—	—	—
2	col_vals_in_set()	category	A, B, C	○→	✓	5	5 1.00	0 0.00	—	—	—	—

1.59 Practical Example: Comprehensive Validation with Briefs

In real-world data validation scenarios, you'll likely work with more complex datasets and apply various types of validation checks. This final example brings together many of the brief-generating techniques we've covered, showing how you can mix different approaches in a single validation workflow.

```
# Create a slightly larger dataset
data_2 = pl.DataFrame({
    "id": [1, 2, 3, 4, 5, 6, 7, 8],
    "value": [10, 20, 30, 40, 50, 60, 70, 80],
    "ratio": [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8],
    "category": ["A", "B", "C", "A", "B", "C", "A", "B"],
    "date": ["2023-01-01", "2023-01-02", "2023-01-03", "2023-01-04",
             "2023-01-05", "2023-01-06", "2023-01-07", "2023-01-08"]
})

(
    pb.Validate(data=data_2)
    .col_vals_gt(
        columns="value", value=0,

        # Plaintext brief ---
        brief="All values must be positive."
    )
    .col_vals_between(
        columns="ratio", left=0, right=1,

        # Template-based brief ---
        brief="**Step {step}**: Ratios should be between `0` and `1`."
    )
    .col_vals_in_set(
        columns="category", set=["A", "B", "C"],

        # Automatically generated brief ---
        brief=True
    )
    .interrogate()
)
```

Pointblank Validation

2025-11-03|00:38:16

POLARS

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_gt() All values must be positive.	value	0	○→	✓	8	8 1.00	0 0.00	—	—	—	—
2	col_vals_between() **Step 2**: Ratios should be between 0 and 1.	ratio	[0, 1]	○→	✓	8	8 1.00	0 0.00	—	—	—	—
3	col_vals_in_set() Expect that values in category should be in the set of A, B, C.	category	A, B, C	○→	✓	8	8 1.00	0 0.00	—	—	—	—

The example above demonstrates:

- plaintext briefs with direct messages
- template-based briefs with Markdown formatting
- automatically generated briefs (`brief=True`)

By combining these different brief styles, you can create validation reports that are informative, consistent, and tailored to your specific data quality requirements.

1.60 Best Practices for Using Briefs

Well-crafted briefs can significantly enhance the readability and usefulness of your validation reports. Here are some guidelines to follow:

1. Be concise: briefs should be short and to the point; they're meant to quickly communicate the purpose of a validation step
2. Be specific: include relevant details or conditions that make the validation meaningful

3. Use templates consistently: if you're using template elements like `"{step}"` or `"{col}"`, try to use them consistently across all briefs for a cleaner look

4. Use auto-generated briefs as a starting point: you can start with `Validate(brief=True)` to see what Pointblank generates automatically, then customize as needed

5. Add custom briefs for complex validations: custom briefs are especially useful for complex validations where the purpose might not be immediately obvious from the code

Following these best practices will help ensure your validation reports are easy to understand for everyone who needs to review them.

1.61 Conclusion

Briefs help make validation reports more readable and understandable. By using global briefs, step-level briefs, or a combination of both, you can create validation reports that clearly communicate what each validation step is checking.

Whether you want automatically generated descriptions or precisely tailored custom messages, the brief system provides the flexibility to make your data validation work more transparent and easier to interpret for all stakeholders.

2 Advanced Validation

While Pointblank offers many specialized validation functions for common data quality checks, sometimes you need more flexibility for complex validation requirements. This is where expression-based validation with `Validate.col_vals_expr()` comes in.

The `Validate.col_vals_expr()` method allows you to:

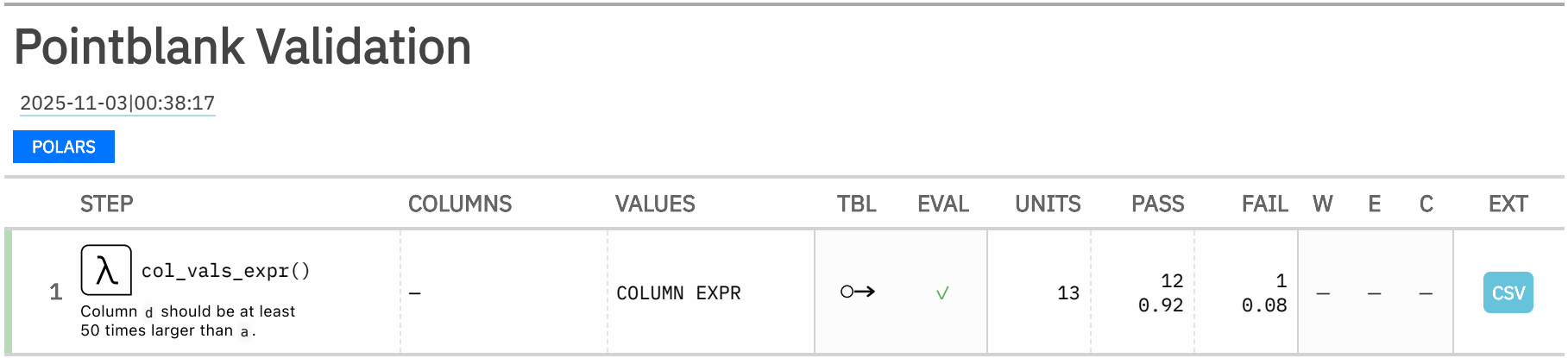
- combine multiple conditions in a single validation step
- access row-wise values across multiple columns

Now let's explore how to use these capabilities through a collection of examples!

2.1 Basic Usage

At its core, `Validate.col_vals_expr()` validates whether an expression evaluates to `True` for each row in your data. Here's a simple example:

```
import pointblank as pb
import polars as pl

# Load small_table dataset as a Polars DataFrame
small_table_pl = pb.load_dataset(dataset="small_table", tbl_type="polars")

(
    pb.Validate(data=small_table_pl)
    .col_vals_expr(

        # Use Polars expression syntax ---
        expr=pl.col("d") > pl.col("a") * 50,
        brief="Column `d` should be at least 50 times larger than `a`."
    )
    .interrogate()
)
```



In this example, we're validating that for each row, the value in column `d` is at least 50 times larger than the value in column `a`.

2.2 Notes on Expression Syntax

The expression syntax depends on your table type:

- **Polars**: uses Polars expression syntax with `pl.col("column_name")`
- **Pandas**: uses standard Python/NumPy syntax

The expression should:

- evaluate to a boolean result for each row
- reference columns using the appropriate syntax for your table type
- use standard operators (`+`, `-`, `*`, `/`, `>`, `<`, `==`, etc.)
- not include assignments

2.3 Complex Expressions

The real power of `Validate.col_vals_expr()` comes with complex expressions that would be difficult to represent using the standard validation functions:

```python
# Load game_revenue dataset as a Polars DataFrame
game_revenue_pl = pb.load_dataset(dataset="game_revenue", tbl_type="polars")

(
    pb.Validate(data=game_revenue_pl)
    .col_vals_expr(

        # Use Polars expression syntax ---
        expr=(pl.col("session_duration") > 20) | (pl.col("item_revenue") > 10),
        brief="Sessions should be either long (>20 min) or high-value (>$10)."
    )
    .interrogate()
)
```

Pointblank Validation

2025-11-03|00:38:17

POLARS

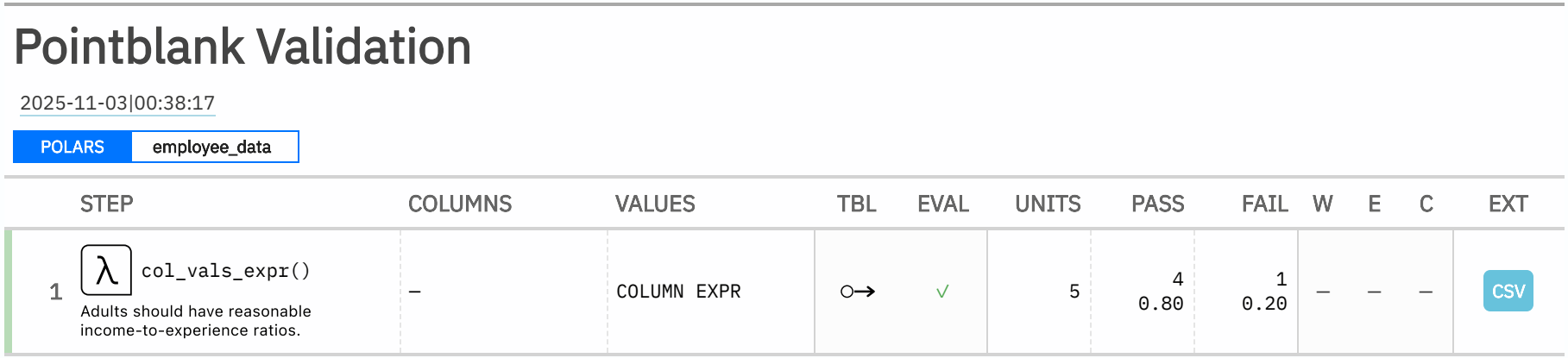
	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	λ col_vals_expr() Sessions should be either long (>20 min) or high-value (>$10).	—	COLUMN EXPR	○→	✓	2000	1518 0.76	482 0.24	—	—	—	CSV

This validates that either the session duration is longer than 20 minutes OR the item revenue is greater than $10.

2.4 Example: Multiple Conditions

You can create sophisticated validations with multiple conditions:

```
# Create a simple Polars DataFrame
employee_df = pl.DataFrame({
    "age": [25, 30, 15, 40, 35],
    "income": [50000, 75000, 0, 100000, 60000],
    "years_experience": [3, 8, 0, 15, 7]
})

(
    pb.Validate(data=employee_df, tbl_name="employee_data")
    .col_vals_expr(

        # Complex condition with multiple comparisons ---
        expr=(
            (pl.col("age") >= 18) &
            (pl.col("income") / (pl.col("years_experience") + 1) <= 25000)
        ),
        brief="Adults should have reasonable income-to-experience ratios."
    )
    .interrogate()
)
```



Pointblank Validation

2025-11-03|00:38:17

POLARS | employee_data

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	λ col_vals_expr() Adults should have reasonable income-to-experience ratios.	—	COLUMN EXPR	○→	✓	5	4 0.80	1 0.20	—	—	—	CSV

2.5 Example: Handling Null Values

When working with expressions, consider how to handle null/missing values:

```
(
    pb.Validate(data=small_table_pl)
    .col_vals_expr(

        # Check for nulls before division ---
        expr=(pl.col("c").is_not_null()) & ((pl.col("c") / pl.col("a")) > 1.5),
        brief="Ratio of `c`/`a` should exceed 1.5 (when `c` is not null)."
    )
    .interrogate()
)
```

Pointblank Validation

2025-11-03|00:38:17

POLARS

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	λ col_vals_expr() Ratio of c/a should exceed 1.5 (when c is not null).	–	COLUMN EXPR	○→	√	13	5 0.38	8 0.62	–	–	–	CSV

2.6 Best Practices

Here are some tips and tricks for effectively using expression-based validation with `Validate.col_vals_expr()`.

2.6.1 Document Your Expressions

Always provide clear documentation in the `brief=` parameter:

```
(
    pb.Validate(data=small_table_pl)
    .col_vals_expr(
        expr=pl.col("d") > pl.col("a") * 1.5,

        # Document which columns are being compared ---
        brief="Column `d` should be at least 1.5 times larger than column `a`."
    )
    .interrogate()
)
```

Pointblank Validation

2025-11-03|00:38:17

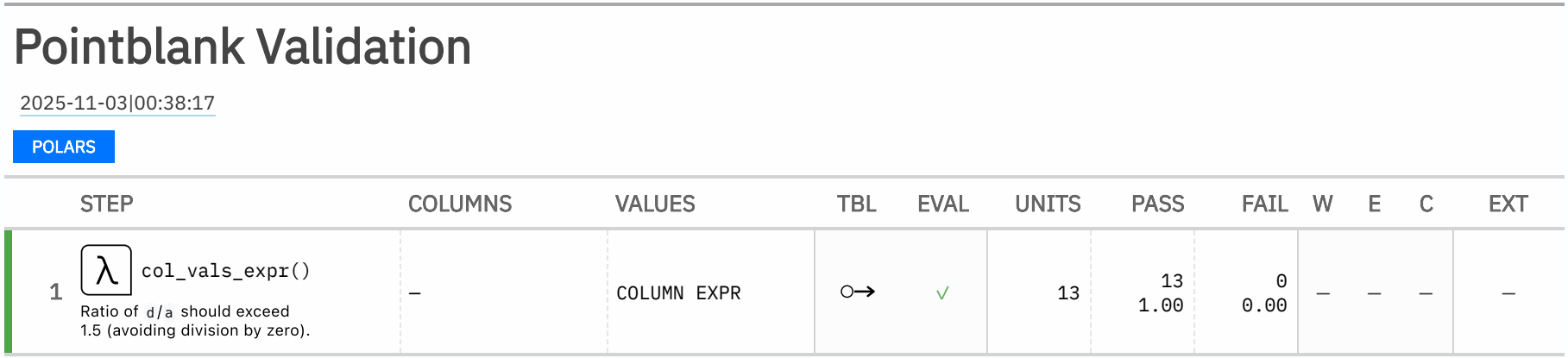
POLARS

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	λ col_vals_expr() Column d should be at least 1.5 times larger than column a.	–	COLUMN EXPR	○→	√	13	13 1.00	0 0.00	–	–	–	–

2.6.2 Handle Edge Cases

Consider potential edge cases like division by zero or nulls:

```
(
    pb.Validate(data=small_table_pl)
    .col_vals_expr(

        # Check denominator before division ---
        expr=(pl.col("a") != 0) & (pl.col("d") / pl.col("a") > 1.5),
        brief="Ratio of `d`/`a` should exceed 1.5 (avoiding division by zero)."
    )
    .interrogate()
)
```



2.6.3 Test on Small Datasets First

When developing complex expressions, test on a small sample of your data first to ensure your logic is correct before applying it to large datasets.

2.7 Conclusion

The `Validate.col_vals_expr()` method provides a powerful way to implement complex validation logic in Pointblank when standard validation methods aren't sufficient. By leveraging expressions, you can create sophisticated data quality checks tailored to your specific requirements, combining conditions across multiple columns and applying transformations as needed.

This flexibility makes expression-based validation an essential tool for addressing complex data quality scenarios in your validation workflows.

Schema validation in Pointblank allows you to verify that your data conforms to an expected structure and type specification. This is particularly useful when ensuring data consistency across systems or validating incoming data against predefined requirements.

Let's first look at the dataset we'll use for the first example:

```
import pointblank as pb

# Preview the small_table dataset we'll use throughout this guide
pb.preview(pb.load_dataset(dataset="small_table", tbl_type="polars"))
```

POLARS | ROWS 13 | COLUMNS 8

	date_time *Datetime*	date *Date*	a *Int64*	b *String*	c *Int64*	d *Float64*	e *Boolean*	f *String*
1	2016-01-04 11:00:00	2016-01-04	2	1-bcd-345	3	3423.29	True	high
2	2016-01-04 00:32:00	2016-01-04	3	5-egh-163	8	9999.99	True	low
3	2016-01-05 13:32:00	2016-01-05	6	8-kdg-938	3	2343.23	True	high
4	2016-01-06 17:23:00	2016-01-06	2	5-jdo-903	None	3892.4	False	mid
5	2016-01-09 12:36:00	2016-01-09	8	3-ldm-038	7	283.94	True	low
9	2016-01-20 04:30:00	2016-01-20	3	5-bce-642	9	837.93	False	high
10	2016-01-20 04:30:00	2016-01-20	3	5-bce-642	9	837.93	False	high
11	2016-01-26 20:07:00	2016-01-26	4	2-dmx-010	7	833.98	True	low
12	2016-01-28 02:51:00	2016-01-28	2	7-dmx-010	8	108.34	False	low
13	2016-01-30 11:23:00	2016-01-30	1	3-dka-303	None	2230.09	True	high

2.8 Schema Definition and Validation

A schema in Pointblank is created using the `Schema` class which defines the expected structure of a table. Once created, you apply schema validation through the `Validate.col_schema_match()` validation step.

```python
# Create a schema definition matching small_table structure
schema = pb.Schema(
    columns=[
        ("date_time",),   # Only check column name
        ("date",),        # Only check column name
        ("a", "Int64"),   # Check name and type
        ("b", "String"),  # Check name and type
        ("c", "Int64"),   # Check name and type
        ("d", "Float64"), # Check name and type
        ("e", "Boolean"), # Check name and type
        ("f",),           # Only check column name
    ]
)

# Validate the small_table against the schema
small_table_validation = (
    pb.Validate(
        data=pb.load_dataset(dataset="small_table", tbl_type="polars"),
        label="Schema validation of `small_table`.",
    )
    .col_schema_match(schema=schema)
    .interrogate()
)

small_table_validation
```

Pointblank Validation

Schema validation of `small_table`.

POLARS

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_schema_match()	—	SCHEMA	○→	✓	1	1 1.00	0 0.00	—	—	—	—

The output shows the validation passed successfully. When all columns have the correct names and types as specified in the schema, the validation passes with a single passing test unit. If there were discrepancies, this would fail, but the basic output wouldn't show specific issues.

For detailed information about validation results, use `Validate.get_step_report()`:

```
small_table_validation.get_step_report(i=1)
```

Report for Validation Step 1 ✓

COLUMN SCHEMA MATCH | COMPLETE | IN ORDER | COLUMN ≠ column | DTYPE ≠ dtype | float ≠ float64

	TARGET		EXPECTED			
	COLUMN	DATA TYPE		COLUMN		DATA TYPE	
1	date_time	Datetime(time_unit='u…	1	date_time	✓	—	
2	date	Date	2	date	✓	—	
3	a	Int64	3	a	✓	Int64	✓
4	b	String	4	b	✓	String	✓
5	c	Int64	5	c	✓	Int64	✓
6	d	Float64	6	d	✓	Float64	✓
7	e	Boolean	7	e	✓	Boolean	✓
8	f	String	8	f	✓	—	

Supplied Column Schema:

```
[('date_time',), ('date',), ('a', 'Int64'), ('b', 'String'), ('c', 'Int64'), ('d', 'Float64'), ('e', 'Boolean'), ('f',)]
```

The step report provides specific details about which columns were checked and whether they matched the schema, helping diagnose issues when validation fails.

2.9 Schema Components and Column Types

When defining a schema, you need to specify column names and optionally their data types. By default, Pointblank enforces strict validation where:

- all columns in your table must match the specified schema
- column order must match the schema
- column types are case-sensitive
- type names must match exactly

The schema definition accepts column types as string representations, which vary depending on your data source:

- `string`: Character data (may also be `"String"`, `"varchar"`, `"character"`, etc.)
- `integer`: Integer values (may also be `"Int64"`, `"int"`, `"bigint"`, etc.)
- `numeric`: Numeric values including integers and floating-point numbers (may also be `"Float64"`, `"double"`, `"decimal"`, etc.)
- `boolean`: Logical values (`True` / `False`) (may also be `"Boolean"`, `"bool"`, etc.)
- `datetime`: Date and time values (may also be `"Datetime"`, `"timestamp"`, etc.)
- `date`: Date values (may also be `"Date"`, etc.)
- `time`: Time values

For specific database engines or DataFrame libraries, you may need to use their exact type names (like `"VARCHAR(255)"` for SQL databases or `"Int64"` for Polars integers).

2.10 Discovering Column Types

To easily determine the correct type string for columns in your data, Pointblank provides two helpful functions:

```
import polars as pl
from datetime import date

# Define a sample dataframe
sample_df = pl.DataFrame({
    "id": [1, 2, 3],
    "name": ["Alice", "Bob", "Charlie"],
    "join_date": [date(2020, 1, 1), date(2021, 3, 15), date(2022, 7, 10)]
})
```

```
# Method 1: Using `preview()` with `show_types=True` to see column types
pb.preview(sample_df)
```

POLARS | ROWS 3 | COLUMNS 3

	id *Int64*	name *String*	join_date *Date*
1	1	Alice	2020-01-01
2	2	Bob	2021-03-15
3	3	Charlie	2022-07-10

```
# Method 2: Using `col_summary_tbl()` which shows column types and other details
pb.col_summary_tbl(sample_df)
```

POLARS | ROWS 3 | COLUMNS 3

Column	NA	UQ	Mean	SD	Min	P_5	Q_1	Med	Q_3	P_{95}	Max	IQ
N id Int64	0 0	3 1	2	1	1	1.01	1.5	2	2.5	2.9	3	
S name String	0 0	3 1	5	2	3	3.02	4	5	6	6.8	7	
D join_date Date	0 0	3 1	-	-	2020 01 01	-	-	-	-	-	2022 07 10	

String columns statistics regard the string's length.

These functions help you identify the exact type strings to use in your schema definitions, eliminating guesswork and ensuring compatibility with your data source.

2.11 Creating a Schema

You can create a schema in four different ways, each with its own advantages. All schema objects can be printed to display their column names and data types.

2.11.1 1. Using a List of Tuples with `columns=`

This approach allows for mixed validation: some columns checked for both name and type, others only for name:

```
schema_tuples = pb.Schema(

    # List of tuples approach: flexible for mixed type/name checking ---
    columns=[
        ("name", "String"), # Check name and type
        ("age", "Int64"),   # Check name and type
        ("height",)         # Check name only
    ]
)

print(schema_tuples)
```

```
Pointblank Schema
  name: String
  age: Int64
  height: <ANY>
```

This is the only method that allows checking just column names for some columns while checking both names and types for others.

2.11.2 2. Using a Dictionary with `columns=`

This approach is often the most readable when defining a schema manually, especially for larger schemas:

```
schema_dict = pb.Schema(

    # Dictionary approach (keys are column names, values are data types) ---
    columns={
        "name": "String",
        "age": "Int64",
        "height": "Float64",
        "created_at": "Datetime"
    }
)

print(schema_dict)
```

```
Pointblank Schema
  name: String
  age: Int64
  height: Float64
  created_at: Datetime
```

With this method, you must always provide both column names (as keys) and their types (as values).

2.11.3 3. Using Keyword Arguments

For more readable code with a small number of columns:

```
schema_kwargs = pb.Schema(

    # Keyword arguments approach (more readable for simple schemas) ---
    name="String",
    age="Int64",
    height="Float64"
)

print(schema_kwargs)
```

```
Pointblank Schema
  name: String
  age: Int64
  height: Float64
```

Like the dictionary method, this approach requires both column names and types.

2.11.4 4. Extracting from an Existing Table with `tbl=`

You can automatically extract a schema from an existing table:

```
import polars as pl

# Create a sample dataframe
df = pl.DataFrame({
    "name": ["Alice", "Bob", "Charlie"],
    "age": [25, 30, 35],
    "height": [5.6, 6.0, 5.8]
})

# Extract schema from table
schema_from_table = pb.Schema(tbl=df)

print(schema_from_table)
```

```
Pointblank Schema
  name: String
  age: Int64
  height: Float64
```

This is especially useful when you want to validate that future data matches the structure of a reference dataset.

2.12 Multiple Data Types for a Column

You can specify multiple acceptable types for a column by providing a list of types:

```
# Schema with multiple possible types for a column
schema_multi_types = pb.Schema(
    columns={
        "name": "String",
        "age": ["Int64", "Float64"],  # Accept either integer or float
        "active": "Boolean"
    }
)

print(schema_multi_types)
```

```
Pointblank Schema
  name: String
  age: ['Int64', 'Float64']
  active: Boolean
```

This is useful when working with data sources that might represent the same information in different ways (e.g., integers sometimes stored as floats).

2.13 Schema Validation Options

When using `col_schema_match()`, you can customize validation behavior with several important options:

Option	Default	Description
`complete`	`True`	Require exact column presence (no extra columns allowed)
`in_order`	`True`	Enforce column order
`case_sensitive_colnames`	`True`	Make column name matching case-sensitive
`case_sensitive_dtypes`	`True`	Make data type matching case-sensitive
`full_match_dtypes`	`True`	Require exact (not partial) type name matches

2.13.1 Controlling Column Presence

By default, `Validate.col_schema_match()` requires a complete match between the schema's columns and the table's columns. You can make this more flexible:

```
# Create a sample table
users_table_extra = pl.DataFrame({
    "id": [1, 2, 3],
    "name": ["Alice", "Bob", "Charlie"],
    "age": [25, 30, 35],
    "extra_col": ["a", "b", "c"]  # Extra column not in schema
})

# Create a schema
schema = pb.Schema(
    columns={"id": "Int64", "name": "String", "age": "Int64"}
)

# Validate without requiring all columns to be present
validation = (
    pb.Validate(data=users_table_extra)
    .col_schema_match(
        schema=schema,

        # Allow schema columns to be a subset ---
        complete=False
    )
    .interrogate()
)

validation.get_step_report(i=1)
```

Report for Validation Step 1 ✓

COLUMN SCHEMA MATCH ~~COMPLETE~~ IN ORDER COLUMN ≠ column DTYPE ≠ dtype float ≠ float64

	TARGET		EXPECTED			
	COLUMN	DATA TYPE		COLUMN		DATA TYPE	
1	id	Int64	1	id	✓	Int64	✓
2	name	String	2	name	✓	String	✓
3	age	Int64	3	age	✓	Int64	✓
4	extra_col	String					

Supplied Column Schema:

```
[('id', 'Int64'), ('name', 'String'), ('age', 'Int64')]
```

2.13.2 Column Order Enforcement

You can control whether column order matters in your validation:

```
# Create a sample table
users_table = pl.DataFrame({
    "id": [1, 2, 3],
    "name": ["Alice", "Bob", "Charlie"],
    "age": [25, 30, 35],
})

# Create a schema
schema = pb.Schema(
    columns={"name": "String", "age": "Int64", "id": "Int64"}
)

# Validate without enforcing column order
validation = (
    pb.Validate(data=users_table)
    .col_schema_match(
        schema=schema,

        # Don't enforce column order ---
        in_order=False
    )
    .interrogate()
)

validation.get_step_report(i=1)
```

Report for Validation Step 1 ✓

COLUMN SCHEMA MATCH COMPLETE ~~IN ORDER~~ COLUMN ≠ column DTYPE ≠ dtype float ≠ float64

TARGET			EXPECTED		
	COLUMN	DATA TYPE		COLUMN	DATA TYPE
1	id	Int64	3	id ✓	Int64 ✓
2	name	String	1	name ✓	String ✓
3	age	Int64	2	age ✓	Int64 ✓

Supplied Column Schema:

```
[('name', 'String'), ('age', 'Int64'), ('id', 'Int64')]
```

2.13.3 Case Sensitivity

Control whether column names and data types are case-sensitive:

```python
# Create schema with different case charactistics
case_schema = pb.Schema(
    columns={"ID": "int64", "NAME": "string", "AGE": "int64"}
)

# Create validation with case-insensitive column names and types
validation = (
    pb.Validate(data=users_table)
    .col_schema_match(
        schema=case_schema,

        # Ignore case in column names ---
        case_sensitive_colnames=False,

        # Ignore case in data type names ---
        case_sensitive_dtypes=False
    )
    .interrogate()
)

validation.get_step_report(i=1)
```

Report for Validation Step 1 ✓

COLUMN SCHEMA MATCH | COMPLETE | IN ORDER | COLUMN = column | DTYPE = dtype | float ≠ float64

	TARGET		EXPECTED			
	COLUMN	DATA TYPE	COLUMN		DATA TYPE	
1	id	Int64	1 ID	✓	int64	✓
2	name	String	2 NAME	✓	string	✓
3	age	Int64	3 AGE	✓	int64	✓

Supplied Column Schema:

```
[('ID', 'int64'), ('NAME', 'string'), ('AGE', 'int64')]
```

2.13.4 Type Matching Precision

Control how strictly data types must match:

```
# Create schema with simplified type names
type_schema = pb.Schema(

    # Using simplified type names ---
    columns={"id": "int", "name": "str", "age": "int"}
)

# Allow partial type matches
validation = (
    pb.Validate(data=users_table)
    .col_schema_match(
        schema=type_schema,

        # Ignore case in data type names ---
        case_sensitive_dtypes=False,

        # Allow partial type name matches ---
        full_match_dtypes=False
    )
    .interrogate()
)

validation.get_step_report(i=1)
```

Report for Validation Step 1 ✓

COLUMN SCHEMA MATCH COMPLETE IN ORDER COLUMN ≠ column DTYPE = dtype float = float64

	TARGET		EXPECTED		
	COLUMN	DATA TYPE	COLUMN		DATA TYPE	
1	id	Int64	id	✓	int	✓
2	name	String	name	✓	str	✓
3	age	Int64	age	✓	int	✓

Supplied Column Schema:

```
[('id', 'int'), ('name', 'str'), ('age', 'int')]
```

2.14 Common Schema Validation Patterns

This section explores common patterns for applying schema validation to different scenarios. Each pattern addresses specific validation needs you might encounter when working with real-world data. We'll examine the step reports for these validations since they provide more detailed information about what was checked and how the validation performed, offering an intuitive way to understand the results beyond simple pass/fail indicators.

2.15 Common Schema Validation Patterns

This section explores common patterns for applying schema validation to different scenarios. Each pattern addresses specific validation needs you might encounter when working with real-world data. We'll examine the step reports (`Validate.get_step_report()`) for these validations since they provide more detailed information about what was checked and how the validation performed, offering an intuitive way to understand the results beyond simple pass/fail indicators.

2.15.1 Structural Validation Only

When you only care about column names but not their types:

```
# Create a schema with only column names
structure_schema = pb.Schema(
    columns=["id", "name", "age", "extra_col"]
)

# Validate structure only
validation = (
    pb.Validate(data=users_table_extra)
    .col_schema_match(schema=structure_schema)
    .interrogate()
)

validation.get_step_report(i=1)
```

Report for Validation Step 1 ✓

COLUMN SCHEMA MATCH COMPLETE IN ORDER COLUMN ≠ column DTYPE ≠ dtype float ≠ float64

	TARGET		EXPECTED		
	COLUMN	DATA TYPE	COLUMN		DATA TYPE
1	id	Int64	id	✓	–
2	name	String	name	✓	–
3	age	Int64	age	✓	–
4	extra_col	String	extra_col	✓	–

Supplied Column Schema:

```
[('id',), ('name',), ('age',), ('extra_col',)]
```

2.15.2 Mixed Validation

Validate types for critical columns but just presence for others:

```
# Mixed validation for different columns
mixed_schema = pb.Schema(
    columns=[
        ("id", "Int64"),     # Check name and type
        ("name", "String"),  # Check name and type
        ("age",),            # Check name only
        ("extra_col",)       # Check name only
    ]
)

# Validate with mixed approach
validation = (
    pb.Validate(data=users_table_extra)
    .col_schema_match(schema=mixed_schema)
    .interrogate()
)

validation.get_step_report(i=1)
```

Report for Validation Step 1 ✓

COLUMN SCHEMA MATCH COMPLETE IN ORDER COLUMN ≠ column DTYPE ≠ dtype float ≠ float64

TARGET			EXPECTED		
	COLUMN	DATA TYPE	COLUMN		DATA TYPE
1	id	Int64	1 id	✓	Int64 ✓
2	name	String	2 name	✓	String ✓
3	age	Int64	3 age	✓	–
4	extra_col	String	4 extra_col	✓	–

Supplied Column Schema:

```
[('id', 'Int64'), ('name', 'String'), ('age',), ('extra_col',)]
```

2.15.3 Progressive Schema Evolution

As your data evolves, you might need to adapt your validation approach:

```
# Original schema
original_schema = pb.Schema(
    columns={
        "id": "Int64",
        "name": "String"
    }
)

# New data with additional columns
evolved_data = pl.DataFrame({
    "id": [1, 2, 3],
    "name": ["Alice", "Bob", "Charlie"],
    "age": [25, 30, 35],           # New column
    "active": [True, False, True]  # New column
})

# Validate with flexible parameters
validation = (
    pb.Validate(evolved_data)
    .col_schema_match(
        schema=original_schema,

        # Allow extra columns ---
        complete=False,

        # Don't enforce order ---
        in_order=False
    )
    .interrogate()
)

validation.get_step_report(i=1)
```

Report for Validation Step 1 ✓

COLUMN SCHEMA MATCH ~~COMPLETE~~ ~~IN ORDER~~ COLUMN ≠ column DTYPE ≠ dtype float ≠ float64

	TARGET		EXPECTED			
	COLUMN	DATA TYPE		COLUMN		DATA TYPE	
1	id	Int64	1	id	✓	Int64	✓
2	name	String	2	name	✓	String	✓
3	age	Int64					
4	active	Boolean					

Supplied Column Schema:

```
[('id', 'Int64'), ('name', 'String')]
```

2.16 Integrating with Larger Validation Workflows

Schema validation is often just one part of a comprehensive data validation strategy. You can combine schema checks with other validation steps:

```
# Define a schema
schema = pb.Schema(
    columns={
        "id": "Int64",
        "name": "String",
        "age": "Int64"
    }
)

# Create a validation plan
validation = (
    pb.Validate(
        users_table,
        label="User data validation",
        thresholds=pb.Thresholds(warning=0.05, error=0.1)
    )

    # Add schema validation ---
    .col_schema_match(schema=schema)

    # Add other validation steps ---
    .col_vals_not_null(columns="id")
    .col_vals_gt(columns="age", value=26)
    .interrogate()
)

validation
```

Pointblank Validation

User data validation

POLARS | WARNING 0.05 | ERROR 0.1 | CRITICAL —

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_schema_match()	—	SCHEMA	○→	√	1	1 1.00	0 0.00	○	○	—	—
2	col_vals_not_null()	id	—	○→	√	3	3 1.00	0 0.00	○	○	—	—
3	col_vals_gt()	age	26	○→	√	3	2 0.67	1 0.33	●	●	—	CSV

This approach allows you to first validate the structure of your data and then check specific business rules or constraints.

2.17 Best Practices

1. Define schemas early: document and define expected data structures early in your data workflow.

2. Choose the right creation method:

   - use `columns=<dict>` for readability with many columns
   - use `columns=<list of tuples>` for mixed name/type validation
   - use `kwargs` for small schemas with simple column names
   - use `tbl=` to extract schemas from reference datasets

3. Be deliberate about strictness: choose validation parameters based on your specific needs:

   - strict validation (`complete=True`) for critical data interfaces
   - flexible validation (`complete=False`, `in_order=False`) for evolving datasets

4. Reuse schemas: create schema definitions that can be reused across multiple validation contexts.

5. Version control schemas: as your data evolves, maintain versions of your schemas to track changes.
6. Extract schemas from reference data: when you have a 'golden' dataset that represents your ideal structure, use `Schema(tbl=reference_data)` to extract its schema.
7. Consider type flexibility: use multiple types per column (`["Int64", "Float64"]`) when working with data from diverse sources.
8. Combine with targeted validation: use schema validation for structural checks and add specific validation steps for business rules.

2.18 Conclusion

Schema validation provides a powerful mechanism for ensuring your data adheres to expected structural requirements. It serves as an excellent first line of defense in your data validation strategy, verifying that the data you're working with has the expected shape before applying more detailed business rule validations.

The `Schema` class offers multiple ways to define schemas, from manual specification with dictionaries or keyword arguments to automatic extraction from reference tables. When combined with the flexible options of `Validate.col_schema_match()`, you can implement validation approaches ranging from strict structural enforcement to more flexible evolution-friendly checks.

By understanding the different schema creation methods and validation options, you can efficiently validate the structure of your data tables and ensure they meet your requirements before processing.

In addition to validation steps that create reports, Pointblank provides **assertions**. This is a lightweight way to confirm data quality by raising exceptions when validation conditions aren't met. Assertions are particularly useful in:

- data processing pipelines where you need to halt execution if data doesn't meet expectations
- testing environments where you want to verify data properties programmatically
- scripts and functions where you need immediate notification of data problems

2.19 Basic Assertion Workflow

The assertion workflow uses your familiar validation steps with assertion methods to check that validations meet your requirements:

```
import pointblank as pb
import polars as pl

# Create sample data
sample_data = pl.DataFrame({
    "id": [1, 2, 3, 4, 5],
    "value": [10.5, 8.3, -2.1, 15.7, 7.2]
})

# Create a validation plan and assert that all steps pass
(
    pb.Validate(data=sample_data)
    .col_vals_gt(columns="id", value=0, brief="IDs must be positive")
    .col_vals_gt(columns="value", value=-5, brief="Values should exceed -5")

    # Will automatically `interrogate()` and raise an AssertionError if any validation fails ---
    .assert_passing()
)
```

This simple pattern allows you to integrate data quality checks into your data pipelines. With it, you can create clear stopping points when data doesn't meet specified criteria.

2.20 Assertion Methods

Pointblank offers two types of assertions:

1. Full Passing Assertions: using `Validate.assert_passing()` to verify that every single test unit passes
2. Threshold-Based Assertions: using `Validate.assert_below_threshold()` to verify that failure rates stay within acceptable thresholds

2.20.1 `assert_passing()`

The `Validate.assert_passing()` method is the strictest form of assertion, requiring every single validation test unit to pass:

```
try:
    (
        pb.Validate(data=sample_data)
        .col_vals_gt(columns="value", value=0)

        # Direct assertion: automatically interrogates ---
        .assert_passing()
    )
except AssertionError as e:
    print("AssertionError:", str(e))
```

```
AssertionError: The following assertions failed:
- Step 1: Expect that values in `value` should be > `0`.
```

2.20.2 `assert_below_threshold()`

The `Validate.assert_below_threshold()` method is more flexible as it allows some failures as long as they stay below specified threshold levels. Pointblank uses three severity thresholds that increase in order of seriousness:

- **'warning'** (least severe): the first threshold that gets triggered when failures exceed this level
- **'error'** (more severe): the middle threshold indicating more serious data quality issues
- **'critical'** (most severe): the highest threshold indicating critical data quality problems

```python
# Create a two-column DataFrame for this example
tbl_pl = pl.DataFrame({
    "a": [4, 6, 9, 7, 12, 8, 7, 12, 10, 7],
    "b": [9, 8, 10, 5, 10, 9, 14, 6, 6, 8],

})

# Set thresholds: warning=0.2 (20%), error=0.3 (30%), critical=0.4 (40%)
validation = (
    pb.Validate(data=tbl_pl, thresholds=(0.2, 0.3, 0.4))
    .col_vals_gt(columns="b", value=5)   # 1/10 failing (10% failure rate)
    .col_vals_lt(columns="a", value=11)  # 2/10 failing (20% failure rate)
    .col_vals_ge(columns="b", value=8)   # 3/10 failing (30% failure rate)
    .interrogate()
)

validation
```

Pointblank Validation

2025-11-03|00:38:17

POLARS | WARNING 0.2 | ERROR 0.3 | CRITICAL 0.4

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_gt()	b	5	○→	✓	10	9 0.90	1 0.10	○	○	○	CSV
2	col_vals_lt()	a	11	○→	✓	10	8 0.80	2 0.20	●	○	○	CSV
3	col_vals_ge()	b	8	○→	✓	10	7 0.70	3 0.30	●	●	○	CSV

The validation report above visually indicates threshold levels with colored circles:

- gray circles in the `W` column indicate the 'warning' threshold
- yellow circles in the `E` column indicate the 'error' threshold
- red circles in the `C` column indicate the 'critical' threshold

This won't pass the `Validate.assert_below_threshold()` assertion for the 'error' level because step 3 exceeds this threshold (30% failure rate matches the error threshold):

```
try:
    validation.assert_below_threshold(level="error")
except AssertionError as e:
    print("AssertionError:", str(e))
```

```
AssertionError: The following steps exceeded the error threshold level:
Step 3: Expect that values in `b` should be >= `8`.
```

We can check against the 'error' threshold for specific steps with the `i=` parameter:

```
validation.assert_below_threshold(level="error", i=[1, 2])
```

This passes because the highest threshold exceeded in steps 1 and 2 is 'warning'.

The `Validate.assert_below_threshold()` method takes these parameters:

- `level=`: threshold level to check against (`"warning"`, `"error"`, or `"critical"`)
- `i=`: optional specific step number(s) to check
- `message=`: optional custom error message

This is particularly useful when:

- working with real-world data where some percentage of failures is acceptable
- implementing different severity levels for data quality rules
- gradually improving data quality with stepped thresholds

> ⓘ **Note**
>
> Assertion methods like `Validate.assert_passing()` and `Validate.assert_below_threshold()` will automatically call `Validate.interrogate()` if needed, so you don't have to explicitly include this step when using assertions directly.

2.21 Using Status Check Methods

In addition to assertion methods that raise exceptions, Pointblank provides status check methods that return boolean values:

2.21.1 `all_passed()`

The `Validate.all_passed()` method will return `True` only if every single test unit in every validation step passed:

```
validation = (
    pb.Validate(data=sample_data)
    .col_vals_gt(columns="value", value=0)
    .interrogate()
)

if not validation.all_passed():
    print("Validation failed: some values are not positive")
```

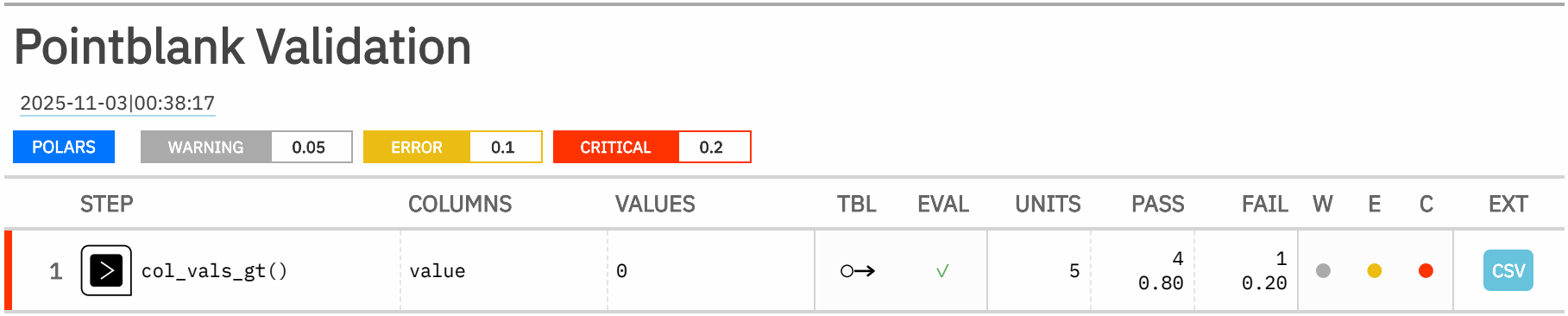
```
Validation failed: some values are not positive
```

2.21.2 `warning()`, `error()`, and `critical()`

The methods `Validate.warning()`, `Validate.error()`, and `Validate.critical()` all return information about whether validation steps exceeded that specific threshold level.

While assertion methods raise exceptions to halt execution when thresholds are exceeded, these status methods give you fine-grained control to implement custom logic based on different validation quality levels.

```
validation = (
    pb.Validate(data=sample_data, thresholds=(0.05, 0.10, 0.20))
    .col_vals_gt(columns="value", value=0)  # Some values are negative
    .interrogate()
)

validation
```



The `Validate.warning()` method returns a dictionary mapping step numbers to boolean values. A `True` value means that step exceeds the warning threshold:

```
# Get dictionary of warning status for each step
warning_status = validation.warning()
print(f"Warning status: {warning_status}")  # {1: True} means step 1 exceeds warning threshold
```

```
Warning status: {1: True}
```

You can check a specific step using the `i=` parameter, and get a single boolean with `scalar=True`:

```
# Check error threshold for specific step
has_errors = validation.error(i=1, scalar=True)

if has_errors:
    print("Step 1 exceeded the error threshold.")
```

```
Step 1 exceeded the error threshold.
```

Similarly, we can check if any steps exceed the 'critical' threshold:

```
# Check against critical threshold
critical_status = validation.critical()
print(f"Critical status: {critical_status}")
```

```
Critical status: {1: True}
```

These methods are particularly useful for:

1. Conditional logic: taking different actions based on threshold severity
2. Reporting: generating summary reports about validation quality
3. Monitoring: tracking data quality trends over time
4. Graceful degradation: implementing fallback logic when quality decreases

Each method has these options:

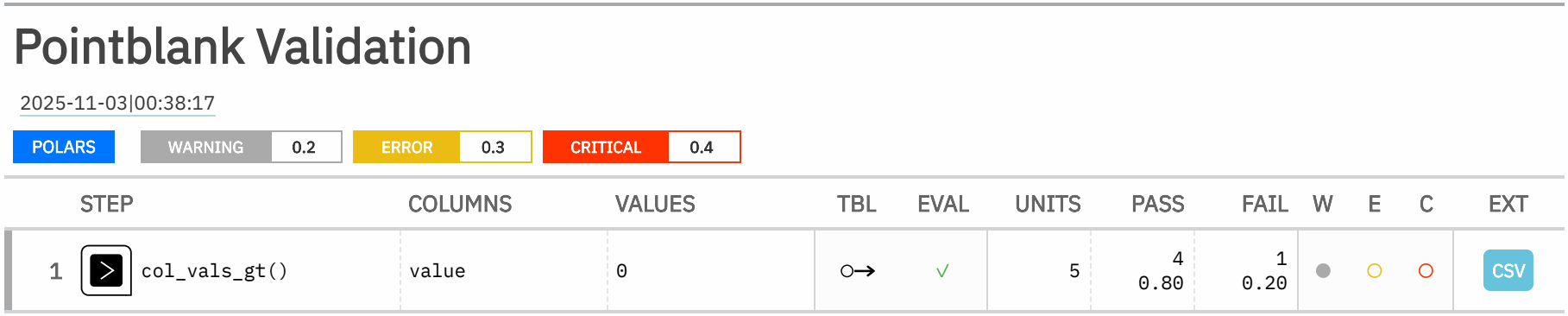
- without parameters: returns a dictionary mapping step numbers to boolean status values
- with `i=`: check specific step(s)
- with `scalar=True`: return a single boolean instead of a dictionary (when checking a specific step)

While assertion methods raise exceptions to halt execution when thresholds are exceeded, these methods give you fine-grained control to implement custom logic based on different validation quality levels.

2.22 Customizing Error Messages

You can provide custom error messages when assertions fail to make them more meaningful in your specific workflow context:

```
# Create a validation with potential failures
validation = (
    pb.Validate(data=sample_data, thresholds=(0.2, 0.3, 0.4))
    .col_vals_gt(columns="value", value=0)
    .interrogate()
)

# Display the validation results
validation
```



When you need to customize the error message that appears when an assertion fails, use the `message=` parameter:

```
try:
    # Custom message for threshold assertion
    validation.assert_below_threshold(
        level="warning",
        message="Data quality too low for processing!"
    )
except AssertionError as e:
    print(f"Custom handling of failure: {e}")
```

```
Custom handling of failure: Data quality too low for processing!
```

Descriptive error messages are essential in production systems where multiple team members might need to interpret validation failures. The custom message lets you provide context appropriate to your specific workflow or data pipeline stage.

2.23 Combining Assertions with Actions

Actions and assertions serve complementary but distinct purposes in data validation workflows:

- Actions trigger during validation but shouldn't raise errors (as this would halt report generation)
- Assertions are designed to raise errors based on specific conditions, making them ideal for flow control after validation completes

Here's a simplified example showing how to use them together. The print statements simulate logging or monitoring that would be valuable in production data pipelines:

```
# Define a simple action function (won't raise errors)
def notify_quality_issue(message="Data quality issue detected"):
    print(f"ACTION TRIGGERED: {message}")

# Create data with known failures
problem_data = pl.DataFrame({
    "id": [1, 2, 3, -4, 5],  # One negative ID
    "value": [10.5, 8.3, -2.1, 15.7, 7.2]  # One negative value
})

# First use actions for automated responses during validation
print("Running validation with actions...")
validation = (
    pb.Validate(data=problem_data, thresholds=(0.1, 0.2, 0.3))
    .col_vals_gt(
        columns="id", value=0,
        brief="IDs must be positive",
        actions=pb.Actions(warning=notify_quality_issue)
    )
    .interrogate()  # Actions trigger here but won't stop report generation
)

# Then use assertions after validation for workflow control
print("\nNow using assertion for flow control...")
try:
    validation.assert_below_threshold(level="warning")
    print("This line won't execute if the assertion fails")
except AssertionError as e:
    print(f"Validation failed threshold check: {e}")
    print("Implementing fallback process...")
```

```
Running validation with actions...
ACTION TRIGGERED: Data quality issue detected

Now using assertion for flow control...
Validation failed threshold check: The following steps exceeded the warning threshold level:
Step 1: Expect that values in `id` should be > `0`.
Implementing fallback process...
```

This approach gives you the best of both worlds:

- Actions provide immediate notification during validation without interrupting the process
- Assertions control workflow execution after validation when important thresholds are exceeded

This pattern works well in data pipelines where you want both: (1) automated responses during validation and (2) clear decision points after validation is complete.

2.24 Best Practices for Assertions

When using assertions in your data workflows, consider these best practices:

1. **Choose the right assertion type**:
    - use `Validate.assert_passing()` for critical validations where any failure is unacceptable
    - use `Validate.assert_below_threshold()` for validations where some failure rate is acceptable
2. **Set appropriate thresholds** that match your data quality requirements:

```
# Example threshold strategy
validation = pb.Validate(
    data=sample_data,
    # warning at 1%, error at 5%, critical at 10%
    thresholds=pb.Thresholds(warning=0.01, error=0.05, critical=0.10)
)
```

3. **Use a graduated approach** to validation severity:

```
# Critical validations: must be perfect
validation_1.assert_passing()

# Important validations: must be below error threshold
validation_2.assert_below_threshold(level="error")

# Monitor-only validations: check warning status
warning_status = validation_3.warning()
```

4. **Placement in pipelines**: place assertions at critical points where data quality is essential
5. **Error handling**: wrap assertions in try-except blocks for better error handling in production systems
6. **Combine with reporting**: use both assertions and reporting approaches for comprehensive quality control

2.25 Conclusion

Pointblank's assertion methods give you flexible options for enforcing data quality requirements:

- `Validate.assert_passing()` for strict validation where every test unit must pass
- `Validate.assert_below_threshold()` for more flexible validation where some failures are tolerable
- Status methods (`Validate.warning()`, `Validate.error()`, and `Validate.critical()`) for programmatic threshold checking

By using these assertion methods appropriately, you can build robust data pipelines with different levels of quality enforcement (from strict validation of critical data properties to more lenient checks for less critical aspects). This graduated approach to data quality helps create systems that are both reliable and practical in real-world data environments.

Draft validation in Pointblank leverages large language models (LLMs) to automatically generate validation plans for your data. This feature is especially useful when starting validation on a new dataset or when you need to quickly establish baseline validation coverage.

The `DraftValidation` class connects to various LLM providers to analyze your data's characteristics and generate a complete validation plan tailored to its structure and content.

2.26 How `DraftValidation` Works

When you use `DraftValidation`, the process works through these steps:

1. a statistical summary of your data is generated using the `DataScan` class
2. this summary is converted to JSON format and sent to your selected LLM provider
3. the LLM uses the summary along with knowledge about Pointblank's validation capabilities to generate a validation plan
4. the result is returned as executable Python code that you can use directly or modify as needed

The entire process happens without sending all of the data to the LLM provider, but only a summary that includes column names, data types, basic statistics, and a small sample of values.

2.27 Requirements and Setup

To use the `DraftValidation` feature, you'll need:

1. an API key from a supported LLM provider
2. the required Python packages installed

You can install all necessary dependencies with:

```
pip install pointblank[generate]
```

This will install the `chatlas` package and other dependencies required for `DraftValidation`.

2.27.1 Supported LLM Providers

The `DraftValidation` class supports multiple LLM providers:

- **Anthropic** (Claude models)
- **OpenAI** (GPT models)
- **Ollama** (local LLMs)
- **Amazon Bedrock** (AWS-hosted models)

Each provider has different capabilities and performance characteristics, but all can be used to generate validation plans through a consistent interface.

2.28 Basic Usage

The simplest way to use `DraftValidation` is to provide your data and specify an LLM model. Let's try it out with the `global_sales` dataset.

```
import pointblank as pb

# Load a dataset
data = pb.load_dataset(dataset="global_sales", tbl_type="polars")

# Generate a validation plan
pb.DraftValidation(
    data=data,
    model="anthropic:claude-sonnet-4-5",
    api_key="your_api_key_here"  # Replace with your actual API key
)
```

```python
import pointblank as pb

# Define schema based on column names and dtypes
schema = pb.Schema(columns=[
    ("product_id", "String"),
    ("product_category", "String"),
    ("customer_id", "String"),
    ("customer_segment", "String"),
    ("region", "String"),
    ("country", "String"),
    ("city", "String"),
    ("timestamp", "Datetime(time_unit='us', time_zone=None)"),
    ("quarter", "String"),
    ("month", "Int64"),
    ("year", "Int64"),
    ("price", "Float64"),
    ("quantity", "Int64"),
    ("status", "String"),
    ("email", "String"),
    ("revenue", "Float64"),
    ("tax", "Float64"),
    ("total", "Float64"),
    ("payment_method", "String"),
    ("sales_channel", "String")
])

# The validation plan
validation = (
    pb.Validate(
        data=your_data,  # Replace your_data with the actual data variable
        label="Draft Validation",
        thresholds=pb.Thresholds(warning=0.10, error=0.25, critical=0.35)
    )
    .col_schema_match(schema=schema)
    .col_vals_not_null(columns=[
        "product_category", "customer_segment", "region", "country",
        "price", "quantity", "status", "email", "revenue", "tax",
        "total", "payment_method", "sales_channel"
    ])
    .col_vals_between(columns="month", left=1, right=12, na_pass=True)
    .col_vals_between(columns="year", left=2021, right=2023, na_pass=True)
    .col_vals_gt(columns="price", value=0)
    .col_vals_gt(columns="quantity", value=0)
```

```
    .col_vals_gt(columns="revenue", value=0)
    .col_vals_gt(columns="tax", value=0)
    .col_vals_gt(columns="total", value=0)
    .col_vals_in_set(columns="product_category", set=[
        "Manufacturing", "Retail", "Healthcare"
    ])
    .col_vals_in_set(columns="customer_segment", set=[
        "Government", "Consumer", "SMB"
    ])
    .col_vals_in_set(columns="region", set=[
        "Asia Pacific", "Europe", "North America"
    ])
    .col_vals_in_set(columns="status", set=[
        "returned", "shipped", "delivered"
    ])
    .col_vals_in_set(columns="payment_method", set=[
        "Apple Pay", "PayPal", "Bank Transfer", "Credit Card"
    ])
    .col_vals_in_set(columns="sales_channel", set=[
        "Partner", "Distributor", "Phone"
    ])
    .row_count_match(count=50000)
    .col_count_match(count=20)
    .rows_distinct()
    .interrogate()
)

validation
```

2.28.1 Managing API Keys

While you can directly provide API keys as shown above, there are more secure approaches:

```
import os

# Get API key from environment variable
api_key = os.getenv("ANTHROPIC_API_KEY")

draft_validation = pb.DraftValidation(
    data=data,
    model="anthropic:claude-sonnet-4-5",
    api_key=api_key
)
```

You can also store API keys in a `.env` file in your project's root directory:

```
# Contents of .env file
ANTHROPIC_API_KEY=your_anthropic_api_key_here
OPENAI_API_KEY=your_openai_api_key_here
```

If your API keys have standard names (like `ANTHROPIC_API_KEY` or `OPENAI_API_KEY`), `DraftValidation` will automatically find and use them:

```
# No API key needed if stored in .env with standard names
draft_validation = pb.DraftValidation(
    data=data,
    model="anthropic:claude-sonnet-4-5"
)
```

2.29 Example Output for `nycflights`

Here's an example of a validation plan that might be generated by `DraftValidation` for the `nycflights` dataset:

```
pb.DraftValidation(
    pb.load_dataset(dataset="nycflights", tbl_type="duckdb",
    model="anthropic:claude-sonnet-4-5"
)
```

```python
import pointblank as pb

# Define schema based on column names and dtypes
schema = pb.Schema(columns=[
    ("year", "int64"),
    ("month", "int64"),
    ("day", "int64"),
    ("dep_time", "int64"),
    ("sched_dep_time", "int64"),
    ("dep_delay", "int64"),
    ("arr_time", "int64"),
    ("sched_arr_time", "int64"),
    ("arr_delay", "int64"),
    ("carrier", "string"),
    ("flight", "int64"),
    ("tailnum", "string"),
    ("origin", "string"),
    ("dest", "string"),
    ("air_time", "int64"),
    ("distance", "int64"),
    ("hour", "int64"),
    ("minute", "int64")
])

# The validation plan
validation = (
    pb.Validate(
        data=your_data,  # Replace your_data with the actual data variable
        label="Draft Validation",
        thresholds=pb.Thresholds(warning=0.10, error=0.25, critical=0.35)
    )
    .col_schema_match(schema=schema)
    .col_vals_not_null(columns=[
        "year", "month", "day", "sched_dep_time", "carrier", "flight",
        "origin", "dest", "distance", "hour", "minute"
    ])
    .col_vals_between(columns="month", left=1, right=12)
    .col_vals_between(columns="day", left=1, right=31)
    .col_vals_between(columns="sched_dep_time", left=106, right=2359)
    .col_vals_between(columns="dep_delay", left=-43, right=1301, na_pass=True)
    .col_vals_between(columns="air_time", left=20, right=695, na_pass=True)
    .col_vals_between(columns="distance", left=17, right=4983)
    .col_vals_between(columns="hour", left=1, right=23)
```

```
    .col_vals_between(columns="minute", left=0, right=59)
    .col_vals_in_set(columns="origin", set=["EWR", "LGA", "JFK"])
    .col_count_match(count=18)
    .row_count_match(count=336776)
    .rows_distinct()
    .interrogate()
)

validation
```

Notice how the generated plan includes:

1. A schema validation with appropriate data types
2. Not-null checks for required columns
3. Range validations for numerical data
4. Set membership checks for categorical data
5. Row and column count validations
6. Appropriate handling of missing values with `na_pass=True`

2.30 Working with Model Providers

2.30.1 Specifying Models

When using `DraftValidation`, you specify the model in the format `"provider:model_name"`:

```
# Using Anthropic's Claude model
pb.DraftValidation(data=data, model="anthropic:claude-sonnet-4-5")

# Using OpenAI's GPT model
pb.DraftValidation(data=data, model="openai:gpt-4-turbo")

# Using a local model with Ollama
pb.DraftValidation(data=data, model="ollama:llama3:latest")

# Using Amazon Bedrock
pb.DraftValidation(data=data, model="bedrock:anthropic.claude-3-sonnet-20240229-v1:0")
```

2.30.2 Model Performance and Privacy

Different models have different capabilities when it comes to generating validation plans:

- Anthropic Claude Sonnet 4.5 generally provides the most comprehensive and accurate validation plans
- OpenAI GPT-4 models also perform well
- Local models through Ollama can be useful for private data but they currently have reduced capabilities here

A key advantage of `DraftValidation` is that your actual dataset is not sent to the LLM provider. Instead, only a summary is transmitted, which includes:

- the number of rows and columns
- column names and data types
- basic statistics (min, max, mean, median, missing values count)
- a small sample of values from each column (usually 5-10 values)

This approach protects your data while still providing enough context for the LLM to generate relevant validation rules.

2.31 Customizing Generated Plans

The validation plan generated by `DraftValidation` is just a starting point. You'll typically want to:

1. review the generated code for correctness
2. replace `your_data` with your actual data variable name that exists in your workspace
3. ensure the data object referenced is actually present in your workspace
4. adjust thresholds and validation parameters
5. add domain-specific validation rules
6. remove any unnecessary checks

For example, you might start by capturing the text representation of your draft validation. This will give you the raw Python code that you can copy into a new code cell in your notebook or script. From there, you can customize it by modifying thresholds to match your organization's data quality standards, adding business-specific validation rules that require domain knowledge, or removing checks that aren't relevant to your use case. Once you've made your modifications, you can execute the customized validation plan as you would any other Pointblank validation.

2.32 Under the Hood

2.32.1 The Generated Data Summary

To understand what the LLM works with, here's an example of the data summary format that's sent:

```
{
  "table_info": {
    "rows": 336776,
    "columns": 18,
    "table_type": "duckdb"
  },
  "column_info": [
    {
      "column_name": "year",
      "column_type": "int64",
      "missing_values": 0,
      "min": 2013,
      "max": 2013,
      "mean": 2013.0,
      "median": 2013,
      "sample_values": [2013, 2013, 2013, 2013, 2013]
    },
    {
      "column_name": "month",
      "column_type": "int64",
      "missing_values": 0,
      "min": 1,
      "max": 12,
      "mean": 6.548819,
      "median": 7,
      "sample_values": [1, 1, 1, 1, 1]
    },
    // Additional columns...
  ]
}
```

2.32.2 The Prompt Strategy

The `DraftValidation` class uses a carefully crafted prompt that instructs the LLM to:

1. use the schema information to create a `Schema` object
2. include `Validate.col_vals_not_null()` for columns with no missing values
3. add appropriate range validations based on min/max values
4. include row and column count validations
5. format the output as clean, executable Python code

The prompt also contains constraints to ensure consistent, high-quality results, such as using line breaks in long lists for readability, applying `na_pass=True` for columns with missing values, and avoiding duplicate validations.

2.33 Best Practices and Troubleshooting

2.33.1 When to Use `DraftValidation`

Drafting a validation is most useful when:

- working with a new dataset for the first time
- needing to quickly establish baseline validation
- exploring potential validation rules before formalizing them
- validating columns with consistent patterns (numeric ranges, categories, etc.)

Consider writing validation plans manually when you need very specific business rules, are working with sensitive data, need complex validation logic, or need to validate relationships between columns.

2.33.2 Recommended Workflow and Common Issues

Here's a recommended workflow incorporating `DraftValidation`:

1. generate an initial plan with `DraftValidation`
2. review the generated validations for relevance
3. adjust thresholds and parameters as needed
4. add specific business logic and cross-column validations
5. store the final validation plan in version control

It's possible that you might bump up against some issues. Here are some common ones and solutions you might try:

- Authentication Errors: ensure your API key is valid and correctly passed to `DraftValidation`
- Package Not Found: make sure you've installed the required packages with `pip install pointblank[generate]`
- Unsupported Model: verify you're using the correct `provider:model` format
- Poor Quality Plans: try a more capable model

2.34 Conclusion

`DraftValidation` provides a powerful way to jumpstart your data validation process by leveraging LLMs to generate context-aware validation plans. By analyzing your data's structure and content, `DraftValidation` can create comprehensive validation rules that would otherwise take significant time to develop manually.

The feature balances privacy (by sending only data summaries) with utility (by generating executable validation code). While the generated plans should always be reviewed and refined, they provide an excellent starting point for ensuring your data meets your quality requirements.

By understanding how `DraftValidation` works and how to customize its output, you can significantly accelerate your data validation workflows and improve the quality of your data throughout your projects.

3 YAML

Pointblank supports defining validation workflows using YAML configuration files, providing a declarative, readable, and maintainable approach to data validation. YAML workflows are particularly useful for teams, version control, automation pipelines, and scenarios where you want to separate validation logic from application code.

YAML validation workflows offer several advantages: they're easy to read and write, can be version controlled alongside your data processing code, enable non-programmers to contribute to data quality definitions, and provide a clear separation between validation logic and execution code.

The YAML approach complements Pointblank's Python API, giving you flexibility to choose the right tool for each situation. Simple, repetitive validations work well in YAML, while complex logic with custom functions might be better suited for the Python API.

3.1 Basic YAML Validation Structure

A YAML validation workflow consists of a few key components:

- `tbl`: specifies the data source (file path, dataset name, or Python expression)
- `steps`: defines the validation checks to perform
- Optional metadata: table name, label, thresholds, actions, and other configuration

Here's a simple example validating the built-in `small_table` dataset:

```yaml
tbl: small_table
df_library: polars                    # Optional: specify DataFrame library
tbl_name: "Small Table Validation"
label: "Basic data quality checks"
steps:
  - rows_distinct
  - col_exists:
      columns: [a, b, c, d]
  - col_vals_not_null:
      columns: [a, b]
```

You can save this configuration to a .yaml file and execute it using the `yaml_interrogate()` function:

```
import pointblank as pb
from pathlib import Path

# Save the YAML configuration to a file
yaml_content = """
tbl: small_table
df_library: polars
tbl_name: "Small Table Validation"
label: "Basic data quality checks"
steps:
  - rows_distinct
  - col_exists:
      columns: [a, b, c, d]
  - col_vals_not_null:
      columns: [a, b]
"""

yaml_file = Path("basic_validation.yaml")
yaml_file.write_text(yaml_content)

# Execute the validation from the file
result = pb.yaml_interrogate(yaml_file)
result
```

Pointblank Validation

Basic data quality checks

POLARS | Small Table Validation

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	rows_distinct()	ALL COLUMNS	—	○→	√	13	9 0.69	2 0.15	—	—	—	CSV
2	col_exists()	a	—	○→	√	1	1 1.00	0 0.00	—	—	—	—
3	col_exists()	b	—	○→	√	1	1 1.00	0 0.00	—	—	—	—
4	col_exists()	c	—	○→	√	1	1 1.00	0 0.00	—	—	—	—
5	col_exists()	d	—	○→	√	1	1 1.00	0 0.00	—	—	—	—
6	col_vals_not_null()	a	—	○→	√	13	13 1.00	0 0.00	—	—	—	—
7	col_vals_not_null()	b	—	○→	√	13	13 1.00	0 0.00	—	—	—	—

The validation table shows the results of each step, just as if you had written the equivalent Python code. You can also pass YAML content directly as a string for quick testing, but working with files is the recommended approach for production workflows.

3.2 Data Sources in YAML

The `tbl` field supports various data source types, making it easy to work with different kinds of data. You can also control the DataFrame library used for loading data with the `df_library` parameter.

3.2.1 DataFrame Library Selection

By default, Pointblank loads data as Polars DataFrames, but you can specify alternative libraries:

```
# Load as Polars DataFrame (default)
tbl: small_table
df_library: polars

# Load as Pandas DataFrame
tbl: small_table
df_library: pandas

# Load as DuckDB table (via Ibis)
tbl: small_table
df_library: duckdb
```

This is particularly useful when using validation expressions that require specific DataFrame APIs:

```
# Using Pandas-specific operations
tbl: small_table
df_library: pandas
steps:
  - specially:
      expr: "lambda df: df.assign(total=df['a'] + df['d'])"

# Using Polars-specific operations
tbl: small_table
df_library: polars
steps:
  - specially:
      expr: "lambda df: df.select(pl.col('a') + pl.col('d') > 0)"
```

3.2.2 File-based Sources

```yaml
# CSV files (respects df_library setting)
tbl: "data/customers.csv"
df_library: pandas

# Parquet files
tbl: "warehouse/sales.parquet"
df_library: polars

# Multiple files with patterns
tbl: "logs/*.parquet"
```

3.2.3 Built-in Datasets

```yaml
# Use Pointblank's built-in datasets
tbl: small_table
tbl: game_revenue
tbl: nycflights
```

3.2.4 Python Expressions for Complex Sources

For more complex data loading, use the `python:` block syntax. This syntax can be used with several parameters throughout your YAML configuration:

- `tbl`: For complex data source loading (as shown below)
- `expr`: For custom validation expressions in `col_vals_expr`
- `pre`: For data preprocessing before validation steps
- `actions`: For callable action functions (`warning`, `error`, `critical`, and `default`)

```yaml
# Load data with custom Polars operations
tbl:
  python: |
    pl.scan_csv("sales_data.csv")
    .filter(pl.col("date") >= "2024-01-01")
    .head(1000)

# Load from a database connection
tbl:
  python: |
    pl.read_database(
        query="SELECT * FROM customers WHERE active = true",
        connection="postgresql://user:pass@localhost/db"
    )
```

3.3 Reusable Templates with `set_tbl=`

One of the most powerful features of YAML validation workflows is the ability to create reusable templates that can be applied to different datasets. Using the `set_tbl=` parameter with `yaml_interrogate()`, you can define validation logic once and apply it to multiple data sources.

3.3.1 Creating Validation Templates

When creating templates for use with `set_tbl=`, the `tbl` field is still required but its value will be overridden. The recommended approach is to use `tbl: null`:

```yaml
tbl: null
tbl_name: "Sales Data Validation Template"
label: "Standard validation checks for sales data"
steps:
  - col_exists:
      columns: [customer_id, revenue, region, date]
  - col_vals_not_null:
      columns: [customer_id, revenue]
  - col_vals_gt:
      columns: [revenue]
      value: 0
  - col_vals_in_set:
      columns: [region]
      set: [North, South, East, West]
```

3.3.2 Applying Templates to Multiple Datasets

Here's a practical example showing how to apply the same validation template to multiple quarterly datasets, demonstrating the power of reusable YAML configurations:

```
import pointblank as pb
import polars as pl

# Define the template once
sales_template = """
tbl: null  # Will be overridden
tbl_name: "Sales Data Validation"
label: "Standard sales validation checks"
thresholds:
  warning: 0.05
  error: 0.1
steps:
  - col_exists:
      columns: [customer_id, revenue, region]
  - col_vals_not_null:
      columns: [customer_id, revenue]
  - col_vals_gt:
      columns: [revenue]
      value: 0
  - col_vals_in_set:
      columns: [region]
      set: [North, South, East, West]
"""

# Create different datasets
q1_data = pl.DataFrame({
    "customer_id": [1, 2, 3, 4],
    "revenue": [100, 200, 150, 300],
    "region": ["North", "South", "East", "West"]
})

q2_data = pl.DataFrame({
    "customer_id": [5, 6, 7, 8],
    "revenue": [250, 180, 220, 350],
    "region": ["South", "North", "West", "East"]
})

# Apply the same template to both datasets
q1_result = pb.yaml_interrogate(sales_template, set_tbl=q1_data)
q2_result = pb.yaml_interrogate(sales_template, set_tbl=q2_data)

print(f"Q1 validation: {all(v.all_passed for v in q1_result.validation_info)}")
print(f"Q2 validation: {all(v.all_passed for v in q2_result.validation_info)}")
```

```
Q1 validation: True
Q2 validation: True
```

3.3.3 Template Best Practices

1. **Use `tbl: null`**: this clearly indicates the template expects a data source to be provided
2. **Include comprehensive metadata**: use `tbl_name`, `label`, and `brief` to make results self-documenting
3. **Set appropriate thresholds**: define warning/error levels that make sense for your use case
4. **Version control templates**: store templates in your repository alongside your data processing code
5. **Test with sample data**: validate your templates work with representative datasets

3.3.4 Common Template Patterns

For API response validation, you can ensure that responses have the expected structure and valid status codes:

```yaml
tbl: null
tbl_name: "API Response Validation"
brief: "Standard checks for API response data"
steps:
  - col_exists:
      columns: [user_id, status, timestamp]
  - col_vals_in_set:
      columns: [status]
      set: [success, error, pending]
  - col_vals_not_null:
      columns: [user_id, timestamp]
```

For file upload validation, you can check file sizes and formats to ensure they meet your requirements:

```yaml
tbl: null
tbl_name: "File Upload Validation"
steps:
  - col_vals_gt:
      columns: [file_size]
      value: 0
  - col_vals_lt:
      columns: [file_size]
      value: 10485760  # 10MB limit
  - col_vals_in_set:
      columns: [file_type]
      set: [csv, json, xlsx, parquet]
```

This template approach is particularly valuable in data pipelines, ETL processes, and automated testing scenarios where you need to apply consistent validation logic across multiple similar datasets.

3.4 Validation Steps

YAML supports all of Pointblank's validation methods. Here are some common patterns:

3.4.1 Column-based Validations

```
tbl: worldcities.csv
steps:
  # Check for missing values
  - col_vals_not_null:
      columns: [city_name, country]

  # Validate value ranges
  - col_vals_between:
      columns: latitude
      left: -90
      right: 90

  # Check set membership
  - col_vals_in_set:
      columns: country_code
      set: [US, CA, MX, UK, DE, FR]

  # Regular expression validation
  - col_vals_regex:
      columns: postal_code
      pattern: "^[0-9]{5}(-[0-9]{4})?$"
```

3.4.2 Row-based Validations

```
tbl: sales_data.csv
steps:
  # Check for duplicate rows
  - rows_distinct

  # Ensure complete rows (no missing values)
  - rows_complete

  # Check row count
  - row_count_match:
      count: 1000
```

3.4.3 Schema Validations

Schema validation ensures your data has the expected structure and column types. The `col_schema_match` validation method uses a `schema` key that contains a `columns` list, where each item in the list can specify a column name alone or a column name with its expected data type.

Each `column` entry can be specified as:

- `column_name`: column name as a scalar string (structure validation, no type checking)
- `[column_name, "data_type"]`: column name with type validation (as a list with two elements)
- `[column_name]`: column name in a single-item list (equivalent to scalar, for consistency)

```
tbl: customer_data.csv
steps:
  # Complete schema validation (structure and types)
  - col_schema_match:
      schema:
        columns:
          - [customer_id, "int64"]
          - [name, "object"]
          - [email, "object"]
          - [signup_date, "datetime64[ns]"]

  # Structure-only validation (column names without types)
  - col_schema_match:
      schema:
        columns:
          - customer_id
          - name
          - email
      complete: false
      brief: "Check that core columns exist"
```

3.4.3.1 Schema Validation Options

Schema validations support the full range of validation options:

```yaml
tbl: data_file.csv
steps:
  - col_schema_match:
      schema:
        columns:
          - [id, "int64"]
          - name
      complete: false                  # Allow extra columns
      in_order: false                  # Column order doesn't matter
      case_sensitive_colnames: false   # Case-insensitive column names
      case_sensitive_dtypes: false     # Case-insensitive type names
      full_match_dtypes: false         # Allow partial type matching
      brief: "Flexible schema validation"
```

3.4.3.2 Other Structure Validations

```yaml
tbl: customer_data.csv
steps:
  # Check column count
  - col_count_match:
      count: 4
```

3.4.4 Trend Validations

Validate that values follow increasing or decreasing patterns across rows:

```yaml
tbl: time_series_data.csv
steps:
  # Ensure timestamp values increase
  - col_vals_increasing:
      columns: timestamp
      brief: "Timestamps must be in chronological order"

  # Validate countdown timer decreases
  - col_vals_decreasing:
      columns: countdown
      allow_stationary: true
      brief: "Countdown values should decrease (ties allowed)"

  # Check trend with tolerance
  - col_vals_increasing:
      columns: temperature
      decreasing_tol: 0.5
      brief: "Temperature trends upward (small drops < 0.5°C allowed)"
```

3.4.5 Specification-based Validations

Validate values against common data specifications like email addresses, URLs, postal codes, and more:

```yaml
tbl: user_contact_info.csv
steps:
  # Validate email addresses
  - col_vals_within_spec:
      columns: email
      spec: "email"

  # Validate US ZIP codes
  - col_vals_within_spec:
      columns: zip_code
      spec: "postal_code[US]"

  # Validate URLs
  - col_vals_within_spec:
      columns: website
      spec: "url"
      na_pass: true
```

Available specifications include: `"email"`, `"url"`, `"phone"`, `"ipv4"`, `"ipv6"`, `"mac"`, `"isbn"`, `"vin"`, `"credit_card"`, `"swift"`, `"postal_code[<country>]"`, `"iban[<country>]"`.

3.4.6 Table Comparison

Validate that an entire table matches a reference table:

```yaml
tbl: processed_output.csv
steps:
  # Compare against expected output
  - tbl_match:
      tbl_compare:
        python: |
          pb.load_dataset("expected_output", tbl_type="polars")
      brief: "Output matches expected results"
```

The `tbl_match()` validation performs comprehensive comparison including column count, row count, schema, and data values. It supports cross-backend validation (e.g., comparing Polars vs. Pandas DataFrames).

3.4.7 AI-Powered Validation

Use Large Language Models to validate data based on natural language criteria:

```yaml
tbl: customer_feedback.csv
steps:
  # Validate sentiment
  - prompt:
      prompt: "Customer feedback should express positive sentiment"
      model: "anthropic:claude-sonnet-4"
      columns_subset: [feedback_text, rating]
      batch_size: 500
      thresholds:
        warning: 0.1

  # Validate semantic correctness
  - prompt:
      prompt: "Product descriptions should mention the product category and at least one benefit"
      model: "openai:gpt-4"
      columns_subset: [product_name, description, category]
```

Note: AI validations require API keys to be set as environment variables (e.g., `ANTHROPIC_API_KEY`, `OPENAI_API_KEY`) or in a `.env` file. These validations are best suited for semantic, context-dependent, or subjective quality checks rather than simple numeric comparisons.

3.5 Thresholds and Severity Levels

Thresholds determine when validation failures trigger different severity levels. You can set global thresholds for the entire workflow:

```yaml
tbl: sales_data.csv
tbl_name: "Sales Data Quality Check"
thresholds:
  warning: 0.05    # 5% failure rate triggers warning
  error: 0.10      # 10% failure rate triggers error
  critical: 0.15   # 15% failure rate triggers critical
steps:
  - col_vals_not_null:
      columns: [customer_id, amount]
  - col_vals_gt:
      columns: amount
      value: 0
```

You can also set thresholds for individual validation steps:

```yaml
tbl: user_data.csv
steps:
  - col_vals_not_null:
      columns: email
      thresholds:
        warning: 1      # Any missing email is a warning
        error: 0.01     # 1% missing emails is an error

  - col_vals_regex:
      columns: email
      pattern: "^[\\w\\.-]+@[\\w\\.-]+\\.[a-zA-Z]{2,}$"
      thresholds:
        error: 1        # Any invalid email format is an error
```

3.6 Actions: Responding to Validation Failures

Actions define what happens when validation thresholds are exceeded. You can use string templates with placeholder variables or callable functions.

3.6.1 String Template Actions

```yaml
tbl: orders.csv
thresholds:
  warning: 0.02
  error: 0.05
actions:
  warning: "Warning: Step {step} found {n_failed} failures in {col} column"
  error: "Error in {TYPE} validation: {n_failed}/{n} rows failed (Step {step})"
  critical: "Critical failure detected at {time}"
steps:
  - col_vals_not_null:
      columns: [order_id, customer_id]
```

Available template variables include:

- `{step}`: validation step number
- `{col}`: column name being validated
- `{val}`: specific failing value (when applicable)
- `{n_failed}`: number of failing rows
- `{n}`: total number of rows checked
- `{TYPE}`: validation method name (e.g., "COL_VALS_NOT_NULL")
- `{LEVEL}`: severity level ("WARNING", "ERROR", "CRITICAL")
- `{time}`: timestamp of the validation

3.6.2 Callable Actions

For more complex responses, use Python callable functions:

```yaml
tbl: critical_data.csv
thresholds:
  error: 1
actions:
  error:
    python: |
      lambda: print("ALERT: Critical data validation failed!")
  critical:
    python: |
      lambda: print("CRITICAL: Validation failure - manual intervention required!")
steps:
  - col_vals_not_null:
      columns: [transaction_id, amount]
```

Note: The Python environment in YAML actions is restricted for security. You can use built-in functions like `print()`, basic operations, and available DataFrame libraries, but cannot import external modules like `requests` or `logging`. For external notifications, consider using string template actions or handling alerts in your application code after the validation completes.

3.6.3 Step-level Actions

You can also define actions for individual validation steps:

```yaml
tbl: financial_data.csv
steps:
  - col_vals_not_null:
      columns: account_balance
      thresholds:
        error: 1
      actions:
        error: "Missing account balance detected in step {step}."

  - col_vals_gt:
      columns: account_balance
      value: 0
      actions:
        warning:
          python: |
            lambda: print("Negative balance warning triggered.")
```

3.7 Advanced Features

3.7.1 Pre-processing with the `pre` Parameter

You can apply data transformations before validation using the `pre` parameter:

```yaml
tbl: transactions.csv
steps:
  # Validate only recent transactions
  - col_vals_gt:
      columns: amount
      value: 0
      pre:
        python: |
          lambda df: df.filter(
              pl.col("transaction_date") >= "2024-01-01"
          )

  # Check completeness for active customers only
  - col_vals_not_null:
      columns: [email, phone]
      pre: |
        lambda df: df.filter(pl.col("status") == "active")
```

Note that you can use either the explicit `python:` block syntax or the shortcut syntax (just `pre: |`) for the lambda expressions.

3.7.2 Complex Expressions

For advanced validation logic, use a `col_vals_expr` step with custom expressions:

```yaml
tbl: sales_data.csv
steps:
  # Custom business logic validation
  - col_vals_expr:
      expr:
        python: |
          (
            pl.when(pl.col("product_type") == "premium")
            .then(pl.col("price") >= 100)
            .when(pl.col("product_type") == "standard")
            .then(pl.col("price").is_between(20, 99))
            .otherwise(pl.col("price") <= 19)
          )
```

3.7.3 Brief Descriptions

Add human-readable descriptions to validation steps. The `brief` parameter supports string templating and automatic generation:

```yaml
tbl: customer_data.csv
brief: "Customer data quality validation for {auto}"
steps:
  - col_vals_not_null:
      columns: customer_id
      brief: "Ensure all customers have valid IDs"

  - col_vals_regex:
      columns: email
      pattern: "^[\\w\\.-]+@[\\w\\.-]+\\.[a-zA-Z]{2,}$"
      brief: "Validate email format compliance"

  - col_vals_between:
      columns: age
      left: 13
      right: 120
      brief: "Check reasonable age ranges"

  # Use automatic brief generation
  - col_vals_not_null:
      columns: phone_number
      brief: true

  # Template variables in briefs
  - col_vals_in_set:
      columns: status
      set: [active, inactive, pending]
      brief: "Column '{col}' must be one of: {set}"
```

Brief Templating Options:

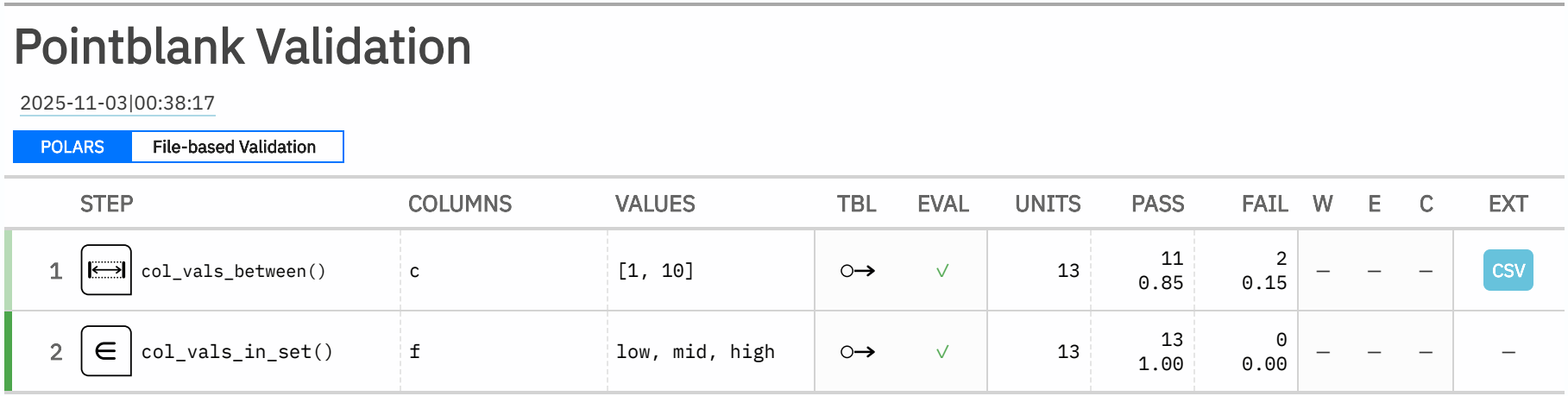
- custom strings: Write your own descriptive text
- `true`: Automatically generates a brief based on the validation method and parameters
- `{auto}`: Placeholder for auto-generated text within custom strings
- template variables: Use the same variables available in actions:
  - `{col}`: column name(s) being validated
  - `{step}`: the step number in the validation plan
  - `{value}`: the comparison value used in the validation (for single-value comparisons)
  - `{pattern}`: for regex validations, the pattern being matched

3.8 Working with YAML Files

3.8.1 Loading from Files

You can save your YAML configuration to files and load them:

```
# Create a YAML file
yaml_content = """
tbl: small_table
tbl_name: "File-based Validation"
steps:
  - col_vals_between:
      columns: c
      left: 1
      right: 10
  - col_vals_in_set:
      columns: f
      set: [low, mid, high]
"""

# Save to file
from pathlib import Path
yaml_file = Path("validation_config.yaml")
yaml_file.write_text(yaml_content)

# Load and execute
result = pb.yaml_interrogate(yaml_file)
result
```



3.8.2 Converting YAML to Python

Use `yaml_to_python()` to generate equivalent Python code from your YAML configuration:

```
yaml_config = """
tbl: small_table
tbl_name: "Example Validation"
thresholds:
  warning: 0.1
  error: 0.2
actions:
  warning: "Warning: {TYPE} validation failed"
steps:
  - col_vals_gt:
      columns: a
      value: 0
  - col_vals_in_set:
      columns: f
      set: [low, mid, high]
"""

# Generate Python code
python_code = pb.yaml_to_python(yaml_config)
print(python_code)
```

```python
import pointblank as pb

(
    pb.Validate(
        data=pb.load_dataset("small_table", tbl_type="polars"),
        tbl_name="Example Validation",
        thresholds=pb.Thresholds(warning=0.1, error=0.2),
        actions=pb.Actions(warning="Warning: {TYPE} validation failed"),
    )
    .col_vals_gt(columns="a", value=0)
    .col_vals_in_set(columns="f", set=["low", "mid", "high"])
    .interrogate()
)
```

This is useful for:

- learning how YAML maps to Python API calls
- transitioning from YAML to code-based workflows
- generating documentation that shows both approaches
- debugging YAML configurations

3.9 Practical Examples

3.9.1 Data Pipeline Validation

Here's a comprehensive example for validating data in a processing pipeline:

```yaml
tbl:
  python: |
    (
      pl.scan_csv("raw_data/customer_events.csv")
      .filter(pl.col("event_date") >= "2024-01-01")
    )

tbl_name: "Customer Events Pipeline Validation"
label: "Daily data quality check for customer events"

thresholds:
  warning: 0.01   # 1% failure rate
  error: 0.05     # 5% failure rate

actions:
  warning: "Pipeline warning: {TYPE} validation found {n_failed} issues"
  error:
    python: |
      lambda: print("ERROR: Pipeline validation failed - manual review required")

steps:
  # Schema validation
  - col_schema_match:
      schema:
        columns:
          - [customer_id, "int64"]
          - [event_type, "object"]
          - [event_date, "object"]
          - [revenue, "float64"]
      brief: "Validate table structure matches expected schema"

  # Data completeness
  - col_vals_not_null:
      columns: [customer_id, event_type, event_date]
      brief: "Critical fields must be complete"

  # Business logic validation
  - col_vals_in_set:
      columns: event_type
      set: [signup, purchase, cancellation, upgrade]
      brief: "Event types must be from approved list"

  # Data quality checks
  - col_vals_gt:
```

```
    columns: revenue
    value: 0
    na_pass: true
    brief: "Revenue values must be positive when present"

# Temporal validation
- col_vals_expr:
    expr:
      python: |
        pl.col("event_date").str.strptime(pl.Date, "%Y-%m-%d").is_not_null()
    brief: "Event dates must be valid YYYY-MM-DD format"
```

3.9.2 Quality Monitoring Dashboard

For ongoing data quality monitoring:

```yaml
tbl: warehouse/daily_metrics.parquet
tbl_name: "Daily Metrics Quality Check"

thresholds:
  warning: 5      # 5 failing rows
  error: 50       # 50 failing rows
  critical: 100   # 100 failing rows

actions:
  warning: "Quality check warning: {n_failed} rows failed {TYPE} validation"
  error: "Quality degradation detected: Step {step} failed for {n_failed}/{n} rows"
  critical:
    python: |
      lambda: print("CRITICAL: Data quality failure detected - immediate attention required")
  highest_only: false

steps:
  - row_count_match:
      count: 10000
      brief: "Verify expected daily record count"

  - col_vals_not_null:
      columns: [date, metric_value, source_system]
      brief: "Core fields must be complete"

  - col_vals_between:
      columns: metric_value
      left: 0
      right: 1000000
      brief: "Metric values within reasonable range"

  - rows_distinct:
      columns_subset: [date, metric_name, source_system]
      brief: "No duplicate metric records per day"
```

3.10 Best Practices

3.10.1 Organization and Structure

1. use descriptive names: give your validations clear `tbl_name` and `label` values

2. add brief descriptions: document what each validation step checks
3. group related validations: organize steps logically (schema, completeness, business rules)
4. version control: store YAML files in git alongside your data processing code

3.10.2 Error Handling and Monitoring

1. set appropriate thresholds: start conservative and adjust based on your data patterns
2. use actions for alerting: set up notifications for critical failures
3. document expected failures: some data quality issues might be acceptable
4. monitor validation results: track validation performance over time

3.10.3 Performance Considerations

1. use the `pre` parameter efficiently: apply filters early to reduce data volume
2. order validations strategically: put fast, likely-to-fail checks first
3. consider data source location: local files are faster than remote sources
4. use appropriate column selections: only validate the columns you need

3.11 Wrapping Up

YAML validation workflows provide a powerful, declarative approach to data validation in Pointblank. Such workflows are great at expressing common validation patterns in a readable format that can be easily shared, version controlled, and maintained by teams.

Key advantages of YAML workflows:

- readable: non-programmers can understand and contribute to validation logic
- maintainable: easy to modify validation rules without changing application code
- portable: YAML files can be shared between projects and teams
- version controlled: track changes to validation logic over time
- flexible: support for simple checks and complex custom logic

Use YAML workflows when you want declarative, maintainable validation definitions, and fall back to the Python API when you need complex programmatic logic or tight integration with application code. The two approaches complement each other well and can be used together as your validation needs evolve.

This reference provides a comprehensive guide to all YAML keys and parameters supported by Pointblank's YAML validation workflows. Use this document as a quick lookup when building validation configurations.

3.12 Global Configuration Keys

3.12.1 Top-level Structure

```yaml
tbl: data_source                      # REQUIRED: Data source specification
df_library: "polars"                  # OPTIONAL: DataFrame library ("polars", "pandas", "duckdb")
tbl_name: "Custom Table Name"         # OPTIONAL: Human-readable table name
label: "Validation Description"       # OPTIONAL: Description for the validation workflow
lang: "en"                            # OPTIONAL: Language code (default: "en")
locale: "en"                          # OPTIONAL: Locale setting (default: "en")
brief: "Global brief: {auto}"         # OPTIONAL: Global brief template
thresholds:                           # OPTIONAL: Global failure thresholds
  warning: 0.1
  error: 0.2
  critical: 0.3
actions:                              # OPTIONAL: Global failure actions
  warning: "Warning message template"
  error: "Error message template"
  critical: "Critical message template"
  highest_only: false
steps:                                # REQUIRED: List of validation steps
  - validation_method_name
  - validation_method_name:
      parameter: value
```

3.12.2 Data Source (`tbl`)

The `tbl` key specifies the data source and supports multiple formats:

```yaml
# File paths
tbl: "data/file.csv"
tbl: "data/file.parquet"

# Built-in datasets
tbl: small_table
tbl: game_revenue
tbl: nycflights

# Python expressions for complex data loading
tbl:
  python: |
    pl.scan_csv("data.csv").filter(pl.col("date") >= "2024-01-01")
```

3.12.2.1 Using Templates with `set_tbl=`

For reusable validation templates that will always use a custom data source via the `set_tbl=` parameter in `yaml_interrogate()`, the `tbl` field is still required but its value doesn't matter since it will be overridden. Recommended approaches:

```yaml
# Option 1: Use a valid dataset name (gets overridden anyway)
tbl: small_table  # Will be ignored when `set_tbl=` is used

# Option 2: Use YAML null (clearest semantic intent)
tbl: null  # Indicates table will be provided via `set_tbl=`
```

When using `yaml_interrogate()` with `set_tbl=`, the validation template becomes fully reusable:

```
# Define reusable template
template = """
tbl: null  # Will be overridden
tbl_name: "Sales Validation"
steps:
  - col_exists:
      columns: [customer_id, revenue, region]
  - col_vals_gt:
      columns: [revenue]
      value: 0
"""

# Apply to different datasets
q1_result = pb.yaml_interrogate(template, set_tbl=q1_data)
q2_result = pb.yaml_interrogate(template, set_tbl=q2_data)
```

3.12.3 DataFrame Library (`df_library`)

The `df_library` key controls which DataFrame library is used to load data sources. This parameter affects both built-in datasets and file loading:

```
# Use Polars DataFrames (default)
df_library: polars

# Use Pandas DataFrames
df_library: pandas

# Use DuckDB tables (via Ibis)
df_library: duckdb
```

Examples with different libraries:

```yaml
# Load built-in dataset as Pandas DataFrame
tbl: small_table
df_library: pandas
steps:
  - specially:
      expr: "lambda df: df.assign(validation_result=df['a'] > 0)"

# Load CSV file as Polars DataFrame
tbl: "data/sales.csv"
df_library: polars
steps:
  - col_vals_gt:
      columns: amount
      value: 0

# Load dataset as DuckDB table
tbl: nycflights
df_library: duckdb
steps:
  - row_count_match:
      count: 336776
```

The `df_library` parameter is particularly useful when:

- using validation expressions that require specific DataFrame APIs (e.g., Pandas `.assign()`, Polars `.select()`)
- integrating with existing pipelines that use a specific DataFrame library
- optimizing performance for different data sizes and operations
- ensuring compatibility with downstream processing steps

3.12.4 Global Thresholds

Thresholds define when validation failures trigger different severity levels:

```yaml
thresholds:
  warning: 0.05    # 5% failure rate triggers warning
  error: 0.10      # 10% failure rate triggers error
  critical: 0.15   # 15% failure rate triggers critical
```

- values: numbers between `0` and `1` (percentages) or integers (row counts)

- levels: `warning`, `error`, `critical`

3.12.5 Global Actions

Actions define responses when thresholds are exceeded. When supplying a string to a severity level ('warning', 'error', 'critical'), you can use template variables that will be automatically substituted with contextual information:

```yaml
actions:
  warning: "Warning: {n_failed} failures in step {step}"
  error:
    python: |
      lambda: print("Error detected!")
  critical: "Critical failure at {time}"
  highest_only: false        # Execute all applicable actions vs. only highest severity
```

Template variables available for action strings:

- `{step}`: current validation step number
- `{col}`: column name(s) being validated
- `{val}`: validation value or threshold
- `{n_failed}`: number of failing records
- `{n}`: total number of records
- `{type}`: validation method type
- `{level}`: severity level ('warning'/'error'/'critical')
- `{time}`: timestamp of validation

3.13 Validation Methods Reference

3.13.1 Column Value Validations

3.13.1.1 Comparison Methods

`col_vals_gt`: are column data greater than a fixed value or data in another column?

```yaml
- col_vals_gt:
    columns: [column_name]             # REQUIRED: Column(s) to validate
    value: 100                         # REQUIRED: Comparison value
    na_pass: true                      # OPTIONAL: Pass NULL values (default: false)
    pre: |                             # OPTIONAL: Data preprocessing
      lambda df: df.filter(condition)
    thresholds:                        # OPTIONAL: Step-level thresholds
      warning: 0.1
    actions:                           # OPTIONAL: Step-level actions
      warning: "Custom message"
    brief: "Values must be > 100"      # OPTIONAL: Step description
```

`col_vals_lt`: are column data less than a fixed value or data in another column?

```yaml
- col_vals_lt:
    columns: [column_name]
    value: 100
    na_pass: true
    # ... (same parameters as col_vals_gt)
```

`col_vals_ge`: are column data greater than or equal to a fixed value or data in another column?

```yaml
- col_vals_ge:
    columns: [column_name]
    value: 100
    na_pass: true
    # ... (same parameters as col_vals_gt)
```

`col_vals_le`: are column data less than or equal to a fixed value or data in another column?

```yaml
- col_vals_le:
    columns: [column_name]
    value: 100
    na_pass: true
    # ... (same parameters as col_vals_gt)
```

`col_vals_eq`: are column data equal to a fixed value or data in another column?

```yaml
- col_vals_eq:
    columns: [column_name]
    value: "expected_value"
    na_pass: true
    # ... (same parameters as col_vals_gt)
```

`col_vals_ne`: are column data not equal to a fixed value or data in another column?

```yaml
- col_vals_ne:
    columns: [column_name]
    value: "forbidden_value"
    na_pass: true
    # ... (same parameters as col_vals_gt)
```

3.13.1.2 Range Methods

`col_vals_between`: are column data between two specified values (inclusive)?

```yaml
- col_vals_between:
    columns: [column_name]             # REQUIRED: Column(s) to validate
    left: 0                            # REQUIRED: Lower bound
    right: 100                         # REQUIRED: Upper bound
    inclusive: [true, true]            # OPTIONAL: Include bounds [left, right]
    na_pass: false                     # OPTIONAL: Pass NULL values
    pre: |                             # OPTIONAL: Data preprocessing
      lambda df: df.filter(condition)
    thresholds:                        # OPTIONAL: Step-level thresholds
      warning: 0.1
    actions:                           # OPTIONAL: Step-level actions
      warning: "Custom message"
    brief: "Values between 0 and 100"  # OPTIONAL: Step description
```

`col_vals_outside`: are column data outside of two specified values?

```yaml
- col_vals_outside:
    columns: [column_name]
    left: 0
    right: 100
    inclusive: [false, false]          # OPTIONAL: Exclude bounds [left, right]
    na_pass: false
    # ... (same parameters as col_vals_between)
```

3.13.1.3 Set Membership Methods

`col_vals_in_set`: are column data part of a specified set of values?

```yaml
- col_vals_in_set:
    columns: [column_name]             # REQUIRED: Column(s) to validate
    set: [value1, value2, value3]      # REQUIRED: Allowed values
    na_pass: false                     # OPTIONAL: Pass NULL values
    pre: |                             # OPTIONAL: Data preprocessing
      lambda df: df.filter(condition)
    thresholds:                        # OPTIONAL: Step-level thresholds
      warning: 0.1
    actions:                           # OPTIONAL: Step-level actions
      warning: "Custom message"
    brief: "Values in allowed set"     # OPTIONAL: Step description
```

`col_vals_not_in_set`: are column data not part of a specified set of values?

```yaml
- col_vals_not_in_set:
    columns: [column_name]
    set: [forbidden1, forbidden2]      # REQUIRED: Forbidden values
    na_pass: false
    # ... (same parameters as col_vals_in_set)
```

3.13.1.4 NULL Value Methods

`col_vals_null`: are column data null (missing)?

```yaml
- col_vals_null:
    columns: [column_name]             # REQUIRED: Column(s) to validate
    pre: |                             # OPTIONAL: Data preprocessing
      lambda df: df.filter(condition)
    thresholds:                        # OPTIONAL: Step-level thresholds
      warning: 0.1
    actions:                           # OPTIONAL: Step-level actions
      warning: "Custom message"
    brief: "Values must be NULL"       # OPTIONAL: Step description
```

`col_vals_not_null`: are column data not null (not missing)?

```yaml
- col_vals_not_null:
    columns: [column_name]
    # ... (same parameters as col_vals_null)
```

3.13.1.5 Pattern Matching Methods

`col_vals_regex`: do string-based column data match a regular expression?

```yaml
- col_vals_regex:
    columns: [column_name]             # REQUIRED: Column(s) to validate
    pattern: "^[A-Z]{2,3}$"            # REQUIRED: Regular expression pattern
    na_pass: false                     # OPTIONAL: Pass NULL values
    pre: |                             # OPTIONAL: Data preprocessing
      lambda df: df.filter(condition)
    thresholds:                        # OPTIONAL: Step-level thresholds
      warning: 0.1
    actions:                           # OPTIONAL: Step-level actions
      warning: "Custom message"
    brief: "Values match pattern"      # OPTIONAL: Step description
```

`col_vals_within_spec`: do column data conform to a specification (email, URL, postal codes, etc.)?

```yaml
- col_vals_within_spec:
    columns: [column_name]             # REQUIRED: Column(s) to validate
    spec: "email"                      # REQUIRED: Specification type
    na_pass: false                     # OPTIONAL: Pass NULL values
    pre: |                             # OPTIONAL: Data preprocessing
      lambda df: df.filter(condition)
    thresholds:                        # OPTIONAL: Step-level thresholds
      warning: 0.1
    actions:                           # OPTIONAL: Step-level actions
      warning: "Custom message"
    brief: "Values match spec"         # OPTIONAL: Step description
```

Available specification types:

- `"email"` - Email addresses
- `"url"` - Internet URLs
- `"phone"` - Phone numbers
- `"ipv4"` - IPv4 addresses
- `"ipv6"` - IPv6 addresses
- `"mac"` - MAC addresses
- `"isbn"` - International Standard Book Numbers (10 or 13 digit)
- `"vin"` - Vehicle Identification Numbers
- `"credit_card"` - Credit card numbers (uses Luhn algorithm)
- `"swift"` - Business Identifier Codes (SWIFT-BIC)
- `"postal_code[<country_code>]"` - Postal codes for specific countries (e.g., `"postal_code[US]"`, `"postal_code[CA]"`)
- `"zip"` - Alias for US ZIP codes (`"postal_code[US]"`)
- `"iban[<country_code>]"` - International Bank Account Numbers (e.g., `"iban[DE]"`, `"iban[FR]"`)

Examples:

```yaml
# Email validation
- col_vals_within_spec:
    columns: user_email
    spec: "email"

# US postal codes
- col_vals_within_spec:
    columns: zip_code
    spec: "postal_code[US]"

# German IBAN
- col_vals_within_spec:
    columns: account_number
    spec: "iban[DE]"
```

3.13.1.6 Custom Expression Methods

`col_vals_expr`: do column data agree with a predicate expression?

```yaml
- col_vals_expr:
    expr:                           # REQUIRED: Custom validation expression
      python: |
        pl.when(pl.col("status") == "active")
        .then(pl.col("value") > 0)
        .otherwise(pl.lit(True))
    pre: |                          # OPTIONAL: Data preprocessing
      lambda df: df.filter(condition)
    thresholds:                     # OPTIONAL: Step-level thresholds
      warning: 0.1
    actions:                        # OPTIONAL: Step-level actions
      warning: "Custom message"
    brief: "Custom validation rule"    # OPTIONAL: Step description
```

3.13.1.7 Trend Validation Methods

`col_vals_increasing`: are column data increasing row-by-row?

```yaml
- col_vals_increasing:
    columns: [column_name]             # REQUIRED: Column(s) to validate
    allow_stationary: false            # OPTIONAL: Allow consecutive equal values (default: false)
    decreasing_tol: 0.5                # OPTIONAL: Tolerance for negative movement (default: null)
    na_pass: false                     # OPTIONAL: Pass NULL values
    pre: |                             # OPTIONAL: Data preprocessing
      lambda df: df.filter(condition)
    thresholds:                        # OPTIONAL: Step-level thresholds
      warning: 0.1
    actions:                           # OPTIONAL: Step-level actions
      warning: "Custom message"
    brief: "Values must increase"      # OPTIONAL: Step description
```

This validation checks whether values in a column increase as you move down the rows. Useful for validating time-series data, sequence numbers, or any monotonically increasing values.

Parameters:

- `allow_stationary`: If `true`, allows consecutive values to be equal (stationary phases). For example, `[1, 2, 2, 3]` would pass when `true` but fail at the third value when `false`.
- `decreasing_tol`: Absolute tolerance for negative movement. Setting this to `0.5` means values can decrease by up to 0.5 units and still pass. Setting any value also sets `allow_stationary` to `true`.

Examples:

```yaml
# Strict increasing validation
- col_vals_increasing:
    columns: timestamp_seconds
    brief: "Timestamps must strictly increase"

# Allow stationary values
- col_vals_increasing:
    columns: version_number
    allow_stationary: true
    brief: "Version numbers should increase (ties allowed)"

# With tolerance for small decreases
- col_vals_increasing:
    columns: temperature
    decreasing_tol: 0.1
    brief: "Temperature trend (small drops allowed)"
```

`col_vals_decreasing`: are column data decreasing row-by-row?

```yaml
- col_vals_decreasing:
    columns: [column_name]             # REQUIRED: Column(s) to validate
    allow_stationary: false            # OPTIONAL: Allow consecutive equal values (default: false)
    increasing_tol: 0.5                # OPTIONAL: Tolerance for positive movement (default: null)
    na_pass: false                     # OPTIONAL: Pass NULL values
    pre: |                             # OPTIONAL: Data preprocessing
      lambda df: df.filter(condition)
    thresholds:                        # OPTIONAL: Step-level thresholds
      warning: 0.1
    actions:                           # OPTIONAL: Step-level actions
      warning: "Custom message"
    brief: "Values must decrease"      # OPTIONAL: Step description
```

This validation checks whether values in a column decrease as you move down the rows. Useful for countdown timers, inventory depletion, or any monotonically decreasing values.

Parameters:

- `allow_stationary`: If `true`, allows consecutive values to be equal (stationary phases). For example, `[10, 8, 8, 5]` would pass when `true` but fail at the third value when `false`.

- `increasing_tol`: Absolute tolerance for positive movement. Setting this to `0.5` means values can increase by up to 0.5 units and still pass. Setting any value also sets `allow_stationary` to `true`.

Examples:

```yaml
# Strict decreasing validation
- col_vals_decreasing:
    columns: countdown_timer
    brief: "Timer must strictly decrease"

# Allow stationary values
- col_vals_decreasing:
    columns: priority_score
    allow_stationary: true
    brief: "Priority scores should decrease (ties allowed)"

# With tolerance for small increases
- col_vals_decreasing:
    columns: stock_level
    increasing_tol: 5
    brief: "Stock levels decrease (small restocks allowed)"
```

3.13.2 Row-based Validations

`rows_distinct`: are row data distinct?

```yaml
- rows_distinct                       # Simple form

- rows_distinct:                      # With parameters
    columns_subset: [col1, col2]      # OPTIONAL: Check subset of columns
    pre: |                            # OPTIONAL: Data preprocessing
      lambda df: df.filter(condition)
    thresholds:                       # OPTIONAL: Step-level thresholds
      warning: 0.1
    actions:                          # OPTIONAL: Step-level actions
      warning: "Custom message"
    brief: "No duplicate rows"        # OPTIONAL: Step description
```

`rows_complete`: are row data complete?

```yaml
- rows_complete                        # Simple form

- rows_complete:                       # With parameters
    columns_subset: [col1, col2]       # OPTIONAL: Check subset of columns
    pre: |                             # OPTIONAL: Data preprocessing
      lambda df: df.filter(condition)
    thresholds:                        # OPTIONAL: Step-level thresholds
      warning: 0.1
    actions:                           # OPTIONAL: Step-level actions
      warning: "Custom message"
    brief: "Complete rows only"        # OPTIONAL: Step description
```

3.13.3 Structure Validations

`col_exists`: does column exist in the table?

```yaml
- col_exists:
    columns: [col1, col2, col3]        # REQUIRED: Column(s) that must exist
    thresholds:                        # OPTIONAL: Step-level thresholds
      warning: 0.1
    actions:                           # OPTIONAL: Step-level actions
      warning: "Custom message"
    brief: "Required columns exist"    # OPTIONAL: Step description
```

`col_schema_match`: does the table have expected column names and data types?

```yaml
- col_schema_match:
    schema:                          # REQUIRED: Expected schema
      columns:
        - [column_name, "data_type"]   # Column with type validation
        - column_name                  # Column name only (no type check)
        - [column_name]                # Alternative syntax
    complete: true                   # OPTIONAL: Require exact column set
    in_order: true                   # OPTIONAL: Require exact column order
    case_sensitive_colnames: true    # OPTIONAL: Case-sensitive column names
    case_sensitive_dtypes: true      # OPTIONAL: Case-sensitive data types
    full_match_dtypes: true          # OPTIONAL: Exact type matching
    thresholds:                      # OPTIONAL: Step-level thresholds
      warning: 0.1
    actions:                         # OPTIONAL: Step-level actions
      warning: "Custom message"
    brief: "Schema validation"       # OPTIONAL: Step description
```

`row_count_match`: does the table have n rows?

```yaml
- row_count_match:
    count: 1000                      # REQUIRED: Expected row count
    thresholds:                      # OPTIONAL: Step-level thresholds
      warning: 0.1
    actions:                         # OPTIONAL: Step-level actions
      warning: "Custom message"
    brief: "Expected row count"      # OPTIONAL: Step description
```

`col_count_match`: does the table have n columns?

```yaml
- col_count_match:
    count: 10                        # REQUIRED: Expected column count
    thresholds:                      # OPTIONAL: Step-level thresholds
      warning: 0.1
    actions:                         # OPTIONAL: Step-level actions
      warning: "Custom message"
    brief: "Expected column count"   # OPTIONAL: Step description
```

`tbl_match`: does the table match a comparison table?

```yaml
- tbl_match:
    tbl_compare:                      # REQUIRED: Comparison table
      python: |
        pb.load_dataset("reference_table", tbl_type="polars")
    pre: |                            # OPTIONAL: Data preprocessing
      lambda df: df.filter(condition)
    thresholds:                       # OPTIONAL: Step-level thresholds
      warning: 0.0
    actions:                          # OPTIONAL: Step-level actions
      warning: "Custom message"
    brief: "Table structure matches"  # OPTIONAL: Step description
```

This validation performs a comprehensive comparison between the target table and a comparison table, using progressively stricter checks:

1. **Column count match**: both tables have the same number of columns
2. **Row count match**: both tables have the same number of rows
3. **Schema match (loose)**: column names and dtypes match (case-insensitive, any order)
4. **Schema match (order)**: columns in correct order (case-insensitive names)
5. **Schema match (exact)**: column names match exactly (case-sensitive, correct order)
6. **Data match**: values in corresponding cells are identical

The validation fails at the first check that doesn't pass, making it easy to diagnose mismatches. This operates over a single test unit (pass/fail for complete table match).

Cross-backend validation: `tbl_match()` supports automatic backend coercion when comparing tables from different backends (e.g., Polars vs. Pandas, DuckDB vs. SQLite). The comparison table is automatically converted to match the target table's backend.

Examples:

```yaml
# Compare against reference dataset
- tbl_match:
    tbl_compare:
      python: |
        pb.load_dataset("expected_output", tbl_type="polars")
    brief: "Output matches expected results"

# Compare against CSV file
- tbl_match:
    tbl_compare:
      python: |
        pl.read_csv("reference_data.csv")
    brief: "Matches reference CSV"

# Compare with preprocessing on target table only
- tbl_match:
    tbl_compare:
      python: |
        pb.load_dataset("reference_table", tbl_type="polars")
    pre: |
      lambda df: df.select(["id", "name", "value"])
    brief: "Selected columns match reference"
```

3.13.4 Special Validation Methods

`conjointly`: are multiple validations having a joint dependency?

```yaml
- conjointly:
    expressions:                      # REQUIRED: List of lambda expressions
      - "lambda df: df['d'] > df['a']"
      - "lambda df: df['a'] > 0"
      - "lambda df: df['a'] + df['d'] < 12000"
    thresholds:                       # OPTIONAL: Step-level thresholds
      warning: 0.1
    actions:                          # OPTIONAL: Step-level actions
      warning: "Custom message"
    brief: "All conditions must pass"  # OPTIONAL: Step description
```

`specially`: do table data pass a custom validation function?

```yaml
- specially:
    expr:                              # REQUIRED: Custom validation function
      "lambda df: df.select(pl.col('a') + pl.col('d') > 0)"
    thresholds:                        # OPTIONAL: Step-level thresholds
      warning: 0.1
    actions:                           # OPTIONAL: Step-level actions
      warning: "Custom message"
    brief: "Custom validation"         # OPTIONAL: Step description
```

Alternative syntax with Python expressions:

```yaml
- specially:
    expr:
      python: |
        lambda df: df.select(pl.col('amount') > 0)
```

For Pandas DataFrames (when using `df_library: pandas`):

```yaml
- specially:
    expr: "lambda df: df.assign(is_valid=df['a'] + df['d'] > 0)"
```

3.13.5 AI-Powered Validation

`prompt`: validate rows using AI/LLM-powered analysis

```yaml
- prompt:
    prompt: "Values should be positive and realistic"  # REQUIRED: Natural language criteria
    model: "anthropic:claude-sonnet-4"                 # REQUIRED: Model identifier
    columns_subset: [column1, column2]                 # OPTIONAL: Columns to validate
    batch_size: 1000                                   # OPTIONAL: Rows per batch (default: 1000)
    max_concurrent: 3                                  # OPTIONAL: Concurrent API requests (default: 3)
    pre: |                                             # OPTIONAL: Data preprocessing
      lambda df: df.filter(condition)
    thresholds:                                        # OPTIONAL: Step-level thresholds
      warning: 0.1
    actions:                                           # OPTIONAL: Step-level actions
      warning: "Custom message"
    brief: "AI validation"                             # OPTIONAL: Step description
```

This validation method uses Large Language Models (LLMs) to validate rows of data based on natural language criteria. Each row becomes a test unit that either passes or fails the validation criteria, producing binary True/False results that integrate with standard Pointblank reporting.

Supported models:

- **Anthropic**: `"anthropic:claude-sonnet-4"`, `"anthropic:claude-opus-4"`
- **OpenAI**: `"openai:gpt-4"`, `"openai:gpt-4-turbo"`, `"openai:gpt-3.5-turbo"`
- **Ollama**: `"ollama:<model-name>"` (e.g., `"ollama:llama3"`)
- **Bedrock**: `"bedrock:<model-name>"`

Authentication: API keys are automatically loaded from environment variables or `.env` files:

- **OpenAI**: Set `OPENAI_API_KEY` environment variable or add to `.env` file
- **Anthropic**: Set `ANTHROPIC_API_KEY` environment variable or add to `.env` file
- **Ollama**: No API key required (runs locally)
- **Bedrock**: Configure AWS credentials through standard AWS methods

Example `.env` file:

```
ANTHROPIC_API_KEY="your_anthropic_api_key_here"
OPENAI_API_KEY="your_openai_api_key_here"
```

Performance optimization: The validation process uses row signature memoization to avoid redundant LLM calls. When multiple rows have identical values in the selected columns, only one representative row is validated, and the result is applied to all matching rows. This dramatically reduces API costs and processing time for datasets with repetitive patterns.

Examples:

```yaml
# Basic AI validation
- prompt:
    prompt: "Email addresses should look realistic and professional"
    model: "anthropic:claude-sonnet-4"
    columns_subset: [email]

# Complex semantic validation
- prompt:
    prompt: "Product descriptions should mention the product category and include at least one benefit"
    model: "openai:gpt-4"
    columns_subset: [product_name, description, category]
    batch_size: 500
    max_concurrent: 5

# Sentiment analysis
- prompt:
    prompt: "Customer feedback should express positive sentiment"
    model: "anthropic:claude-sonnet-4"
    columns_subset: [feedback_text, rating]

# Context-dependent validation
- prompt:
    prompt: "For high-value transactions (amount > 1000), a detailed justification should be provided"
    model: "openai:gpt-4"
    columns_subset: [amount, justification, approver]
    thresholds:
      warning: 0.05
      error: 0.15

# Local model with Ollama
- prompt:
    prompt: "Transaction descriptions should be clear and professional"
    model: "ollama:llama3"
    columns_subset: [description]
```

Best practices for AI validation:

- Be specific and clear in your prompt criteria
- Include only necessary columns in `columns_subset` to reduce API costs
- Start with smaller `batch_size` for testing, increase for production
- Adjust `max_concurrent` based on API rate limits

- Use thresholds appropriate for probabilistic validation results
- Consider cost implications for large datasets
- Test prompts on sample data before full deployment

When to use AI validation:

- Semantic checks (e.g., “does the description match the category?”)
- Context-dependent validation (e.g., “is the justification appropriate for the amount?”)
- Subjective quality assessment (e.g., “is the text professional?”)
- Pattern recognition that’s hard to express programmatically
- Natural language understanding tasks

When NOT to use AI validation:

- Simple numeric comparisons (use `col_vals_gt`, `col_vals_lt`, etc.)
- Exact pattern matching (use `col_vals_regex`)
- Schema validation (use `col_schema_match`)
- Performance-critical validations with large datasets
- When deterministic results are required

3.14 Column Selection Patterns

All validation methods that accept a `columns` parameter support these selection patterns:

```yaml
# Single column
columns: column_name

# Multiple columns as list
columns: [col1, col2, col3]

# Column selector functions (when used in Python expressions)
columns:
  python: |
    starts_with("prefix_")

# Examples of common patterns
columns: [customer_id, order_id]     # Specific columns
columns: user_email                  # Single column
```

3.15 Parameter Details

3.15.1 Common Parameters

These parameters are available for most validation methods:

- `columns`: column selection (string, list, or selector expression)
- `na_pass`: whether to pass NULL/missing values (boolean, default: false)
- `pre`: data preprocessing function (Python lambda expression)
- `thresholds`: step-level failure thresholds (dict)
- `actions`: step-level failure actions (dict)
- `brief`: step description (string, boolean, or template)

3.15.2 Brief Parameter Options

The `brief` parameter supports several formats:

```yaml
brief: "Custom description"          # Custom text
brief: true                         # Auto-generated description
brief: false                        # No description
brief: "Step {step}: {auto}"        # Template with auto-generated text
brief: "Column '{col}' validation"  # Template with variables
```

template variables: `{step}`, `{col}`, `{value}`, `{set}`, `{pattern}`, `{auto}`

3.15.3 Python Expressions

Several parameters support Python expressions using the `python:` block syntax:

```yaml
# Data source loading
tbl:
  python: |
    pl.scan_csv("data.csv").filter(pl.col("active") == True)

# Preprocessing
pre:
  python: |
    lambda df: df.filter(pl.col("date") >= "2024-01-01")

# Custom expressions
expr:
  python: |
    pl.col("value").is_between(0, 100)

# Callable actions
actions:
  error:
    python: |
      lambda: print("VALIDATION ERROR: Critical data quality issue detected!")
```

Note: The Python environment in YAML is restricted for security. Only built-in functions (`print`, `len`, `str`, etc.), `Path` from pathlib, and available DataFrame libraries (`pl`, `pd`) are accessible. You cannot import additional modules like `requests`, `logging`, or custom libraries.

You can also use the shortcut syntax for lambda expressions:

```yaml
# Shortcut syntax (equivalent to python: block)
pre: |
  lambda df: df.filter(pl.col("status") == "active")
```

3.15.4 Restricted Python Environment

For security reasons, the Python environment in YAML configurations is restricted to a safe subset of functionality. The available namespace includes:

Built-in functions:

- basic types: `str`, `int`, `float`, `bool`, `list`, `dict`, `tuple`, `set`
- math functions: `sum`, `min`, `max`, `abs`, `round`, `len`
- iteration: `range`, `enumerate`, `zip`
- output: `print`

Available modules:

- `Path` from pathlib for file path operations
- `pb` (`pointblank`) for dataset loading and validation functions
- `pl` (`polars`) if available on the system
- `pd` (`pandas`) if available on the system

Restrictions:

- cannot import external libraries (`requests`, `logging`, `os`, `sys`, etc.)
- cannot use `__import__`, `exec`, `eval`, or other dynamic execution functions
- file operations are limited to `Path` functionality

Examples of valid callable actions:

```yaml
# Simple output with built-in functions
actions:
  warning:
    python: |
      lambda: print(f"WARNING: {sum([1, 2, 3])} validation issues detected")

# Using available variables and string formatting
actions:
  error:
    python: |
      lambda: print("ERROR: Data validation failed at " + str(len("validation")))

# Multiple statements in lambda (using parentheses)
actions:
  critical:
    python: |
      lambda: (
          print("CRITICAL ALERT:"),
          print("Immediate attention required"),
          print("Contact data team")
      )[-1]  # Return the last value
```

For complex alerting, logging, or external system integration, use string template actions instead of callable actions, and handle the external communication in your application code after validation completes.

3.16 Best Practices

3.16.1 Organization

- use descriptive `tbl_name` and `label` values
- add `brief` descriptions for complex validations
- group related validations logically
- use consistent indentation and formatting

3.16.2 Performance

- apply `pre` filters early to reduce data volume
- order validations from fast to slow

- use `columns_subset` for row-based validations when appropriate
- consider data source location (local vs. remote)
- choose `df_library` based on data size and operations:
  - `polars`: fastest for large datasets and analytical operations
  - `pandas`: best for complex transformations and data science workflows
  - `duckdb`: optimal for analytical queries on very large datasets

3.16.3 Maintainability

- store YAML files in version control
- use template variables in actions and briefs
- document expected failures with comments
- test configurations with `validate_yaml()` before deployment
- specify `df_library` explicitly when using library-specific validation expressions
- keep DataFrame library choice consistent within related validation workflows

3.16.4 Error Handling

- set appropriate thresholds based on data patterns
- use actions for monitoring and alerting
- start with conservative thresholds and adjust
- consider using `highest_only: false` for comprehensive reporting

4 Post Interrogation

After interrogating your data with a validation plan, Pointblank automatically generates a *validation report*. That tabular report comprehensively summarizes the results of all validation steps. It'll be your primary tool for understanding data quality at a glance, identifying issues, and communicating results to stakeholders.

Validation reports are Great Tables objects that provide rich information about each validation step. It includes: identifying information for the step, pass/fail statistics, threshold exceedances, and visual status indicators. The report makes it easy to quickly assess overall data quality and pinpoint specific areas that need attention.

4.1 Viewing the Validation Report

The most straightforward way to view a validation report is to simply print the `Validate` object after calling `interrogate()`:

```
import pointblank as pb
import polars as pl

# Sample data
data = pl.DataFrame({
    "id": range(1, 11),
    "value": [120, 85, 47, 210, 30, 155, 175, 95, 205, 140],
    "category": ["A", "B", "C", "A", "D", "B", "A", "E", "A", "C"],
    "ratio": [0.5, 0.7, 0.3, 1.2, 0.8, 0.9, 0.4, 1.5, 0.6, 0.2],
})

# Create and interrogate a validation
validation = (
    pb.Validate(data=data, tbl_name="sales_data")
    .col_vals_gt(columns="value", value=50, brief=True)
    .col_vals_in_set(columns="category", set=["A", "B", "C"], brief=True)
    .col_exists(columns=["id", "value"], brief=True)
    .interrogate()
)

# Display the validation report
validation
```

Pointblank Validation

2025-11-03|00:38:18

POLARS | sales_data

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_gt() Expect that values in value should be > 50.	value	50	○→	✓	10	8 0.80	2 0.20	—	—	—	CSV
2	col_vals_in_set() Expect that values in category should be in the set of A, B, C.	category	A, B, C	○→	✓	10	8 0.80	2 0.20	—	—	—	CSV
3	col_exists() Expect that column id exists.	id	—	○→	✓	1	1 1.00	0 0.00	—	—	—	—
4	col_exists() Expect that column value exists.	value	—	○→	✓	1	1 1.00	0 0.00	—	—	—	—

In a notebook or interactive environment, simply typing the validation object name displays the report automatically. In a script or REPL, you might need to explicitly call `validation.get_tabular_report().show()` to display the table.

> **Note**
>
> You can display a validation report even before calling `interrogate()`. The report will show your validation plan with all the steps you've defined, but it won't contain any interrogation results. Additionally, validation steps that use column selection patterns (like validating multiple columns at once) won't be expanded into individual rows yet, as that expansion happens during interrogation.

4.2 Understanding Report Components

The validation report table consists of several key components that work together to provide a complete picture of your data quality:

4.2.0.1 Report Header

The report header (title and subtitle area) contains important metadata about the validation:

- **Title**: by default, shows "Pointblank Validation" but can be customized
- **Label**: your custom label for the validation (if provided via the `label=` parameter)
- **Table Information**: the table name and type (Polars, Pandas, DuckDB, etc.)
- **Thresholds**: the warning, error, and critical threshold values used

This header information provides essential context for interpreting the validation results, especially when sharing reports with stakeholders or reviewing historical validations.

4.2.0.2 Report Footer

The report footer contains a timestamp showing when the interrogation was performed. This timestamp helps track when data quality checks were executed, which is especially useful when archiving reports or monitoring data quality over time.

> ⓘ **Note**
>
> Throughout this documentation, the footer is hidden in example reports for brevity. This is controlled through a global option (see the section on controlling header and footer display later in this guide). In practice, including the footer provides valuable timestamp information for tracking when validations were executed.

4.2.1 Report Columns

The validation report table includes the following columns, each providing specific information about the validation steps:

4.2.1.1 Status Indicator (first column, unlabeled)

The first column is an unlabeled vertical colored bar that provides instant visual feedback about each step's status:

- **Green**: all test units passed the validation
- **Light green (semi-transparent)**: some test units failed but no thresholds were exceeded
- **Gray**: the 'warning' threshold was exceeded
- **Yellow**: the 'error' threshold was exceeded
- **Red**: the 'critical' threshold was exceeded

This visual indicator allows you to quickly scan the report and identify problem areas.

4.2.1.2 Step Number (second column, unlabeled)

The second column is unlabeled and contains the sequential step number, starting from 1. This number is used when referencing specific steps in other methods like `get_step_report(i=2)` or when extracting data from specific validation steps.

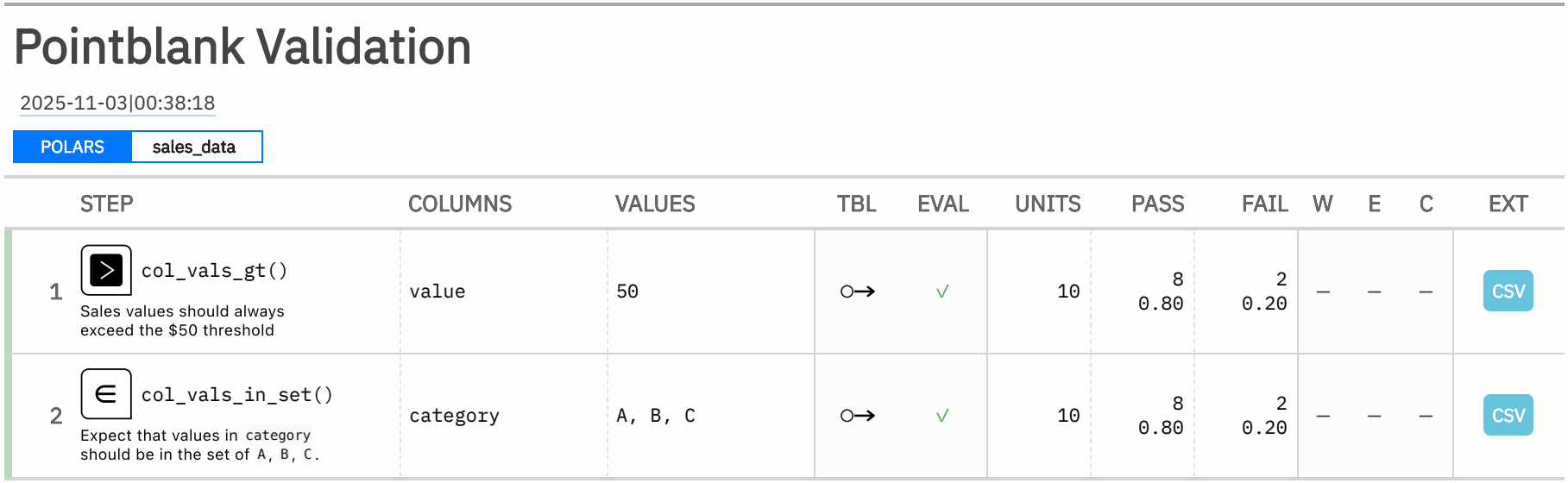
4.2.1.3 TYPE

The TYPE column displays the validation method name along with an icon that visually represents the type of validation being performed. The validation method indicates what aspect of data quality is being checked, such as:

- `col_vals_gt()`: column values greater than
- `col_vals_in_set()`: column values in a set
- `col_exists()`: column existence check
- `rows_distinct()`: row uniqueness check
- and many others...

When you provide a brief message (via `brief=True` for auto-generated briefs or `brief="custom text"` for custom messages), it appears within the TYPE column below the validation method name. These briefs provide human-readable explanations of what each validation step is checking, making the report more accessible to non-technical stakeholders.

```python
# Example showing brief messages in the TYPE column
validation_with_briefs = (
    pb.Validate(data=data, tbl_name="sales_data")
    .col_vals_gt(
        columns="value",
        value=50,
        brief="Sales values should always exceed the $50 threshold"
    )
    .col_vals_in_set(
        columns="category",
        set=["A", "B", "C"],
        brief=True  # Auto-generated brief
    )
    .interrogate()
)

validation_with_briefs
```



Pointblank Validation

2025-11-03|00:38:18

POLARS | sales_data

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_gt() Sales values should always exceed the $50 threshold	value	50	○→	✓	10	8 0.80	2 0.20	—	—	—	CSV
2	col_vals_in_set() Expect that values in category should be in the set of A, B, C.	category	A, B, C	○→	✓	10	8 0.80	2 0.20	—	—	—	CSV

In the above report, you'll see the custom brief message appear below the `col_vals_gt` method name in the first step, and an automatically generated brief below `col_vals_in_set` in the second step.

4.2.1.4 COLUMNS

The column(s) being validated in this step. For validation methods that don't target specific columns (like `row_count_match`), this will show an em dash (—).

4.2.1.5 VALUES

The comparison value(s) or criteria used in the validation. For example:

- for `col_vals_gt(value=100)`, this shows `100`
- for `col_vals_in_set(set=["A", "B", "C"])`, this shows `A | B | C`
- for existence checks, this shows an em dash (—)

4.2.1.6 TBL

Icons indicating whether any preprocessing or segmentation was applied:

- **Table icon**: standard validation on the original data
- **Transformation icon**: preprocessing function was applied via `pre=`
- **Segmentation icon**: data was segmented via `segments=`

These icons help you understand if you're validating transformed or segmented data.

4.2.1.7 EVAL

Indicates whether the validation step was evaluated:

- **Checkmark**: step was successfully evaluated
- **Error icon**: an evaluation error occurred (e.g., column not found)
- **Inactive icon**: step was marked as inactive

This column is crucial for identifying validation steps that couldn't be executed properly.

4.2.1.8 UNITS

The number of units tested in this validation step. A 'test unit' is the atomic unit being validated, which varies by validation type:

- for column value checks: each cell in the target column(s)

- for row checks: each row
- for table checks: typically 1 (the table itself)

This number is formatted with locale-appropriate thousand separators for readability. Also, since space is limited, values are often abbreviated so a figure like 43,534 will appear as `43.5K`.

4.2.1.9 PASS

The number and fraction of test units that passed the validation, displayed as:

```
n_passed
f_passed
```

For example, the cell with

```
8
0.80
```

means 8 test units passed out of the total, representing an 80% success rate (though `f_passed` is always expressed as a fractional value from `0` to `1`).

4.2.1.10 FAIL

The number and fraction of test units that failed the validation, displayed similarly to PASS:

```
n_failed
f_failed
```

For example, the cell with

```
2
0.20
```

means 2 test units failed, representing a 20% failure rate from a fractional value of `0.20`. Note that this fractional `f_failed` value is what's used to set failure thresholds for 'warning', 'error', and 'critical' states.

4.2.1.11 W, E, C (Warning, Error, Critical)

Three columns showing whether each threshold level was exceeded for the three different states.

- **Long dash**: threshold wasn't set for a state
- **Empty colored circle**: threshold was set but wasn't exceeded for a given state
- **Filled colored circle**: threshold was set and exceeded

In terms of colors, the 'warning' state is gray, the 'error' state is yellow, and the 'critical' state is red.

Having visual indicators makes it easy to identify which validation steps have crossed into warning, error, or critical territory.

4.2.1.12 EXT

Indicates whether failing row data was extracted for this step:

- **Em dash (—)**: no extract available
- **Download button**: click to download failing rows as CSV

When extracts are available, you can download them directly from the report for further analysis or to share with data stewards who need to fix the issues.

4.3 Understanding Validation Status

The validation report helps you quickly understand the overall status of your data:

- **All green status indicators**: all validations passed completely
- **Light green indicators**: minor failures below warning threshold
- **Gray, yellow, or red indicators**: threshold exceedances requiring attention
- **Error icons in EVAL column**: validation steps that couldn't be evaluated

By scanning the status indicators column, you can immediately identify which validation steps need attention and prioritize your data quality efforts accordingly.

4.4 Customizing the Report Title

You can customize the validation report's title using the `title=` parameter in `get_tabular_report()`. This is particularly useful when generating multiple reports or when you want to provide more context:

```
# Default title
validation.get_tabular_report()
```

Pointblank Validation

2025-11-03|00:38:18

POLARS | sales_data

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_gt() Expect that values in `value` should be > `50`.	value	50	○→	√	10	8 0.80	2 0.20	—	—	—	CSV
2	col_vals_in_set() Expect that values in `category` should be in the set of `A`, `B`, `C`.	category	A, B, C	○→	√	10	8 0.80	2 0.20	—	—	—	CSV
3	col_exists() Expect that column `id` exists.	id	—	○→	√	1	1 1.00	0 0.00	—	—	—	—
4	col_exists() Expect that column `value` exists.	value	—	○→	√	1	1 1.00	0 0.00	—	—	—	—

```
# Use the table name as the title
validation.get_tabular_report(title=":tbl_name:")
```

sales_data

2025-11-03|00:38:18

POLARS | sales_data

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_gt() Expect that values in `value` should be > `50`.	value	50	○→	✓	10	8 0.80	2 0.20	—	—	—	CSV
2	col_vals_in_set() Expect that values in `category` should be in the set of `A, B, C`.	category	A, B, C	○→	✓	10	8 0.80	2 0.20	—	—	—	CSV
3	col_exists() Expect that column `id` exists.	id	—	○→	✓	1	1 1.00	0 0.00	—	—	—	—
4	col_exists() Expect that column `value` exists.	value	—	○→	✓	1	1 1.00	0 0.00	—	—	—	—

```
# Provide a custom title (supports Markdown)
validation.get_tabular_report(title="**Sales Data** Quality Report")
```

Sales Data Quality Report

2025-11-03|00:38:18

POLARS | sales_data

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_gt() Expect that values in `value` should be > `50`.	value	50	○→	√	10	8 0.80	2 0.20	—	—	—	CSV
2	col_vals_in_set() Expect that values in `category` should be in the set of `A, B, C`.	category	A, B, C	○→	√	10	8 0.80	2 0.20	—	—	—	CSV
3	col_exists() Expect that column `id` exists.	id	—	○→	√	1	1 1.00	0 0.00	—	—	—	—
4	col_exists() Expect that column `value` exists.	value	—	○→	√	1	1 1.00	0 0.00	—	—	—	—

```
# No title
validation.get_tabular_report(title=":none:")
```

2025-11-03|00:38:18

POLARS | sales_data

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_gt() Expect that values in `value` should be > `50`.	value	50	○→	√	10	8 0.80	2 0.20	—	—	—	CSV
2	col_vals_in_set() Expect that values in `category` should be in the set of `A, B, C`.	category	A, B, C	○→	√	10	8 0.80	2 0.20	—	—	—	CSV
3	col_exists() Expect that column `id` exists.	id	—	○→	√	1	1 1.00	0 0.00	—	—	—	—
4	col_exists() Expect that column `value` exists.	value	—	○→	√	1	1 1.00	0 0.00	—	—	—	—

The title customization options are:

- `":default:"` (default): shows `"Pointblank Validation"`
- `":tbl_name:"`: uses the table name from `tbl_name=` parameter
- `":none:"`: hides the title completely
- Any string: custom title text (Markdown is supported)

4.5 Customizing with Great Tables

Since the validation report is a Great Tables object, you can leverage the full power of Great Tables to customize its appearance. This allows you to match your organization's branding, highlight specific information, or adjust the presentation for different audiences.

4.5.1 Guide to Internal Column Names

When working with Great Tables methods to customize the validation report, you'll need to use the *internal column names* rather than the display labels you see in the rendered table. This is because Great Tables operates on the underlying data table structure, where columns have technical names that differ from their user-facing labels.

For example, the column labeled `"STEP"` in the report is actually stored internally as `"i"`, and the `"TYPE"` column is internally named `"type_upd"`. Most Great Tables methods that target specific columns (like `tab_style()`, `cols_width()`, `cols_hide()`, etc.) require these internal names.

Here's the complete mapping from display labels to internal column names:

1. Status indicator (no label): `"status_color"`
2. Step number (no label): `"i"`
3. `TYPE`: `"type_upd"`
4. `COLUMNS`: `"columns_upd"`
5. `VALUES`: `"values_upd"`
6. `TBL`: `"tbl"`
7. `EVAL`: `"eval"`
8. `UNITS`: `"test_units"`
9. `PASS`: `"pass"`
10. `FAIL`: `"fail"`
11. `W`: `"w_upd"`
12. `E`: `"e_upd"`
13. `C`: `"c_upd"`
14. `EXT`: `"extract_upd"`

Always use these internal names when calling Great Tables methods. Using the display labels (like `"STEP"` or `"TYPE"`) will result in errors since these labels only exist in the rendered output, not in the underlying data structure.

In the examples that follow, you'll see how to use these internal column names to customize various aspects of the validation report.

4.5.2 Adding Custom Styling

You can apply custom styles to the report table:

```python
from great_tables import style, loc

# Get the report as a Great Tables object
report = validation.get_tabular_report()

# Add custom styling using internal column names
report = (
    report
    .tab_style(
        style=style.fill(color="#F0F8FF"),
        locations=loc.body(columns="i")  # Internal name for step number
    )
    .tab_style(
        style=style.text(weight="bold"),
        locations=loc.body(columns="type_upd")  # Internal name for TYPE
    )
)

report
```

Pointblank Validation

2025-11-03|00:38:18

POLARS | sales_data

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_gt() Expect that values in `value` should be > `50`.	value	50	○→	√	10	8 0.80	2 0.20	—	—	—	CSV
2	col_vals_in_set() Expect that values in `category` should be in the set of `A, B, C`.	category	A, B, C	○→	√	10	8 0.80	2 0.20	—	—	—	CSV
3	col_exists() Expect that column `id` exists.	id	—	○→	√	1	1 1.00	0 0.00	—	—	—	—
4	col_exists() Expect that column `value` exists.	value	—	○→	√	1	1 1.00	0 0.00	—	—	—	—

4.5.3 Modifying Column Widths

Adjust column widths to optimize the layout:

```
report = (
    validation
    .get_tabular_report()
    .cols_width(
        cases={
            "status_color": "20px", # Status indicator column
            "i": "40px",            # Step number column
            "type_upd": "170px",    # TYPE column
            "columns_upd": "100px", # COLUMNS column
        }
    )
)

report
```

Pointblank Validation

2025-11-03|00:38:18

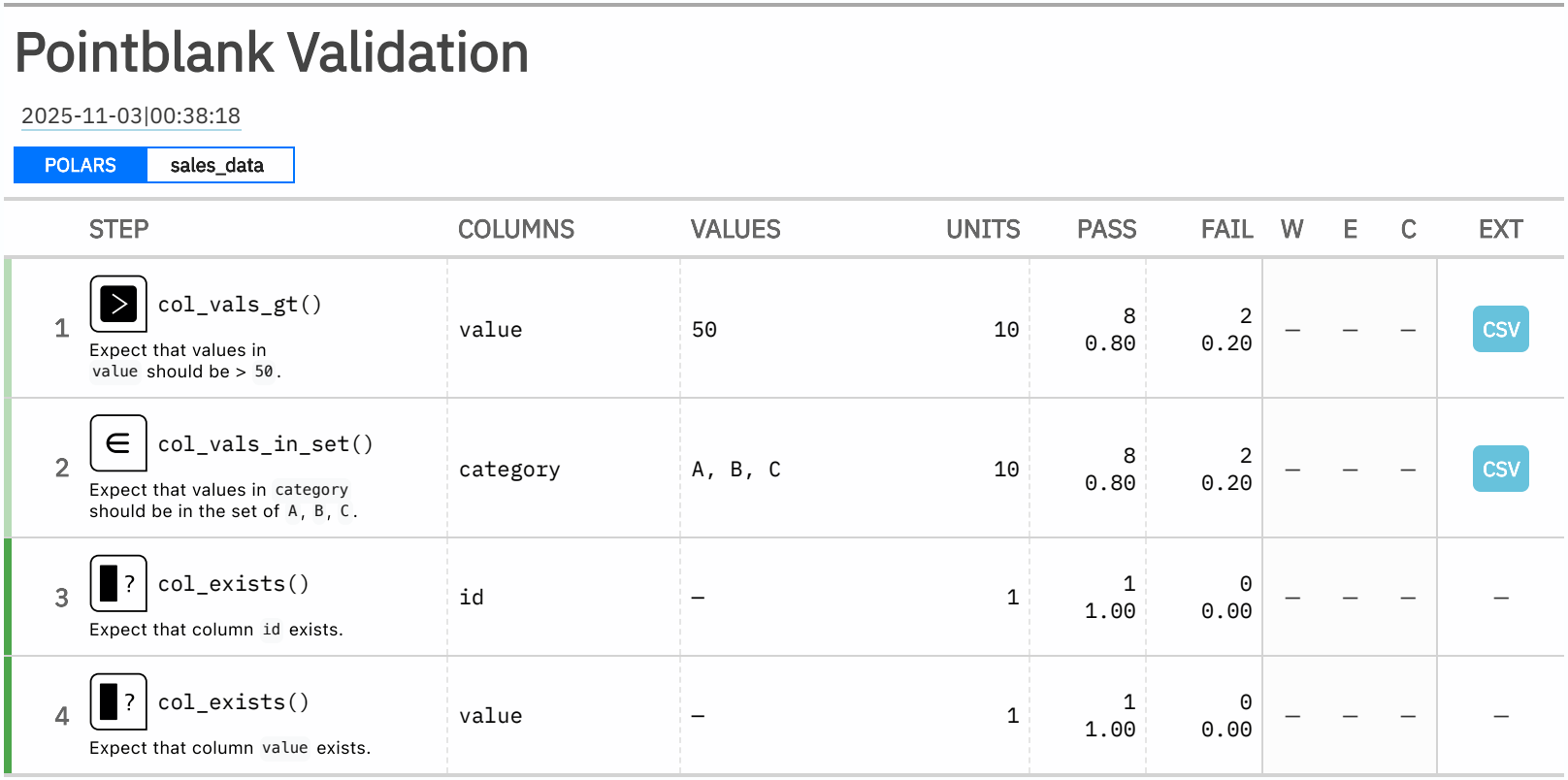
POLARS | sales_data

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_gt() Expect that values in `value` should be > `50`.	value	50	○→	✓	10	8 0.80	2 0.20	—	—	—	CSV
2	col_vals_in_set() Expect that values in `category` should be in the set of `A, B, C`.	category	A, B, C	○→	✓	10	8 0.80	2 0.20	—	—	—	CSV
3	col_exists() Expect that column `id` exists.	id	—	○→	✓	1	1 1.00	0 0.00	—	—	—	—
4	col_exists() Expect that column `value` exists.	value	—	○→	✓	1	1 1.00	0 0.00	—	—	—	—

4.5.4 Hiding Columns

Hide specific columns that aren't relevant for your audience:

```
# Hide the TBL and EVAL columns for a cleaner presentation (using internal names)
report = (
    validation
    .get_tabular_report()
    .cols_hide(columns=["tbl", "eval"])  # Use internal column names
)

report
```



Pointblank Validation

2025-11-03|00:38:18

POLARS | sales_data

	STEP	COLUMNS	VALUES	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_gt() Expect that values in value should be > 50.	value	50	10	8 0.80	2 0.20	—	—	—	CSV
2	col_vals_in_set() Expect that values in category should be in the set of A, B, C.	category	A, B, C	10	8 0.80	2 0.20	—	—	—	CSV
3	col_exists() Expect that column id exists.	id	—	1	1 1.00	0 0.00	—	—	—	—
4	col_exists() Expect that column value exists.	value	—	1	1 1.00	0 0.00	—	—	—	—

4.5.5 Adding a Source Note

Add information about data source or validation context:

```
report = (
    validation
    .get_tabular_report()
    .tab_source_note(
        source_note="Data validated on 2025-10-10 | Production database snapshot"
    )
)

report
```

Pointblank Validation

2025-11-03|00:38:18

POLARS | sales_data

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_gt() Expect that values in `value` should be > `50`.	value	50	○→	√	10	8 0.80	2 0.20	—	—	—	CSV
2	col_vals_in_set() Expect that values in `category` should be in the set of `A`, `B`, `C`.	category	A, B, C	○→	√	10	8 0.80	2 0.20	—	—	—	CSV
3	col_exists() Expect that column `id` exists.	id	—	○→	√	1	1 1.00	0 0.00	—	—	—	—
4	col_exists() Expect that column `value` exists.	value	—	○→	√	1	1 1.00	0 0.00	—	—	—	—

Data validated on 2025-10-10 | Production database snapshot

4.6 Exporting the Report

Great Tables provides multiple export options for sharing validation reports:

```
# Save as a standalone HTML file
validation.get_tabular_report().write_raw_html("validation_report.html")

# Save as a PNG image
validation.get_tabular_report().save("validation_report.png")

# Open in browser
validation.get_tabular_report().show("browser")
```

4.7 Controlling Header and Footer Display

You can control whether the header and footer appear in the validation report:

```
# Hide the footer
validation.get_tabular_report(incl_footer=False)
```

Pointblank Validation

2025-11-03|00:38:18

POLARS | sales_data

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_gt() Expect that values in `value` should be > `50`.	value	50	○→	√	10	8 0.80	2 0.20	—	—	—	CSV
2	col_vals_in_set() Expect that values in `category` should be in the set of `A, B, C`.	category	A, B, C	○→	√	10	8 0.80	2 0.20	—	—	—	CSV
3	col_exists() Expect that column `id` exists.	id	—	○→	√	1	1 1.00	0 0.00	—	—	—	—
4	col_exists() Expect that column `value` exists.	value	—	○→	√	1	1 1.00	0 0.00	—	—	—	—

```
# Hide the header
validation.get_tabular_report(incl_header=False)
```

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_gt() Expect that values in `value` should be > `50`.	value	50	○→	✓	10	8 0.80	2 0.20	—	—	—	CSV
2	col_vals_in_set() Expect that values in `category` should be in the set of `A, B, C`.	category	A, B, C	○→	✓	10	8 0.80	2 0.20	—	—	—	CSV
3	col_exists() Expect that column `id` exists.	id	—	○→	✓	1	1 1.00	0 0.00	—	—	—	—
4	col_exists() Expect that column `value` exists.	value	—	○→	✓	1	1 1.00	0 0.00	—	—	—	—

```
# Hide both
validation.get_tabular_report(incl_header=False, incl_footer=False)
```

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_gt() Expect that values in `value` should be > `50`.	value	50	○→	✓	10	8 0.80	2 0.20	—	—	—	CSV
2	col_vals_in_set() Expect that values in `category` should be in the set of `A, B, C`.	category	A, B, C	○→	✓	10	8 0.80	2 0.20	—	—	—	CSV
3	col_exists() Expect that column `id` exists.	id	—	○→	✓	1	1 1.00	0 0.00	—	—	—	—
4	col_exists() Expect that column `value` exists.	value	—	○→	✓	1	1 1.00	0 0.00	—	—	—	—

You can also set these preferences globally using `pb.config()`:

```
# Set global preferences
pb.config(report_incl_header=True, report_incl_footer=False)
```

4.8 Best Practices for Validation Reports

Here are some guidelines for creating effective validation reports:

4.8.0.1 1. Use Descriptive Table Names and Labels

Provide meaningful names and labels to make reports self-documenting:

```
validation = pb.Validate(
    data=sales_df,
    tbl_name="Q3_2025_sales",
    label="Quarterly sales data validation for financial reporting"
)
```

4.8.0.2 2. Add Brief Messages for Stakeholder Reports

When sharing reports with non-technical stakeholders, always include briefs:

```
.col_vals_between(
    columns="price",
    left=0, right=10000,
    brief="Product prices must be between $0 and $10,000"
)
```

4.8.0.3 3. Set Appropriate Thresholds

Configure thresholds that align with your data quality requirements:

```
validation = pb.Validate(
    data=data,
    tbl_name="customer_data",
    thresholds=pb.Thresholds(
        warning=0.01,  # 1% failure triggers warning
        error=0.05,    # 5% failure triggers error
        critical=0.10  # 10% failure triggers critical
    )
)
```

4.8.0.4 4. Customize for Your Audience

Tailor the report presentation to your audience:

- **Technical teams**: include all columns, show preprocessing indicators
- **Management**: hide technical columns, emphasize status indicators
- **Data stewards**: include extract download buttons, detailed briefs

4.8.0.5 5. Combine with Other Reporting Tools

Use validation reports alongside other Pointblank features:

- **Step reports**: drill down into specific failing steps with `get_step_report()`
- **Extracts**: use `get_data_extracts()` to get all failing data for analysis
- **Sundered data**: use `get_sundered_data()` to split data into passing/failing sets

4.8.0.6 6. Archive Reports for Trend Analysis

Save validation reports over time to track data quality trends:

```
from datetime import datetime

# Save with timestamp
timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
validation.get_tabular_report().write_raw_html(f"validation_report_{timestamp}.html")
```

4.9 Conclusion

The validation report is your primary interface for understanding data quality after running a validation. By providing a comprehensive overview of all validation steps, visual status indicators, and detailed statistics, it enables you to:

- quickly assess overall data quality across multiple dimensions
- identify specific validation steps that need attention
- communicate data quality status to technical and non-technical stakeholders
- track threshold exceedances and their severity levels
- access failing data through extract downloads

Combined with customization options from Great Tables, you can create reports that perfectly match your organization's needs and workflows. Whether you're validating data in an interactive notebook, generating automated quality reports, or presenting findings to stakeholders, the validation report provides the clarity and detail you need to maintain high data quality standards.

While validation reports provide a comprehensive overview of all validation steps, sometimes you need to focus on a specific validation step in greater detail. This is where *step reports* come in. A step report is a detailed examination of a single validation step, providing in-depth information about the test units that were validated and their pass/fail status.

Step reports are especially useful when debugging validation failures, investigating problematic data, or communicating detailed findings to colleagues who are responsible for specific data quality issues.

4.10 Creating a Step Report

To create a step report, you first need to run a validation and then use the `Validate.get_step_report()` method, specifying which validation step you want to examine:

```
import pointblank as pb
import polars as pl

# Sample data as a Polars DataFrame
data = pl.DataFrame({
    "id": range(1, 11),
    "value": [10, 20, 3, 35, 50, 2, 70, 8, 20, 4],
    "category": ["A", "B", "C", "A", "D", "F", "A", "E", "H", "G"],
    "ratio": [0.5, 0.7, 0.3, 1.2, 0.8, 0.9, 0.4, 1.5, 0.6, 0.2],
    "status": ["active", "active", "inactive", "active", "inactive",
               "active", "inactive", "active", "active", "inactive"]
})

# Create a validation
validation = (
    pb.Validate(data=data, tbl_name="example_data")
    .col_vals_gt(
        columns="value",
        value=10
    )
    .col_vals_in_set(
        columns="category",
        set=["A", "B", "C"]
    )
    .interrogate()
)

# Get step report for the second validation step (i=2)
step_report = validation.get_step_report(i=2)

step_report
```

Report for Validation Step 2

ASSERTION `category ∈ {A, B, C}`

5 / 10 TEST UNIT FAILURES IN COLUMN 3
EXTRACT OF ALL 5 ROWS (WITH TEST UNIT FAILURES IN RED):

	id *Int64*	value *Int64*	category *String*	ratio *Float64*	status *String*
5	5	50	D	0.8	inactive
6	6	2	F	0.9	active
8	8	8	E	1.5	active
9	9	20	H	0.6	active
10	10	4	G	0.2	inactive

In this example, we first create and interrogate a validation object with two steps. We then generate a step report for the second validation step (`i=2`), which checks if the values in the `category` column are in the set `["A", "B", "C"]`.

Note that step numbers in Pointblank start at `1`, matching what you see in the validation report's `STEP` column (i.e., not 0-based indexing). So the first step is referred to with `i=1`, the second step with `i=2`, and so on.

4.11 Understanding Step Report Components

A step report consists of several key components that provide detailed information about the validation step:

1. Header: displays the validation step number, type of validation, and a brief description
2. Table Body: presents either the failing rows, a sample of completely passing data, or an expected/actual comparison (for a `Validate.col_schema_match()` step)

The step report table highlights passing and failing rows, making it easy to identify problematic data points. This is especially useful for diagnosing issues when dealing with large datasets.

4.12 Different Types of Step Reports

It's important to note that step reports vary in appearance and structure depending on the type of validation method used:

- Value-based validations (like `Validate.col_vals_gt()`, `Validate.col_vals_in_set()`): show individual rows that failed validation
- Uniqueness checks (`Validate.rows_distinct()`): group together the duplicate records in order of appearance
- Schema validations (`Validate.col_schema_match()`): display column-level information about expected vs. actual data types

Additionally, step reports for value-based validations and uniqueness checks operate in two distinct modes:

1. When errors are present: The report shows only the failing rows and, for value-based validations, clearly highlights the column under study
2. When no errors exist: The report header clearly indicates success, and a sample of the data is shown (along with the studied column highlighted, for value-based validations)

This variation in reporting style allows step reports to effectively communicate the specific type of validation being performed and display relevant information in the most appropriate format. When you're working with different validation types, expect to see different step report layouts optimized for each context.

4.12.1 Value-Based Validation Step Reports

Value-based step reports focus on showing individual rows where values in the target column failed the validation check. These reports highlight the specific column being validated and clearly display which values violated the condition.

```
# Create sample data with some validation failures
data = pl.DataFrame({
    "id": range(1, 8),
    "value": [120, 85, 47, 210, 30, 10, 5],
    "category": ["A", "B", "C", "A", "D", "B", "E"]
})

# Create a validation with a value-based check
validation_values = (
    pb.Validate(data=data, tbl_name="sales_data")
    .col_vals_gt(
        columns="value",
        value=50,
        brief="Sales values should exceed $50"
    )
    .interrogate()
)

# Display the step report for the value-based validation
validation_values.get_step_report(i=1)
```

Report for Validation Step 1

ASSERTION `value > 50`

4 / **7** TEST UNIT FAILURES IN COLUMN **2**
EXTRACT OF ALL **4** ROWS (WITH TEST UNIT FAILURES IN RED):

	id *Int64*	value *Int64*	category *String*
3	3	47	C
5	5	30	D
6	6	10	B
7	7	5	E

This report clearly identifies which rows contain values that don't meet our threshold, making it easy to investigate these specific data points.

4.12.2 Uniqueness Validation Step Reports

Uniqueness checks produce a different type of step report that groups duplicate records together. This format makes it easy to identify patterns in duplicate data.

```
# Create sample data with some duplicate rows based on the combination of columns
data = pl.DataFrame({
    "customer_id": [101, 102, 103, 101, 104, 105, 102],
    "order_date": ["2023-01-15", "2023-01-16", "2023-01-16",
                   "2023-01-15", "2023-01-17", "2023-01-18", "2023-01-19"],
    "product": ["Laptop", "Phone", "Tablet", "Laptop",
                "Monitor", "Keyboard", "Headphones"]
})

# Create a validation checking for unique customer-product combinations
validation_duplicates = (
    pb.Validate(data=data, tbl_name="order_data")
    .rows_distinct(
        columns_subset=["customer_id", "product"],
        brief="Customer should not order the same product twice"
    )
    .interrogate()
)

# Display the step report for the uniqueness validation
validation_duplicates.get_step_report(i=1)
```

Report for Validation Step 1

Rows are distinct across a subset of columns

2 / **7** TEST UNIT FAILURES
EXTRACT OF ALL **2** ROWS:

	customer_id	product
	Int64	*String*
1	101	Laptop
4	101	Laptop

The report organizes duplicate records together, making it easy to see which combinations are repeated and how many times they appear.

4.12.3 Schema Validation Step Reports

Schema validation step reports have a completely different structure, comparing expected versus actual column data types and presence.

```
schema = pb.Schema(
    columns=[
        ("date_time", "timestamp"),
        ("dates", "date"),
        ("a", "int64"),
        ("b",),
        ("c",),
        ("d", "float64"),
        ("e", ["bool", "boolean"]),
        ("f", "str"),
    ]
)

validation_schema = (
    pb.Validate(
        data=pb.load_dataset(dataset="small_table", tbl_type="duckdb"),
        tbl_name="small_table",
        label="Step report for a schema check"
    )
    .col_schema_match(schema=schema)
    .interrogate()
)

# Display the step report for the schema validation
validation_schema.get_step_report(i=1)
```

Report for Validation Step 1 ✗

COLUMN SCHEMA MATCH [COMPLETE] [IN ORDER] [COLUMN ≠ column] [DTYPE ≠ dtype] [float ≠ float64]

	TARGET		EXPECTED			
	COLUMN	DATA TYPE		COLUMN		DATA TYPE	
1	date_time	timestamp(6)	1	date_time	✓	timestamp	✗
2	date	date	2	dates	✗	date	–
3	a	int64	3	a	✓	int64	✓
4	b	string	4	b	✓	–	
5	c	int64	5	c	✓	–	
6	d	float64	6	d	✓	float64	✓
7	e	boolean	7	e	✓	bool | boolean	✓
8	f	string	8	f	✓	str	✗

Supplied Column Schema:

```
[('date_time', 'timestamp'), ('dates', 'date'), ('a', 'int64'), ('b',), ('c',), ('d', 'float64'), ('e', ['bool', 'boolean']), ('f', 'str')]
```

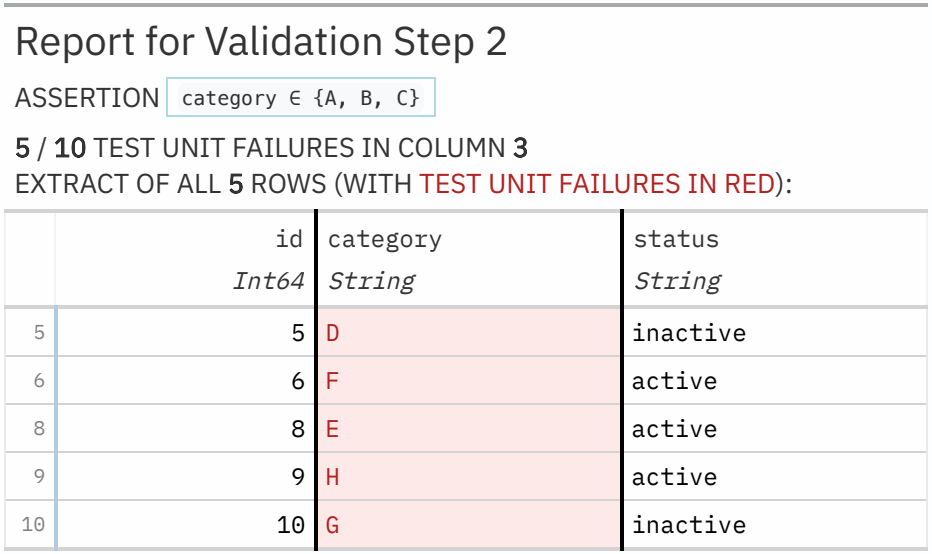
This report style focuses on comparing the expected schema against the actual table structure, highlighting mismatches in data types or missing/extra columns. The table format makes it easy to see exactly where the schema expectations differ from reality.

4.13 Customizing Step Reports

Step reports can be customized with several parameters to better focus your analysis and tailor the output to your specific needs. The `Validate.get_step_report()` method offers multiple customization options to help you create more effective reports.

When a dataset has many columns, you might want to focus on just those relevant to your analysis. You can create a step report containing only a subset of the columns in the target table:

```
validation.get_step_report(
    i=2,

    # Only show these columns ---
    columns_subset=["id", "category", "status"]
)
```



Report for Validation Step 2

ASSERTION `category ∈ {A, B, C}`

5 / **10** TEST UNIT FAILURES IN COLUMN **3**
EXTRACT OF ALL **5** ROWS (WITH TEST UNIT FAILURES IN RED):

	id *Int64*	category *String*	status *String*
5	5	D	inactive
6	6	F	active
8	8	E	active
9	9	H	active
10	10	G	inactive

This approach makes step reports much easier to interpret by highlighting just the essential columns that help understand the validation failures.

For large datasets with many failing rows, you might want to use `limit=` to set a cap on the number of rows shown in the report:

```
validation.get_step_report(
    i=2,

    # Only show up to 2 failing rows ---
    limit=2
)
```

Report for Validation Step 2

ASSERTION `category ∈ {A, B, C}`

5 / **10** TEST UNIT FAILURES IN COLUMN **3**
EXTRACT OF FIRST **2** ROWS (WITH TEST UNIT FAILURES IN RED):

	id *Int64*	value *Int64*	category *String*	ratio *Float64*	status *String*
5	5	50	D	0.8	inactive
6	6	2	F	0.9	active

The report header can also be extensively customized to provide more specific context. You can replace the default header with plain text or Markdown formatting:

```
validation.get_step_report(
    i=2,
    header="Category Values Validation: *Critical Analysis*"
)
```

Category Values Validation: *Critical Analysis*

	id *Int64*	value *Int64*	category *String*	ratio *Float64*	status *String*
5	5	50	D	0.8	inactive
6	6	2	F	0.9	active
8	8	8	E	1.5	active
9	9	20	H	0.6	active
10	10	4	G	0.2	inactive

For more advanced header customization, you can use the templating system with the `{title}` and `{details}` elements to retain parts of the default header while adding your own content. The `{title}` template is the default title whereas `{details}` provides information on the assertion, number of failures, etc. Let's move away from the default template of `{title}{details}` and provide a custom title to go with the details text:

```
validation.get_step_report(
    i=2,
    header="Custom Category Validation Report {details}"
)
```

Custom Category Validation Report

ASSERTION `category ∈ {A, B, C}`

5 / **10** TEST UNIT FAILURES IN COLUMN **3**
EXTRACT OF ALL **5** ROWS (WITH TEST UNIT FAILURES IN RED):

	id *Int64*	value *Int64*	category *String*	ratio *Float64*	status *String*
5	5	50	D	0.8	inactive
6	6	2	F	0.9	active
8	8	8	E	1.5	active
9	9	20	H	0.6	active
10	10	4	G	0.2	inactive

We can keep `{title}` and `{details}` and add some more context in between the two:

```
validation.get_step_report(
    i=2,
    header=(
        "{title}<br>"
        "<span style='font-size: 0.75em;'>"
        "This validation is critical for our data quality standards."
        "</span><br>"
        "{details}"
    )
)
```

Report for Validation Step 2

This validation is critical for our data quality standards.

ASSERTION `category ∈ {A, B, C}`

5 / **10** TEST UNIT FAILURES IN COLUMN **3**
EXTRACT OF ALL **5** ROWS (WITH TEST UNIT FAILURES IN RED):

	id *Int64*	value *Int64*	category *String*	ratio *Float64*	status *String*
5	5	50	D	0.8	inactive
6	6	2	F	0.9	active
8	8	8	E	1.5	active
9	9	20	H	0.6	active
10	10	4	G	0.2	inactive

You could always use more HTML and CSS to do *a lot* of customization:

```
validation.get_step_report(
    i=2,
    header=(
        "VALIDATION SUMMARY\n\n{details}\n\n"
        "<hr style='color: lightblue;'>"
        "<div style='font-size: smaller; padding-bottom: 5px; text-transform: uppercase'>"
        "{title}"
        "</div>"
    )
)
```

VALIDATION SUMMARY

ASSERTION `category ∈ {A, B, C}`

5 / **10** TEST UNIT FAILURES IN COLUMN **3**
EXTRACT OF ALL **5** ROWS (WITH TEST UNIT FAILURES IN RED):

REPORT FOR VALIDATION STEP 2

	id *Int64*	value *Int64*	category *String*	ratio *Float64*	status *String*
5	5	50	D	0.8	inactive
6	6	2	F	0.9	active
8	8	8	E	1.5	active
9	9	20	H	0.6	active
10	10	4	G	0.2	inactive

If you prefer no header at all, simply set `header=None`:

```
validation.get_step_report(
    i=2,
    header=None
)
```

	id *Int64*	value *Int64*	category *String*	ratio *Float64*	status *String*
5	5	50	D	0.8	inactive
6	6	2	F	0.9	active
8	8	8	E	1.5	active
9	9	20	H	0.6	active
10	10	4	G	0.2	inactive

These customization options can be combined to create highly focused reports tailored to specific needs:

```
validation.get_step_report(
    i=2,
    columns_subset=["id", "category"],
    header="*Category Validation:* Top Issues",
    limit=2
)
```

Category Validation: Top Issues

	id	category
	Int64	*String*
5	5	D
6	6	F

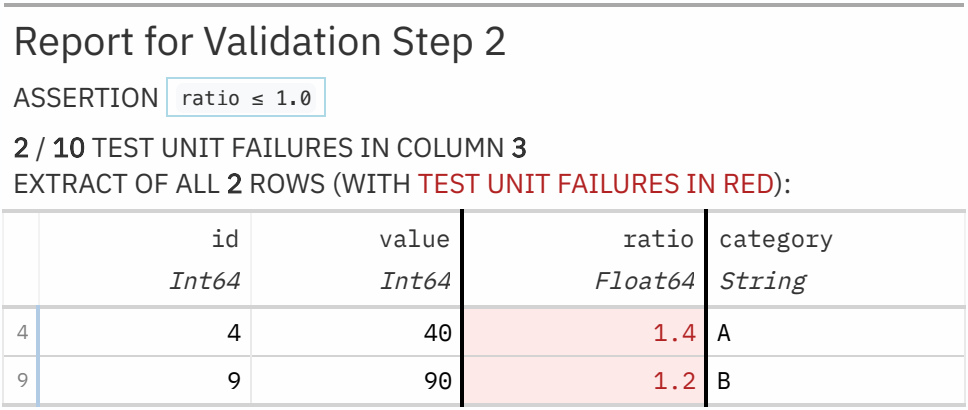
Through these customization options, you can craft step reports that effectively communicate the most important information to different audiences. Technical teams might benefit from seeing all columns but with a limited number of examples. Business stakeholders might prefer a focused view with only the most relevant columns. For documentation purposes, custom headers provide important context about what's being validated.

Remember that customizing your step reports is about more than aesthetics: it's about making complex validation information more accessible and actionable for all stakeholders involved in data quality.

4.14 Using Step Reports for Data Investigation

Step reports can be powerful tools for investigating data quality issues. Let's look at a more complex example:

```
# Create a more complex dataset with multiple issues
complex_data = pl.DataFrame({
    "id": range(1, 11),
    "value": [10, 20, 3, 40, 50, 2, 70, 80, 90, 7],
    "ratio": [0.1, 0.2, 0.3, 1.4, 0.5, 0.6, 0.7, 0.8, 1.2, 0.9],
    "category": ["A", "B", "C", "A", "D", "B", "A", "C", "B", "E"]
})

# Create a validation with multiple steps
validation_complex = (
    pb.Validate(data=complex_data, tbl_name="complex_data")
    .col_vals_gt(columns="value", value=10)
    .col_vals_le(columns="ratio", value=1.0)
    .col_vals_in_set(columns="category", set=["A", "B", "C"])
    .interrogate()
)

# Get step report for the ratio validation (step 2)
ratio_report = validation_complex.get_step_report(i=2)

ratio_report
```



Report for Validation Step 2

ASSERTION `ratio ≤ 1.0`

2 / 10 TEST UNIT FAILURES IN COLUMN 3
EXTRACT OF ALL 2 ROWS (WITH TEST UNIT FAILURES IN RED):

	id	value	ratio	category
	Int64	*Int64*	*Float64*	*String*
4	4	40	1.4	A
9	9	90	1.2	B

In this example, we're investigating issues with the `ratio` column by generating a step report specifically for that validation step. The step report shows exactly which rows have values that exceed our maximum threshold of `1.0`.

4.15 Combining Step Reports with Extracts

For more advanced analysis, you can extract the actual data from a step report into a DataFrame:

```
# Extract the data from the step report
failing_ratios = validation_complex.get_data_extracts(i=2)

failing_ratios
```

```
{2: shape: (2, 5)
┌──────────┬─────┬───────┬───────┬──────────┐
│ _row_num_ ┆ id  ┆ value ┆ ratio ┆ category │
│ ---      ┆ --- ┆ ---   ┆ ---   ┆ ---      │
│ u32      ┆ i64 ┆ i64   ┆ f64   ┆ str      │
╞══════════╪═════╪═══════╪═══════╪══════════╡
│ 4        ┆ 4   ┆ 40    ┆ 1.4   ┆ A        │
│ 9        ┆ 9   ┆ 90    ┆ 1.2   ┆ B        │
└──────────┴─────┴───────┴───────┴──────────┘}
```

This extracts the failing rows from the validation step, which you can then further analyze or fix as needed. Note that the parameter `i=2` corresponds directly to the step number shown in the validation report; it's the same numbering system used for `Validate.get_step_report()`.

These extracts are particularly valuable for analysts who need to:

- perform additional calculations on problematic data
- feed failing records into correction pipelines
- create visualizations of data patterns that led to validation failures
- export problem records to share with data owners

It's worth noting that the validation report itself includes export buttons on the far right of each row that allow you to download CSV files of the failing data directly. This serves as a convenient delivery mechanism for sharing extracts with colleagues who may not be working in Python, making the validation report not just a visual tool but also a practical means of distributing problematic data for further investigation.

4.16 Step Reports with Segmented Data

When working with segmented validation, step reports become even more valuable as they allow you to investigate issues within specific segments:

```
# Create data with different regions
segmented_data = pl.DataFrame({
    "id": range(1, 10),
    "value": [10, 20, 3, 40, 50, 2, 6, 8, 60],
    "region": ["North", "North", "South", "South", "East", "East", "West", "West", "West"]
})

# Create a validation with segments
segmented_validation = (
    pb.Validate(data=segmented_data, tbl_name="regional_data")
    .col_vals_gt(
        columns="value",
        value=10,
        segments="region"  # Segment by region
    )
    .interrogate()
)

# Get step report for a specific segment (the 'West' region)
# For segmented validations, each segment gets its own step number
north_report = segmented_validation.get_step_report(i=4)

north_report
```

Report for Validation Step 4

ASSERTION `value > 10`

2 / 3 TEST UNIT FAILURES IN COLUMN 2
EXTRACT OF ALL 2 ROWS (WITH TEST UNIT FAILURES IN RED):

	id *Int64*	value *Int64*	region *String*
1	7	6	West
2	8	8	West

For segmented validations, each segment is treated as a separate validation step with its own step number. This allows you to investigate issues specific to each data segment using the appropriate step number from the validation report.

4.17 Best Practices for Using Step Reports

Here are some guidelines for effectively using step reports in your data validation workflow:

1. Generate step reports selectively: create reports only for steps that require detailed investigation rather than for all steps
2. Use the `limit=` parameter for large datasets: when working with large datasets, focus only on a subset of failing rows to avoid information overload
3. Share specific step reports with stakeholders: when collaborating with domain experts, share relevant step reports to help them understand and address specific data quality issues (and customize the header to improve clarity)
4. Combine with extracts for deeper analysis: use the `Validate.get_data_extracts()` method to extract the failing rows for further analysis or correction
5. Document findings from step reports: when you discover patterns or insights from step reports, document them to inform future data quality improvements

Remember that step reports are most valuable when used strategically as part of a broader data quality framework. By following these best practices, you can use step reports not just for troubleshooting, but to develop a deeper understanding of your data's characteristics

and quality patterns over time. This approach transforms step reports from simple debugging tools into strategic assets for continuous data quality improvement.

4.18 Conclusion

Step reports provide a focused lens into specific validation steps, allowing you to investigate data quality issues in detail. By generating targeted reports for specific validation steps, you can:

- pinpoint exactly which data points are causing validation failures
- communicate specific issues to relevant stakeholders
- gather insights that might be missed in the aggregate validation report
- track improvements in specific aspects of data quality over time

Whether you're debugging validation failures, investigating edge cases, or communicating specific data quality issues to colleagues, step reports can give you the detailed information you need to understand and resolve data quality problems effectively.

When validating data, identifying exactly which rows failed is critical for diagnosing and resolving data quality issues. This is where *data extracts* come in. Data extracts consist of target table rows containing at least one cell that failed validation. While the validation report provides an overview of pass/fail statistics, data extracts give you the actual problematic records for deeper investigation.

This article will cover:

- which validation methods collect data extracts
- multiple ways to access and work with data extracts
- practical examples of using extracts for data quality improvement
- advanced techniques for analyzing extract patterns

4.19 The Validation Methods that Work with Data Extracts

The following validation methods operate on column values and will have rows extracted when there are failing test units in those rows:

- `Validate.col_vals_gt()`
- `Validate.col_vals_lt()`

- `Validate.col_vals_ge()`
- `Validate.col_vals_le()`
- `Validate.col_vals_eq()`
- `Validate.col_vals_ne()`
- `Validate.col_vals_between()`
- `Validate.col_vals_outside()`
- `Validate.col_vals_in_set()`
- `Validate.col_vals_not_in_set()`
- `Validate.col_vals_null()`
- `Validate.col_vals_not_null()`
- `Validate.col_vals_regex()`
- `Validate.col_vals_expr()`
- `Validate.conjointly()`

These row-based validation methods will also have rows extracted should there be failing rows:

- `Validate.rows_distinct()`
- `Validate.rows_complete()`

Note that some validation methods like `Validate.col_exists()` and `Validate.col_schema_match()` don't generate data extracts because they validate structural aspects of the table rather than checking column values.

4.20 Accessing Data Extracts

There are three primary ways to access data extracts in Pointblank:

1. the **CSV** buttons in validation reports
2. through the `Validate.get_data_extracts()` method
3. inspecting a subset of failed rows in step reports

Let's explore each approach using examples.

4.20.1 CSV Data from Validation Reports

Data extracts are embedded within validation report tables. Let's look at an example, using the `small_table` dataset, where data extracts are collected in a single validation step due to failing test units:

```
import pointblank as pb

validation = (
    pb.Validate(data=pb.load_dataset(dataset="small_table", tbl_type="polars"))
    .col_vals_lt( columns="d", value=3000)
    .interrogate()
)

validation
```

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_lt()	d	3000	○→	✓	13	9 0.69	4 0.31	—	—	—	CSV

The single validation step checks whether values in `d` are less than `3000`. Within that column, values range from `108.34` to `9999.99` so it makes sense that we can see 4 failing test units in the `FAIL` column.

If you look at the far right of the validation report you'll find there's a `CSV` button. Pressing it initiates the download of a CSV file, and that file contains the data extract for this validation step. The `CSV` button only appears when:

1. there is a non-zero number of failing test units
2. the validation step is based on the use of a column-value or a row-based validation method (the methods outlined in the section entitled *The Validation Methods that Work with Data Extracts*)

Access to CSV data for the test unit errors is useful when the validation report is shared with other data quality stakeholders, since it is easily accessible and doesn't require further use of Pointblank. The stakeholder can simply open the downloaded CSV in their preferred spreadsheet software, import it into a different analysis environment like R or Julia, or process it with any tool that supports CSV files. This cross-platform compatibility makes the CSV export particularly valuable in mixed-language data teams where different members might be working with different tools.

4.20.2 `get_data_extracts()`

For programmatic access to data extracts, Pointblank provides the `Validate.get_data_extracts()` method. This allows you to work with extract data directly in your Python workflow:

```
# Get data extracts from step 1
extract_1 = validation.get_data_extracts(i=1, frame=True)

extract_1
```

shape: (4, 9)

_row_num_	date_time	date	a	b	c	d	e	f
u32	datetime[μs]	date	i64	str	i64	f64	bool	str
1	2016-01-04 11:00:00	2016-01-04	2	"1-bcd-345"	3	3423.29	true	"high"
2	2016-01-04 00:32:00	2016-01-04	3	"5-egh-163"	8	9999.99	true	"low"
4	2016-01-06 17:23:00	2016-01-06	2	"5-jdo-903"	null	3892.4	false	"mid"
6	2016-01-11 06:15:00	2016-01-11	4	"2-dhe-923"	4	3291.03	true	"mid"

The extracted table is of the same type (a Polars DataFrame) as the target table. Previously we used `load_dataset()` with the `tbl_type="polars"` option to fetch the dataset in that form.

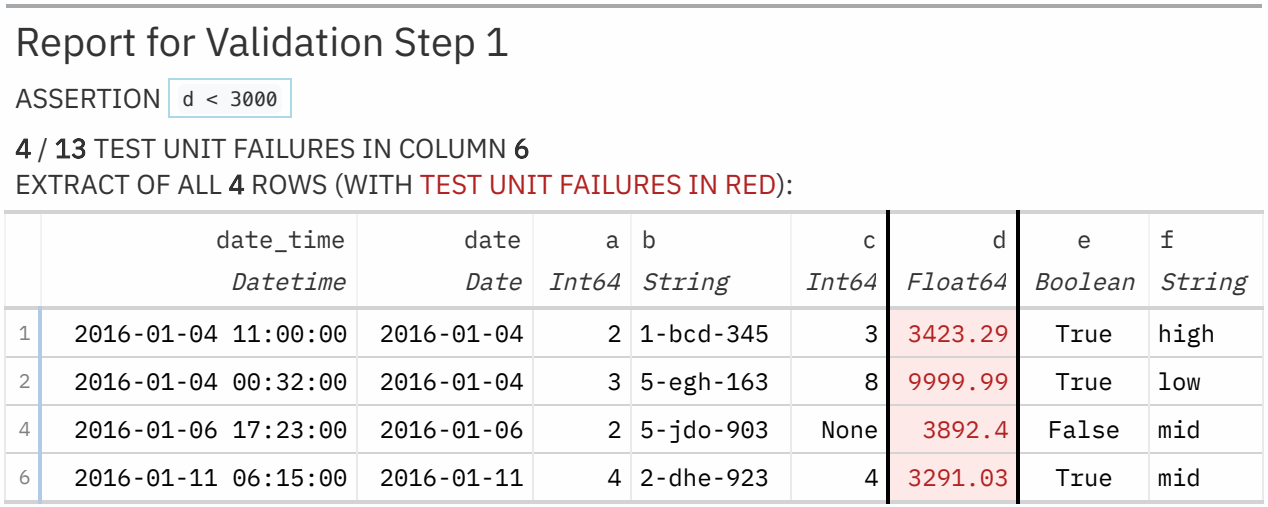
Note these important details about using `Validate.get_data_extracts()`:

- the parameter `i=1` corresponds to the step number shown in the validation report (1-indexed, not 0-indexed)
- setting `frame=True` returns the data as a DataFrame rather than a dictionary (only works when `i` is a single integer)
- the extract includes all columns from the original data, not just the column being validated
- an additional `_row_num_` column is added to identify the original row positions

4.20.3 Step Reports

Step reports provide another way to access and visualize failing data. When you generate a step report for a validation step that has failing rows, those failing rows are displayed directly in the report:

```
# Get a step report for the first validation step
step_report = validation.get_step_report(i=1)

step_report
```



Report for Validation Step 1

ASSERTION `d < 3000`

4 / 13 TEST UNIT FAILURES IN COLUMN 6
EXTRACT OF ALL 4 ROWS (WITH TEST UNIT FAILURES IN RED):

	date_time *Datetime*	date *Date*	a *Int64*	b *String*	c *Int64*	d *Float64*	e *Boolean*	f *String*
1	2016-01-04 11:00:00	2016-01-04	2	1-bcd-345	3	3423.29	True	high
2	2016-01-04 00:32:00	2016-01-04	3	5-egh-163	8	9999.99	True	low
4	2016-01-06 17:23:00	2016-01-06	2	5-jdo-903	None	3892.4	False	mid
6	2016-01-11 06:15:00	2016-01-11	4	2-dhe-923	4	3291.03	True	mid

Step reports offer several advantages for working with data extracts as they:

1. provide immediate visual context by highlighting the specific column being validated
2. format the data for better readability, especially useful when sharing results with colleagues
3. include additional metadata about the validation step and failure statistics

For steps with many failures, you can customize how many rows to display:

```
# Limit to just 2 rows of failing data
limited_report = validation.get_step_report(i=1, limit=2)

limited_report
```

Report for Validation Step 1

ASSERTION `d < 3000`

4 / 13 TEST UNIT FAILURES IN COLUMN 6
EXTRACT OF FIRST 2 ROWS (WITH TEST UNIT FAILURES IN RED):

	date_time *Datetime*	date *Date*	a *Int64*	b *String*	c *Int64*	d *Float64*	e *Boolean*	f *String*
1	2016-01-04 11:00:00	2016-01-04	2	1-bcd-345	3	3423.29	True	high
2	2016-01-04 00:32:00	2016-01-04	3	5-egh-163	8	9999.99	True	low

Step reports are particularly valuable when you want to quickly inspect the failing data without extracting it into a separate DataFrame. They provide a bridge between the high-level validation report and the detailed data extracts.

4.21 Viewing Data Extracts with `preview()`

To get a consistent HTML representation of any data extract (regardless of the table type), we can use the `preview()` function:

```
pb.preview(data=extract_1)
```

POLARS | ROWS 4 | COLUMNS 9

	date_time *Datetime*	date *Date*	a *Int64*	b *String*	c *Int64*	d *Float64*	e *Boolean*	f *String*
1	2016-01-04 11:00:00	2016-01-04	2	1-bcd-345	3	3423.29	True	high
2	2016-01-04 00:32:00	2016-01-04	3	5-egh-163	8	9999.99	True	low
4	2016-01-06 17:23:00	2016-01-06	2	5-jdo-903	None	3892.4	False	mid
6	2016-01-11 06:15:00	2016-01-11	4	2-dhe-923	4	3291.03	True	mid

The view is optimized for readability, with column names and data types displayed in a compact format. Notice that the `_row_num_` column is now part of the table stub and doesn't steal focus from the table's original columns.

The `preview()` function is designed to provide the head and tail (5 rows each) of the table so very large extracts won't overflow the display.

4.22 Working with Multiple Validation Steps

When validating data with multiple steps, you can extract failing rows from any step or combine extracts from multiple steps:

```
# Create a validation with multiple steps
multi_validation = (
    pb.Validate(data=pb.load_dataset(dataset="small_table", tbl_type="polars"))
    .col_vals_gt(columns="a", value=3)                                  # Step 1
    .col_vals_lt(columns="d", value=3000)                               # Step 2
    .col_vals_regex(columns="b", pattern="^[0-9]-[a-z]{3}-[0-9]{3}$")   # Step 3
    .interrogate()
)

multi_validation
```

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_gt()	a	3	○→	√	13	6 0.46	7 0.54	—	—	—	CSV
2	col_vals_lt()	d	3000	○→	√	13	9 0.69	4 0.31	—	—	—	CSV
3	col_vals_regex()	b	^[0-9]-[a-z]{3}…	○→	√	13	13 1.00	0 0.00	—	—	—	—

4.22.1 Extracting Data from a Specific Step

You can access extracts from any specific validation step:

```
# Get extracts from step 2 (`d < 3000` validation)
less_than_failures = multi_validation.get_data_extracts(i=2, frame=True)

less_than_failures
```

shape: (4, 9)

_row_num_	date_time	date	a	b	c	d	e	f
u32	datetime[μs]	date	i64	str	i64	f64	bool	str
1	2016-01-04 11:00:00	2016-01-04	2	"1-bcd-345"	3	3423.29	true	"high"
2	2016-01-04 00:32:00	2016-01-04	3	"5-egh-163"	8	9999.99	true	"low"
4	2016-01-06 17:23:00	2016-01-06	2	"5-jdo-903"	null	3892.4	false	"mid"
6	2016-01-11 06:15:00	2016-01-11	4	"2-dhe-923"	4	3291.03	true	"mid"

Using `frame=True` means that returned value will be a DataFrame (not a dictionary that contains a single DataFrame).

If a step has no failing rows, an empty DataFrame will be returned:

```
# Get extracts from step 3 (regex check)
regex_failures = multi_validation.get_data_extracts(i=3, frame=True)

regex_failures
```

shape: (0, 9)

_row_num_	date_time	date	a	b	c	d	e	f
u32	datetime[μs]	date	i64	str	i64	f64	bool	str

4.22.2 Getting All Extracts at Once

To retrieve extracts from all steps with failures in one command:

```
# Get all extracts ()
all_extracts = multi_validation.get_data_extracts()

# Display the step numbers that have extracts
print(f"Steps with data extracts: {list(all_extracts.keys())}")
```

```
Steps with data extracts: [1, 2, 3]
```

A dictionary of DataFrames is returned and only steps with failures will appear in this dictionary.

4.22.3 Getting Specific Extracts

You can also retrieve data extracts from several specified steps as a dictionary:

```
# Get extracts from steps 1 and 2 as a dictionary
extract_dict = multi_validation.get_data_extracts(i=[1, 2])

# The keys are the step numbers
print(f"Dictionary keys: {list(extract_dict.keys())}")

# Get the number of failing rows in each extract
for step, extract in extract_dict.items():
    print(f"Step {step}: {len(extract)} failing rows")
```

```
Dictionary keys: [1, 2]
Step 1: 7 failing rows
Step 2: 4 failing rows
```

Note that `frame=True` cannot be used when retrieving multiple extracts.

4.23 Applications of Data Extracts

Once you have extracted the failing data, there are numerous ways to analyze and use this information to improve data quality. Let's explore some practical applications.

4.23.1 Finding Patterns Across Validation Steps

You can analyze patterns across different validation steps by combining extracts:

```python
# Get a consolidated view of all rows that failed any validation
all_failure_rows = set()
for step, extract in all_extracts.items():
    if len(extract) > 0:
        all_failure_rows.update(extract["_row_num_"])

print(f"Total unique rows with failures: {len(all_failure_rows)}")
print(f"Row numbers with failures: {sorted(all_failure_rows)}")
```

```
Total unique rows with failures: 8
Row numbers with failures: [1, 2, 4, 6, 9, 10, 12, 13]
```

4.23.2 Identifying Rows with Multiple Failures

You might want to find rows that failed multiple validation checks, as these often represent more serious data quality issues:

```python
# Get row numbers from each extract
step1_rows = set(multi_validation.get_data_extracts(i=1, frame=True)["_row_num_"])
step2_rows = set(multi_validation.get_data_extracts(i=2, frame=True)["_row_num_"])

# Find rows that failed both validations
common_failures = step1_rows.intersection(step2_rows)
print(f"Rows failing both step 1 and step 2: {common_failures}")
```

```
Rows failing both step 1 and step 2: {1, 2, 4}
```

4.23.3 Statistical Analysis of Failing Values

Once you have data extracts, you can perform statistical analysis to identify patterns in the failing data:

```
# Get extracts from step 2
d_value_failures = multi_validation.get_data_extracts(i=2, frame=True)

# Basic statistical analysis of the failing values
if len(d_value_failures) > 0:
    print(f"Min failing value: {d_value_failures['d'].min()}")
    print(f"Max failing value: {d_value_failures['d'].max()}")
    print(f"Mean failing value: {d_value_failures['d'].mean()}")
```

```
Min failing value: 3291.03
Max failing value: 9999.99
Mean failing value: 5151.6775
```

These analysis techniques help you thoroughly investigate data quality issues by examining failing data from multiple perspectives. Rather than treating failures as isolated incidents, you can identify patterns that might indicate systematic problems in your data pipeline.

4.23.4 Detailed Analysis with `col_summary_tbl()`

For a more comprehensive view of the statistical properties of your extract data, you can use the `col_summary_tbl()` function:

```
# Get extracts from step 2
d_value_failures = multi_validation.get_data_extracts(i=2, frame=True)

# Generate a comprehensive statistical summary of the failing data
pb.col_summary_tbl(d_value_failures)
```

POLARS | ROWS 4 | COLUMNS 9

	Column	NA	UQ	Mean	SD	Min	P_5	Q_1	Med	Q_3	P_{95}	Max	IQ
N	_row_num_ UInt32	0 0	4 1	3.25	2.22	1	1.01	1.75	3	4.5	5.7	6	2
D	date_time Datetime(time_unit='us', time_zone=None)	0 0	4 1	-	-	2016 01 04 00:32:00	-	-	-	-	-	2016 01 11 06:15:00	
D	date Date	0 0	3 0.75	-	-	2016 01 04	-	-	-	-	-	2016 01 11	
N	a Int64	0 0	3 0.75	2.75	0.96	2	2	2	2.5	3.25	3.85	4	1
S	b String	0 0	4 1	9	0	9	9	9	9	9	9	9	
N	c Int64	1 0.25	4 1	5	2.65	3	3.01	3.5	4	6	7.6	8	
N	d Float64	0 0	4 1	5,151.68	3,242.49	3291.03	3,293.01	3,390.22	3,657.85	5,419.3	9,083.85	9999.99	2,029
T/F	e Boolean	0 0	T0.75 F0.25	-	-	-	-	-	-	-	-	-	
S	f String	0 0	3 0.75	3.25	0.5	3	3	3	3	3.25	3.85	4	0

String columns statistics regard the string's length.

This statistical overview provides:

1. a count of values (including missing values)
2. type information for each column
3. distribution metrics like min, max, mean, and quartiles for numeric columns

4. frequency of common values for categorical columns
5. missing value counts and proportions

Using `col_summary_tbl()` on data extracts lets you quickly understand the characteristics of failing data without writing custom analysis code. This approach is particularly valuable when:

- You need to understand the statistical properties of failing records
- You want to compare distributions of failing vs passing data
- You're looking for anomalies or unexpected patterns within the failing rows

For example, if values failing a validation check are concentrated at certain quantiles or have an unusual distribution shape, this might indicate a systematic data collection or processing issue rather than random errors.

4.24 Using Extracts for Data Quality Improvement

Data extracts are especially valuable for:

1. **Root Cause Analysis**: examining the full context of failing rows to understand why they failed
2. **Data Cleaning**: creating targeted cleanup scripts that focus only on problematic records
3. **Feedback Loops**: sharing specific examples with data providers to improve upstream quality
4. **Pattern Recognition**: identifying systemic issues by analyzing groups of failing records

Here's an example of using extracts to create a corrective action plan:

```
import polars as pl

# Create a new sample of an extract DF
sample_extract = pl.DataFrame({
    "id": range(1, 11),
    "value": [3500, 4200, 3800, 9800, 5500, 7200, 8300, 4100, 7600, 3200],
    "category": ["A", "B", "A", "C", "B", "A", "C", "B", "A", "B"],
    "region": [
        "South", "South", "North", "East", "South",
        "South", "East", "South", "West", "South"
    ]
})

# Identify which regions have the most failures
region_counts = (
    sample_extract
    .group_by("region")
    .agg(pl.len().alias("failure_count"))
    .sort("failure_count", descending=True)
)

region_counts
```

shape: (4, 2)

region	failure_count
str	u32
"South"	6
"East"	2
"North"	1
"West"	1

Analysis shows that 6 out of 10 failing records (60%) are from the `"South"` region, making it the highest priority area for data quality investigation. This suggests a potential systemic issue with data collection or processing in that specific region.

4.25 Best Practices for Working with Data Extracts

When incorporating data extracts into your data quality workflow:

1. Use extracts for investigation, not just reporting: the real value is in the insights you gain from analyzing the problematic data
2. Combine with other Pointblank features: data extracts work well with step reports and can inform threshold settings for future validations
3. Consider sampling for very large datasets: if your extracts contain thousands of rows, focus your investigation on a representative sample
4. Look beyond individual validation steps: cross-reference extracts from different steps to identify complex issues that span multiple validation rules
5. Document patterns in failing data: record and share insights about common failure modes to build organizational knowledge about data quality issues.

By integrating these practices into your data validation workflow, you'll transform data extracts from simple error lists into powerful diagnostic tools. The most successful data quality initiatives treat extracts as the starting point for investigation rather than the end result of validation. When systematically analyzed and documented, patterns in failing data can reveal underlying issues in data systems, collection methods, or business processes that might otherwise remain hidden. Remember that the ultimate goal isn't just to identify problematic records, but to use that information to implement targeted improvements that prevent similar issues from occurring in the future.

4.26 Conclusion

Data extracts bridge the gap between high-level validation statistics and the detailed context needed to fix data quality issues. By providing access to the actual failing records, Pointblank enables you to:

- pinpoint exactly which data points caused validation failures
- understand the full context around problematic values
- develop targeted strategies for data cleanup and quality improvement
- communicate specific examples to stakeholders

Whether you're accessing extracts through CSV downloads, the `Validate.get_data_extracts()` method, or step reports, this feature provides the detail needed to move from identifying problems to implementing solutions.

Sundering data? First off, let's get the correct meaning across here. Sundering is really just splitting, dividing, cutting into two pieces. And it's a useful thing we can do in Pointblank to any data that we are validating. When you interrogate the data, you learn about which rows have test failures within them. With more validation steps, we get an even better picture of this simply by virtue of more testing.

The power of sundering lies in its ability to separate your data into two distinct categories:

1. rows that pass all validation checks (clean data)
2. rows that fail one or more validation checks (problematic data)

This approach allows you to:

- focus your analysis on clean, reliable data
- isolate problematic records for investigation or correction
- create pipelines that handle good and bad data differently

Let's use the `small_table` in our examples to show just how sundering is done. Here's that table:

POLARS ROWS 13 COLUMNS 8

	date_time *Datetime*	date *Date*	a *Int64*	b *String*	c *Int64*	d *Float64*	e *Boolean*	f *String*
1	2016-01-04 11:00:00	2016-01-04	2	1-bcd-345	3	3423.29	True	high
2	2016-01-04 00:32:00	2016-01-04	3	5-egh-163	8	9999.99	True	low
3	2016-01-05 13:32:00	2016-01-05	6	8-kdg-938	3	2343.23	True	high
4	2016-01-06 17:23:00	2016-01-06	2	5-jdo-903	None	3892.4	False	mid
5	2016-01-09 12:36:00	2016-01-09	8	3-ldm-038	7	283.94	True	low
6	2016-01-11 06:15:00	2016-01-11	4	2-dhe-923	4	3291.03	True	mid
7	2016-01-15 18:46:00	2016-01-15	7	1-knw-093	3	843.34	True	high
8	2016-01-17 11:27:00	2016-01-17	4	5-boe-639	2	1035.64	False	low
9	2016-01-20 04:30:00	2016-01-20	3	5-bce-642	9	837.93	False	high
10	2016-01-20 04:30:00	2016-01-20	3	5-bce-642	9	837.93	False	high
11	2016-01-26 20:07:00	2016-01-26	4	2-dmx-010	7	833.98	True	low
12	2016-01-28 02:51:00	2016-01-28	2	7-dmx-010	8	108.34	False	low
13	2016-01-30 11:23:00	2016-01-30	1	3-dka-303	None	2230.09	True	high

4.27 A Simple Example Where Data is Torn Asunder

We'll begin with a very simple validation plan, having only a single step. There *will be* failing test units here.

```
import pointblank as pb

validation = (
    pb.Validate(data=pb.load_dataset(dataset="small_table"))
    .col_vals_ge(columns="d", value=1000)
    .interrogate()
)

validation
```

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_ge()	d	1000	○→	✓	13	7 0.54	6 0.46	—	—	—	CSV

We see six failing test units in `FAIL` column of the above validation report table. There is a data extract (collection of failing rows) available. Let's use the `Validate.get_data_extracts()` method to have a look at it.

```
validation.get_data_extracts(i=1, frame=True)
```

shape: (6, 9)

_row_num_	date_time	date	a	b	c	d	e	f
u32	datetime[μs]	date	i64	str	i64	f64	bool	str
5	2016-01-09 12:36:00	2016-01-09	8	"3-ldm-038"	7	283.94	true	"low"
7	2016-01-15 18:46:00	2016-01-15	7	"1-knw-093"	3	843.34	true	"high"
9	2016-01-20 04:30:00	2016-01-20	3	"5-bce-642"	9	837.93	false	"high"
10	2016-01-20 04:30:00	2016-01-20	3	"5-bce-642"	9	837.93	false	"high"
11	2016-01-26 20:07:00	2016-01-26	4	"2-dmx-010"	7	833.98	true	"low"
12	2016-01-28 02:51:00	2016-01-28	2	"7-dmx-010"	8	108.34	false	"low"

This is six rows of data that had failing test units in column `d`. Indeed we can see that all values in that column are less than `1000` (and we asserted that values should be greater than or equal to `1000`). This is the 'bad' data, if you will. Using the `Validate.get_sundered_data()` method, we get the 'good' part:

```
validation.get_sundered_data()
```

shape: (7, 8)

date_time	date	a	b	c	d	e	f
datetime[μs]	date	i64	str	i64	f64	bool	str
2016-01-04 11:00:00	2016-01-04	2	"1-bcd-345"	3	3423.29	true	"high"
2016-01-04 00:32:00	2016-01-04	3	"5-egh-163"	8	9999.99	true	"low"
2016-01-05 13:32:00	2016-01-05	6	"8-kdg-938"	3	2343.23	true	"high"
2016-01-06 17:23:00	2016-01-06	2	"5-jdo-903"	null	3892.4	false	"mid"
2016-01-11 06:15:00	2016-01-11	4	"2-dhe-923"	4	3291.03	true	"mid"
2016-01-17 11:27:00	2016-01-17	4	"5-boe-639"	2	1035.64	false	"low"
2016-01-30 11:23:00	2016-01-30	1	"3-dka-303"	null	2230.09	true	"high"

This is a Polars DataFrame of seven rows. All values in `d` were passing test units (i.e., fulfilled the expectation outlined in the validation step) and, in many ways, this is like a 'good extract'.

You can always collect the failing rows with `Validate.get_sundered_data()` by using the `type="fail"` option. Let's try that here:

```
validation.get_sundered_data(type="fail")
```

shape: (6, 8)

date_time	date	a	b	c	d	e	f
datetime[μs]	date	i64	str	i64	f64	bool	str
2016-01-09 12:36:00	2016-01-09	8	"3-ldm-038"	7	283.94	true	"low"
2016-01-15 18:46:00	2016-01-15	7	"1-knw-093"	3	843.34	true	"high"
2016-01-20 04:30:00	2016-01-20	3	"5-bce-642"	9	837.93	false	"high"
2016-01-20 04:30:00	2016-01-20	3	"5-bce-642"	9	837.93	false	"high"
2016-01-26 20:07:00	2016-01-26	4	"2-dmx-010"	7	833.98	true	"low"
2016-01-28 02:51:00	2016-01-28	2	"7-dmx-010"	8	108.34	false	"low"

It gives us the same rows as in the DataFrame obtained from using `validation.get_data_extracts(i=1, frame=True)`. Two important things to know about `Validate.get_sundered_data()` are that the table rows returned from `type=pass` (the default) and `type=fail` are:

- the sum of rows across these returned tables will be equal to that of the original table

- the rows in each split table are mutually exclusive (i.e., you won't find the same row in both)

You can think of sundered data as a filtered version of the original dataset based on validation results. While the simple example illustrates how this process works on a basic level, the value of the method is better seen in a slightly more complex example.

4.28 Using `get_sundered_data()` with a More Comprehensive Validation

The previous example used exactly one validation step. You're likely to use more than that in standard practice so let's see how `Validate.get_sundered_data()` works in those common situations. Here's a validation with three steps:

```
validation_2 = (
    pb.Validate(data=pb.load_dataset(dataset="small_table"))
    .col_vals_ge(
        columns="d",
        value=1000
    )
    .col_vals_not_null(columns="c")
    .col_vals_gt(
        columns="a",
        value=2
    )
    .interrogate()
)

validation_2
```

	STEP	COLUMNS	VALUES	TBL	EVAL	UNITS	PASS	FAIL	W	E	C	EXT
1	col_vals_ge()	d	1000	○→	✓	13	7 0.54	6 0.46	—	—	—	CSV
2	col_vals_not_null()	c	—	○→	✓	13	11 0.85	2 0.15	—	—	—	CSV
3	col_vals_gt()	a	2	○→	✓	13	9 0.69	4 0.31	—	—	—	CSV

There are quite a few failures here across the three validation steps. In the `FAIL` column of the validation report table, there are 12 failing test units if we were to tally them up. So if the input table has 13 rows in total, does this mean there would be one row in the table returned by `Validate.get_sundered_data()`? Not so:

```
validation_2.get_sundered_data()
```

shape: (4, 8)

date_time	date	a	b	c	d	e	f
datetime[μs]	date	i64	str	i64	f64	bool	str
2016-01-04 00:32:00	2016-01-04	3	"5-egh-163"	8	9999.99	true	"low"
2016-01-05 13:32:00	2016-01-05	6	"8-kdg-938"	3	2343.23	true	"high"
2016-01-11 06:15:00	2016-01-11	4	"2-dhe-923"	4	3291.03	true	"mid"
2016-01-17 11:27:00	2016-01-17	4	"5-boe-639"	2	1035.64	false	"low"

There are four rows. This is because the different validation steps tested values in different columns of the table. Some of the failing test units had to have occurred in more than once in certain rows. The rows that didn't have any failing test units across the three different tests (in three different columns) are the ones seen above. This brings us to the third important thing about the sundering process:

- the absence of test-unit failures in a row across all validation steps means those rows are returned as the 'passing' set, all others are placed in the 'failing' set

In validations where many validation steps are used, we can be more confident about the level of data quality for those rows returned in the passing set. But not every type of validation step is considered within this splitting procedure. The next section will explain the rules on that.

4.29 The Validation Methods Considered When Sundering

The sundering procedure relies on row-level validation types to be used. This makes sense as it's impossible to judge the quality of a row when using the `col_exists()` validation method, for example. Luckily, we have many row-level validation methods; here's a list:

- `Validate.col_vals_gt()`
- `Validate.col_vals_lt()`
- `Validate.col_vals_ge()`

- `Validate.col_vals_le()`
- `Validate.col_vals_eq()`
- `Validate.col_vals_ne()`
- `Validate.col_vals_between()`
- `Validate.col_vals_outside()`
- `Validate.col_vals_in_set()`
- `Validate.col_vals_not_in_set()`
- `Validate.col_vals_null()`
- `Validate.col_vals_not_null()`
- `Validate.col_vals_regex()`
- `Validate.col_vals_expr()`
- `Validate.rows_distinct()`
- `Validate.rows_complete()`
- `Validate.conjointly()`

This is the same list of validation methods that are considered when creating data extracts.

There are some additional caveats though. Even if using a validation method drawn from the set above, the validation step won't be used for sundering if:

- the `active=` parameter for that step has been set to `False`
- the `pre=` parameter has been used

The first one makes intuitive sense (you decided to skip this validation step entirely), the second one requires some explanation. Using `pre=` allows you to modify the target table, there's no easy or practical way to compare rows in a mutated table compared to the original table (e.g., a mutation may drastically reduce the number of rows).

4.30 Practical Applications of Sundering

4.30.1 1. Creating Clean Datasets for Analysis

One of the most common use cases for sundering is preparing validated data for downstream analysis:

```python
# Comprehensive validation for analysis-ready data
analysis_validation = (
    pb.Validate(data=pb.load_dataset(dataset="small_table"))
    .col_vals_not_null(columns=["a", "b", "c", "d", "e", "f"])  # No missing values
    .col_vals_gt(columns="a", value=0)                          # Positive values only
    .col_vals_lt(columns="d", value=10000)                      # No extreme outliers
    .interrogate()
)

# Extract only the clean data that passed all checks
clean_data = analysis_validation.get_sundered_data(type="pass")

# Use the clean data for your analysis
pb.preview(clean_data)
```

POLARS | ROWS 11 | COLUMNS 8

	date_time *Datetime*	date *Date*	a *Int64*	b *String*	c *Int64*	d *Float64*	e *Boolean*	f *String*
1	2016-01-04 11:00:00	2016-01-04	2	1-bcd-345	3	3423.29	True	high
2	2016-01-04 00:32:00	2016-01-04	3	5-egh-163	8	9999.99	True	low
3	2016-01-05 13:32:00	2016-01-05	6	8-kdg-938	3	2343.23	True	high
4	2016-01-09 12:36:00	2016-01-09	8	3-ldm-038	7	283.94	True	low
5	2016-01-11 06:15:00	2016-01-11	4	2-dhe-923	4	3291.03	True	mid
7	2016-01-17 11:27:00	2016-01-17	4	5-boe-639	2	1035.64	False	low
8	2016-01-20 04:30:00	2016-01-20	3	5-bce-642	9	837.93	False	high
9	2016-01-20 04:30:00	2016-01-20	3	5-bce-642	9	837.93	False	high
10	2016-01-26 20:07:00	2016-01-26	4	2-dmx-010	7	833.98	True	low
11	2016-01-28 02:51:00	2016-01-28	2	7-dmx-010	8	108.34	False	low

This approach ensures that any subsequent analysis is based on data that meets your quality standards, reducing the risk of misleading results or spurious conclusions due to problematic records. By making validation an explicit step in your analytical workflow, you create a natural quality gate that prevents invalid data from influencing your findings.

4.30.2 2. Creating Parallel Workflows for Clean and Problematic Data

You can use sundering to create parallel processing paths:

```
# Get both clean and problematic data
clean_data = analysis_validation.get_sundered_data(type="pass")
problem_data = analysis_validation.get_sundered_data(type="fail")

# Process clean data (in real applications, you'd do more here)
print(f"Clean data size: {len(clean_data)} rows")

# Log problematic data for investigation
print(f"Problematic data size: {len(problem_data)} rows")
```

```
Clean data size: 11 rows
Problematic data size: 2 rows
```

This approach enables you to build robust data processing pathways with separate handling for clean and problematic data. In production environments, you could save problematic records to a separate location for further investigation, generate detailed logs of validation failures, and trigger automated notifications to data stewards when issues arise. By establishing clear protocols for handling both data streams, you create a systematic approach to data quality that balances immediate analytical needs with longer-term data improvement goals.

4.30.3 3. Data Quality Monitoring and Improvement

Tracking the ratio of passing to failing rows over time can help monitor data quality trends:

```
# Calculate data quality metrics
total_rows = len(pb.load_dataset(dataset="small_table"))
passing_rows = len(clean_data)
quality_score = passing_rows / total_rows

print(f"Data quality score: {quality_score:.2%}")
print(f"Passing rows: {passing_rows} out of {total_rows}")
```

```
Data quality score: 84.62%
Passing rows: 11 out of 13
```

By tracking these metrics over time, you can measure the impact of your data quality improvement efforts and communicate progress to stakeholders. This approach transforms sundering from a one-time filtering tool into an ongoing data quality management system, where improving the ratio of passing rows becomes a measurable business objective aligned with broader data governance goals.

4.31 Best Practices for Using Sundered Data

When incorporating data sundering into your workflow, consider these best practices:

1. Be comprehensive in your validation: the more validation steps you include (assuming they're meaningful), the more confidence you can have in your passing dataset
2. Document your validation criteria: when sharing sundered data with others, always document the criteria used to determine passing rows
3. Consider traceability: for audit purposes, it may be valuable to add a column indicating whether a record was originally in the passing or failing set
4. Balance strictness and practicality: if you're too strict with validation rules, you might end up with very few passing rows; consider the appropriate level of strictness for your use case
5. Use sundering as part of a pipeline: automate the process of validation, sundering, and subsequent handling of the two resulting datasets
6. Continually refine validation rules: as you learn more about your data and domain, update your validation rules to improve the accuracy of your sundering process

By following these best practices, data scientists and engineers can transform sundering from a simple utility into a strategic component of their data quality framework. When implemented thoughtfully, sundering enables a shift from reactive data cleaning to proactive quality management, where validation criteria evolve alongside your understanding of the data.

The ultimate goal isn't just to separate good data from bad, but to gradually improve your entire dataset over time by addressing the root causes of validation failures that appear in the failing set. This approach turns data validation from a gatekeeper function into a continuous improvement process.

4.32 Conclusion

Data sundering provides a powerful way to separate your data based on validation results. While the concept is simple (splitting data into passing and failing sets) the feature can very useful in many data workflows. By integrating sundering into your data pipeline, you can:

- ensure that downstream analysis only works with validated data
- create focused datasets for different purposes
- improve overall data quality through systematic identification and isolation of problematic records
- build more robust data pipelines that explicitly handle data quality issues

So long as you're aware of the rules and limitations of sundering, you're likely to find it to be a simple and useful way to filter your input table on the basis of a validation plan, turning data validation from a passive reporting tool into an active component of your data processing workflow.

5 Data Inspection

In many cases, it's *good* to look at your data tables. Before validating a table, you'll likely want to inspect a portion of it before diving into the creation of data-quality rules. This is pretty easily done with Polars and Pandas DataFrames, however, it's not as easy with database tables and each table backend displays things differently.

To make this common task a little better, you can use the `preview()` function in Pointblank. It has been designed to work with every table that the package supports (i.e., DataFrames and Ibis-backend tables, the latter of which are largely database tables). Plus, what's shown in the output is consistent, no matter what type of data you're looking at.

5.1 Viewing a Table with `preview()`

Let's look at how `preview()` works. It requires only a table and, for this first example, let's use the `nycflights` dataset:

```
import pointblank as pb

nycflights = pb.load_dataset(dataset="nycflights", tbl_type="polars")

pb.preview(nycflights)
```

POLARS | ROWS 336,776 | COLUMNS 18

	year *Int64*	month *Int64*	day *Int64*	dep_time *Int64*	sched_dep_time *Int64*	dep_delay *Int64*	arr_time *Int64*	sched_arr_time *Int64*	arr_delay *Int64*	carrier *String*	flight *Int64*	tailnum *String*
1	2013	1	1	517	515	2	830	819	11	UA	1545	N14228
2	2013	1	1	533	529	4	850	830	20	UA	1714	N24211
3	2013	1	1	542	540	2	923	850	33	AA	1141	N619AA
4	2013	1	1	544	545	-1	1004	1022	-18	B6	725	N804JB
5	2013	1	1	554	600	-6	812	837	-25	DL	461	N668DN
336772	2013	9	30	None	1455	None	None	1634	None	9E	3393	None
336773	2013	9	30	None	2200	None	None	2312	None	9E	3525	None
336774	2013	9	30	None	1210	None	None	1330	None	MQ	3461	N535MQ
336775	2013	9	30	None	1159	None	None	1344	None	MQ	3572	N511MQ
336776	2013	9	30	None	840	None	None	1020	None	MQ	3531	N839MQ

This is an HTML table using the style of the other reporting tables in the library. The header is more minimal here, only showing the type of table we're looking at (`POLARS` in this case) along with the table dimensions. The column headers provide both the column names and the column data types.

By default, we're getting the first five rows and the last five rows. Row numbers (from the original dataset) provide an indication of which rows are the head and tail rows. The blue lines provide additional demarcation of the column containing the row numbers and the head and tail row groups. Finally, any cells with missing values are prominently styled with red lettering and a lighter red background.

If you'd rather not see the row numbers in the table, you can use the `show_row_numbers=False` option. Let's try that with the `game_revenue` dataset as a DuckDB table:

```
game_revenue = pb.load_dataset(dataset="game_revenue", tbl_type="duckdb")

pb.preview(game_revenue, show_row_numbers=False)
```

DUCKDB ROWS 2,000 COLUMNS 11

player_id *string*	session_id *string*	session_start *timestamp*	time *timestamp*	item_type *string*	item_name *string*	ite
ECPANOIXLZHF896	ECPANOIXLZHF896-eol2j8bs	2015-01-01 01:31:03+00:00	2015-01-01 01:31:27+00:00	iap	offer2	
ECPANOIXLZHF896	ECPANOIXLZHF896-eol2j8bs	2015-01-01 01:31:03+00:00	2015-01-01 01:36:57+00:00	iap	gems3	
ECPANOIXLZHF896	ECPANOIXLZHF896-eol2j8bs	2015-01-01 01:31:03+00:00	2015-01-01 01:37:45+00:00	iap	gold7	
ECPANOIXLZHF896	ECPANOIXLZHF896-eol2j8bs	2015-01-01 01:31:03+00:00	2015-01-01 01:42:33+00:00	ad	ad_20sec	
ECPANOIXLZHF896	ECPANOIXLZHF896-hdu9jkls	2015-01-01 11:50:02+00:00	2015-01-01 11:55:20+00:00	ad	ad_5sec	
NAOJRDMCSEBI281	NAOJRDMCSEBI281-j2vs9ilp	2015-01-21 01:57:50+00:00	2015-01-21 02:02:50+00:00	ad	ad_survey	
NAOJRDMCSEBI281	NAOJRDMCSEBI281-j2vs9ilp	2015-01-21 01:57:50+00:00	2015-01-21 02:22:14+00:00	ad	ad_survey	
RMOSWHJGELCI675	RMOSWHJGELCI675-vbhcsmtr	2015-01-21 02:39:48+00:00	2015-01-21 02:40:00+00:00	ad	ad_5sec	
RMOSWHJGELCI675	RMOSWHJGELCI675-vbhcsmtr	2015-01-21 02:39:48+00:00	2015-01-21 02:47:12+00:00	iap	offer5	
GJCXNTWEBIPQ369	GJCXNTWEBIPQ369-9elq67md	2015-01-21 03:59:23+00:00	2015-01-21 04:06:29+00:00	ad	ad_5sec	

With the above preview, the row numbers are gone. The horizontal blue line still serves to divide the top and bottom rows of the table, however.

5.2 Adjusting the Number of Rows Shown

It could be that displaying the five top and bottom rows is not preferred. This can be changed with the `n_head=` and `n_tail=`. Maybe, you want three from the top along with the last row? Let's try that out with the `small_table` dataset as a Pandas DataFrame:

```
small_table = pb.load_dataset(dataset="small_table", tbl_type="pandas")

pb.preview(small_table, n_head=3, n_tail=1)
```

PANDAS ROWS 13 COLUMNS 8

	date_time *datetime64[ns]*	date *datetime64[ns]*	a *int64*	b *object*	c *float64*	d *float64*	e *bool*	f *object*
1	2016-01-04 11:00:00	2016-01-04 00:00:00	2	1-bcd-345	3.0	3423.29	True	high
2	2016-01-04 00:32:00	2016-01-04 00:00:00	3	5-egh-163	8.0	9999.99	True	low
3	2016-01-05 13:32:00	2016-01-05 00:00:00	6	8-kdg-938	3.0	2343.23	True	high
13	2016-01-30 11:23:00	2016-01-30 00:00:00	1	3-dka-303	NA	2230.09	True	high

If you're looking at a small table and want to see the entirety of it, you can enlarge the `n_head=` and `n_tail=` values:

```
small_table = pb.load_dataset(dataset="small_table", tbl_type="pandas")

pb.preview(small_table, n_head=10, n_tail=10)
```

PANDAS ROWS 13 COLUMNS 8

	date_time *datetime64[ns]*	date *datetime64[ns]*	a *int64*	b *object*	c *float64*	d *float64*	e *bool*	f *object*
1	2016-01-04 11:00:00	2016-01-04 00:00:00	2	1-bcd-345	3.0	3423.29	True	high
2	2016-01-04 00:32:00	2016-01-04 00:00:00	3	5-egh-163	8.0	9999.99	True	low
3	2016-01-05 13:32:00	2016-01-05 00:00:00	6	8-kdg-938	3.0	2343.23	True	high
4	2016-01-06 17:23:00	2016-01-06 00:00:00	2	5-jdo-903	NA	3892.4	False	mid
5	2016-01-09 12:36:00	2016-01-09 00:00:00	8	3-ldm-038	7.0	283.94	True	low
6	2016-01-11 06:15:00	2016-01-11 00:00:00	4	2-dhe-923	4.0	3291.03	True	mid
7	2016-01-15 18:46:00	2016-01-15 00:00:00	7	1-knw-093	3.0	843.34	True	high
8	2016-01-17 11:27:00	2016-01-17 00:00:00	4	5-boe-639	2.0	1035.64	False	low
9	2016-01-20 04:30:00	2016-01-20 00:00:00	3	5-bce-642	9.0	837.93	False	high
10	2016-01-20 04:30:00	2016-01-20 00:00:00	3	5-bce-642	9.0	837.93	False	high
11	2016-01-26 20:07:00	2016-01-26 00:00:00	4	2-dmx-010	7.0	833.98	True	low
12	2016-01-28 02:51:00	2016-01-28 00:00:00	2	7-dmx-010	8.0	108.34	False	low
13	2016-01-30 11:23:00	2016-01-30 00:00:00	1	3-dka-303	NA	2230.09	True	high

Given that the table has 13 rows, asking for 20 rows to be displayed effectively shows the entire table.

5.3 Previewing a Subset of Columns

The preview scales well to tables that have many columns by allowing for a horizontal scroll. However, previewing data from all columns can be impractical if you're only concerned with a key set of them. To preview only a subset of a table's columns, we can use the `columns_subset=` argument. Let's do this with the `nycflights` dataset and provide a list of six columns from that table.

```
pb.preview(
    nycflights,
    columns_subset=["hour", "minute", "sched_dep_time", "year", "month", "day"]
)
```

POLARS | ROWS 336,776 | COLUMNS 18

	hour *Int64*	minute *Int64*	sched_dep_time *Int64*	year *Int64*	month *Int64*	day *Int64*
1	5	15	515	2013	1	1
2	5	29	529	2013	1	1
3	5	40	540	2013	1	1
4	5	45	545	2013	1	1
5	6	0	600	2013	1	1
336772	14	55	1455	2013	9	30
336773	22	0	2200	2013	9	30
336774	12	10	1210	2013	9	30
336775	11	59	1159	2013	9	30
336776	8	40	840	2013	9	30

What we see are the six columns we specified from the `nycflights` dataset.

Note that the columns are displayed in the order provided in the `columns_subset=` list. This can be useful for making quick, side-by-side comparisons. In the example above, we placed `hour` and `minute` next to the `sched_dep_time` column. In the original dataset, `sched_dep_time` is far apart from the other two columns, but, it's useful to have them next to each other in the preview since `hour` and `minute` are derived from `sched_dep_time` (and this lets us spot check any issues).

We can also use column selectors within `columns_subset=`. Suppose we want to only see those columns that have `"dep_"` or `"arr_"` in the name. To do that, we use the `matches()` column selector function:

```
pb.preview(nycflights, columns_subset=pb.matches("dep_|arr_"))
```

POLARS | ROWS 336,776 | COLUMNS 18

	dep_time *Int64*	sched_dep_time *Int64*	dep_delay *Int64*	arr_time *Int64*	sched_arr_time *Int64*	arr_delay *Int64*
1	517	515	2	830	819	11
2	533	529	4	850	830	20
3	542	540	2	923	850	33
4	544	545	-1	1004	1022	-18
5	554	600	-6	812	837	-25
336772	None	1455	None	None	1634	None
336773	None	2200	None	None	2312	None
336774	None	1210	None	None	1330	None
336775	None	1159	None	None	1344	None
336776	None	840	None	None	1020	None

Several selectors can be combined together through use of the `col()` function and operators such as `&` (*and*), `|` (*or*), `-` (*difference*), and `~` (*not*). Let's look at a column selection case where:

- the first three columns are selected
- all columns containing `"dep_"` or `"arr_"` are selected
- any columns beginning with `"sched"` are omitted

This is how we put that together within `col()`:

```
pb.preview(
    nycflights,
    columns_subset=pb.col((pb.first_n(3) | pb.matches("dep_|arr_")) & ~ pb.starts_with("sched"))
)
```

POLARS | ROWS 336,776 | COLUMNS 18

	year *Int64*	month *Int64*	day *Int64*	dep_time *Int64*	dep_delay *Int64*	arr_time *Int64*	arr_delay *Int64*
1	2013	1	1	517	2	830	11
2	2013	1	1	533	4	850	20
3	2013	1	1	542	2	923	33
4	2013	1	1	544	-1	1004	-18
5	2013	1	1	554	-6	812	-25
336772	2013	9	30	None	None	None	None
336773	2013	9	30	None	None	None	None
336774	2013	9	30	None	None	None	None
336775	2013	9	30	None	None	None	None
336776	2013	9	30	None	None	None	None

This gives us a preview with only the columns that fit the specific selection rules. Incidentally, using selectors with a dataset through `preview()` is a good way to test out the use of selectors more generally. Since they are primarily used to select columns for validation, trying them beforehand with `preview()` can help verify that your selection logic is sound.

While previewing a table with `preview()` is undoubtedly a good thing to do, sometimes you need more. This is where summarizing a table comes in. When you view a summary of a table, the column-by-column info can quickly increase your understanding of a dataset. Plus, it allows you to quickly catch anomalies in your data (e.g., the maximum value of a column could be far outside the realm of possibility).

Pointblank provides a function to make it extremely easy to view column-level summaries in a single table. That function is called `col_summary_tbl()` and, just like `preview()` does, it supports the use of any table that Pointblank can use for validation. And no matter what the input data is, the resultant reporting table is consistent in its design and construction.

5.4 Trying out `col_summary_tbl()`

The function only requires a table. Let's use the `small_table` dataset (a very simple table) to start us off:

```
import pointblank as pb

small_table = pb.load_dataset(dataset="small_table", tbl_type="polars")

pb.col_summary_tbl(small_table)
```

POLARS | ROWS 13 | COLUMNS 8

	Column	NA	UQ	Mean	SD	Min	P_5	Q_1	Med	Q_3	P_{95}	Max	IQ
D	date_time Datetime(time_unit='us', time_zone=None)	0 0	12 0.92	-	-	2016-01-04 00:32:00	-	-	-	-	-	2016-01-30 11:23:00	
D	date Date	0 0	11 0.85	-	-	2016-01-04	-	-	-	-	-	2016-01-30	
N	a Int64	0 0	7 0.54	3.77	2.09	1	1.06	2	3	4	7.4	8	
S	b String	0 0	12 0.92	9	0	9	9	9	9	9	9	9	
N	c Int64	2 0.15	7 0.54	5.73	2.72	2	2.05	3	7	8	9	9	
N	d Float64	0 0	12 0.92	2,304.7	2,631.36	108.34	118.88	837.93	1,035.64	3,291.03	6,335.44	9999.99	2,45
T/F	e Boolean	0 0	T0.62 F0.38	-	-	-	-	-	-	-	-	-	
S	f String	0 0	3 0.23	3.46	0.52	3	3	3	3	4	4	4	

String columns statistics regard the string's length.

The header provides the type of table we're looking at (`POLARS`, since this is a Polars DataFrame) and the table dimensions. The rest of the table focuses on the column-level summaries. As such, each row represents a summary of a column in the `small_table` dataset. There's a

lot of information in this summary table to digest. Some of it is intuitive since this sort of table summarization isn't all that uncommon, but other aspects of it could also give some pause. So we'll carefully wade through how to interpret this report.

5.5 Data Categories in the Column Summary Table

On the left side of the table are icons of different colors. These represent categories that the columns fall into. There are only five categories and columns can only be of one type. The categories (and their letter marks) are:

- `N`: numeric
- `S`: string-based
- `D`: date/datetime
- `T/F`: boolean
- `O`: object

The numeric category (`N`) takes data types such as floats and integers. The `S` category is for string-based columns. Date or datetime values are lumped into the `D` category. Boolean columns (`T/F`) have their own category and are *not* considered numeric (e.g., `0`/`1`). The `O` category is a catchall for all other types of columns. Given the disparity of these categories and that we want them in the same table, some statistical measures will be sensible for certain column categories but not for others. Given that, we'll explain how each category is represented in the column summary table.

5.6 Numeric Data

Three columns in `small_table` are numeric: `a` (`Int64`), `c` (`Int64`), and `d` (`Float64`). The common measures of the missing count/proportion (`NA`) and the unique value count/proportion (`UQ`) are provided for the numeric data type. For these two measures, the top number is the absolute count of missing values and the count of unique values. The bottom number is a proportion of the absolute count divided by the row count; this makes each proportion a value between `0` and `1` (bounds included).

The next two columns represent the mean (`Mean`) and the standard deviation (`SD`). The minumum (`Min`), maximum, (`Max`) and a set of quantiles occupy the next few columns (includes `P5`, `Q1`, `Med` for median, `Q3`, and `P95`). Finally, the interquartile range (`IQR`: `Q3 - Q1`) is the last measure provided.

5.7 String Data

String data is present in `small_table`, being in columns `b` and `f`. The missing value (`NA`) and uniqueness (`UQ`) measures are accounted for here. The statistical measures are all based on string lengths, so what happens is that all strings in a column are converted to those numeric values and a subset of stats values is presented. To avoid some understandable confusion when reading the table, the stats values in each of the cells with values are annotated with the text `"SL"`. It makes less sense to provide a full suite of quantile values so only the minimum (`Min`), median (`Med`), and maximum (`Max`) are provided.

5.8 Date/Datetime Data and Boolean Data

We see that in the first two rows of our summary table there are summaries of the `date_time` and `date` columns. The summaries we provide for a date/datetime category (notice the green `D` to the left of the column names) are:

1. the missing count/proportion (`NA`)
2. the unique value count/proportion (`UQ`)
3. the minimum and maximum dates/datetimes

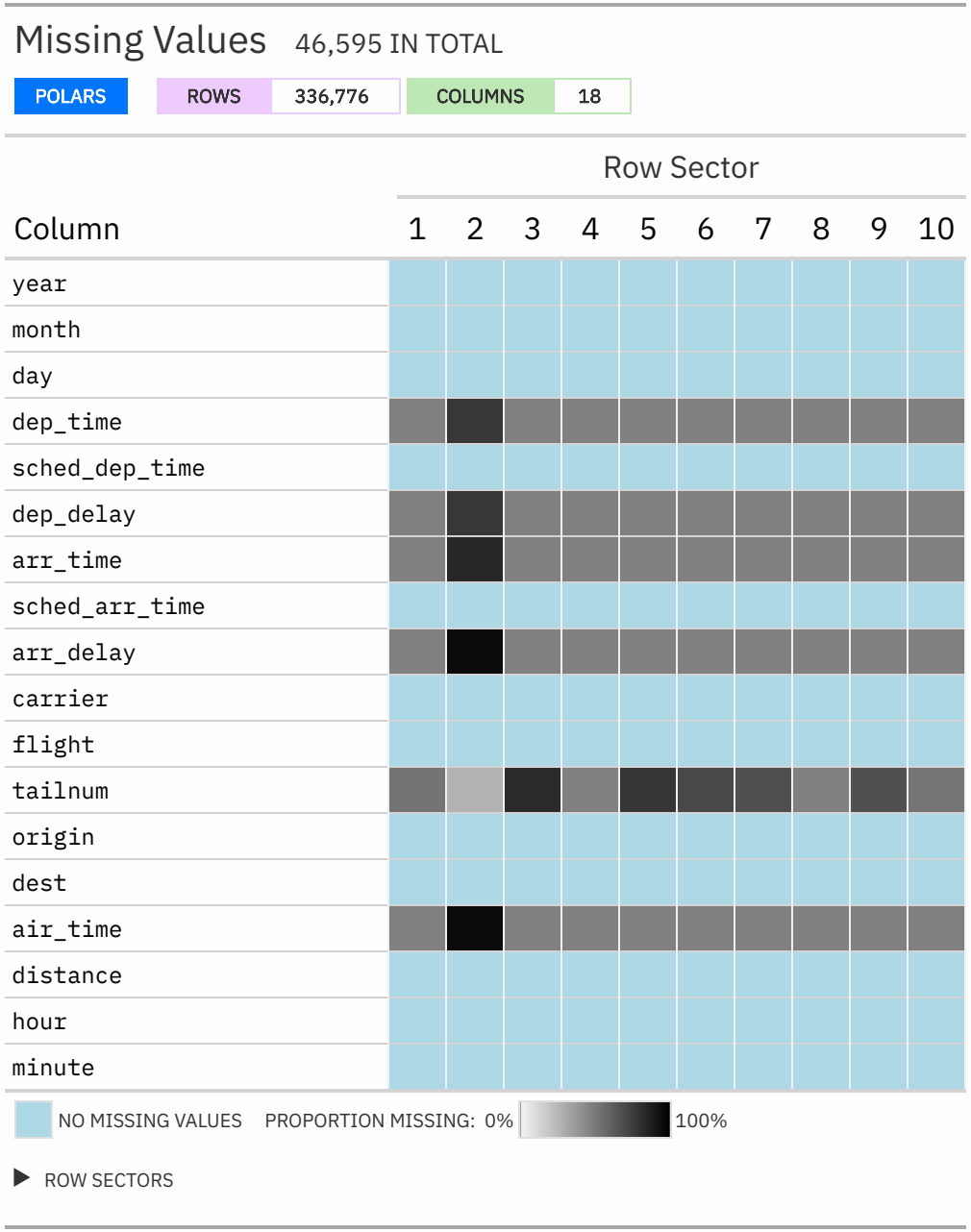
One column, `e`, is of the `Boolean` type. Because columns of this type could only have `True`, `False`, or missing values, we provide summary data for missingness (under `NA`) and proportions of `True` and `False` values (under `UQ`).

Sometimes values just aren't there: they're missing. This can either be expected or another thing to worry about. Either way, we can dig a little deeper if need be and use the `missing_vals_tbl()` function to generate a summary table that can elucidate how many values are missing, and roughly where.

5.9 Using and Understanding `missing_vals_tbl()`

The missing values table is arranged a lot like the column summary table (generated via the `col_summary_tbl()` function) in that columns of the input table are arranged as rows in the reporting table. Let's use `missing_vals_tbl()` on the `nycflights` dataset, which has a lot of missing values:

```
import pointblank as pb

nycflights = pb.load_dataset(dataset="nycflights", tbl_type="polars")

pb.missing_vals_tbl(nycflights)
```



There are 18 columns in `nycflights` and they're arranged down the missing values table as rows. To the right we see column headers indicating 10 columns that are row sectors. Row sectors are groups of rows and each sector contains a tenth of the total rows in the table.

The leftmost sectors are the rows at the top of the table whereas the sectors on the right are closer to the bottom. If you'd like to know which rows make up each row sector, there are details on this in the table footer area (click the `ROW SECTORS` text or the disclosure triangle).

Now that we know about row sectors, we need to understand the visuals here. A light blue cell indicates there are no (`0`) missing values within a given row sector of a column. For `nycflights` we can see that several columns have no missing values at all (i.e., the light blue color makes up the entire row in the missing values table).

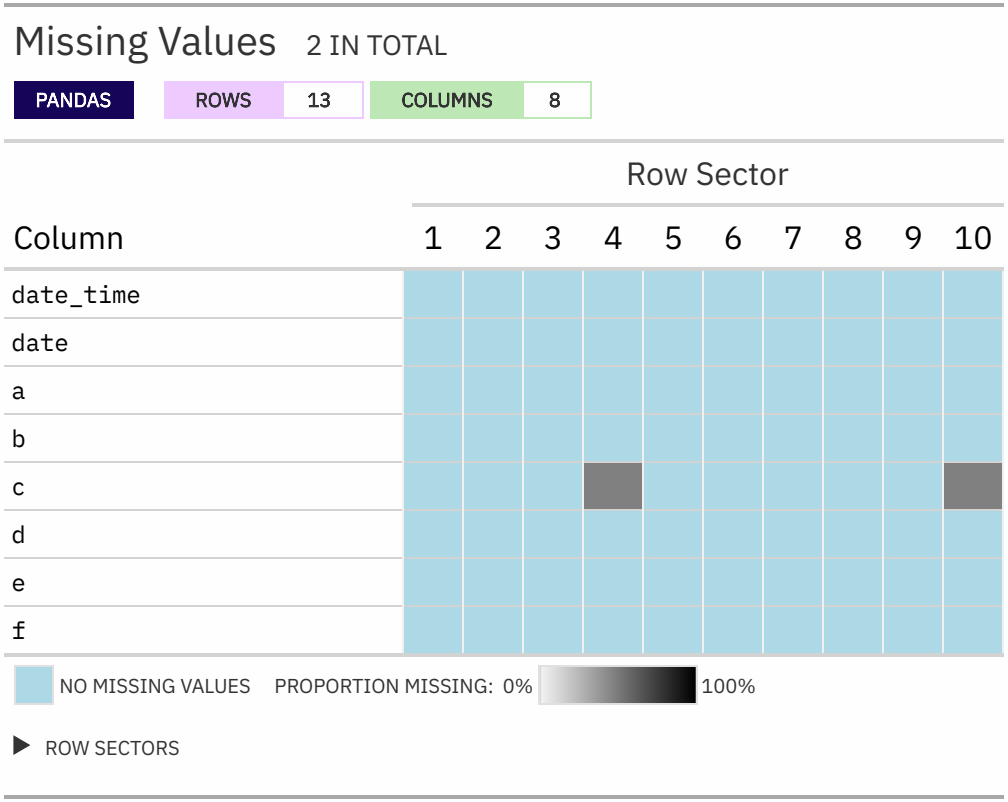
When there are missing values in a column's row sector, you'll be met with a grayscale color. The proportion of missing values corresponds to the color ramp from light gray to solid black. Interestingly, most of the columns that have missing values appear to be related to each other in terms of the extent of missing values (i.e., the appearance in the reporting table looks roughly the same, indicating a sort of systematic missingness). These columns are `dep_time`, `dep_delay`, `arr_time`, `arr_delay`, and `air_time`.

The odd column out with regard to the distribution of missing values is `tailnum`. By scanning the row and observing that the grayscale color values are all a little different we see that the degree of missingness of more variable and not related to the other columns containing missing values.

5.10 Missing Value Tables from the Other Datasets

The `small_table` dataset has only 13 rows to it. Let's use that as a Pandas DataFrame with `missing_vals_tbl()`:

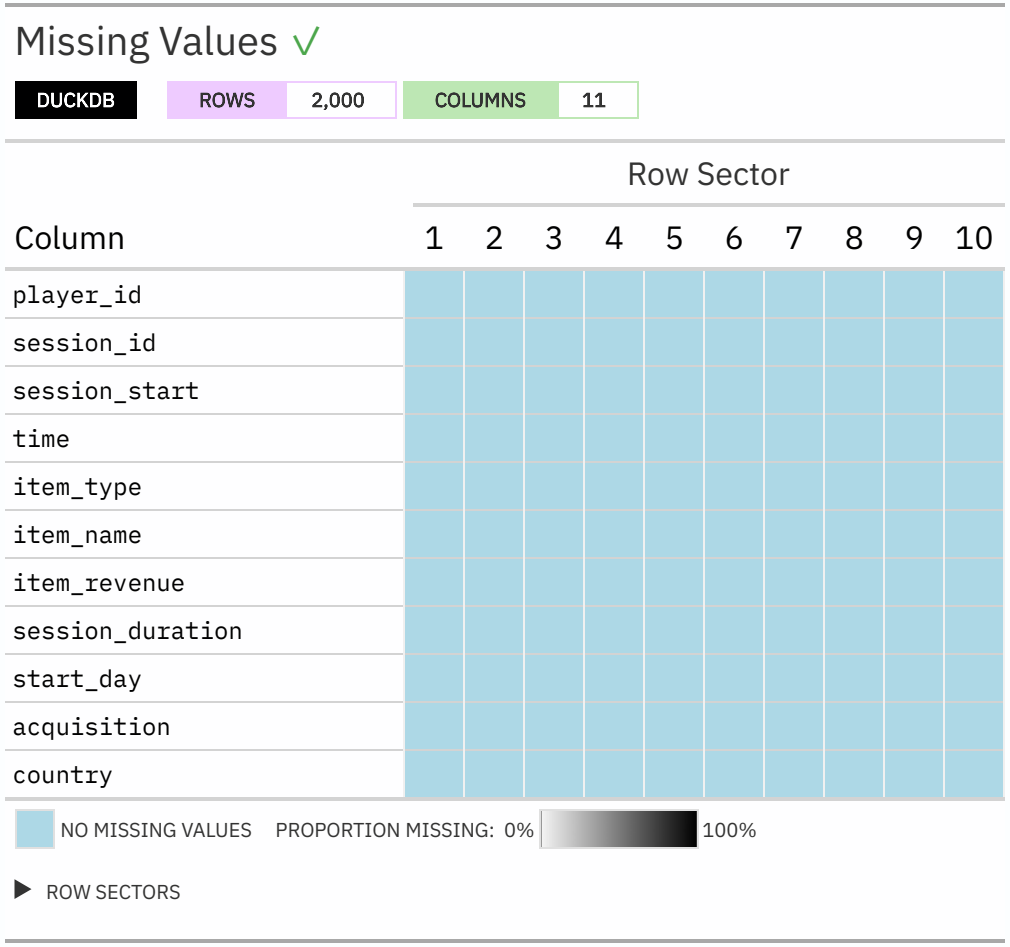
```
small_table = pb.load_dataset(dataset="small_table", tbl_type="pandas")

pb.missing_vals_tbl(small_table)
```



It appears that only column `c` has missing values. And since the table is very small in terms of row count, most of the row sectors contain only a single row.

The `game_revenue` dataset has *no* missing values. And this can be easily proven by using `missing_vals_tbl()` with it:

```
game_revenue = pb.load_dataset(dataset="game_revenue", tbl_type="duckdb")

pb.missing_vals_tbl(game_revenue)
```



We see nothing but light blue in this report! The header also indicates that there are no missing values by displaying a large green check mark (the other report tables provided a count of total missing values across all columns).

6 The Pointblank CLI

Pointblank's CLI (`pb`) makes it easy to view your data before running validations. It has several commands that are exceedingly useful for understanding your data's structure, checking for obvious issues, and confirming that your data source is being read correctly. We also make it easy to explore data in various formats and locations. Let's go through each of the commands for inspecting and exploring data.

6.1 `pb info`: Inspecting the Data Structure

Use `pb info` to display basic information about your data source. Here's how this works with a local CSV file:

```
pb info worldcities.csv
```

This command shows the (1) table type (e.g., `pandas`, `polars`, etc.), (2) the number of rows and columns, and (3) the data source path or identifier.

That example used a local CSV file. The same file is also present in Pointblank's GitHub repository (in the `data-raw` directory) and the CLI is able to load the data from there as well:

```
pb info https://github.com/posit-dev/pointblank/blob/main/data_raw/worldcities.csv
```

The `pb info` command is useful before running validations to confirm your data source's dimensions, and, whether it can even be loaded.

You can inspect a wide variety of data sources using the CLI! Here are some examples with `pb info`:

```
pb info small_table         # built in dataset
pb info worldcities.csv     # single CSV file
pb info meteo.parquet       # single Parquet file
pb info "*.parquet"         # several Parquet files
pb info "data/*.parquet"    # partitioned Parquet files
pb info "duckdb:///warehouse/analytics.ddb::customer_metrics" # DB table via connection string
pb info https://github.com/posit-dev/pointblank/blob/main/data_raw/global_sales.csv # GitHub URL
```

And these input schemes work with all other commands that accept a `DATA_SOURCE`.

6.2 `pb preview`: Previewing Data

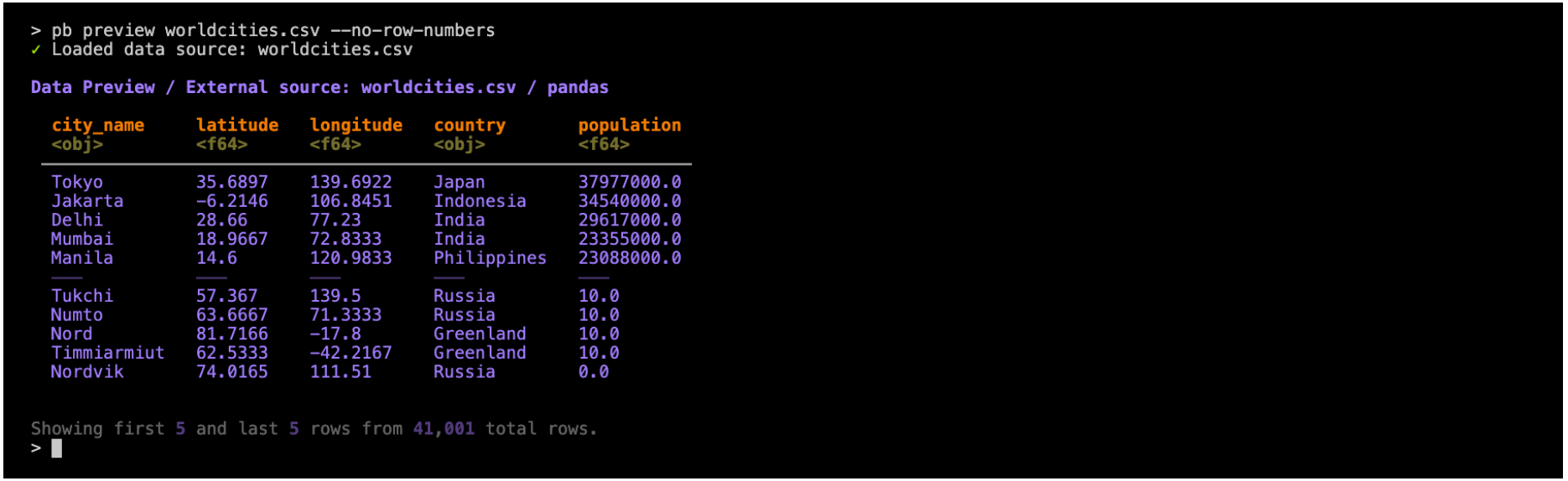
Use `pb preview` to view the first and last rows of your data. Let's try it out with the `worldcities.csv` file:

```
pb preview worldcities.csv
```

As can be seen, `pb preview` gives you a preview of the dataset as a table in the console. The dataset has 41K rows but we're electing to show only five rows from the head and from the tail.

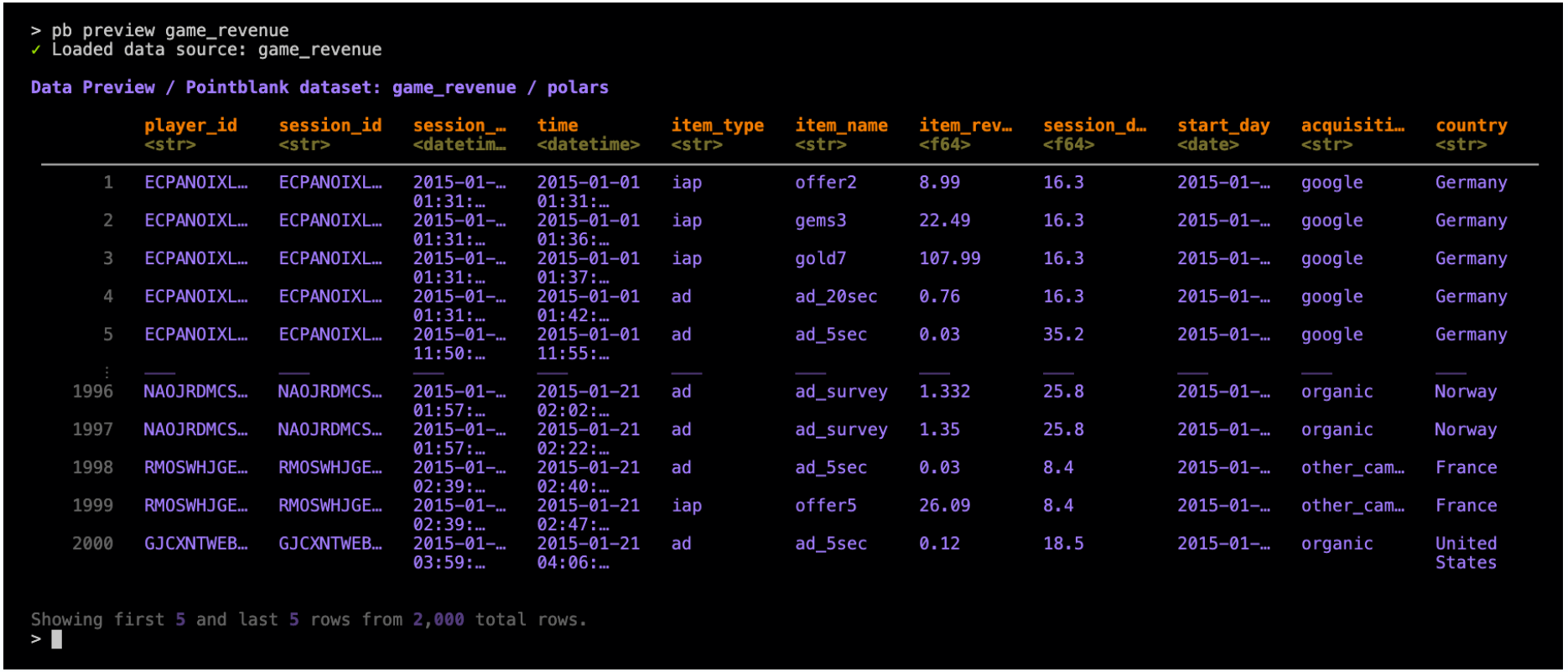
Let's go over some features of the table preview. First off, the table header provides information on the data source and the DataFrame library that handled the reading of the CSV. Below the column names are simplified representations of the data types (e.g., `<obj>` for `object`, `<f64>` for `Float64`). We provide row numbers (in gray) in the table stub to indicate which of the rows are from the head or the tail (and a divider helps to distinguish these row groups). If you'd prefer to eliminate the row numbers, use the `--no-row-numbers` option:

```
pb preview worldcities.csv --no-row-numbers
```



While `pb preview` purposefully displays only a few rows, the number of columns shown can be more than you might need. Furthermore, if a table has *a lot* of columns, you'll only see some of the first and some of the last columns. This is where column selection becomes useful and there are a few methods available for subsetting the preview table's columns. A good one (provided you know the column names) is to use the `--columns` option along with a comma-delimted set of column names. Let's look at a preview of the included `game_revenue` dataset before subsetting the columns:

```
pb preview game_revenue
```



```
> pb preview game_revenue
✓ Loaded data source: game_revenue

Data Preview / Pointblank dataset: game_revenue / polars
```

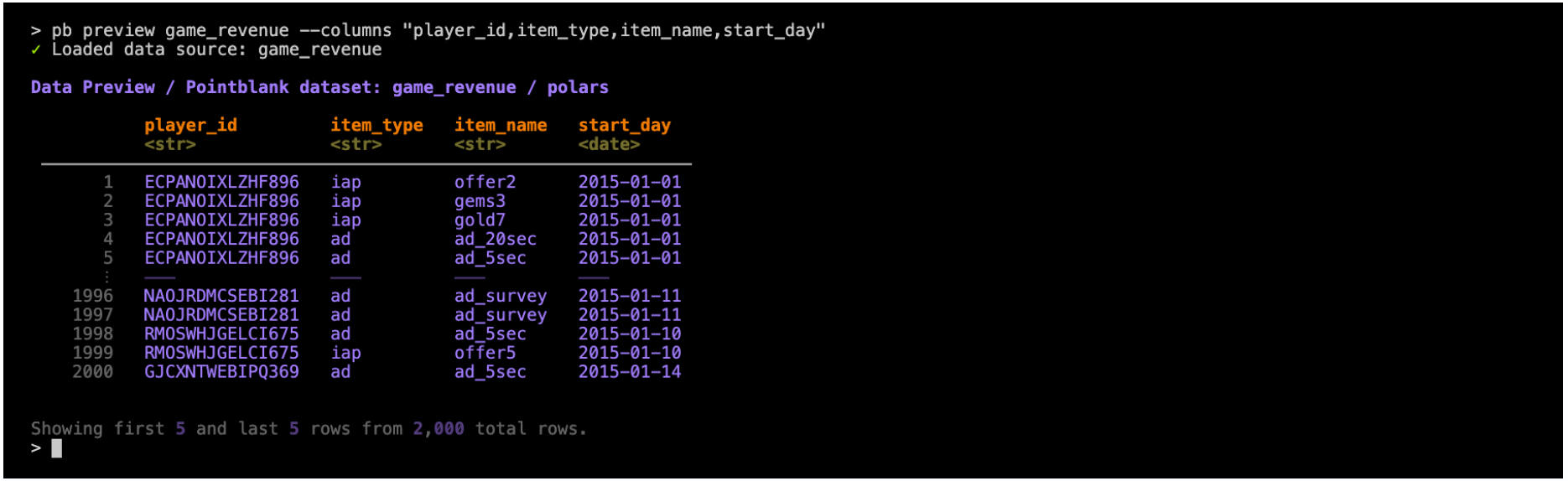
	player_id <str>	session_id <str>	session_… <datetim…	time <datetime>	item_type <str>	item_name <str>	item_rev… <f64>	session_d… <f64>	start_day <date>	acquisiti… <str>	country <str>
1	ECPANOIXL…	ECPANOIXL…	2015-01-… 01:31:…	2015-01-01 01:31:…	iap	offer2	8.99	16.3	2015-01-…	google	Germany
2	ECPANOIXL…	ECPANOIXL…	2015-01-… 01:31:…	2015-01-01 01:36:…	iap	gems3	22.49	16.3	2015-01-…	google	Germany
3	ECPANOIXL…	ECPANOIXL…	2015-01-… 01:31:…	2015-01-01 01:37:…	iap	gold7	107.99	16.3	2015-01-…	google	Germany
4	ECPANOIXL…	ECPANOIXL…	2015-01-… 01:31:…	2015-01-01 01:42:…	ad	ad_20sec	0.76	16.3	2015-01-…	google	Germany
5	ECPANOIXL…	ECPANOIXL…	2015-01-… 11:50:…	2015-01-01 11:55:…	ad	ad_5sec	0.03	35.2	2015-01-…	google	Germany
⋮	…	…	…	…	…	…	…	…	…	…	…
1996	NAOJRDMCS…	NAOJRDMCS…	2015-01-… 01:57:…	2015-01-21 02:02:…	ad	ad_survey	1.332	25.8	2015-01-…	organic	Norway
1997	NAOJRDMCS…	NAOJRDMCS…	2015-01-… 01:57:…	2015-01-21 02:22:…	ad	ad_survey	1.35	25.8	2015-01-…	organic	Norway
1998	RMOSWHJGE…	RMOSWHJGE…	2015-01-… 02:39:…	2015-01-21 02:40:…	ad	ad_5sec	0.03	8.4	2015-01-…	other_cam…	France
1999	RMOSWHJGE…	RMOSWHJGE…	2015-01-… 02:39:…	2015-01-21 02:47:…	iap	offer5	26.09	8.4	2015-01-…	other_cam…	France
2000	GJCXNTWEB…	GJCXNTWEB…	2015-01-… 03:59:…	2015-01-21 04:06:…	ad	ad_5sec	0.12	18.5	2015-01-…	organic	United States

```
Showing first 5 and last 5 rows from 2,000 total rows.
>
```

That's 11 columns in total and while the all columns *are* shown (i.e., none in the middle are truncated from view), we start to see some necessary instances of abbreviating via `…` within the column names and in the displayed values.

Let's now use the `--columns` with a set of column names:

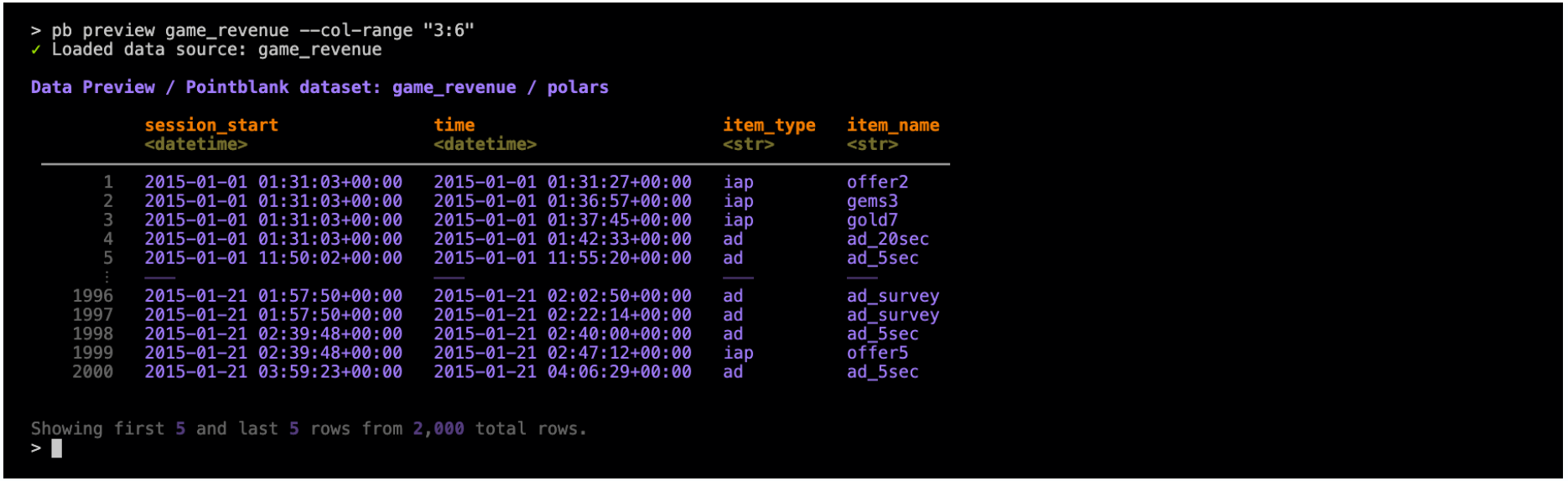
```
pb preview game_revenue -columns "player_id, item_type, item_name, start_day"
```



With that, the few columns that are displayed no longer have to abbreviate their data values. This is an important consideration since a selective display of column becomes more necessary if column content is large or if the width of the terminal (in terms of characters) cannot be increased.

You may want to view ranges of columns by their indices. This is convenient when you want to get a closer look at a few side-by-side columns and you don't want to bother with getting the set of column names exactly right (i.e., for quick inspection). For this, we need to use the `--col-range` option with the desired left/right column bounds separated by a colon:

```
pb preview game_revenue --col-range "3:6"
```



In the case that you want to save a table preview as an HTML table in a standalone file, you can add in the `--output-html` option (just add a path/filename with an .html extension).

And there are many more options that allow for quick iteration while previewing a table. Use `pb preview --help` to get a helpful listing.

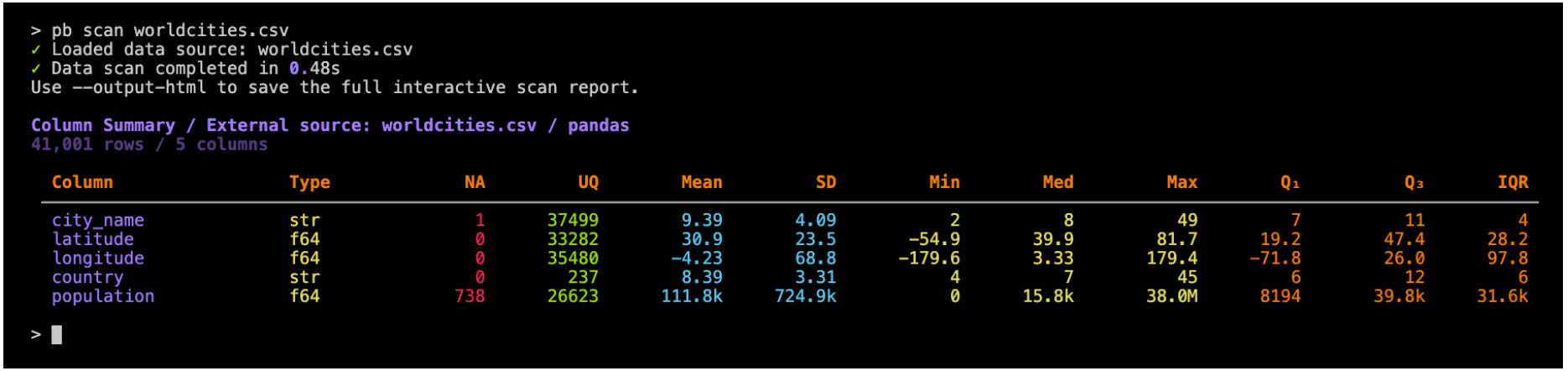
6.3 `pb scan`: Getting Column Summaries

We can use `pb scan` for fairly comprehensive summaries of column data, including:

- data types
- missing value counts
- unique value counts
- summary statistics (mean, standard deviation, min, max, quartiles, and the interquartile range)

Let's use this on the `worldcities.csv` dataset:

```
pb scan worldcities.csv
```



Each row in the summary table represents a column in the input dataset. Just as in `pb preview` we get simplified dtypes (in the `Type` column). The `NA` and `UQ` indicate how many missing and unique values are in the column. The remaining columns are statistical measures and there's an important thing to note here: the values provided for any string-based columns (here, `city_name` and `country`) are derived from string lengths.

When using `pb scan`, it's helpful to know that large numbers in the summary table are automatically abbreviated for readability, so you'll see values like `39.8k` or `38.0M` instead of long numbers that would require many more characters. For the best experience, try to use a terminal window that's at least 150 characters wide. This will help ensure that all column values are fully visible and not adversely abbreviated by the underlying table mechanism.

If your table has many columns, that's not much of a problem for the reporting! Each column is represented as a row in the report, so you'll simply see more lines in the output (and you could always limit the number of columns reported).

There are two options for `pb scan`:

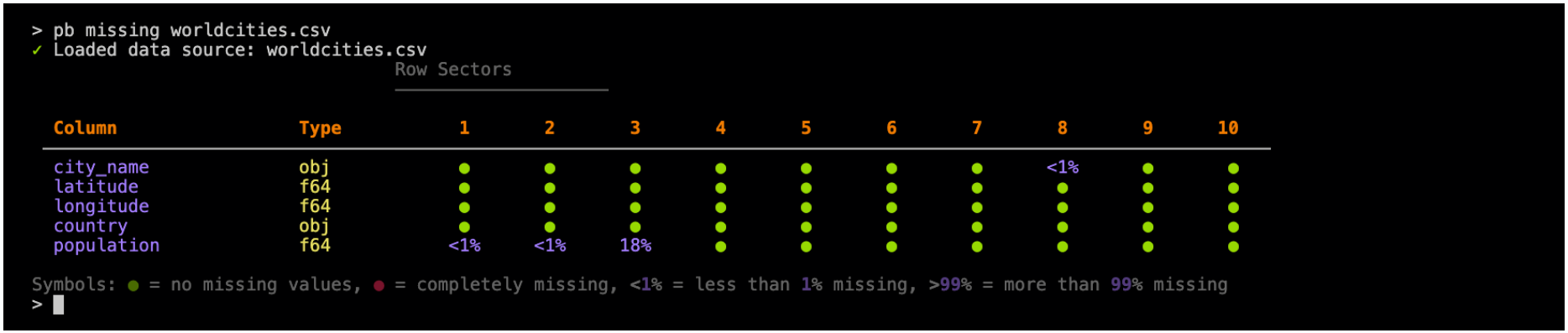
- `--columns "col1,col2"`: scan only specified columns
- `--output-html "file.html"`: save scan as an HTML file

Both of these options are also in the `pb preview` command and they behave the same way here.

6.4 `pb missing`: Reporting on Missing Values

Use `pb missing` to generate a missing values report, visualizing missingness across columns and 10 *row sectors*. Here's an example using `worldcities.csv`:

```
pb missing worldcities.csv
```



This report is arranged similarly to that of `pb scan`, where each column in the input table gets a row in this report table. Each of the 10 row sectors represents 1/10 of the rows in the dataset, where sector `1` encompasses the head of the table, and `10` the tail.

More often than not, we expect few missing values so a filled green circle signifies that the collection of rows in a sector (for a column) has no missing values. We don't see any red circles in the `worldcities.csv`-based example but, if we did, that would mean that sectors for a given column are entirely filled with missing values.

What's in between the no-missing and completely-missing cases are percentages of missing values. For instance, we can see that row sector `3` of the `population` column has 18% missing values (which is very odd for a table with the sole purpose of providing population values).

We also have cases where we see <1% of values in a row sector missing. The reporting of `pb missing` is very careful not to 'round down' in cases where there could be very few missing values (or even just one) in a large table.

Seeing this type of missing value report can be really important! You might not expect *any* missing values but finding them will inform decisions on whether to institute checks for them. Another case is that missing values will pop up in specific sectors, indicating a change in how data is processed and appended to the table.

By way of options, there's only one for `pb missing` and it is `--output-html`. With that (as in the previous two commands discussed), we can write the missing values report to a standalone HTML file.

6.5 Wrapping Up

Pointblank's CLI provides a set of commands that make it easy to inspect, understand, and diagnose your data before you move on to validation or analysis. Using these tools can help you catch issues early and gain confidence in your data sources.

- use `pb info` and before running validations to confirm your data source can be loaded
- use `pb preview` to quickly understand what the data looks like
- use `pb scan` for a quick data profile and to spot outliers or data quality issues
- use `pb missing` to visualize and diagnose missing data patterns

By incorporating these commands into your workflow, you'll be better equipped to work efficiently with your data (and avoid surprises down the line).

Validating data directly in the terminal with the Pointblank CLI offers a fast, scriptable, and repeatable way to check your data. This approach is especially useful for quick checks, CI/CD pipelines, and automation workflows, where you want immediate feedback and clear pass/fail results.

The CLI commands are designed for efficiency: you can run validations with a single line, integrate them easily into shell scripts or data pipelines, and benefit from clear, color-coded output that's easy to interpret at a glance.

The `pb validate` command lets you perform common validation checks directly on your data source with a simple command-line interface. This works well both for quick, one-off checks and for use in automated pipelines.

For more complex validation logic, the `pb run` command serves as a runner for validation scripts written with the Pointblank Python API, allowing you to execute custom validation workflows from the command line.

6.6 `pb validate`: Quick, One-Line Data Checks

The `pb validate` command is your go-to for running common validation checks directly on your data source. It's perfect for quick, one-off checks or for use in automated pipelines. You specify exactly which check you want to run using the `--check` option, making your

intent clear and your validation explicit.

Here's how you construct a validation command:

```
pb validate worldcities.csv --check <check-name> [other options]
```

You always provide the data source first, then specify one or more checks with `--check`. Each check can have its own options, such as `--column` or `--value`, depending on what you want to validate.

6.6.1 Checking for Duplicate and Complete Rows

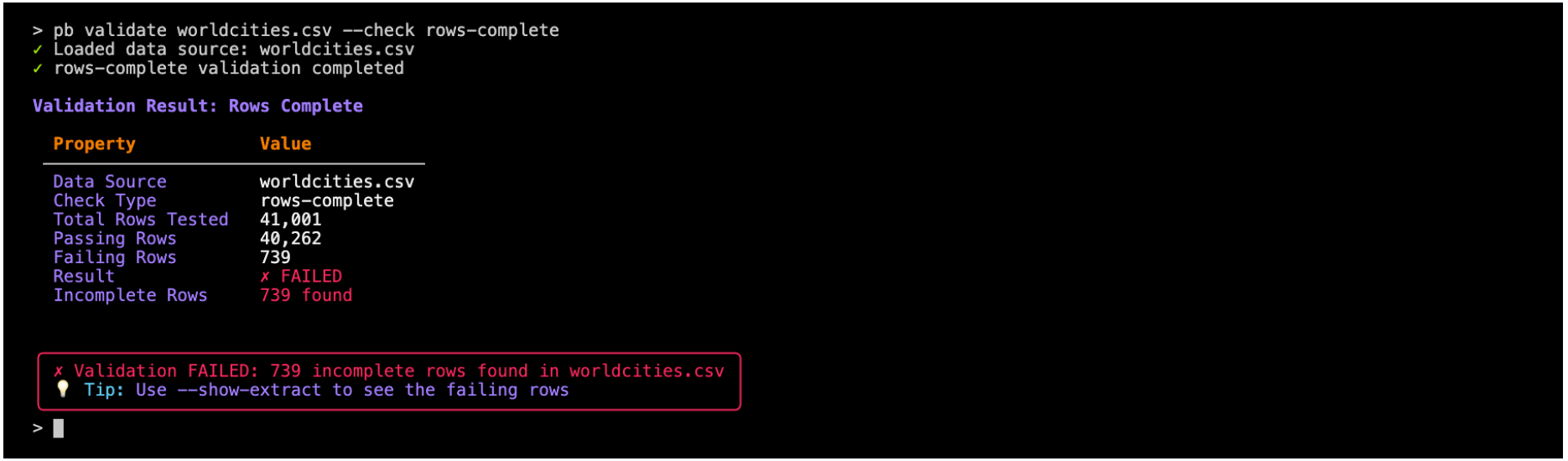
To check for duplicate rows, use the `rows-distinct` check:

```
pb validate worldcities.csv --check rows-distinct
```

The output shows you whether your data contains any duplicate rows, how many rows were checked, and if any duplicates were found. The color-coding of the results helps you quickly interpret the results, using green for pass and red for fail. Here, no duplicate rows were detected out of the 41K rows checked.

To check that every row is complete (i.e., no missing values in any column), use the `rows-complete` check:

```
pb validate worldcities.csv --check rows-complete
```

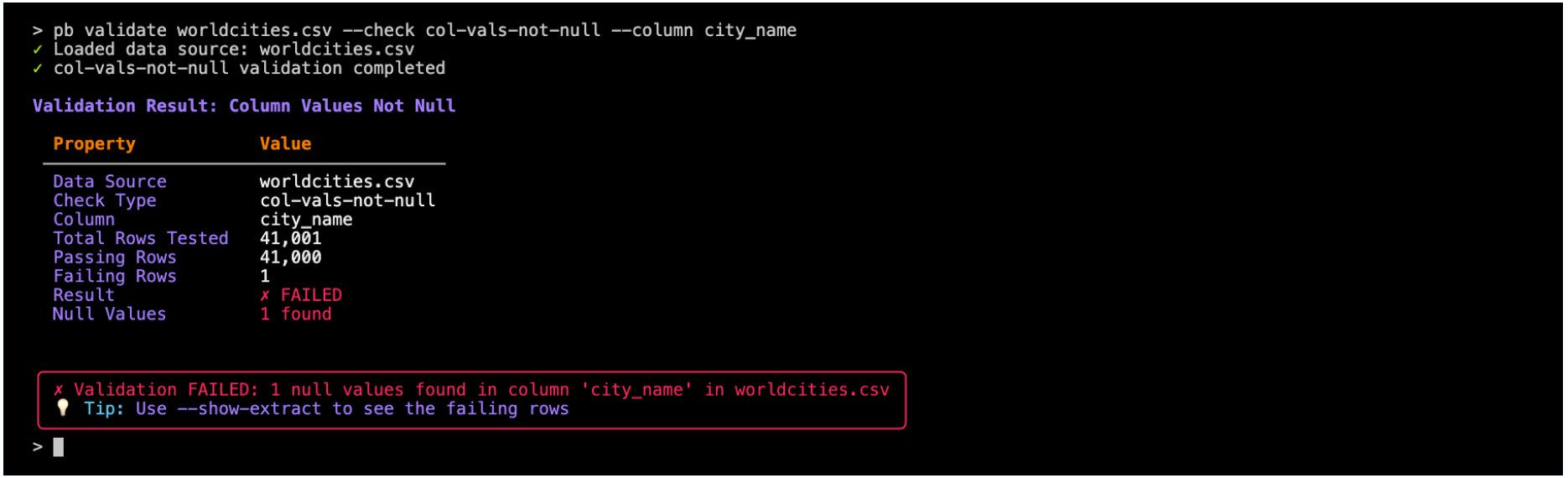


With this check we see that the `worldcities.csv` dataset has 739 rows containing at least one Null/missing value. And with any dataset, it's easy to quickly spot if there are any rows with missing data using this command.

6.6.2 Checking for Nulls and Value Ranges

You can easily check for missing values in a column, or ensure that values fall within a certain range. Here's how to check that all values in the `population` column are not null:

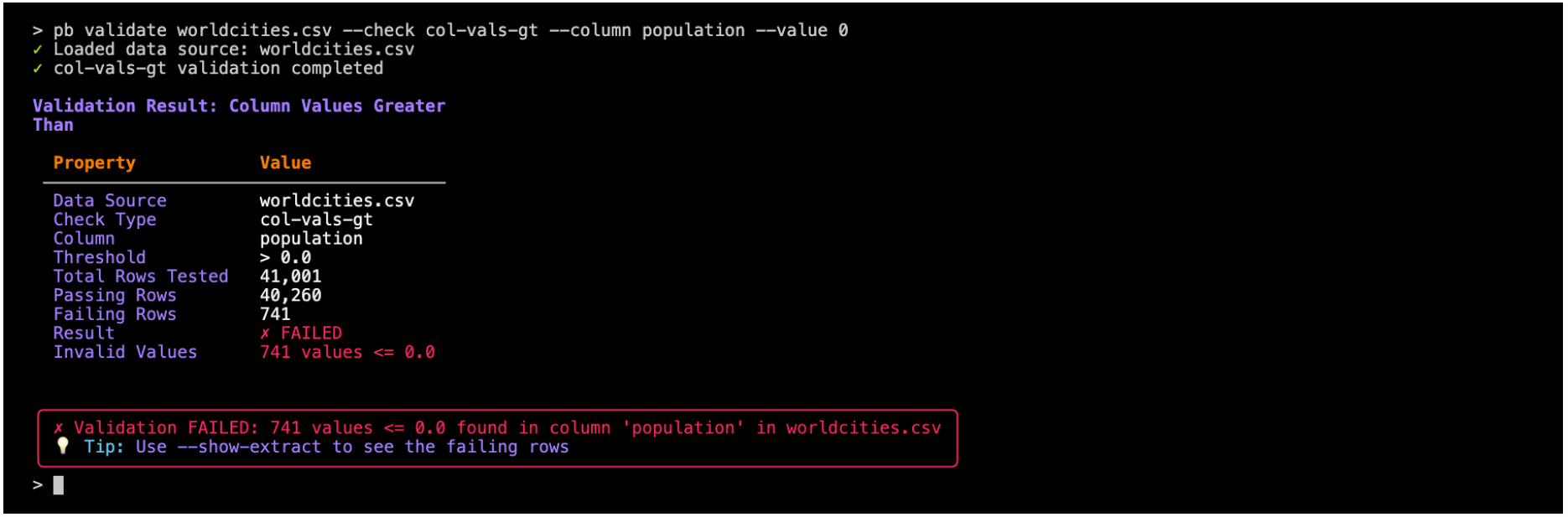
```
pb validate worldcities.csv --check col-vals-not-null --column city_name
```



Perhaps surprisingly, we find that one row has a missing city name.

Let's now check whether all values in the `population` column are greater than zero:

```
pb validate worldcities.csv --check col-vals-gt --column population --value 0
```



With that we find that there are 741 rows where the `population` value is not greater than 0 (note that this check also fails when cells are null or missing).

6.6.3 Multiple Checks in One Command

You can chain several checks together in a single command. This is handy for comprehensive data quality checks:

```
pb validate worldcities.csv --check rows-distinct --check col-vals-not-null --column city_name --check col-vals-gt --column population --value 0
```

```
> pb validate worldcities.csv --check rows-distinct --check col-vals-not-null --column city_name --check col-vals-gt --column population --value 0
✓ Loaded data source: worldcities.csv
✓ 3 validations completed


Validation Result (1 of 3): Rows
Distinct

  Property            Value
 ──────────────────────────────────
  Data Source         worldcities.csv
  Check Type          rows-distinct
  Total Rows Tested   41,001
  Passing Rows        41,001
  Failing Rows        0
  Result              ✓ PASSED
  Duplicate Rows      None found

 ✓ Validation PASSED: No duplicate rows found in worldcities.csv


Validation Result (2 of 3): Column Values
Not Null

  Property            Value
 ──────────────────────────────────
  Data Source         worldcities.csv
  Check Type          col-vals-not-null
  Column              city_name
  Total Rows Tested   41,001
  Passing Rows        41,000
  Failing Rows        1
  Result              ✗ FAILED
  Null Values         1 found

 ✗ Validation FAILED: 1 null values found in column 'city_name' in worldcities.csv


Validation Result (3 of 3): Column Values
Greater Than

  Property            Value
 ──────────────────────────────────
  Data Source         worldcities.csv
  Check Type          col-vals-gt
  Column              population
  Threshold           > 0.0
  Total Rows Tested   41,001
  Passing Rows        40,260
  Failing Rows        741
  Result              ✗ FAILED
  Invalid Values      741 values <= 0.0

 ✗ Validation FAILED: 741 values <= 0.0 found in column 'population' in worldcities.csv

💡 Tip: Use --show-extract to see the failing rows
>
```

Each check is shown one after the other in the terminal output, so you can review the result of each validation step individually as the command proceeds.

6.6.4 Seeing and Saving Failing Rows

If a check fails, you might want to see which rows caused the failure. Use the `--show-extract` option to display failing rows right in the terminal:

```
pb validate worldcities.csv --check rows-complete --show-extract
```

```
> pb validate worldcities.csv --check rows-complete --show-extract
✓ Loaded data source: worldcities.csv
✓ rows-complete validation completed

Validation Result: Rows Complete

  Property            Value
 ──────────────────────────────
  Data Source         worldcities.csv
  Check Type          rows-complete
  Total Rows Tested   41,001
  Passing Rows        40,262
  Failing Rows        739
  Result              ✗ FAILED
  Incomplete Rows     739 found


Extract of failing rows (incomplete rows):
Showing first 10 of 500 incomplete rows

          city_name      latitude   longitude   country                    population
          <obj>          <f64>      <f64>       <obj>                      <f64>
 ──────────────────────────────────────────────────────────────────────────────────
     844   Gaza           31.5069    34.456      Gaza Strip                 NaN
     852   Al Quds        31.7764    35.2269     West Bank                  NaN
     854   Basse-Terre    16.0       -61.7167    Guadeloupe                 NaN
     860   Marigot        18.0706    -63.0847    Saint Martin               NaN
     861   Philipsburg    18.0256    -63.0492    Sint Maarten               NaN
     862   Road Town      18.4167    -64.6167    Virgin Islands, British    NaN
     863   Gustavia       17.8958    -62.8508    Saint Barthelemy           NaN
     864   Saint-Pierre   46.7811    -56.1764    Saint Pierre And Miquel…   NaN
     865   The Valley     18.2167    -63.05      Anguilla                   NaN
     866   Mata-Utu       -13.2825   -176.1736   Wallis And Futuna          NaN


 ✗ Validation FAILED: 739 incomplete rows found in worldcities.csv
>
```

Or, save the failing rows to a CSV file for further investigation:

```
pb validate worldcities.csv --check rows-complete --show-extract --write-extract incomplete_failing_rows
```

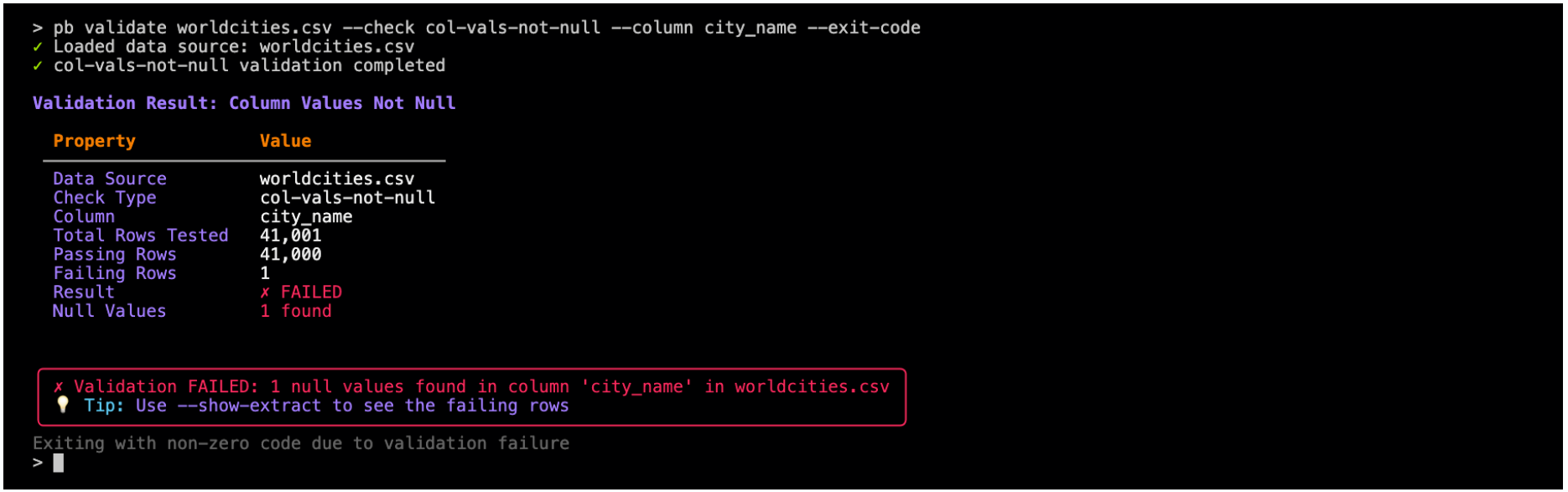
```
> pb validate worldcities.csv --check rows-complete --show-extract --write-extract incomplete_failing_rows
✓ Loaded data source: worldcities.csv
✓ rows-complete validation completed

Validation Result: Rows Complete

  Property            Value
 ──────────────────────────────
  Data Source         worldcities.csv
  Check Type          rows-complete
  Total Rows Tested   41,001
  Passing Rows        40,262
  Failing Rows        739
  Result              ✗ FAILED
  Incomplete Rows     739 found


Extract of failing rows (incomplete rows):
Showing first 10 of 500 incomplete rows

          city_name      latitude   longitude   country                  population
          <obj>          <f64>      <f64>       <obj>                    <f64>
 ──────────────────────────────────────────────────────────────────────────────────
     844   Gaza           31.5069    34.456      Gaza Strip               NaN
     852   Al Quds        31.7764    35.2269     West Bank                NaN
     854   Basse-Terre    16.0       -61.7167    Guadeloupe               NaN
     860   Marigot        18.0706    -63.0847    Saint Martin             NaN
     861   Philipsburg    18.0256    -63.0492    Sint Maarten             NaN
     862   Road Town      18.4167    -64.6167    Virgin Islands, British  NaN
     863   Gustavia       17.8958    -62.8508    Saint Barthelemy         NaN
     864   Saint-Pierre   46.7811    -56.1764    Saint Pierre And Miquel…  NaN
     865   The Valley     18.2167    -63.05      Anguilla                 NaN
     866   Mata-Utu       -13.2825   -176.1736   Wallis And Futuna        NaN

✓ Failing rows saved to folder: incomplete_failing_rows
  – step_01_rows_complete.csv: 10 rows

 ✗ Validation FAILED: 739 incomplete rows found in worldcities.csv
>
```

Note here in the output the additional lines stating that failing rows were saved to a folder (`incomplete_failing_rows`) and, within that folder the `step_01_rows_complete.csv` file was written. Using a folder for extracts is necessary in practice since there may be multiple validations defined in a `pb validate` command.

6.6.5 Advanced Options and CI/CD Integration

- use `--exit-code` to make the command exit with a non-zero code if any check fails; useful for CI/CD pipelines
- use `--limit` to control how many failing rows are shown or saved
- use `--list-checks` to see all available validation checks and their options

```
pb validate worldcities.csv --check col-vals-not-null --column city_name --exit-code
```



6.7 `pb run`: Custom Validation Workflows with Python

For more complex validation logic, use the `pb run` command. This lets you execute a Python script containing Pointblank validation steps, combining the flexibility of the Python API with the convenience of the CLI.

You can always scaffold a template script using the `pb make-template` command:

```
pb make-template my_validation.py
```

```
> pb make-template my_validation.py
✓ Validation script template created: my_validation.py

Edit the template to add your data loading and validation rules, then run:
pb run my_validation.py
pb run my_validation.py --data your_data.csv  # Override data source
>
```

But for our example, we'll elect to make our own `worldcities_validation.py` file from scratch. It will:

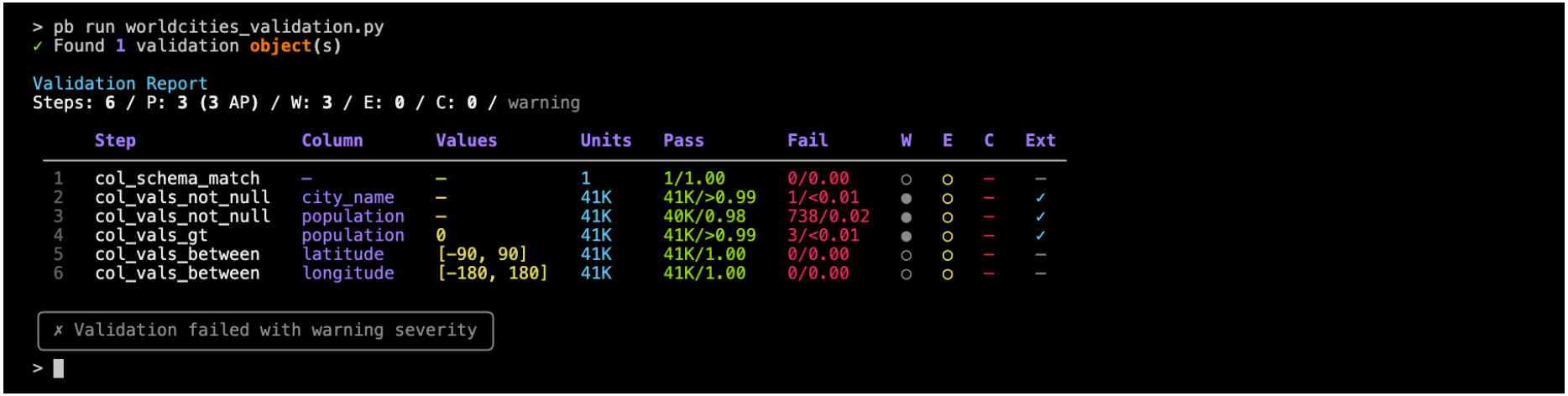
- use the `worldcities.csv` file
- apply two thresholds (one for 'warning', another for 'error')
- have six validation steps

Here's what it looks like:

```
import pointblank as pb

validation = (
    pb.Validate(
        data="worldcities.csv",
        thresholds=pb.Thresholds(
            warning=1,  # 1 failure
            error=0.05,  # 5% of rows failing
        ),
    )
    .col_schema_match(
        schema=pb.Schema(
            columns=[
                ("city_name", "object"),
                ("latitude", "float64"),
                ("longitude", "float64"),
                ("country", "object"),
                ("population", "float64"),
            ]
        ),
    )
    .col_vals_not_null(columns="city_name")
    .col_vals_not_null(columns="population")
    .col_vals_gt(columns="population", value=0, na_pass=True)
    .col_vals_between(columns="latitude", left=-90, right=90)
    .col_vals_between(columns="longitude", left=-180, right=180)
    .interrogate()
)
```

Now, we'll run the .py script from the terminal:

```
pb run worldcities_validation.py
```

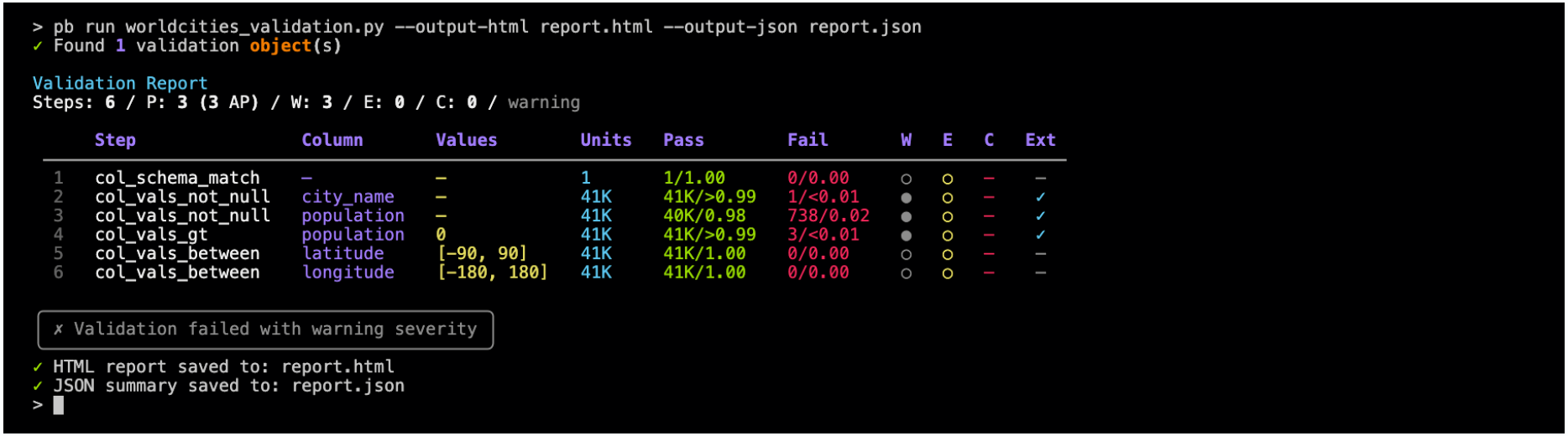


You'll see a summary table that lists all of the steps and their results and you can include as many steps and as much logic as you need.

6.7.1 Output Options

You could save the validation report as HTML or JSON (or both) for the purposes of sharing or for automation:

```
pb run worldcities_validation.py --output-html report.html --output-json report.json
```



There are also the options to produce extracts (subset of failing rows) with `--show-extract` or `--write-extract` (just like with `pb validate`). Let's do both in the following example:

```
pb run worldcities_validation.py --show-extract --write-extract worldcities_failures
```

```
> pb run worldcities_validation.py --show-extract --write-extract worldcities_failures
✓ Found 1 validation object(s)

Validation Report
Steps: 6 / P: 3 (3 AP) / W: 3 / E: 0 / C: 0 / warning
```

	Step	Column	Values	Units	Pass	Fail	W	E	C	Ext
1	col_schema_match	–	–	1	1/1.00	0/0.00	○	○	–	–
2	col_vals_not_null	city_name	–	41K	41K/>0.99	1/<0.01	●	○	–	✓
3	col_vals_not_null	population	–	41K	40K/0.98	738/0.02	●	○	–	✓
4	col_vals_gt	population	0	41K	41K/>0.99	3/<0.01	●	○	–	✓
5	col_vals_between	latitude	[-90, 90]	41K	41K/1.00	0/0.00	○	○	–	–
6	col_vals_between	longitude	[-180, 180]	41K	41K/1.00	0/0.00	○	○	–	–

```
✗ Validation failed with warning severity

Extract of failing rows from validation steps:

Step 2: col_vals_not_null
Showing all 1 failing rows
```

	city_name <obj>	latitude <f64>	longitude <f64>	country <obj>	population <f64>
31835	NaN	44.9333	7.5333	Italy	8015.0

```
Step 3: col_vals_not_null
Showing first 10 of 500 failing rows
```

	city_name <obj>	latitude <f64>	longitude <f64>	country <obj>	population <f64>
844	Gaza	31.5069	34.456	Gaza Strip	NaN
852	Al Quds	31.7764	35.2269	West Bank	NaN
854	Basse-Terre	16.0	-61.7167	Guadeloupe	NaN
860	Marigot	18.0706	-63.0847	Saint Martin	NaN
861	Philipsburg	18.0256	-63.0492	Sint Maarten	NaN
862	Road Town	18.4167	-64.6167	Virgin Islands, British	NaN
863	Gustavia	17.8958	-62.8508	Saint Barthelemy	NaN
864	Saint-Pierre	46.7811	-56.1764	Saint Pierre And Miquel…	NaN
865	The Valley	18.2167	-63.05	Anguilla	NaN
866	Mata-Utu	-13.2825	-176.1736	Wallis And Futuna	NaN

```
Step 4: col_vals_gt
Showing all 3 failing rows
```

	city_name <obj>	latitude <f64>	longitude <f64>	country <obj>	population <f64>
40381	Logashkino	70.8536	153.8744	Russia	0.0
40947	Ağdam	40.9053	45.5564	Azerbaijan	0.0
41001	Nordvik	74.0165	111.51	Russia	0.0

```
✓ Failing rows saved to folder: worldcities_failures
  - step_02_col_vals_not_null.csv: 1 rows
```

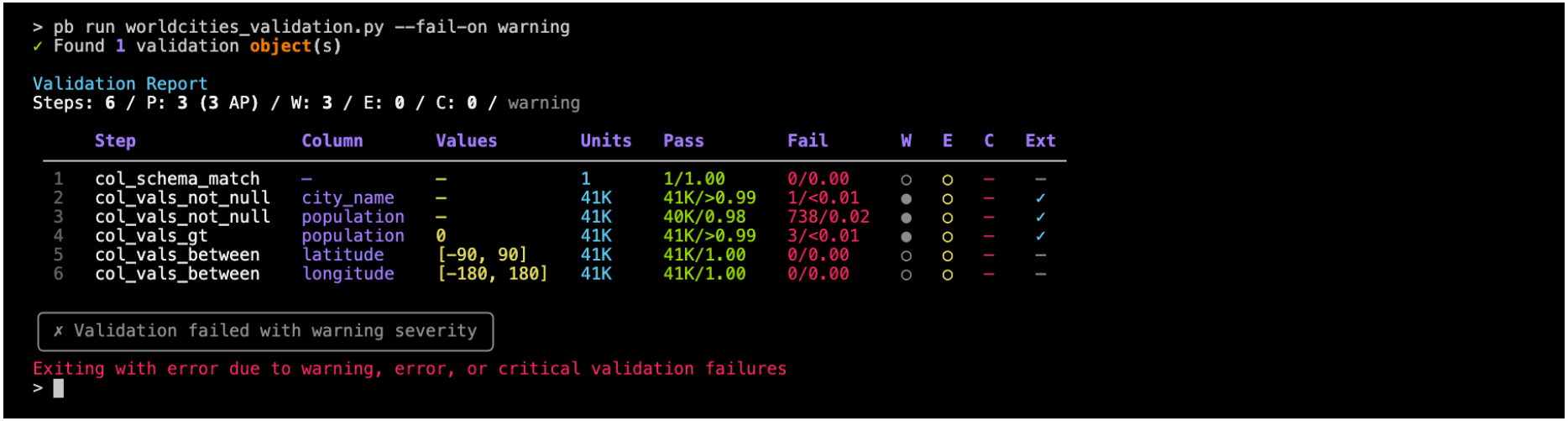
This shows a preview of each extract for those validation steps where extracts were produced (steps 2, 3, and 4). Individual CSV files with extracted rows for those steps were written to the `worldcities_failures` directory.

6.7.2 Controlling Failure Behavior

It's possible to use the `--fail-on` option to control when the command should exit with an error, based on the severity of validation failures. This is especially useful for automated workflows and CI/CD pipelines.

Let's try that with our `worldcities_validation.py` validation, which we've seen exeeds the 'warning' in steps 2, 3, and 4:

```
pb run worldcities_validation.py --fail-on warning
```



Notice the final line states `Exiting with error due to warning, error, or critical validation failures`. Because we applied `--fail-on warning`, any presence of `warning' (or higher levels such as 'error' or 'critical') will yield a non-zero exit code that should stop a pipeline process. We can prove this by running the following lines in the terminal

```
pb run worldcities_validation.py --fail-on warning > /dev/null 2>&1
echo $?
```

which returns `1`.

6.8 Wrapping Up

Pointblank's CLI gives you powerful tools for validating your data, whether you need a quick check or a custom workflow. Use `pb validate` for fast, one-liner checks and `pb run` for more advanced, scriptable validation logic. With clear output and flexible options, you can catch data issues early and keep your workflows running smoothly.

This page provides a complete reference for all Pointblank CLI commands. Each section shows the full help text as it appears in the terminal, giving you quick access to all available options and examples.

For practical usage examples and workflows, see the CLI Data Validation and CLI Data Inspection guides.

6.9 `pb` - Main Command

The main entry point for all Pointblank CLI operations:

```
Usage: pb [OPTIONS] COMMAND [ARGS]...

  Pointblank CLI: Data validation and quality tools for data engineers.

  Use this CLI to validate data quality, explore datasets, and generate
  comprehensive reports for CSV, Parquet, and database sources. Suitable for
  data pipelines, ETL validation, and exploratory data analysis from the
  command line.

  Quick Examples:

    pb preview data.csv              Preview your data
    pb scan data.csv                 Generate data profile
    pb validate data.csv             Run basic validation

  Use pb COMMAND --help for detailed help on any command.

Options:
  -v, --version  Show the version and exit.
  -h, --help     Show this message and exit.

Commands:
  info           Display information about a data source.
  preview        Preview a data table showing head and tail rows.
  scan           Generate a data scan profile report.
  missing        Generate a missing values report for a data table.
  validate       Perform single or multiple data validations.
  run            Run a Pointblank validation script or YAML configuration.
  make-template  Create a validation script or YAML configuration template.
  pl             Execute Polars expressions and display results.
  datasets       List available built-in datasets.
  requirements   Check installed dependencies and their availability.
```

6.10 `pb info` - Data Source Information

Display basic information about a data source:

```
Usage: pb info [OPTIONS] [DATA_SOURCE]

  Display information about a data source.

  Shows table type, dimensions, column names, and data types.

  DATA_SOURCE can be:

  - CSV file path (e.g., data.csv)
  - Parquet file path or pattern (e.g., data.parquet, data/*.parquet)
  - GitHub URL to CSV/Parquet (e.g., https://github.com/user/repo/blob/main/data.csv)
  - Database connection string (e.g., duckdb:///path/to/db.ddb::table_name)
  - Dataset name from pointblank (small_table, game_revenue, nycflights, global_sales)

Options:
  --help  Show this message and exit.
```

6.11 `pb preview` - Data Table Preview

Preview data showing head and tail rows:

```
Usage: pb preview [OPTIONS] [DATA_SOURCE]

  Preview a data table showing head and tail rows.

  DATA_SOURCE can be:

  - CSV file path (e.g., data.csv)
  - Parquet file path or pattern (e.g., data.parquet, data/*.parquet)
  - GitHub URL to CSV/Parquet (e.g., https://github.com/user/repo/blob/main/data.csv)
  - Database connection string (e.g., duckdb:///path/to/db.ddb::table_name)
  - Dataset name from pointblank (small_table, game_revenue, nycflights, global_sales)
  - Piped data from pb pl command

  COLUMN SELECTION OPTIONS:

  For tables with many columns, use these options to control which columns are
  displayed:

  - --columns: Specify exact columns (e.g., --columns "name,age,email")
  - --col-range: Select column range (e.g., --col-range "1:10", --col-range "5:", --col-range ":15")
  - --col-first: Show first N columns (e.g., --col-first 5)
  - --col-last: Show last N columns (e.g., --col-last 3)

  Tables with >15 columns automatically show first 7 and last 7 columns with
  indicators.

Options:
  --columns TEXT             Comma-separated list of columns to display
  --col-range TEXT           Column range like '1:10' or '5:' or ':15'
                             (1-based indexing)
  --col-first INTEGER        Show first N columns
  --col-last INTEGER         Show last N columns
  --head INTEGER             Number of rows from the top (default: 5)
  --tail INTEGER             Number of rows from the bottom (default: 5)
  --limit INTEGER            Maximum total rows to display (default: 50)
  --no-row-numbers           Hide row numbers
  --max-col-width INTEGER    Maximum column width in pixels (default: 250)
  --min-table-width INTEGER  Minimum table width in pixels (default: 500)
  --no-header                Hide table header
  --output-html PATH         Save HTML output to file
  --help                     Show this message and exit.
```

6.12 `pb scan` - Data Profile Reports

Generate comprehensive data profiles:

```
Usage: pb scan [OPTIONS] [DATA_SOURCE]

  Generate a data scan profile report.

  Produces a comprehensive data profile including:

  - Column types and distributions
  - Missing value patterns
  - Basic statistics
  - Data quality indicators

  DATA_SOURCE can be:

  - CSV file path (e.g., data.csv)
  - Parquet file path or pattern (e.g., data.parquet, data/*.parquet)
  - GitHub URL to CSV/Parquet (e.g., https://github.com/user/repo/blob/main/data.csv)
  - Database connection string (e.g., duckdb:///path/to/db.ddb::table_name)
  - Dataset name from pointblank (small_table, game_revenue, nycflights, global_sales)
  - Piped data from pb pl command

Options:
  --output-html PATH  Save HTML scan report to file
  -c, --columns TEXT  Comma-separated list of columns to scan
  --help              Show this message and exit.
```

6.13 `pb missing` - Missing Values Reports

Generate reports focused on missing values:

```
Usage: pb missing [OPTIONS] [DATA_SOURCE]

  Generate a missing values report for a data table.

  DATA_SOURCE can be:

  - CSV file path (e.g., data.csv)
  - Parquet file path or pattern (e.g., data.parquet, data/*.parquet)
  - GitHub URL to CSV/Parquet (e.g., https://github.com/user/repo/blob/main/data.csv)
  - Database connection string (e.g., duckdb:///path/to/db.ddb::table_name)
  - Dataset name from pointblank (small_table, game_revenue, nycflights, global_sales)
  - Piped data from pb pl command

Options:
  --output-html PATH  Save HTML output to file
  --help              Show this message and exit.
```

6.14 `pb validate` - Quick Data Validations

Perform single or multiple data validations:

```
Usage: pb validate [OPTIONS] [DATA_SOURCE]

  Perform single or multiple data validations.

  Run one or more validation checks on your data in a single command. Use
  multiple --check options to perform multiple validations.

  DATA_SOURCE can be:

  - CSV file path (e.g., data.csv)
  - Parquet file path or pattern (e.g., data.parquet, data/*.parquet)
  - GitHub URL to CSV/Parquet (e.g., https://github.com/user/repo/blob/main/data.csv)
  - Database connection string (e.g., duckdb:///path/to/db.ddb::table_name)
  - Dataset name from pointblank (small_table, game_revenue, nycflights, global_sales)

  AVAILABLE CHECK_TYPES:

  Require no additional options:

  - rows-distinct: Check if all rows in the dataset are unique (no duplicates)
  - rows-complete: Check if all rows are complete (no missing values in any column)

  Require --column:

  - col-exists: Check if a specific column exists in the dataset
  - col-vals-not-null: Check if all values in a column are not null/missing

  Require --column and --value:

  - col-vals-gt: Check if column values are greater than a fixed value
  - col-vals-ge: Check if column values are greater than or equal to a fixed value
  - col-vals-lt: Check if column values are less than a fixed value
  - col-vals-le: Check if column values are less than or equal to a fixed value

  Require --column and --set:

  - col-vals-in-set: Check if column values are in an allowed set

  Use --list-checks to see all available validation methods with examples. The
  default CHECK_TYPE is 'rows-distinct' which checks for duplicate rows.

  Examples:

  pb validate data.csv                              # Uses default validation (rows-distinct)
```

```
  pb validate data.csv --list-checks                # Show all available checks
  pb validate data.csv --check rows-distinct
  pb validate data.csv --check rows-distinct --show-extract
  pb validate data.csv --check rows-distinct --write-extract failing_rows_folder
  pb validate data.csv --check rows-distinct --exit-code
  pb validate data.csv --check col-exists --column price
  pb validate data.csv --check col-vals-not-null --column email
  pb validate data.csv --check col-vals-gt --column score --value 50
  pb validate data.csv --check col-vals-in-set --column status --set "active,inactive,pending"

  Multiple validations in one command: pb validate data.csv --check rows-
  distinct --check rows-complete

Options:
  --list-checks         List available validation checks and exit
  --check CHECK_TYPE    Type of validation check to perform. Can be used
                        multiple times for multiple checks.
  --column TEXT         Column name or integer position as #N (1-based index)
                        for validation.
  --set TEXT            Comma-separated allowed values for col-vals-in-set
                        checks.
  --value FLOAT         Numeric value for comparison checks.
  --show-extract        Show extract of failing rows if validation fails
  --write-extract TEXT  Save failing rows to folder. Provide base name for
                        folder.
  --limit INTEGER       Maximum number of failing rows to save to CSV
                        (default: 500)
  --exit-code           Exit with non-zero code if validation fails
  --help                Show this message and exit.
```

6.15 `pb run` - Validation Scripts and YAML

Run Python validation scripts or YAML configurations:

```
Usage: pb run [OPTIONS] [VALIDATION_FILE]

  Run a Pointblank validation script or YAML configuration.

  VALIDATION_FILE can be: - A Python file (.py) that defines validation logic
  - A YAML configuration file (.yaml, .yml) that defines validation steps

  Python scripts should load their own data and create validation objects.
  YAML configurations define data sources and validation steps declaratively.

  If --data is provided, it will automatically replace the data source in your
  validation objects (Python scripts) or override the 'tbl' field (YAML
  configs).

  To get started quickly, use 'pb make-template' to create templates.

  DATA can be:

  - CSV file path (e.g., data.csv)
  - Parquet file path or pattern (e.g., data.parquet, data/*.parquet)
  - GitHub URL to CSV/Parquet (e.g., https://github.com/user/repo/blob/main/data.csv)
  - Database connection string (e.g., duckdb:///path/to/db.ddb::table_name)
  - Dataset name from pointblank (small_table, game_revenue, nycflights, global_sales)

  Examples:

  pb make-template my_validation.py  # Create a Python template
  pb run validation_script.py
  pb run validation_config.yaml
  pb run validation_script.py --data data.csv
  pb run validation_config.yaml --data small_table --output-html report.html
  pb run validation_script.py --show-extract --fail-on error
  pb run validation_config.yaml --write-extract extracts_folder --fail-on critical

Options:
  --data TEXT                     Data source to replace in validation objects
                                  (Python scripts and YAML configs)
  --output-html PATH              Save HTML validation report to file
  --output-json PATH              Save JSON validation summary to file
  --show-extract                  Show extract of failing rows if validation
                                  fails
  --write-extract TEXT            Save failing rows to folders (one CSV per
                                  step). Provide base name for folder.
  --limit INTEGER                 Maximum number of failing rows to save to
```

```
                                  CSV (default: 500)
  --fail-on [critical|error|warning|any]
                                  Exit with non-zero code when validation
                                  reaches this threshold level
  --help                          Show this message and exit.
```

6.16 `pb make-template` - Template Generation

Create validation script or YAML configuration templates:

```
Usage: pb make-template [OPTIONS] [OUTPUT_FILE]

  Create a validation script or YAML configuration template.

  Creates a sample Python script or YAML configuration with examples showing
  how to use Pointblank for data validation. The template type is determined
  by the file extension: - .py files create Python script templates -
  .yaml/.yml files create YAML configuration templates

  Edit the template to add your own data loading and validation rules, then
  run it with 'pb run'.

  OUTPUT_FILE is the path where the template will be created.

  Examples:

  pb make-template my_validation.py        # Creates Python script template
  pb make-template my_validation.yaml      # Creates YAML config template
  pb make-template validation_template.yml # Creates YAML config template

Options:
  --help  Show this message and exit.
```

6.17 `pb pl` - Polars Expression Execution

Execute Polars expressions and display results:

```
Usage: pb pl [OPTIONS] [POLARS_EXPRESSION]

  Execute Polars expressions and display results.

  Execute Polars DataFrame operations from the command line and display the
  results using Pointblank's visualization tools.

  POLARS_EXPRESSION should be a valid Polars expression that returns a
  DataFrame. The 'pl' module is automatically imported and available.

  Examples:

  # Direct expression
  pb pl "pl.read_csv('data.csv')"
  pb pl "pl.read_csv('data.csv').select(['name', 'age'])"
  pb pl "pl.read_csv('data.csv').filter(pl.col('age') > 25)"

  # Multi-line with editor (supports multiple statements)
  pb pl --edit

  # Multi-statement code example in editor:
  # csv = pl.read_csv('data.csv')
  # result = csv.select(['name', 'age']).filter(pl.col('age') > 25)

  # Multi-line with a specific editor
  pb pl --edit --editor nano
  pb pl --edit --editor code
  pb pl --edit --editor micro

  # From file
  pb pl --file query.py

  Piping to other pb commands
  pb pl "pl.read_csv('data.csv').head(20)" --pipe | pb validate --check rows-distinct
  pb pl --edit --pipe | pb preview --head 10
  pb pl --edit --pipe | pb scan --output-html report.html
  pb pl --edit --pipe | pb missing --output-html missing_report.html

  Use --output-format to change how results are displayed:
  pb pl "pl.read_csv('data.csv')" --output-format scan
  pb pl "pl.read_csv('data.csv')" --output-format missing
  pb pl "pl.read_csv('data.csv')" --output-format info

  Note: For multi-statement code, assign your final result to a variable like
```

```
  'result', 'df', 'data', or ensure it's the last expression.

Options:
  -e, --edit                      Open editor for multi-line input
  -f, --file PATH                 Read query from file
  --editor TEXT                   Editor to use for --edit mode (overrides
                                  $EDITOR and auto-detection)
  -o, --output-format [preview|scan|missing|info]
                                  Output format for the result
  --preview-head INTEGER          Number of head rows for preview
  --preview-tail INTEGER          Number of tail rows for preview
  --output-html PATH              Save HTML output to file
  --pipe                          Output data in a format suitable for piping
                                  to other pb commands
  --pipe-format [parquet|csv]     Format for piped output (default: parquet)
  --help                          Show this message and exit.
```

6.18 `pb datasets` - Built-in Datasets

List available built-in datasets:

```
Usage: pb datasets [OPTIONS]

  List available built-in datasets.

Options:
  --help  Show this message and exit.
```

6.19 `pb requirements` - Dependency Check

Check installed dependencies and their availability:

```
Usage: pb requirements [OPTIONS]

  Check installed dependencies and their availability.

Options:
  --help  Show this message and exit.
```

6.20 Common Data Source Types

All commands that accept a `DATA_SOURCE` parameter support these formats:

- **CSV files**: `data.csv`, `path/to/data.csv`
- **Parquet files**: `data.parquet`, `data/*.parquet` (patterns supported)
- **GitHub URLs**: `https://github.com/user/repo/blob/main/data.csv`
- **Database connections**: `duckdb:///path/to/db.ddb::table_name`
- **Built-in datasets**: `small_table`, `game_revenue`, `nycflights`, `global_sales`
- **Piped data**: Output from `pb pl` command (where supported)

6.21 Exit Codes and Automation

Many commands support options useful for automation and CI/CD:

- `--exit-code`: Exit with non-zero code on validation failure
- `--fail-on [critical|error|warning|any]`: Control failure thresholds
- `--output-html`, `--output-json`: Save reports for external consumption
- `--write-extract`: Save failing rows for investigation

These features make Pointblank CLI commands suitable for integration into data pipelines, quality gates, and automated workflows.